Poly-Controlled Partial Evaluation

Germán Puebla
School of Computer Science
Technical University of Madrid
28660 Boadilla del Monte, Spain
german@fi.upm.es

Claudio Ochoa
School of Computer Science
Technical University of Madrid
28660 Boadilla del Monte, Spain
claudio@fi.upm.es

Abstract

Existing algorithms for on-line partial evaluation of logic programs, given an initial program and a description of run-time queries, deterministically produce a specialized program. In this work we propose a novel framework for partial evaluation of logic programs which is *poly-controlled* in that it can take into account repertoires of global control and local control rules instead of a single, predetermined combination. This approach is more flexible than existing ones since it allows assigning *different* global and local control rules to different call patterns, thus obtaining results that cannot be obtained using traditional partial evaluation. This modification transforms partial evaluation from a *greedy* algorithm into a *search-based* algorithm and, as a result, sets of candidate specialized programs can be achieved, instead of a single one. In order to make the algorithm fully automatic, it requires the use of self-tuning techniques which allow automatically measuring the quality of the different candidate specialized programs. Our approach is resource aware in that it uses fitness functions which consider multiple factors such as run-time and code size for the specialized programs. The framework has been implemented in the CiaoPP system, and tested on some benchmarks. The preliminary experimental results we present show that our proposal obtains better specializations than those achieved using traditional partial evaluation.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming

General Terms Languages, Performance

Keywords Partial Evaluation, Control Rules, Optimization

1. Introduction

The aim of partial evaluation (*PE*) is to specialize a program w.r.t. part of its input, which is known as the *static data*[11]. The quality of the code generated by partial evaluation greatly depends on the *control strategy* used. Traditional algorithms for partial evaluation (PE) of logic programs (LP) are parametric w.r.t. the so-called *global control* and *local control* rules. The issue of devising good control rules has received considerable attention (see for example [8] and its references).

PPDP'06 July 10–12, 2006, Venice, Italy.

However, the existence of sophisticated control rules which behave (almost) optimally for all programs is still far from reality. Furthermore, the existing control rules focus on time-efficiency by trying to reduce the number of resolution steps which are performed in the residual program. Other factors such as the size of and the memory required to run the residual program are most often neglected, a relevant exception being the work in [4]. In addition to potentially generating larger programs, it is well known that partial evaluation can slow-down programs due to lower level issues such as clause indexing, cache sizes, etc.

Existing partial evaluators usually provide several global and local control strategies, as well as many other parameters (global trees, computation rules, etc.) directly affecting the quality of the obtained solution. For a novice user, it is extremely hard to find the right combination of parameters to achieve the desired results (reduction of size of compiled code, reduction of execution time, etc.). Even for an experienced user, it is rather difficult to predict the behavior of partial evaluation, especially in terms of space-efficiency (size of the residual program).

Also, once a choice of global and local control rules is made, such a combination will be applied to all call patterns in the residual program. Obviously, in practice, it can be very useful to be able to use *different* global and local control rules for *different* call patterns, thus obtaining results that cannot be produced using traditional partial evaluation with any given control strategy.

In this work we propose a novel framework for on-line partial evaluation which:

1. allows using different global and local control rules for different call patterns (atoms) and
2. can generate several candidate specializations. These specializations can then be empirically compared for efficiency, in terms of multiple factors such as size of the specialized program and time- and memory-efficiency of such specialized program.

The framework is *self-tuning* in that, as mentioned above, it uses empirical evaluations for selecting the best candidates by means of a *fitness function*. It is also *resource-aware* in that multiple factors, such as size of specialized programs and their memory consumption can be taken into account by the fitness function in addition to the natural consideration of time-efficiency of the specialized programs. In [3], a self-tuning, resource aware *off-line* specialization technique is introduced. In contrast, our approach performs *on-line* partial evaluation, and thus can take advantage of the great body of work available for *on-line* partial evaluation of logic programs.

The rest of the paper is organized as follows. In Sec. 2 some required background on the basics of partial evaluation of logic programs is provided. Sec. 3 illustrates, by means of an example, the difficulty of choosing the right control strategies. In Sec. 4 we express a traditional partial evaluation framework as a greedy

algorithm. Then, in Sec. 5 we introduce a first approach to poly-controlled partial evaluation and, finally, in Sec. 6 we show an improved approach to poly-controlled partial evaluation. In Sec. 7 we describe how to make this algorithm self-tuning and resource-aware. Sec. 8 presents some preliminary experimental results. Finally, in Sec. 9 we discuss some related work and conclude.

2. Background

We assume some basic knowledge on the terminology of logic programming. See for example [12] for details.

Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function $pred$ applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $\exists \sigma.\ t\sigma = s$. A *unifier* of a pair of terms $\{t_1, t_2\}$ is a substitution σ such that $t_1\sigma = t_2\sigma$. A unifier σ is called *most general unifier* (mgu) if for every other unifier σ', $t_1\sigma \leq t_2\sigma'$.

2.1 Basics of Partial Evaluation in LP

Partial evaluation of LP is traditionally presented in terms of SLD semantics. We briefly recall the terminology here. We will provide a concrete algorithm for partial evaluation in Section 4. The concept of *computation rule* is used to select an atom within a goal for its evaluation.

DEFINITION 2.1 (computation rule).

A computation rule *is a function $\mathcal{R}$ from goals to atoms. Let G be a goal of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, $k \geq 1$. If $\mathcal{R}(G) = A_R$ we say that A_R is the* selected *atom in G.*

The operational semantics of definite programs is based on derivations [12].

DEFINITION 2.2 (derivation step).

Let G be $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P. Then G' is derived *from G and C via $\mathcal{R}$ if the following conditions hold:*

$$\theta = mgu(A_R, H)$$

G' is the goal $\leftarrow \theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$

As customary, given a program P and a goal G, an *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n.

In partial evaluation, SLD semantics is extended in order to also allow *incomplete derivations* which are finite derivations of the form $G = G_0, G_1, G_2, \ldots, G_n$ and where no atom is selected in G_n for further resolution. This is needed in order to avoid (local) non-termination of the specialization process. Also, the substitution $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer substitution for goal G. An *incomplete SLD tree* possibly contains incomplete derivations.

In order to compute a *partial evaluation* (PE) [11], given an input program and a set of atoms (goal), the first step consists in applying an *unfolding rule* to compute finite incomplete SLD trees for these atoms. Then, a set of *resultant*s or residual rules are systematically extracted from the SLD trees.[1]

DEFINITION 2.3 (unfolding rule).

Given an atom A, an unfolding rule computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with computer answer substitution θ_i for $i = 1, \ldots, n$ whose associated resultants *are $\theta_i(A) \leftarrow G_i$.*

Therefore, this step returns the set of resultants, i.e., a program, associated to the root-to-leaf derivations of these trees. The set of resultants for the computed SLD tree is called a *partial evaluation* for the initial goal (query). The partial evaluation for a set of goals is defined as the union of the partial evaluations for each goal in the set. We refer to [8] for details.

In order to ensure the local termination of the PE algorithm while producing useful specializations, the unfolding rule must incorporate some non-trivial mechanism to stop the construction of SLD trees. Nowadays, well-founded orderings (wfo) [2, 13] and well-quasi orderings (wqo) [15, 9] are broadly used in the context of on-line partial evaluation techniques (see, e.g., [5, 10, 15]).

In addition to local termination, an *abstraction operator* is applied to properly add the atoms in the right-hand sides of resultants to the set of atoms to be partially evaluated. This abstraction operator performs the *global control* and is in charge of guaranteeing that the number of atoms which are generated remains finite by replacing atoms by more general ones, i.e., by losing precision in order to guarantee termination. The abstraction phase yields a new set of atoms, some of which may in turn need further evaluation and, thus, the process is iteratively repeated while new atoms are introduced.

3. The Dilemma of Controlling PE

As mentioned above, there exist many powerful local and global control rules to choose from. Just as an example, in the case of local control, a decision to be taken is whether to allow non-leftmost unfolding or not. It is well known that performing unfolding steps w.r.t. atoms which are not leftmost can slow-down programs, and, in the presence of *impure* predicates, non-leftmost unfolding can even produce incorrect results [1]. On the other hand, performing non-leftmost unfolding can provide important gains in other cases. See, for example, the program in Listing 1.

```
exp(Base,Exp,Res):- exp_ac(Exp,Base,1,Res).

exp_ac(0,_,Res,Res).
exp_ac(Exp,Base,Tmp,Res):-
        Exp > 0,
        Exp1 is Exp - 1,
        NTmp is Tmp * Base,
        exp_ac(Exp1,Base,NTmp,Res).
```

Listing 1. The `exponential/3` example

If we specialize it w.r.t. the query `exp(Base,2,Res)`, enabling non-leftmost unfolding allows to unroll the recursive calls. The residual code, after some simple arithmetic simplifications[2], is shown in Listing 2.

[1] Let us note that the definition of a partial evaluation *algorithm* requires, in addition to an unfolding rule, the so-called global control level (see Section 1).

[2] The specializer in CiaoPP actually performs such simplifications of arithmetic operations.

```
exp(A,2,B) :-  B is A*A.
```

Listing 2. Residual code of the `exponential/3` example

Consider now the program in Listing 3 below. Since the call "`C is B + 1`" to the built-in predicate `is/2` is not sufficiently instantiated to be executed (`B` is not yet bound to an arithmetic expression), it is required to enable non-leftmost unfolding in order to unfold the call `q(C)`.

```
p(B):-  C is B + 1, q(C).

q(1).
q(2).
q(3).
q(4).
q(5).
q(6).
```

Listing 3. The `p/1` example

However, such unfolding generates the residual code shown in Listing 4.

```
p(A) :- 1 is A+1.
p(A) :- 2 is A+1.
p(A) :- 3 is A+1.
p(A) :- 4 is A+1.
p(A) :- 5 is A+1.
p(A) :- 6 is A+1.
```

Listing 4. The residual code for `p/1`

This code is less efficient than the original definition of `p/1`, since the indexing for predicate `q/1` is lost and the calls to built-in `is/2` have to be speculatively performed until a success is found, if any. For example for values of $A \geq 6$, six calls to `is/2` are always performed, whereas just one was needed in the original program.

In summary, the same feature of a local control rule, i.e., whether to allow non-leftmost unfolding, can be beneficial for certain calls (atoms) and can be counterproductive in others. Though one could argue that a good rule of thumb can be to only perform non-leftmost unfolding for determinate atoms, i.e., those which only unify with a single clause head, this heuristic does not guarantee to always achieve the best specialization possible: an atom whose resolution is not determinate can become deterministic later on, since maybe just one (or even none) of the derivations which contain such step is successful or incomplete (i.e., all the rest are failing derivations). Note that the problem of deciding whether an atom is deterministic is undecidable: it can always happen that an SLD tree which contains several non-failing derivations at some depth, contains at most one non-failing derivation in the next depth level.

3.1 A Motivating Example

We now show in Listing 5 a program which defines the predicate `main/3` containing calls to the predicates `exp/3` and `p/1` defined before:

```
main(A,B,C):- exp(B,2,Result), p(A).
```

Listing 5. A motivating example

In Listing 6 we can see the residual code obtained when specializing this program w.r.t. the query `main(A,B,C)` using leftmost unfolding. Note that none of the calls to the built-in predicate `is/2` are sufficiently instantiated to be executed at specialization time. Since only leftmost unfolding is allowed, the unfolding trees computed are not very deep, resulting in a large number of residual predicates. On the other hand, if we choose to enable non-leftmost unfolding, we obtain the residual program shown in Listing 7, where only an SLD tree has been required, and thus no auxiliary predicates are defined.

```
main(A,B,C) :-
        D is 1*B,
        exp_ac_1(1,B,D,C),
        p_1(A).

exp_ac_1(1,A,B,C) :- D is B*A, exp_ac_2(0,A,D,C).

exp_ac_2(0,_1,A,A).

p_1(A) :- B is A+1, q_1(B).

q_1(1).
q_1(2).
q_1(3).
q_1(4).
q_1(5).
q_1(6).
```

Listing 6. Result with leftmost unfolding

Unfortunately, neither the program in Listing 6 nor the one in Listing 7 is optimal. This is because, in order to achieve an optimal result, non-leftmost unfolding should be used for atoms for predicate `exp/3`, but only leftmost unfolding should be used for atoms for predicate `p/1`.

```
main(A,B,C) :- D is 1*B, C is D*B, 1 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 2 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 3 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 4 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 5 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 6 is A+1.
```

Listing 7. Result with non-leftmost unfolding

Note that although the rule of thumb discussed above for non-leftmost unfolding happens to provide good results in this example, clearly there is no unfolding rule which uniformly obtains the optimal results in all cases.

4. PE as a Greedy Algorithm

As it is well known, *greedy algorithms* are characterized by starting from an initial *configuration* (or state) $Conf_0$ and repeatedly applying a *transformation rule* T which given a configuration $Conf_i$ produces a successor configuration $Conf_{i+1}$ s.t. $Conf_{i+1}$=$T(Conf_i)$. This process is repeated until a configuration $Conf_n$, $n \geq 0$, is reached which satisfies certain conditions guaranteeing that $Conf_n$ is *final*.

As we show in Algorithm 1 below, it is possible to consider traditional partial evaluation frameworks as greedy algorithms. A configuration $Conf_i$ is a pair $\langle S_i, H_i \rangle$ s.t. S_i is the set of atoms yet to be handled by the algorithm and H_i is the set of atoms already handled by the algorithm. Indeed, in H_i not only we store atoms but also the result of applying global control to such atoms, i.e., members of H_i are pairs of the form $\langle A_i, A'_i \rangle$. Correctness of the algorithm requires that each A'_i is an *abstraction* of A_i, i.e., $A_i = A'_i\theta$.

Given a set of atoms S which describe the potential queries to the program, the initial configuration is of the form $\langle S, \emptyset \rangle$. In each iteration of the algorithm, an atom A_i from S is selected (line 5). Then, global control and local control as defined by the *Abstract* and *Unfold* rules, respectively, are applied (lines 6 and 7). This builds an SLD-tree for A'_i, a generalization of A_i as determined by *Abstract*, using the predefined unfolding rule *Unfold*. Once the SLD-tree τ_i is computed, the leaves in its resultants, i.e., the atoms in the residual code for A'_i are collected by the function *leaves*. Those atoms in *leaves*(τ_i) which are not a variant of an atom handled in previous iterations of the algorithm are added to the set of atoms to be considered (S_{i+1}). We use $B \equiv A$ to denote that B and A are *variants*, i.e., they are equal modulo variable renaming. A

Algorithm 1 Greedy Partial Evaluation algorithm

Input: Program P
Input: Set of atoms of interest S
Input: A global control rule *Abstract*
Input: A local control rule *Unfold*
Output: A partial evaluation for P and S, encoded by H_i

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: **repeat**
5: $A_i = Select(S_i)$
6: $A'_i =$ *Abstract*(H_i, A_i)
7: $\tau_i =$ *Unfold*(P, A'_i)
8: $H_{i+1} = H_i \cup \{\langle A_i, A'_i\rangle\}$
9: $S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_\rangle \in H_{i+1} \,.\, B \not\equiv A\}$
10: $i = i + i$
11: **until** $S_i = \emptyset$

configuration is final when it is of the form $\langle\emptyset, H\rangle$. The specialized program corresponds to $\bigcup_{\langle A,A'\rangle\in H_n} resultants(A')$.

Note that this algorithm differs from those in [5, 8] in that once an atom A_i is abstracted into A'_i, code for A'_i will be generated, and it will not be abstracted any further no matter which other atoms are handled in later iterations of the algorithm. As a result, the set of atoms for which code is generated are not guaranteed to be *independent*. Two atoms are independent when they have no common instance. However, the pairs in H uniquely determine the version used at each program point. Since code generation produces a new predicate name per entry in H, independence is guaranteed, and thus the specialized program will not produce more solutions than the original one. The ECCE system [10] can be made to behave as Algorithm 1 by setting the *parent abstraction* flag to *off*.

5. A Poly-Controlled PE Algorithm

As we have seen, in the greedy algorithm given a configuration $\langle S_i, H_i\rangle$, and once we decide to continue the computation using $A_i \in S_i$, there is only one successor configuration which is $T(\langle S_i, H_i\rangle)$. However, it is well known that several control strategies exist which can be of interest in different circumstances. It is indeed a rather difficult endeavor to find a pair of global control and local control rules which behaves well in all settings. Thus, rather than considering a single global control and local control rule, at least in principle one can be interested in applying *different* local and global control rules to *different* atoms (call patterns). Unfortunately, this is something which existing algorithms for partial evaluation do not cater for. If we allow different combinations of global and local control rules, given a configuration, there is no longer a single successor in the computation of the algorithm but possibly several ones. In fact, given a set of unfolding rules $\mathcal{U} = \{$*Unfold*$_1, \ldots,$ *Unfold*$_i\}$, and a set of abstraction functions $\mathcal{G} = \{$*Abstract*$_1, \ldots,$ *Abstract*$_j\}$, there are $i \times j$ possible combinations.

Algorithm 2 shows a *poly-controlled* partial evaluation algorithm. We refer to this algorithm as poly-controlled because it allows the use of multiple control strategies and use different ones for different atoms. The choice of the control strategy to apply during the handling of each atom is performed by the $Pick$ function which given an atom A_i, a history H_i, a set of global control rules, and a set of local control rules, picks up a pair $\langle$*Abstract*, *Unfold*$\rangle$ among all possible ones.

This algorithm differs from the greedy algorithm seen in Section 4 in several ways. One is that rather than receiving as input an abstraction function and an unfolding rule, it receives *a set* of global control rules and *a set* of local control rules. Another difference is that the tuples in set H_i now contain not only an atom and the result of abstracting it, but also the unfolding rule which has been picked for unfolding such atom. That is needed in order to use exactly such unfolding rule during the code generation phase. Indeed, the specialized program corresponds to $\bigcup_{\langle A,A',Unfold\rangle\in H_n} resultants(A', Unfold)$, where the function *resultants* is now parametric w.r.t. the unfolding rule. This allows applying the same unfolding rule during code generation as it was applied during the execution of Algorithm 2. The third and final difference corresponds to the addition of the function *Pick* used in line 6, and whose role has already been described above.

Algorithm 2 Poly-Controlled Partial Evaluation algorithm

Input: Program P
Input: Set of atoms of interest S
Input: Set of unfolding rules $\mathcal{U}$
Input: Set of generalization functions $\mathcal{G}$
Input: Selection function $Pick$
Output: A partial evaluation for P and S, encoded by H_i

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: **repeat**
5: $A_i = Select(S_i)$
6: $\langle$*Abstract*, *Unfold*$\rangle = Pick(A_i, H_i, \mathcal{G}, \mathcal{U})$
7: $A'_i =$ *Abstract*(H_i, A_i)
8: $\tau_i =$ *Unfold*(P, A'_i)
9: $H_{i+1} = H_i \cup \{\langle A_i, A'_i,$ *Unfold*$\rangle\}$
10: $S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_, \_\rangle \in H_{i+1} \,.\, B \not\equiv A\}$
11: $i = i + 1$
12: **until** $S_i = \emptyset$

Clearly, different choices for the *Pick* function will result in different specialized programs. It is important to note that the finer-grained control of poly-controlled partial evaluation can potentially produce specialized programs which are hard or even impossible to obtain by using off-the-shelf control strategies. Also, the addition of the *Pick* function conceptually makes the poly-controlled partial evaluation algorithm being composed of three levels of control, the local control, the global control, and the *search control*, which is determined by the function *Pick*. Note that the inclusion of the history as an input argument to *Pick* allows to make hopefully more informed decisions.

6. Search-based Specialization

Poly-controlled algorithms can provide better specializations than those achievable by traditional partial evaluation algorithms by assigning different control strategies to different atoms. However, the improvements achieved rely on the behavior of the function *Pick*. Unfortunately, choosing a good *Pick* function can be a very hard task.

Another alternative is, instead of deciding *a priori* the control strategy to apply to each atom, to generate several (or even all) candidate partial evaluations and then decide *a posteriori* which specialized program to use. In the extreme, this can be done by computing all possible combinations of global and local control rules and exploring the whole search space in order to generate not just a specialized program but rather a collection of specialized programs.

Algorithm 3 shows an all-solutions search-based algorithm. In this case, there is no longer a single successor configuration state for each atom to unfold, but several of them. This can be interpreted as, given $\mathcal{G}$=$\{A_1, \ldots, A_j\}$ and $\mathcal{U}$=$\{U_1, \ldots, U_i\}$, we now have a *set* of transformation operators $T_{U_1}^{A_1}, \ldots, T_{U_i}^{A_1}$, $\ldots, T_{U_i}^{A_j}$. Obviously, in general we will be interested in selecting only one specialized program out of all final programs obtained. Clearly, generating all possible candidate specialized programs is more costly than computing just one. However, selecting the best candidate a posteriori allows to make much more informed decisions than selecting it a priori, as in Algorithm 2.

Algorithm 3 All-candidates Search-based Partial Evaluation algorithm

Input: Program P
Input: Set of atoms of interest S
Input: Set of unfolding rules $\mathcal{U}$
Input: Set of generalization functions $\mathcal{G}$
Output: Set of partial evaluations *Sols*

1: $H_0 = \emptyset$
2: $S_0 = S$
3: create(*Confs*); *Confs* = push($\langle S_0, H_0 \rangle$, *Confs*)
4: *Sols* = $\emptyset$
5: **repeat**
6: $\langle S_i, H_i \rangle$ = pop(*Confs*)
7: $A_i = Select(S_i)$
8: $Candidates$ = $\{\langle$*Abstract*, *Unfold*$\rangle$ | *Abstract* $\in \mathcal{G}$, *Unfold* $\in \mathcal{U}\}$
9: **repeat**
10: $Candidates = Candidates - \{\langle$*Abstract*, *Unfold*$\rangle\}$
11: A'_i = *Abstract*(H_i, A_i)
12: τ_i = *Unfold*(P, A'_i)
13: $H_{i+1} = H_i \cup \{\langle A_i, A'_i,$ *Unfold*$\rangle\}$
14: $S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_, \_\rangle \in H_{i+1} \,.\, B \not\equiv A\}$
15: **if** S_{i+1}=$\emptyset$ **then**
16: *Sols* = *Sols* $\cup \{H_{i+1}\}$
17: **else**
18: push($\langle S_{i+1}, H_{i+1} \rangle$,*Confs*)
19: **end if**
20: **until** $Candidates$ = $\emptyset$
21: $i = i + 1$
22: **until** empty_stack(*Confs*)

The main difference between Algorithm 3 and Algorithm 2 is that there are now two additional data structures. One is *Confs*, which contains the configurations which are currently being explored. The other one is *Sols*, which stores the set of solutions found by the algorithm. As it is well known, the use of different data structures for *Confs* provides different traversals of the search space. Currently, Algorithm 3 uses a stack, thus the search space is traversed in a depth-first fashion. Note that Algorithm 3 does not work with single configurations but rather with stacks of configurations. The process terminates when the stack of configurations to handle is empty, i.e. all final configurations have been reached.

6.1 Searching for All Specializations in our Motivating Example

Consider again the motivating example in Listing 5. Consider also two local control rules, one performing leftmost unfolding only, and the other one performing also non-leftmost unfolding, i.e., $\mathcal{U}$=$\{$*leftmost*, *nonleftmost*$\}$ and one global control rule *id* returning always the same atom, i.e., $\mathcal{G}$=$\{id\}$. By applying Algorithm 3 we get five different specialized programs. In particular, *Solution1* corresponds to the program in Listing 6 and *Solution5* to the program in Listing 7. In addition, our algorithm also produces three other candidate programs which are hybrid in the sense that they use different control rules for different atoms, and thus cannot be achieved using *leftmost+id* nor *nonleftmost+id* only.

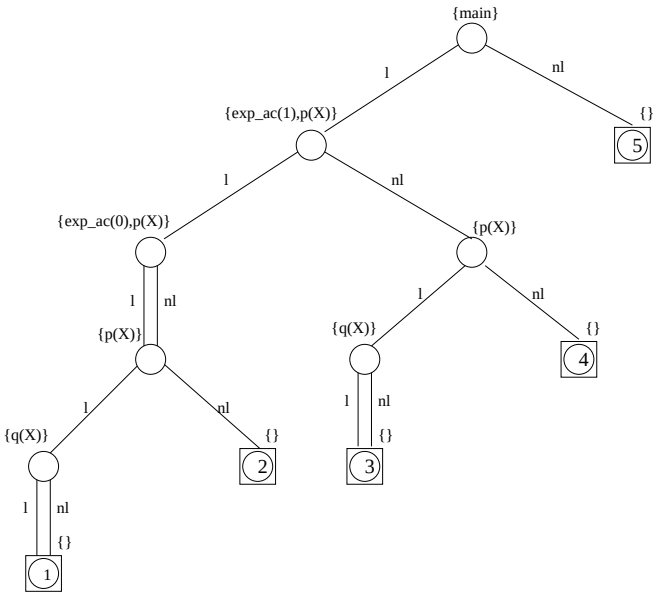

Figure 1. Search space for the motivating example

Program	Run Time	Size	Speedup	Code Reduc
Original	5890	1606	1.00	1.00
Solution1	3652	1596	1.61	1.01
Solution2	5138	1543	1.15	1.04
Solution3	2931	1379	2.01	1.16
Solution4	3962	1326	1.49	1.21
Solution5	7223	1321	0.82	1.22

Table 1. Comparison of Solutions

The search space for this example is shown in Figure 1. There, each configuration is represented with a circle. Configurations which are final are marked with a square around the circle. As can be seen, the whole search space for the example consists of 12 configurations, 7 of which are not final and 5 are final, and thus correspond to different candidate solutions, as already mentioned.

Each configuration is adorned with the set of atoms yet to be handled, i.e., S_i in the algorithm. Each node can have two descendants, which are indicated with arcs. Arcs are labeled either `l`, for *leftmost* or `nl` for *non-leftmost*. The set of nodes already handled is not shown explicitly in each node, but it is implicitly represented by traversing the tree from each node upwards up to the root, since an atom is handled in each node. For example, in the case of *Solution3*, the history is $\{\langle q(B), q(B), nl\rangle, \langle p(A), p(A), l\rangle, \quad \langle exp\_ac(1, A, B, C), exp\_ac(1, A, B, C), nl\rangle, \langle main(A, B, C), \; main(A, B, C), l\rangle\}$. Also, some nodes only have one descendant linked by two arcs to its parent. This indicates that the two control strategies considered produce equivalent configurations, reducing the search space.

Table 1 provides a comparison of the different candidate solutions together with the original program. The first column indicates the program we refer to in each row. The second column provides

an indication of the run-time efficiency of the different programs. This time has been obtained by running a million times the query `main(8,9,Result)` and subtracting the time required by an empty loop which performs a million iterations. The third column compares the sizes of the different programs. This size is in number of bytes of the program compiled into byte-code using Ciao-1.13 and after subtracting the size of an empty program. Finally, the last two columns compare the run-time and code-size of the different programs with that of the original program.

As it can be seen, not all programs obtained by partial evaluation are necessarily faster than the original one. In particular, *Solution5*, the one obtained using non-leftmost unfolding for all cases is less efficient than the original one. This is indicated by an speedup lower than 1, which is 0.82 in this case. On the other hand, the speedup obtained by *Solution1* is 1.61, but it is still far from the fastest program, which is *Solution3* with an speedup of 2.01. As regards code size, in this particular case all solutions achieved are smaller than the original program, though it is well-known that in some cases partial evaluation can produce programs which are significantly larger than the original one. The smallest program is *Solution5*, with a code reduction of 1.22, but which happens to be the slowest program of all, including the original one.

If both the speedup and code reduction factors are taken into account, the most promising programs are probably *Solution3* and *Solution4*, neither of which are achievable by using one unfolding rule for all atoms. If code size is not a very pressing issue, then *Solution3* is probably the best one, but otherwise *Solution4* should be used, since a relative small increase in program size provides significant time performance improvement. The choice between the two solutions mentioned will depend on the fitness function used, which can put more emphasis in one factor or another.

7. Self-Tuning, Resource-Aware PE

Though Algorithm 3 can be used to automatically generate a large number of candidate specialized programs to choose from, we need some mechanism to automatically select just one of them since, obviously, the goal of partial evaluation is to obtain a specialized program, not many. There are certainly several criteria which can be used in order to decide how good a specialized program is. The framework we propose in this work is *resource-aware* since it can take the following criteria into account.

Time efficiency: currently we are measuring speedup w.r.t the original program. In this case, we need a set of test cases which are representative of the class of run-time queries which will be performed. Another possibility to be explored is the use of static cost analysis. Cost analysis can aim at obtaining upper or lower bounds on computational cost or even average cost.

Size of compiled code: fairly easy to measure. It can be an important factor if the program will run on devices with limited resources, as is the case in embedded systems and pervasive computing. Also, even in cases where code size is not much of an issue, it can happen that different specialized programs have similar time-efficiency but some of them can be significantly larger than others.

Memory-consumption: it can be of interest when resources are scarce, similarly to the case of size of compiled code.

Our framework is fully automatic, i.e., there is no need for human intervention in order to decide which is the best among the candidate specializations. We refer to this as a *self-tuning* approach. A *fitness function* assigns a numeric value to each candidate specialization, reflecting how good the corresponding program is. The framework is parametric w.r.t. the fitness function so that the method can be applied with different aims in mind. Sometimes we may be interested in achieving code which is as time-efficient as possible, whereas in other cases space-efficiency can be a primary aim. It is important to note that this search-based approach to partial evaluation is also of interest when only run-time is taken into account. Even in such case there is no control strategy alone which is guaranteed to always produce the most-efficient code for all compilers and architectures.

8. Preliminary Evaluation

In order to perform a preliminary assessment of the benefits and practicality of search based poly-controlled partial evaluation (PCPE), we have conducted a series of experiments using the `CiaoPP` [14, 6] system.

Although the search-based approach presented in Section 6 above is definitely appealing, it is worth investigating whether it can actually produce better specializations than traditional partial evaluation (PE) and also whether it produces too large a number of candidates, even for small programs.

In the implementation, a first obvious optimization was to eliminate equivalent configurations which were descendants of the same node in the search tree. I.e., it is often the case that given a configuration *Conf* there are more than one T_U^A and $T_{U'}^{A'}$ with $(A,U) \neq (A',U')$ s.t. $T_U^A(Conf) = T_{U'}^{A'}(Conf)$. This optimization is easy to implement, not very costly to execute, and reduces search space significantly. For example, in the search space in Figure 1, which already includes this optimization, if this optimization were not applied then it would contain 19 configurations, instead of 12 and there would be 9 candidate solutions instead of 5.

In our evaluation we have compared two extreme cases, i.e., traditional partial evaluation with the search-based algorithm in all-solutions mode. In our experiments, we have used the following set of global control rules $\mathcal{G}$=$\{dynamic, hom\_emb\}$. The hom_emb global control rule is based on homeomorphic embedding [8, 9] and flags atoms as potentially dangerous (and are thus generalized) when they homeomorphically embed any of the previously visited atoms. Then, $dynamic$ is the most abstract possible global control rule, which abstracts away the value of all arguments of the atom and replaces them with distinct variables. Also, the set of local control rules used is $\mathcal{U}$=$\{one\_step, df\_hom\_emb\_as\}$. The rule one_step is the simplest possible unfolding rule which always performs just one unfolding step for any atom. Finally, $df\_hom\_emb\_as$ is an unfolding rule based on homeomorphic embedding. More details on this unfolding rule can be found in [14]. It can handle external predicates safely and can perform non-leftmost unfolding as long as unfolding is safe (see [1]) and *local* (see [14]). We have chosen these particular global and local control rules since, on the one hand, they guarantee termination, and, on the other hand, they allow us to contrast aggressive and conservative unfolding. In this way, we expect to obtain more heterogeneous candidate solutions.

When testing search-based poly-controlled partial evaluation in all solutions mode, for the control rules mentioned above, we have found out that the approach copes with many of the benchmarks by Lam & Kusalik [7]. However, these benchmarks are of relatively little interest to our technique, since many of them can be fully unfolded. Thus, in general, traditional partial evaluation obtains good results, and the solutions provided by PCPE were solutions achievable by traditional PE, i.e., solutions using always one global control rule and one local control rule for all atoms.

However, in practice, it is often the case that programs being partially evaluated cannot be fully unfolded since the static information available is not sufficient to do so. In our experimentation with the technique we have found out that, at least for the com-

Input query	#solutions
rev(L,R)	6
rev([_|L],R)	48
rev([_,_|L],R)	117
rev([_,_,_|L],R)	186
rev([_,_,_,_|L],R)	255
rev([1|L],R)	129
rev([1,2|L],R)	480

Table 2. Solutions generated by PCPE for rev benchmark

bination of control rules used, the number of candidate specializations grows rapidly with the amount of static data available in the specialization query. In order to illustrate this phenomenon, let us consider the program in Listing 8, which implements a naive reverse algorithm:

```
:- module(_,[rev/2],[]).

:- entry rev([_,_|L],R).

rev([],[]).
rev([H|L],R):-
        rev(L,Tmp),
        app(Tmp,[H],R).


app([],L,L).
app([X|Xs],Y,[X|Zs]):-
        app(Xs,Y,Zs).
```

Listing 8. The `rev/2` example

In CiaoPP, the description of initial queries (i.e., the set of atoms of interest S in algorithms 1 to 3) is obtained by taking into account the set of predicates exported by the module, in this case `rev/2`, possibly qualified by means of *entry* declarations. For example, the *entry* declaration in Listing 8 is used to specialize the naive reverse procedure for lists containing *at least* two elements. Table 2 shows the number of candidate solutions generated by Algorithm 3 in all solutions mode (eliminating equivalent configurations in the search tree), for several *entry* declarations. As can be observed in the table, as the length of the list provided as entry grows, the number of candidate solutions computed quickly grows. Furthermore, if the elements of the input list are static, then the number of candidates grows even faster, as can be seen in the last two rows in Table 2, where we provide the first elements of the list. From this small example, it is clear that, in order to be able to cope with realistic Prolog programs, it is mandatory to reduce the search space.

In spite of the phenomenon just described, we have been able to test our approach on a heterogeneous set of benchmarks, and compare these results against those produced by traditional partial evaluation. The set of benchmarks used in our experiments follows:

example_pcpe The motivating example from Listing 5.

nrev The naive reverse algorithm described in Listing 8. It does not contain builtins nor negations. With the specialization query used, this benchmark cannot be fully unfolded.

permute A program which computes all possible permutations of the elements of the input list. An important feature of this program is that its results, when fully unfolded, are much larger than the original program. The specialization query used is a fully-instantiated, closed list.

qsortapp A naive quicksort algorithm implemented using append. It contains arithmetic builtins. With the specialization query used it cannot be fully unfolded.

Benchmark	Compiled size	#versions
example_pcpe	5504	27
permute	4687	70
nrev	4623	117
qsortapp	5390	40
sublists	5638	58
relative	5909	61

Table 3. Size and number of versions of benchmarks

Benchmark	Specialization query
example_pcpe	main(A,B,2,D)
nrev	rev([_,_|L],R)
permute	permute([1,2,3,4,5,6],L)
qsortapp	qsort([_,_|L],R)
sublists	sublists(A,B,C)
relative	relative(john,X)

Table 4. Specialization queries used in our experiment

relative A Lam & Kusalik [7] simple expert system which contains neither builtins nor negations. With the specialization query considered it can be fully unfolded.

sublists A predicate taking a list of pairs of numbers as the first argument, and an arbitrary list as a second argument. Every pair of numbers of the first list denotes the beginning and end of a sublist of the second argument. Sublists are returned in the third argument of the predicate. This benchmark contains builtins. For the specialization query considered it cannot be fully unfolded.

Table 3 shows the size in bytes of the compiled bytecode of each benchmark, as well as the number of candidate solutions being generated by the PCPE approach. In order to keep the number of candidate solutions reasonable, in most cases we have provided specialization queries containing small static data. As a result, in some of the programs the speed-up achieved by partially evaluating the program is not very high using any of the strategies, since little information is known at specialization time. The specialization queries used in our experiments for each benchmark are shown in Table 4.

As we have mentioned previously, the solutions computed by PCPE are evaluated using a fitness function, and the best solution is considered to be the output of the whole algorithm. In our experiments, we have used the following three different fitness functions:

speedup compares programs based on their time-efficiency, measuring run-time speedup w.r.t the original program. For this, the user needs to provide a set of run-time queries with which to time the execution of the program. Thus, such queries should be representative of the real executions of the program. It is computed as

$$\textit{speedup}=T_{spec}/T_{orig},$$

where T_{spec} is the execution time taken by the specialized program to run the given run-time queries, and T_{orig} the time taken by the original program.

reduction compares programs based on their space-efficiency, measuring reduction of size of compiled bytecode w.r.t the original program. It is computed as

$$\textit{reduction}=(S_{orig}-S_{empty})/\,(S_{spec}-S_{empty}),$$

ID	Global control	Local control
c1	hom_emb	one_step
c2	hom_emb	$df\_hom\_emb\_as$
c3	$dynamic$	one_step
c4	$dynamic$	$df\_hom\_emb\_as$

Table 5. Combinations of local and global control rules

where S_{spec} is the size of the compiled bytecode of the specialized program, S_{orig} is the size of the compiled bytecode of the original program, and S_{empty} is the size of the compiled bytecode of an empty program.

balance a combination of the previous two. It is defined as

$$balance = speedup^{\alpha} \times reduction^{\beta}.$$

Exponents α and β can be given different weights depending on whether time- or space-efficiency are considered more important for our purposes. In our runs, we used $\alpha = \beta = 0.5$.

8.1 Benefits of PCPE

We now try to evaluate whether PCPE can actually produce better results than traditional PE. Tables 6, 7, and 8 show how PCPE solutions behave when compared to the solutions obtained by traditional PE, using different fitness functions. In order to be as informative as possible, the best solution obtained by PCPE has been compared against all specialized programs obtained by PE when running *every* combination of the selected global and local control rules.

Each table shows the benchmark being considered, the fitness value obtained by the solution of poly-controlled partial evaluation, and its *composition* (columns *c1* through *c4*, see below), and the fitness value of every solution found by traditional partial evaluation using the different combinations of local and global control rules. Note that all fitness functions are defined in such a way that the original program has fitness 1, and values greater than one indicate improvements over the original program, whereas values less than one indicate that the considered program is worse than the original program (under the corresponding criterion).

For brevity, each of the four combination of global and local control rules is given an identifier, described in Table 5. In the case of the PCPE solution, columns *c1* through *c4* describe the percentage of atoms in the selected best solution whose specialization behaviour can be achieved using the corresponding combination of global and local control. Note that the addition of the values of *c1* through *c4* for a given program will be 100 or more. The latter can occur because different controls can result in exactly the same specialization for certain atoms. A value of a 100 in a given column means that such best solution can be obtained by traditional partial evaluation by using the corresponding combination of global and local control rules. Note that in our implementation, when eliminating equivalent configurations in the search tree, we still keep track of their control rules, in order to produce accurate percentages.

Table 6 shows the results achieved when we use *speedup* as a fitness function. In general, speedup values in most cases should be greater than 1. However, since we are providing very little static information to the partial evaluation algorithms, in the case of *nrev*, *qsortapp*, and *sublist* the speedup achieved w.r.t. the original program is very small, and in many cases (especially in traditional partial evaluation) the specialized program is somewhat slower than the original one. Speedups are however evident in the *relative* and *permute* benchmarks, since they can be fully unfolded. In these two cases, and considering only speedup as the fitness function, the solution obtained by PCPE is a solution that can be obtained by traditional PE. In the case of *permute*, it is achieved by *c2*, i.e., using hom_emb as a global control rule and $df\_hom\_emb\_as$ as a local control rule. This is indicated by the 100 in *c2* column. We can also observe that the speedup of both the PCPE solution and the solution obtained by traditional PE using such control rules are pretty much the same and the difference lies only in timing errors during the experiments, since they correspond to the same program. In the case of *relative*, PCPE obtains two (best) solutions, one containing a 100 in column *c2*, not shown in this table, and one containing a 100 in column *c4*. As can be seen in the table, this speedup value is very similar to the one obtained by traditional PE using such control rules since, again, they correspond to the same code. For this particular fitness function, the rest of benchmarks are the interesting ones, since the solution obtained by PCPE cannot be obtained by PE, as there is no 100 in any column. In all cases the PCPE solution gets a better fitness value than any of the solutions provided by traditional PE, i.e., the obtained specialized program is faster. The PCPE solution in these cases is between 6% and 95% faster than the corresponding best PE solution. Note that this result is interesting in itself: PCPE can achieve better results than any single control rule even in the case where only speedup is taken into account. These experiments were performed on a 1.5 GHz PowerPC G4 processor, with 1Gb of RAM, running on a Darwin 8.5 kernel. Times are given in milliseconds and are computed as the arithmetic mean of five runs.

Table 7 compares PCPE and traditional PE using *reduction* as a fitness function. As can be expected from the selected set of benchmarks, the solutions obtained by PE have a fitness value below 1 in most cases, indicating that the specialized programs are larger than the original one. This usually is due to the fact that these benchmarks contains just a few predicates, and partial evaluation creates many new specialized predicates which then cannot be unfolded very much. When programs can be fully unfolded, as is the case of *relative* and *permute*, the use of $df\_hom\_emb\_as$ as a local control rule usually achieves such full unfolding. In the case of *permute*, the fitness is almost 0 for *c2* since the final fully unfolded program is much larger than the original one. Furthermore, programs produced using *c3*, i.e., $dynamic$ as a global control rule and one_step as a local control rule, are indeed isomorphic to the original program. In this case, fitness values are slightly lower than 1 (0.98) due to predicate renamings, which creates slightly larger predicate names. As can be seen in the table, most of the programs returned by PCPE are not achievable using PE. The only exception is *sublists*, where the best PCPE corresponds to the original program. Thus, it seems that PCPE is able to find a solution that is smaller than any of the solutions found by PE.

Finally, Table 8 shows the results achieved by using *balance* as a fitness function. As can be seen, most of PCPE solutions cannot be obtained via traditional PE, with the exception of the solution for *sublists*, where the optimal coincides with not partially evaluating the program, i.e., the original program. In most cases, PCPE obtains better fitness values than any of the solutions obtained by PE, meaning that PCPE outperforms PE in most cases when both time- and space-efficiency are simultaneously considered.

8.2 Cost of PCPE

We now evaluate the cost of performing PCPE when compared to PE. Though one can argue that, in the case of compile-time specialization, the time required to specialize a program is not very important, the results presented here provide some information of the additional compile-time cost of PCPE when compared to PE. Depending on the situation, the developer may choose to

Benchmark	Best PCPE					Fitness Trad PE			
	c1	c2	c3	c4	Fit	c1	c2	c3	c4
example_pcpe	75	50	50	25	1.96	0.91	0.57	1.01	0.49
permute	0	100	0	0	5.26	0.75	5.14	1.01	2.00
nrev	57	57	0	14	1.20	0.51	0.77	0.99	0.91
qsortapp	50	50	83	67	1.06	0.86	0.87	0.99	0.94
sublists	57	43	71	43	1.08	0.97	0.99	0.98	0.88
relative	0	0	0	100	14.08	0.98	14.05	0.98	14.02

Table 6. PCPE behaviour (speedup).

Benchmark	Best PCPE					Fitness Trad PE			
	c1	c2	c3	c4	Fit	c1	c2	c3	c4
example_pcpe	75	50	50	25	1.22	0.82	1.15	0.98	0.39
permute	25	50	50	50	1.15	0.37	0.00	0.98	0.80
nrev	20	60	60	80	0.98	0.55	0.29	0.98	0.76
qsortapp	33	67	67	83	0.98	0.78	0.43	0.98	0.66
sublists	100	25	100	25	0.98	0.98	0.52	0.98	0.61
relative	20	60	40	60	1.17	0.66	0.89	0.98	0.13

Table 7. PCPE behaviour (reduction).

Benchmark	Best PCPE					Fitness Trad PE			
	c1	c2	c3	c4	Fit	c1	c2	c3	c4
example_pcpe	75	50	50	25	1.54	0.87	0.81	0.99	0.44
permute	40	40	40	40	1.30	0.54	0.14	1.01	1.29
nrev	60	60	0	40	1.12	0.52	0.48	0.98	0.82
qsortapp	50	50	83	67	1.00	0.81	0.61	0.99	0.78
sublists	100	25	100	25	1.01	1.00	0.70	0.99	0.73
relative	20	60	40	80	4.05	0.80	3.55	0.98	1.33

Table 8. PCPE behaviour (balance).

Benchmark	Specialization Time						
	PE			PCPE			PCPE
	Analys	Gen	Total	Analys	Gen	Total	/PE
example_pcpe	26	43	69	111	304	415	6
permute	1153	744	1897	1271	1242	2513	1
nrev	16	27	44	453	1166	1619	37
qsortapp	22	39	61	153	425	578	10
sublists	22	41	63	206	649	854	14
relative	216	166	382	1038	1187	2225	6

Table 9. Cost of PCPE (Specialization Time in msecs.)

spend more resources on specializing the program in return for a (hopefully) better specialized program.

Specialization in both traditional and poly-controlled partial evaluation involves a phase commonly referred to as *analysis* (corresponding to algorithms 1 to 3 described in this paper), and another phase for code generation. Table 9 shows the times (expressed in milliseconds) spent for both approaches during these two phases. The last column shows the ratio between PCPE and PE. As can be seen, the burden introduced by the PCPE approach is usually acceptable. In many cases this overhead is directly related with the amount of candidate solutions generated by the algorithm. Thus, *nrev* is the benchmark where the ratio *PCPE/PE* is bigger, since there are 117 candidate solutions produced by the algorithm. Observe that in those cases that can be fully unfolded, i.e., *permute* and *relative*, specialization time is usually high in both approaches, so the ratio *PCPE/PE* is not very high. In the rest of benchmarks this ratio ranges between 6 and 14.

However, PCPE involves an additional step of evaluation after code generation which is not required in PE. In this step, all candidate solutions are evaluated using the corresponding fitness function, and the solution having the highest fitness value represents the output of the algorithm for the given input queries. Note that there may exist several solutions sharing the same fitness value.

When evaluating all candidates, the fitness function used for such purpose makes an important difference in the time required by such phase. In the case of *reduction*, we need to compile each candidate solution and compare the sizes of the compiled code of all of them. Even though this involves disk accesses, the comparison among solutions can be done pretty quickly, and thus, the increment in time due to evaluation is acceptable, as shown

Benchmark	Evaluation Time(speedup)			
	PE	PCPE		
		Spec	Eval	Total
example_pcpe	69	415	11540	11955
permute	1897	2513	19971	22484
nrev	44	1619	66692	68312
qsortapp	61	578	17159	17737
sublists	63	854	77104	77958
relative	382	2225	17007	19232

Table 10. Total Cost of PCPE (Time in msecs.)

Benchmark	Evaluation Time(reduction)			
	PE	PCPE		
		Spec	Eval	Total
example_pcpe	69	415	1230	1645
permute	1897	2513	3587	6100
nrev	44	1619	5826	7444
qsortapp	61	578	2332	2909
sublists	63	854	4016	4870
relative	382	2225	5173	7399

Table 11. Total Cost of PCPE (Time in msecs.)

in Table 11. In most cases, evaluation takes between two and four times the time spent in the previous two phases.

However, when the fitness function involves measuring time-efficiency, i.e. in *speedup* and *balance*, we need to run each specialized program a number of iterations in order to obtain more accurate measurements, thus increasing the time spent in evaluation (see tables 10 and 12). In our implementation, we have a constant K for estimating the desired amount of time we want to evaluate each candidate. By running the original program for K milliseconds, we estimate the amount of iterations to be run for each of the final candidates. By increasing or decreasing this constant K, we increase or decrease the time spent by the evaluation step of our algorithm. In this way, we have a trade-off between the time spent in this phase, and the accuracy of the obtained solution. For our experiments, we set this constant to 500 milliseconds. As a result, we spend roughly about 500 milliseconds evaluating each candidate solution.

9. Discussion and Related Work

In this work we have introduced a framework for on-line partial evaluation which allows using different global and local control rules for different atoms, obtaining results that are not achievable by traditional partial evaluation. The framework is self-tuning, employing resource-aware *fitness functions* to select the best solutions from a resulting set of candidate solutions.

The poly-controlled partial evaluation framework opens up the door to many interesting possibilities. Experiments have shown that results obtained by poly-controlled partial evaluation are very promising, in the sense that often these results cannot be obtained using traditional partial evaluation, and they obtain better fitness values than their PE counterparts, for fitness functions assessing time- and space-efficiency performance.

As regards related work, the work in [3] is probably the most related one. There, a self-tuning, resource aware *off-line* specialization technique is introduced. The algorithm is based on mutation of annotations for offline partial evaluation. In contrast, our approach performs *on-line* partial evaluation, and thus can take advantage of the great body of work available for *on-line* partial evaluation of logic programs. To the best of our knowledge, there are no similar approaches for *on-line* partial evaluation. Arguably, on-line techniques for partial evaluation of logic programs are very relevant, since on-line techniques have received a lot of attention in the logic programming paradigm.

It remains as future work to develop effective techniques for reducing the search space in PCPE and then apply the resulting algorithm on a larger set of programs, using different search controls. For this purpose, we would like to be able to prune away branches which are not promising, or even guaranteed not to lead to an optimal solution, as in branch and bound algorithms. For this, we need to be able to apply the fitness function not only to configurations which are final, but also to intermediate configurations.

Benchmark	Evaluation Time(balance)			
	PE	PCPE		
		Spec	Eval	Total
example_pcpe	69	415	12887	13302
permute	1897	2513	24408	26920
nrev	44	1619	73538	75157
qsortapp	61	578	19886	20463
sublists	63	854	82898	83752
relative	382	2225	22755	24980

Table 12. Total Cost of PCPE (Time in msecs.)

In addition to this, there are a number of relatively simple ideas which we hope can be used in order to greatly reduce the complexity of PCPE. For example, different procedures have different relative importance on the overall time efficiency of the program. Thus, it can be a good idea to obtain data on the cost of the different procedures by means of profiling, in order to be able to make more informed decisions at partial evaluation time. For example, for procedures with little impact on the run-time of the program, less aggressive control strategies can be used, whereas in calls to predicates with an important cost, more aggressive strategies should be used. Also, the branching factor could be varied for different atoms according to the importance of the atom being handled. If the atom has important weight, we should probably try out more different alternatives than in other less important predicates.

Acknowledgments

The authors would like to thank Michael Leuschel for his comments on a preliminary version of this work. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the *PROMESAS* project.

References

[1] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, LNCS. Springer-Verlag, 2006.

[2] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. *New Generation Computing*, 1(11):47–79, 1992.

[3] Stephen-John Craig and Michael Leuschel. Self-tuning resource aware specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[4] Saumya K. Debray. Resource-Bounded Partial Evaluation. In *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 179–192. ACM Press, 1997.

[5] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[6] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[7] J. Lam and Kusalik A. A comparative analysis of partial deductors for pure prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1991. Revised April 1991.

[8] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[9] Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[10] Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[11] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[12] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[13] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 28(2):89–146, 1996. To Appear, abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993.

[14] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 149–165. Springer-Verlag, 2005.

[15] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.