Poly-Controlled Partial Evaluation in Practice

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

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

Abstract

Poly-Controlled Partial Evaluation (PCPE) is a powerful approach to partial evaluation, which has recently been proposed. PCPE takes into account sets of control strategies instead of a single one. Thus, different control strategies can be assigned to different call patterns, possibly obtaining results that cannot be obtained using a single control strategy. PCPE can be implemented as a *search-based* algorithm, producing *sets* of candidate optimized programs. The quality of each of these programs is assessed through the use of a fitness function, which can be *resource aware*, in the sense that it can take multiple factors into account, such as run-time and code size. Unfortunately, PCPE suffers from an inherent blowup of its search space when implemented as an all-solutions, search-based algorithm. Thus, in order to use it in practice we must be able to prune its search space without losing the (most) interesting solutions. In this work we explore several techniques for pruning the search space of PCPE. Some of these techniques are based on heuristics, while others are based on branch and bound and are guaranteed to obtain an optimal solution. Our experimental results show that, when combined with the proposed pruning techniques, PCPE can cope with realistic programs. Also, that the solutions obtained by PCPE outperform the solutions found by PE under similar conditions.

Keywords Partial Evaluation, Resource-Aware Specialization, Logic Programming

1. Introduction

The aim of partial evaluation (*PE*) [9] is to optimize programs by specializing them w.r.t. part of their input, which is known as the *static data*. The quality of the code generated by partial evaluation greatly depends on the *control strategy* used. Unfortunately, the existence of sophisticated control strategies which behave (almost) optimally for all programs is still far from reality, especially when we take factors such as size of the residual code into account. Poly-controlled partial evaluation [18] (*PCPE*) is a powerful approach to partial evaluation which has recently been proposed in the context of *on-line* partial evaluation of logic programs (*LP*). Among the main advantages of PCPE we can mention:

It can obtain better solutions than traditional PE Preliminary experiments in [18] show that PCPE can produce *hybrid* solutions with better fitness value than any of the solutions achievable by traditional PE, for a number of different resource-aware fitness functions.

It is a resource-aware approach In traditional PE, existing control strategies generally 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 the residual program, and the memory required to run it are most often neglected. Some relevant exceptions are the works in [6],[4]. Also, it is well known that partial evaluation can slow-down programs due to lower level issues, such as clause indexing, cache sizes, etc.

It is not yet another control strategy The topic of control strategies for partial evaluation has received considerable attention. As already mentioned, finding an optimal control strategy is not trivial. However, it is important to note that PCPE is not a control strategy, but a new framework allowing the co-existence and cooperation of any set of control strategies. In fact, PCPE will benefit from any further research on control strategies.

It is more user-friendly Often, partial evaluators provide a good number of parameters which affect the quality of the obtained solution. It can be extremely hard to find the right combination of parameters in order to achieve the desired results (reduction of size of compiled code, reduction of execution time, etc.). PCPE allows the user to simultaneously experiment with different combinations of parameters.

It performs on-line partial evaluation As opposed to other approaches (e.g. [4]), PCPE performs *on-line* partial evaluation, and thus it can take advantage of the great body of work available for *on-line* partial evaluation of logic programs.

In [18], two algorithms for PCPE were introduced. Given a set of control strategies, the first algorithm uses a function called *pick* to decide *a priori* which control strategy (among those in the set) should be used for each call pattern. The second algorithm, which we will refer to as the *all-solutions* PCPE, applies all the control strategies in the set to each call pattern, thus possibly generating several candidate partial evaluations, and decides *a posteriori* which partial evaluation is the best one by empirically comparing the final configurations (candidate partial evaluations) using a *fitness function*. The process can be *resource-aware* by taking into account factors such as the size and time- and memory-efficiency of the partial evaluations. Since choosing a good *pick* function is a very hard task, and in the need of a proof of concept, in [18] a pre-

liminary experimental evaluation was performed based on the all-solutions algorithm whose conclusions are two-fold. First, PCPE can obtain better results than traditional partial evaluation. This is encouraging, since it actually indicates that it is worth pursuing the idea of PCPE. Second, all-solutions PCPE is too costly in practice. Thus, the main question which we address in this work is: is it possible to obtain an algorithm for PCPE which is capable of achieving results comparable to those computed by the all-solutions algorithm while staying within reasonable cost?

2. Background

We assume some basic knowledge on the terminology of logic programming. See for example [15] 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 an atom A, i.e., $pred(A)$, returns the predicate symbol 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.

We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for all $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $\sigma(X) = X$ for any other variable X, where t_i are terms. A *unifier* for a finite set S of simple expressions is a substitution θ if $S\theta$ is a singleton. A unifier θ is called *most general unifier* (mgu) for S, if for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$. Two terms t and t' are *variants*, denoted $t \equiv t'$, if there exist substitutions θ and σ s.t. $t = t'\theta$ and $t' = t\sigma$.

2.1 Basics of On-Line Partial Evaluation in LP

On-line partial evaluation of LP is traditionally presented in terms of SLD semantics. We briefly recall the terminology here. 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 [15].

DEFINITION 2.2 (derivation step). *Let $\mathcal{R}$ be a computation rule, let G be $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, 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:*

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

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

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) [14], given an input program and a set of atoms (goals), 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 is systematically extracted from the SLD trees.

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 σ_i for $i = 1, \ldots, n$ whose associated* resultants *are $\sigma_i(A) \leftarrow G_i$.*

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 [12] for details.

In order to ensure 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, 16] and well-quasi orderings (wqo) [19, 11] are broadly used in the context of on-line partial evaluation techniques (see, e.g., [7, 13, 19]).

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. This usually implies some loss of precision (abstracting newly added atoms by more general ones) 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. All-Solutions PCPE

In Algorithm 1 we recall the all-solutions algorithm of [18]. In this algorithm, 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 and H_i is the set of atoms already handled by the algorithm. Indeed, in H_i we store tuples of the form $\langle A_i, A'_i, Unfold \rangle$ where in addition to each atom A_i we also store the result A'_i of applying global control to A_i (i.e., A'_i is an abstraction of A_i) and the unfolding rule $Unfold$ which has been used to unfold A_i. We store $Unfold$ in order to use exactly such unfolding rule during the code generation phase. Correctness of the algorithm requires that each A'_i is an *abstraction* of A_i, i.e., $A_i = A'_i\theta$ for some substitution θ.

Algorithm 1 employs two auxiliary data structures. One is *Confs*, which contains the *configurations* (or states) which are currently being explored. The other one is *Sols*, which stores the set of solutions currently found by the algorithm. As it is well known, the use of a stack results in a depth-first traversal of the search space.

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, a configuration $\langle S_i, H_i \rangle$ is popped from *Confs* (line 6), and an atom A_i from S_i is selected (line 7). Then, all combinations of global control ($Abstract \in \mathcal{G}$) and local control ($Unfold \in \mathcal{U}$) rules, respectively, are applied (lines 11 and 12). Each application builds an SLD-tree for A'_i, a generalization of A_i as determined by *Abstract*, using the cor-

Algorithm 1 Search-Based 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 $\mathcal{G}$
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: $Candidates = \{\langle Abstract, Unfold\rangle \mid Abstract \in \mathcal{G}, Unfold \in \mathcal{U}\}$
9: **repeat**
10: $Candidates = Candidates - \{\langle Abstract, Unfold\rangle\}$
11: $A'_i = Abstract(H_i, A_i)$
12: $\tau_i = Unfold(P, A'_i)$
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*)

responding 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* (line 14). Those atoms in *leaves*(τ_i) which are not a variant (equal modulo variable renaming) of any of the atoms handled in previous iterations of the algorithm are added to the set of atoms to be considered (S_{i+1}) and pushed on *Confs*. The process terminates when *Confs* is empty. A configuration $\langle S_i, H_i\rangle$ is *final* if $S_i = \emptyset$, otherwise it is called an *intermediate* configuration. The residual program which corresponds to a final configuration $\langle\emptyset, H_i\rangle$ is obtained as $\bigcup_{\langle A,A',Unfold\rangle\in H_n} resultants(A', Unfold)$, where the function *resultants* is parametric w.r.t. the unfolding rule.

4. Blowup of the Search Space

Given a configuration, a set of unfolding rules $\mathcal{U}$, and a set of abstraction functions $\mathcal{G}$ such that $|\mathcal{U}| = i$ and $|\mathcal{G}| = j$, Algorithm 1 can generate $i \times j$ successor configurations in the *worst* case. Thus, and as already mentioned, this represents an inherent exponential blowup in the size of the search space, and it makes the algorithm impractical for dealing with realistic programs.

Of course, several optimizations can be done to the base algorithm shown above, in order to deal with this problem. A first obvious optimization is to eliminate equivalent configurations which are descendants of the same node in the search tree. This optimization, already proposed in [18], significantly reduces the search space. However, even with this optimization, a simple experiment shows the magnitude of this problem.

Let us consider the program in Figure 1, which implements a naive reverse algorithm. In this experiment, let us choose the set of global control rules $\mathcal{G}$={$dynamic$, hom_emb}. The hom_emb global control rule is based on homeomorphic embedding [12, 11] and flags atoms as potentially dangerous (and are thus generalized) when they homeomorphically embed any of the previously visited atoms at the global control level. 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, let us choose the set of local control rules $\mathcal{U}$={one_step, hom_emb_aggr}. The rule one_step is the simplest possible unfolding rule which always performs just one unfolding step for any atom. Finally, hom_emb_aggr is an unfolding rule based on homeomorphic embedding. More details on this unfolding rule can be found in [17] and in Section 7.

```
:- 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).
```

Figure 1. The `nrev` example

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 1. Number of solutions generated by Alg. 1

In `CiaoPP` [8], the description of initial queries —the set of atoms of interest S in Algorithm 1— 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 Figure 1 is used to specialize the naive reverse procedure for lists containing *at least* two elements. Table 1 shows the number of candidate solutions generated by Algorithm 1 (eliminating equivalent configurations in the search tree), for several *entry* declarations. It can be observed in the table that 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 1, 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 programs, it is mandatory to reduce the search space. In the following sections we propose different techniques for doing so.

5. Heuristic Pruning

Although PCPE is definitely appealing, we need to *prune* the search space in order to deal with realistic programs, as already discussed. This pruning can be performed using some heuristics—possibly losing optimal solutions—, or we can try to preserve optimal solutions, for instance, by means of branch and bound techniques. In this section we explore some pruning techniques of the first kind, and present a technique of the second kind in Sec. 6.

Atom	$Pred$	$modes_{\alpha_{SD}}$	$modes_{\alpha_{SDL}}$
$p(X,a)$	$p/2$	$p(D,S)$	$p(D,S)$
$p(a,q(X,b),X)$	$p/3$	$p(S,D,D)$	$p(S,D,D)$
$p(a,[],[a,X])$	$p/3$	$p(S,S,D)$	$p(S,S,L)$
$p(a,q(b,X,r(Y,[])))$	$p/2$	$p(S,D)$	$p(S,D)$

Table 2. Abstraction of calls using different domains

5.1 Predicate-Consistency Heuristics

Given the selected atom A_i in the current configuration, rather than trying all possible control strategies, herein we propose to consider only those control strategies which are *consistent* with the choices previously taken in previous configurations. The first notion of consistency we are going to consider is that we must use the same control strategy for all atoms which correspond to the same predicate. We will refer to configurations which satisfy this restriction as *predicate-consistent*. This restriction will often significantly reduce the branching factor since handling of an atom A_i will become deterministic as soon as we have previously considered an atom for the same predicate in any configuration which is an ancestor of the current one in the search space, i.e., it is compulsory to use for A_i exactly the same control strategy used before. Though this simplification may look too restrictive at first sight, the intuition behind it is that though it is often a good idea to allow using different control strategies for different predicates (i.e., allowing hybrid solutions) it is also often the case that it is possible to obtain optimal solutions where we consistently use the same control strategy for all atoms of the same predicate. In other words, we believe that it is often the case that, in the context of a given program, there exists a control strategy which behaves well for all atoms which correspond to the same predicate.

We thus propose to modify Algorithm 1 so that only consistent configurations are further processed. For this we need to store, together with every atom in every configuration, the global control rule used to generalize such an atom. We now provide a formal definition of consistent configurations:

DEFINITION 5.1 (predicate-consistent configuration). *Given a configuration* Conf $= \langle S, H\rangle$, *we say that* Conf *is* predicate-consistent *iff* $\forall\ \langle A_1, A'_1, G_1, U_1\rangle,\ \langle A_2, A'_2, G_2, U_2\rangle \in H,\ pred(A_1) = pred(A_2) \Rightarrow (G_1 = G_2 \wedge U_1 = U_2)$.

Note that this definition can be applied to both intermediate and final configurations. Thus, if a given intermediate configuration *Conf* is inconsistent, it will be pruned, i.e., it will not be pushed on *Confs*. By doing this we are pruning not only *Conf*, but also all the successor configurations that would have been generated from *Conf*. This means that early prunings will achieve significant reductions of the search space.

5.2 Mode-Consistency Heuristics

A possible improvement over predicate-consistency, in order to increase accuracy, is to define consistency at the level of *modes* for a predicate. This means that two calls to a predicate with similar *modes* (instantiation level in their arguments) have to use the same control strategy, but not if they have different modes.

In order to check whether two atoms A and A' with $pred(A) = pred(A')$ have the same modes, we apply to them a function $modes$ that abstracts their arguments one by one w.r.t. a given abstract domain [3], i.e., given an abstraction function α, we define $modes_\alpha(p(t_1,\ldots,t_n))$ as $p(\alpha(t_1),\ldots,\alpha(t_n))$. Then, we say that A and A' have the same modes under α iff $modes_\alpha(A) = modes_\alpha(A')$. In a way, this is similar to the *binding types* used in the binding-time analysis (*BTA*) of *offline* partial evaluation [9]. In *BTA*, each argument of a predicate is given a *binding type* that provides some information about the instantiation state of an argument at specialization time.

The basic binding types in *BTA* are *static*, indicating that the argument is completely known at specialization time, and *dynamic*, indicating that the argument is possibly unknown at specialization time. Thus, we define the α_{SD} *abstraction* as follows:

DEFINITION 5.2 (α_{SD} abstraction). *Given a term t, the α_{SD} abstraction over t is defined as follows:*

$$\alpha_{SD}(t) = \begin{cases} S & \text{if } vars(t) = \emptyset \\ D & \text{if } vars(t) \neq \emptyset \end{cases}$$

A more precise binding type can be defined by means of regular type declarations, and combined with basic binding types. For example, one can define types such as list skeletons. We can define the α_{SDL} *abstraction* as follows:

DEFINITION 5.3 (α_{SDL} abstraction). *Given a term t, the α_{SDL} abstraction over t is defined as follows:*

$$\alpha_{SDL}(t) = \begin{cases} S & \text{if } vars(t) = \emptyset \\ L & \text{if } t \text{ is bound to a list skeleton} \wedge vars(t) \neq \emptyset \\ D & \text{otherwise} \end{cases}$$

In Table 2 we can observe some examples of atoms, and how they are abstracted using the definitions introduced above. We now provide a formal definition of consistent configurations w.r.t. the mode-consistent heuristics.

DEFINITION 5.4 (mode-consistent configuration). *Let α be an abstraction function. Then, given a configuration* Conf $= \langle S, H\rangle$, *we say that* Conf *is* mode-consistent *iff* $\forall\langle A_1, A'_1, G_1, U_1\rangle$, $\forall\langle A_2, A'_2, G_2, U_2\rangle \in H$, $pred(A_1) = pred(A_2) \wedge modes_\alpha(A_1) = modes_\alpha(A_2) \Rightarrow (G_1 = G_2 \wedge U_1 = U_2)$.

6. Branch and Bound Pruning

The main advantages of the heuristic-based pruning techniques shown in Sec. 5 are twofold: they are simple to implement, and they drastically reduce the search space of PCPE, thus reducing the specialization time and memory requirements and making the PCPE algorithm able to cope with many more programs than the original all solutions algorithm. On the other hand, it is well known that heuristics can perform well for some cases and not so well for others. In the case of the heuristics just presented, this could mean that, in particular situations, optimal solutions may be lost. In this section we explore a different pruning technique, which guarantees the preservation of an optimal solution. Ideally, the new pruning technique should be applicable either in isolation or combined with the heuristic pruning already presented. Our new pruning is based on *branch and bound* [10] (*BnB*). Here, the basic idea is to store the fitness value of the best solution found so far and prune away those configurations which are guaranteed not to improve the (temporary) optimal solution. In order to implement a BnB-based algorithm we need:

- to devise a mechanism for computing an *upper bound* of the fitness value of any intermediate configuration. This involves the use of two components for estimating the fitness of intermediate configurations. One which is *actual* and another which *maximizes* the possible value in case of completing the configuration.
- to decide how often we should evaluate candidates and try to prune. Clearly, if we evaluate very often we will be able to prune more. However, evaluating introduces a non negligible cost. Thus, evaluating too often can make branch and bound even slower than the all-solutions algorithm. As a result, in

our implementation we do not evaluate every configuration. Instead, the implementation is parametric w.r.t. a *depth-level*. Those configurations which appear at a depth which is a multiple of the depth-level are evaluated. All others are not.

6.1 Fitness Functions

As already mentioned, we need to obtain upper bounds on the fitness value of intermediate configurations. The way we do it is tightly coupled to the fitness function used. In all cases we assume that the fitness function returns values in the interval $[0, \infty)$ and that larger values are preferable to smaller values. In our framework, we have implemented the following *resource-aware* fitness functions, in the spirit of those in [4]:

speedup compares programs based on their time-efficiency, measuring the run-time of the candidate program w.r.t. the original one. In order to use this fitness function, the user needs to provide a set of run-time queries with which to time the execution of the program. Such queries should be representative of the real executions of the program[1]. This fitness function is computed as

$$speedup = T_{orig} / T_{candid},$$

where T_{candid} is the execution time taken by the candidate 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 the size of compiled bytecode of the candidate program w.r.t. the original one. It is computed as

$$reduction = (S_{orig} - S_{empty}) / (S_{candid} - S_{empty}),$$

where S_{candid} is the size of the compiled bytecode of the candidate 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[2].

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

$$balance = speedup \times reduction.$$

and thus it takes into account both the size and the efficiency of the candidate programs. As it stands, it gives equal importance to both factors. It is easy to obtain variations of this formula which assign different weights to them, as best suited to each situation.

6.2 Estimating Fitness Values

In the case of *reduction*—or for any fitness function that takes the size of the resulting program as one factor—, for any intermediate configuration *conf*$=\langle S_i, H_i\rangle$ we can take as current value the size of the resultants in the atoms in H_i, and take 0 as estimate for the size of the atoms in S_i. In other words, the actual component of the fitness value can be obtained by simply compiling the current intermediate configuration, i.e., we obtain an incomplete program out of the atoms contained in the set of already handled atoms H_i.

The fitness value thus computed is guaranteed to be an *upper bound* on the fitness value of any possible final configuration reachable from the current state. This is because the size for the code of the atoms in S_i will definitely be greater than 0, which is the value

[1] Though the issue of finding representative run-time queries is an interesting research topic in its own right, it is out of the scope of this paper to automate such process.

[2] In the current framework, we have implemented also a similar function that takes into account the memory taken by the residual program, i.e., including bytecode and data.

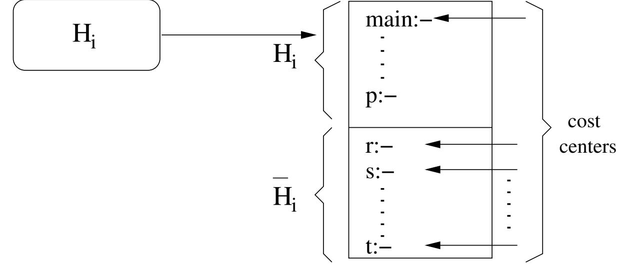

Figure 2. Profiling an intermediate configuration

we have taken. Thus, if the size of the compiled incomplete program resulting from the current configuration $\langle S_i, H_i\rangle$ is already *larger* than the bytecode of the current best solution, then we can safely prune away the current node and all of its descendants, since it will be impossible to obtain a program containing the already visited atoms which is smaller than the best solution.

This approximation makes use of only the actual part of the fitness value of the current intermediate configuration. A more accurate approximation can make use of the atoms in S_i to create extra facts with different variables in the output program, and in this way better approximate the real size of the residual program.

In the case of *speedup*, or for any fitness function that takes execution time of the residual program into consideration, we can make use of a profiler (see e.g. [5]) which allows defining *cost centers*. The profiler splits the total execution time among the different predicates in the program. When a cost center is defined, it accumulates the execution time of all computations started from such predicate.

The main challenge we must cope with when implementing this approach is the fact that the profiler takes complete programs as input. Instead, we want to use it with intermediate configurations which correspond to incomplete programs. This poses a problem, since the partial evaluation algorithm is not devised in such a way that a consistent program can be obtained for any set of atoms. If such set of atoms is not *closed* [14], then the union of the partial deductions for the atoms in the set does not correspond to a self-contained program.

The solution we propose in this case is to use the original predicate definition for all atoms in S_i, since we do not have a specialized version for them yet. This allows running the incomplete program. However, the run-time thus obtained will in general not be a lower bound of the execution time of the descendant final configurations. In order to solve this second problem we propose to use cost centers for all predicates in the original program plus a cost center for the main entry (exported predicate) of the program (see Figure 2). Note that such exported atom must belong to H_i since it is the first atom handled by Algorithm 1. This way, when running the residual program, the time reported by the profiler for predicates in H_i will not include the time actually required for atoms which are not in H_i (represented by $\overline{H_i}$ in Figure 2).

Note that the execution time reported by the profiler for the main entry is, modulo timing noise, a lower bound of the execution time of any candidate solution reachable from the current configuration since we are using 0 as an estimate for the execution of all atoms in S_i. Thus, if the time for the main entry is higher than the best time already found, again there is no point in further expanding the current configuration, and we can safely prune the corresponding branch.

EXAMPLE 6.1. *Figure 2 depicts an intermediate state where we have processed a set of atoms H_i, including the initial predicate* `main/0` *and other atoms such as* `p/0`*, but the set S_i of atoms*

Benchmark		PCPE						Best PE
		all	Heuristic		BnB	BnB+Heur		
Name	Size		Preds	Modes		Preds	Modes	
datetime	17689	1.93	1.94	1.89	1.89	1.89	1.90	1.31
nrev	4623	-	3.65	3.71	3.65	3.64	3.66	1.98
qsortapp	5390	-	2.30	2.59	3.73	2.26	2.58	1.77
contains	5549	4.16	4.16	4.21	4.17	4.17	4.16	3.15
grammar	11381	8.41	8.42	8.43	8.42	8.44	8.44	4.71
groundunify_simple	10368	-	3.02	-	2.96	2.95	2.97	2.95
liftsolve_app	7111	1.17	1.16	1.19	1.18	1.17	1.17	1.17
match	4781	1.59	1.58	1.58	1.61	1.61	1.60	1.09
transpose	5005	2.46	2.46	2.45	2.44	2.44	2.46	2.46
Geom Mean		**2.60**	**2.60**	**2.60**	**2.60**	**2.59**	**2.60**	**1.98**

Table 3. Fitness for Several PCPE Algorithms and Traditional PE

to be processed is not empty, and in our case contains `r/0` *and* `s/0`*, which in turn may call other atoms not yet processed such as* `t/0`*. In such case, we take the original definitions of the predicates* pred(`r/0`)*,* pred(`s/0`)*, and* pred(`t/0`) *and define a cost center for each of these predicates. Thus, when profiling the execution of* `main/0`*, the time reported will not include the time spent in the execution of any of the predicates for which a cost center has been defined. This guarantees that the time obtained is a lower bound of the time that the residual program would take.*

Finally, in the case of balance, we can simply estimate the upper bounds of *speedup* and *reduction* as above, and apply the balance function to obtain an approximated fitness value which, as we need, is guaranteed to be an upper bound of the fitness value reachable from the current state.

Note that BnB can be combined with the predicate- and mode-consistency heuristics presented in Section 5. In this case, we will obtain a solution with a fitness value which is guaranteed to be optimal among those final configurations which are consistent w.r.t. the abstraction used.

7. Experimental Results

With the purpose of assessing the effectiveness of the different improvements proposed in this paper over the all-solutions algorithm for PCPE, we have performed a series of experiments using several benchmarks.[3] The size in bytes of the compiled version of each benchmark, using compilation to WAM bytecode, is indicated in column `Size` of Table 3.

In all our experiments, we have used the following two global control strategies: $\mathcal{G}$=$\{hom_emb, dynamic\}$ and two local control strategies $\mathcal{U}$=$\{hom_emb_aggr,\ hom_emb_cons\}$. The behaviour of the global control strategies hom_emb and $dynamic$ has already been explained in Section 4. Also, hom_emb_aggr and hom_emb_cons are unfolding rules which are both based on homeomorphic embedding for flagging possible non-termination (see [17] for more details). In both cases, non-leftmost unfolding is performed only when it is guaranteed to be *safe* (see [1]). However, the first one is more aggressive, whereas the second one is more conservative. More precisely, they differ in two ways: 1) the first one uses the *binding-insensitive* computation rule, whereas the second uses the *safe* computation rule of `CiaoPP`. The former is more aggressive, but it is only guaranteed to be correct in programs which are well typed. The second is correct for all programs. 2) more importantly, the second one only performs non-leftmost unfolding steps which are *determinate*, i.e., the selected atom must unify at most with one program rule. Note that this is important since it is well known that performing non-determinate non-leftmost unfolding can sometimes speedup the program but it often slowdowns the program. Thus, conservative rules tend not to perform this kind of unfolding steps.

[3] The source code of these benchmarks (including the partial evaluation and runtime queries) can be found at `http://www.clip.dia.fi.upm.es/Systems/pcpe`.

7.1 Benefits of PCPE

Probably, the first question that we need to answer about PCPE is whether it is actually able to improve the results of traditional PE. With this aim, Table 3 compares the quality of the residual programs obtained using the different approaches to PCPE presented in this paper. As a measure of the quality of programs we have used the *balance* fitness function. We believe that this fitness function is particularly interesting since it is resource-aware: both time-efficiency and size of the residual programs are taken into account. Thus, it is a useful fitness function in the context of pervasive and embedded systems, where it is important that the program does not exceed the storage capabilities of the device. All our experiments have been run using `Ciao` 1.13 over a 2.6 Linux kernel, on a Pentium IV 3.4GHz processor, with 512Mb of RAM.

Since the *balance* fitness function takes into account run-times of the residual programs, and there is always some noise associated to time measurement, times are taken as the arithmetic mean of ten consecutive runs. Nevertheless, when using the balance function, is difficult to determine whether the selected solution is of optimal fitness or not, since fitness cannot be computed with full accuracy, in contrast to the *reduction* fitness function, which is exact.

In order to make a fair comparison of PCPE w.r.t. traditional PE, we have run PE over all benchmarks with all four combinations of the control strategies discussed above, looking for the combination which achieves the best overall fitness. For these particular benchmarks the best combination was ($hom_emb + hom_emb_cons$). Thus, we compare the different algorithms for PCPE against PE using this particular control strategy.

In all our tables, the column `all` represents the all-solutions mode, where no pruning is performed. The two following columns, under the `Heuristic` label, show the results of using heuristic pruning. The column `Preds` presents the case where the predicate-consistency heuristics is used, and column `Modes` shows the results when the $modes_{\alpha_{SD}}$ heuristics is used. The following column, labeled `BnB` shows the results for the branch and bound algorithm. The next two columns, labeled `BnB+Heur` show the cases where BnB is combined with each of the two heuristics considered. In all our experiments we have set the parameter *depth-level* to 3. This means that those configurations which appear at depths 3, 6, 9... are evaluated and pruned if possible, but not those which are

Benchmark	PCPE						Best PE
	all	Heuristic		BnB	BnB+Heur		
		Preds	Modes		Preds	Modes	
States							
datetime	21.2	3.6	5.0	16.1	3.0	4.4	21
nrev	-	2.2	10.1	7.7	2.1	4.9	14
qsortapp	-	3.3	10.6	8.4	2.5	4.9	32
contains	8.3	3.4	8.3	4.4	2.4	4.1	11
grammar	11.8	9.7	11.8	5.0	4.3	4.8	6
groundunify_simple	-	420.8	-	4.0	4.0	4.0	4
liftsolve_app	223.7	31.7	178.3	4.3	2.7	4.3	3
match	9.2	3.8	3.8	4.0	2.6	2.6	5
transpose	4.5	3.5	4.5	2.5	2.5	2.5	2
Geom Mean	**16.57**	**6.07**	**10.69**	**4.98**	**2.85**	**3.66**	**5.89**
Evaluations							
datetime	105	6	12	149	18	31	0
nrev	-	4	24	52	7	25	0
qsortapp	-	6	18	111	18	50	0
contains	12	4	12	16	7	15	0
grammar	16	12	16	12	10	12	0
groundunify_simple	-	85	-	6	6	6	0
liftsolve_app	49	5	33	5	3	5	0
match	11	4	4	7	4	4	0
transpose	3	2	3	2	2	2	0
Geom Mean	**17.87**	**4.75**	**9.85**	**11.23**	**5.51**	**7.73**	**0.00**

Table 4. Normalized Size of Search Space and Number of Evaluations Performed

not multiples of 3. We have empirically determined that 3 is an appropriate value for this parameter.

In order to have a global view of the values in the different columns, in all tables we have included a row `Geom Mean` with the *geometric mean* of (part of) the values in the corresponding column. Since some columns do not have values for some benchmarks (because the corresponding algorithm has run out of memory), and in order to make comparisons meaningful, we compute the geometric mean only over those benchmarks which all algorithms can handle without problems, i.e., `nrev`, `qsortapp`, and `groundunify_simple` are not considered in this calculation. It should be noted that the specialization queries used in our runs contain a good amount of static information, thus causing PCPE to run out of memory in the original all-solutions mode (we indicate this fact by a $-$ in the table) for the programs mentioned above. Fortunately, by using the heuristic-based pruning techniques presented in this work, PCPE has finished in most cases, and when using the branch-and-bound pruning PCPE has always finished.

Several important conclusions can be drawn from Table 3. We can see that PCPE outperforms PE in most cases, achieving a mean fitness value of about 2.60 (vs 1.98 achieved by PE). The ratio PCPE/PE is around 1.31, which indicates that PCPE obtains residual programs which are about 31% better under the balance fitness function than those achieved by traditional PE for this particular set of benchmarks. However, there are some cases where PCPE does not improve the results of PE because the optimal program is *pure* rather than hybrid. This happens, for example, in `transpose` and `liftsolve_app`.

Also, it can be seen that heuristic pruning provides results whose fitness values are identical in most cases to the fitness obtained without pruning. An exception is `qsortapp`, where the fitness obtained by BnB (3.73) is considerably larger than that obtained by BnB+Preds (2.26). In turn, this fitness is lower than that obtained by BnB+Modes (2.58), which is the only case where the additional accuracy inherent to the mode-consistency heuristic seems to be of interest.

Another interesting observation is that BnB allows handling all considered benchmarks. This is because BnB reduces the search space significantly, thus avoiding *out of memory* problems.

7.2 Search Space of PCPE

Once we have established that PCPE actually obtains better results than PE and that the fitness value basically remains the same even when the proposed pruning techniques are applied, the next question we need to address is whether the pruning techniques proposed can actually reduce the search space to levels which are comparable to those of traditional PE.

The upper half of Table 4 shows the ratios of the number of configurations generated by the different PCPE algorithms versus those generated by traditional PE. Column *Best PE* shows the actual number of configurations generated by traditional PE. As can be seen, predicate-consistency achieves a significant reduction of the search space. It requires to explore, on average (only) 6.07 times the amount of states generated by PE, rather than 16.57 in the case of the all-solutions algorithm. As regards the mode-consistency heuristics, it can be seen that it generates around twice as many states as predicate-consistency, and it even runs out of memory for `groundunify_simple`. This, together with the fact that the fitness of the best solutions found using this heuristic are quite close to those obtained using the predicate-consistency heuristics (with the only exception of `qsortapp`), allows concluding that the latter heuristics is preferable in practice. As regards BnB, we can see that it allows a further reduction of the search space. On average, we (only) need to explore 4.98 times as many configurations as in traditional PE. This is an important advantage of this technique, since it copes directly with the main problem of PCPE.

Another important conclusion which can be drawn from the upper half of Table 4 is that, fortunately, the reduction of the search space achievable by the proposed heuristics gets along very well

Benchmark	PCPE						Best PE
	all	Heuristic		BnB	BnB+Heur		
		Preds	Modes		Preds	Modes	
Analysis							
datetime	1466	472	511	3084	1306	1407	371
nrev	-	166	329	1022	674	847	150
qsortapp	-	227	583	2289	799	1195	116
contains	532	326	417	1092	880	966	267
grammar	626	458	553	1354	1288	1280	290
groundunify_simple	-	4275	-	819	789	778	121
liftsolve_app	5924	418	1802	793	726	750	145
match	207	107	121	670	614	625	99
transpose	278	282	279	822	808	810	259
Geom Mean	**741.71**	**310.10**	**439.14**	**1121.73**	**900.54**	**933.28**	**218.41**
Code generation							
datetime	11972	654	1273	0	0	0	143
nrev	-	115	1025	0	0	0	58
qsortapp	-	672	2503	0	0	0	180
contains	710	209	720	0	0	0	82
grammar	1491	1094	1508	0	0	0	59
groundunify_simple	-	14456	-	0	0	0	13
liftsolve_app	4152	330	2665	0	0	0	30
match	266	93	108	0	0	0	32
transpose	55	34	53	0	0	0	15
Geom Mean	**957.39**	**232.04**	**525.64**	**0.00**	**0.00**	**0.00**	**42.43**
Evaluation							
datetime	39704	2906	5016	100928	11298	18032	0
nrev	-	1419	5135	17351	2148	8542	0
qsortapp	-	2611	6404	71189	9543	34624	0
contains	3218	1516	3200	5822	2117	5543	0
grammar	4753	3670	4620	6117	5423	6092	0
groundunify_simple	-	45329	-	2585	2691	2694	0
liftsolve_app	20305	2445	13427	2249	1239	2246	0
match	4202	1802	1777	2185	1239	1260	0
transpose	1296	1035	1322	504	507	498	0
Geom Mean	**6375.71**	**2047.86**	**3643.59**	**4552.73**	**2157.94**	**3082.83**	**0.00**

Table 5. Partial Execution Times of PCPE Algorithms (fitness = balance)

Benchmark	PCPE						Best PE
	all	Heuristic		BnB	BnB+Heur		
		Preds	Modes		Preds	Modes	
datetime	53142	4032	6800	104012	12604	19439	514
nrev	-	1700	6489	18373	2822	9389	208
qsortapp	-	3510	9491	73479	10342	35819	296
contains	4460	2051	4337	6914	2997	6509	349
grammar	6870	5222	6681	7471	6711	7372	349
groundunify_simple	-	64060	-	3404	3480	3472	134
liftsolve_app	30382	3193	17894	3042	1965	2996	175
match	4675	2002	2006	2855	1853	1885	131
transpose	1629	1351	1654	1326	1315	1308	274
Geom Mean	**8498.84**	**2683.08**	**4764.77**	**6289.36**	**3266.22**	**4362.44**	**267.37**

Table 6. Total Execution Times of PCPE Algorithms (fitness = balance)

with the reduction achieved by BnB. In fact, the best result in terms of search space is the combination of *BnB*+predicate-consistency, where on average it is only needed to explore less than *thrice* (2.85) as many states as PE.

Another important difference between PCPE and PE is that since the former allows computing several candidate residual programs, we need to *evaluate* them in order to choose which is the best of them. The lower half of Table 4 shows the number of configurations which need to be evaluated, i.e., the number of times we need to compute or approximate fitness values. In the case of the three leftmost algorithms, only final configurations are evaluated, as a *post-step* of the PCPE algorithm. Thus, for this part of the ta-

ble, the number of evaluations actually coincides with the number of solutions generated by PCPE. On the other hand, for the next three columns, in which BnB is applied, only one solution is obtained. However, the reported evaluations are actually performed *during* the analysis phase of the PCPE algorithm, sometimes on final configurations, sometimes on incomplete configurations with the aim of pruning them.

Again, the results shown in the table indicate that both heuristic pruning and BnB require an acceptable number of evaluations. One important point is that the number of evaluations performed when using BnB can sometimes be larger than when it is not. This is because in the case of BnB, also intermediate configurations can be evaluated. If little pruning is achieved, then more evaluations are needed. However, this will often be compensated by pruning, which reduces the number of configurations to be explored and of final solutions to be evaluated.

All in all, we believe that the results shown in Table 4 are indeed quite promising and show that the search space and the number of evaluations required by PCPE are manageable, when the proposed pruning techniques are applied.

7.3 Time Cost of PCPE

Tables 5 and 6 show the time required by PCPE. All figures are in milliseconds. In order to have a clear understanding on how the total time required by PCPE, shown in Table 6, is used, we have separated in Table 5 such total time into different categories: i.e., analysis, code generation, and evaluation of the configurations. By *analysis* we mean the time required to explore the search space, *code generation* refers to the time required to generate code for the final configurations. Finally, *evaluation* is the time needed to apply the fitness function to the required configurations. Whereas in the three leftmost algorithms these three phases take place sequentially in the order mentioned, in BnB-based PCPE these stages are interleaved. However, we discriminate the time spent in evaluating both intermediate and final configurations during analysis, and show this information in the evaluation part of the table. Note that code generation is also included in the evaluation part since it is required to generate code prior to evaluating a configuration. As a result, the generation phase has time zero for all BnB-based algorithms, since all candidates are generated during the analysis phase.

When considering analysis and generation times, especially when both predicate-consistency pruning and BnB are used, PCPE performs around 4 times slower than PE. However, when compared to PE, PCPE requires an additional phase to select the best candidate. The time required for evaluating candidates can vary a great deal from some fitness functions to others. In particular, if the fitness function measures time-efficiency, then in order to obtain accurate results several runs have to be performed. This is the case with the fitness function used in our experiments. When evaluation time is taken into account, PCPE is an order or magnitude slower than PE. This however, can be improved in several ways. Also, in many situations it can be argued that the cost of partial evaluation is not crucial since it takes place at compile-time. In fact, we believe that there can be a good number of cases where it is actually worthwhile to have the possibility of using a more powerful, though more expensive (but also completely automatic) framework for optimizing *relevant* code. This includes code which is going to be executed very often or code which has to be executed on devices with limited computing capabilities, as is the case in embedded systems.

8. Conclusions

In this work we have presented a series of improvements over the all-solutions algorithm for poly-controlled partial evaluation. Some of the improvements are based on heuristics, with the advantage of being easy to implement and the disadvantage of not guaranteeing that an optimal solution is found. However, our experimental results show that such heuristics work pretty well in practice since they reduce the search space significantly and the resulting fitness value is almost as high as in the all-solutions algorithm. Besides, we have presented a branch and bound-based pruning technique, which is more complex than the previous ones, but with the advantages of achieving solutions which are guaranteed to be optimal, and consuming much less memory than the rest of approaches, which is PCPE's main bottleneck. The experimental results show that this technique is able to reduce the search space significantly and by using an appropriate depth-level the time required for the combined algorithm is reasonable. Interestingly, the branch and bound-based pruning can be further enhanced by combining it with the heuristics presented in this work.

Experimental results show that in many cases PCPE outperforms traditional PE for the *balance* fitness function, i.e., PCPE is able to generate smaller and faster programs, which can be of interest in the case of pervasive computing, for instance. In summary, we believe that by using the optimized algorithms proposed in this work, PCPE is a relevant technique in practice, since it allows obtaining important improvements at a reasonable cost.

Acknowledgments

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 by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*. Springer LNCS 3901, April 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] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.

[4] S.J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.

[5] S. K. Debray. Profiling prolog programs. *Software Practice and Experience*, 18(9):821–839, 1983.

[6] Saumya K. Debray. Resource-Bounded Partial Evaluation. In *Proc. of PEPM'97*, pages 179–192. ACM Press, 1997.

[7] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.

[8] M. Hermenegildo, G. Puebla, F. Bueno, and P. 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.

[9] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[10] A.H. Land and A.G. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrica*, 28:497–520, 1960.

[11] M. Leuschel. On the power of homeomorphic embedding for online termination. In *Proc. of SAS'98*, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[12] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *TPLP*, 2(4 & 5):461–515,

July & September 2002.

[13] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM TOPLAS*, 20(1):208–258, January 1998.

[14] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *JLP*, 11:217–242, 1991.

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

[16] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *JLP*, 28(2):89–146, 1996.

[17] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.

[18] G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of PPDP'06*, pages 261–271. ACM Press, 2006.

[19] 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.