Oracle-Based Partial Evaluation

Claudio Ochoa[1]

Software Solutions Group
Intel
Argentina

Germán Puebla[2]

School of Computer Science
Technical University of Madrid
Boadilla del Monte, Spain

Abstract

We present *Oracle-Based Partial Evaluation (OBPE)*, a novel approach to on-line Partial Evaluation (PE) which decides the control strategy to use for each call pattern by using an *oracle* function which compares the results of specializing such call pattern w.r.t. a set of strategies. Our proposal is motivated by *Poly-Controlled Partial Evaluation (PCPE)*, which allows using different control strategies for different call patterns. Given a set $\mathcal{CS}$ of control strategies, the best PCPE specialized programs outperform the specialized programs obtained by traditional PE for any of the control strategies in $\mathcal{CS}$, especially when *resource-aware* specialization is performed. Unfortunately, computing all PCPE specialized programs and then choosing *a posteriori* the best one is too costly in practice. In contrast, in OBPE a single specialized program is computed. We have developed an *empirical oracle* whose parameters are approximated from a set of training data, by using constraint logic programming. Our experimental results show that the additional cost of OBPE when compared with traditional PE is a constant factor and that, at least in our experiments, OBPE obtains significantly better specializations. We argue that our proposal is relevant in practice and introduces clear improvements over standard PE. Our work is developed in the context of logic programs, though the ideas are in principle of interest to the PE of any programming language.

Keywords: Program Transformation, Partial Evaluation, Resource-Aware Specialization, Logic Programming

1 Introduction

Partial Evaluation (*PE*) [7] optimizes programs by specializing them w.r.t. part of their input, which is known as the *static data.* Those computations which only depend on the static data are performed at specialization-time, whereas those which depend on dynamic data remain in the *specialized program.* The idea, of course, is that the running-time of the specialized program should be smaller than that of the original program, since in the former fewer execution steps are performed at run-time.

[1] Email: claudio.j.ochoa@intel.com

[2] Email: german@fi.upm.es

Partial evaluation algorithms are typically parametric w.r.t. a *control strategy*, which guides the PE process. Different control strategies produce different specialized programs with varying degrees of quality. The study of control strategies for PE has received considerable attention (see e.g. [9] and its references for an overview of control strategies in PE of logic programs) and sophisticated strategies exist which allow obtaining powerful specializations. However, it is well known (see e.g. [14,3]) that a given control strategy can be very appropriate for some cases but produce low quality results in others. This is especially the case if we take factors such as the size of the specialized program into account, since most of the work on partial evaluation has concentrated on improving time-efficiency, largely ignoring such other factors. Some relevant exceptions are the works of Debray [4] and Craig-Leuschel [3]. The latter is able to perform resource-aware specialization based on off-line partial evaluation techniques.

Poly-Controlled Partial Evaluation [13] (*PCPE*) is a powerful approach to *online* partial evaluation of logic programs. Rather than using a fixed control strategy (as done in traditional partial evaluation algorithms), PCPE allows considering a *set* $\mathcal{CS}$ of control strategies. For each call pattern, PCPE can choose any of the control strategies in $\mathcal{CS}$. This allows using *different* control (or specialization) strategies for different call patterns. Thus, PCPE can produce residual programs that are *not directly achievable* by traditional partial evaluation using any of the considered control strategies in isolation. This often results in hybrid solutions with better fitness value than any of the solutions achievable by traditional PE, for a number of different resource-aware fitness functions [11].

As a result of the use of sets of control strategies, given a program, a description of the initial call patterns, and a set of control strategies, PCPE can obtain multiple specialized programs. The execution of PCPE can be depicted as a *tree* whose leaves correspond to the different specialized programs.

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

exp(B,E,R):-
  exp_ac(E,B,1,R).

exp_ac(E,_,R,R):-
  E =< 0.
exp_ac(E,B,T,R):-
  E > 0,
  E1 is E - 1,
  N is T * B,
  exp_ac(E1,B,N,R).

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

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

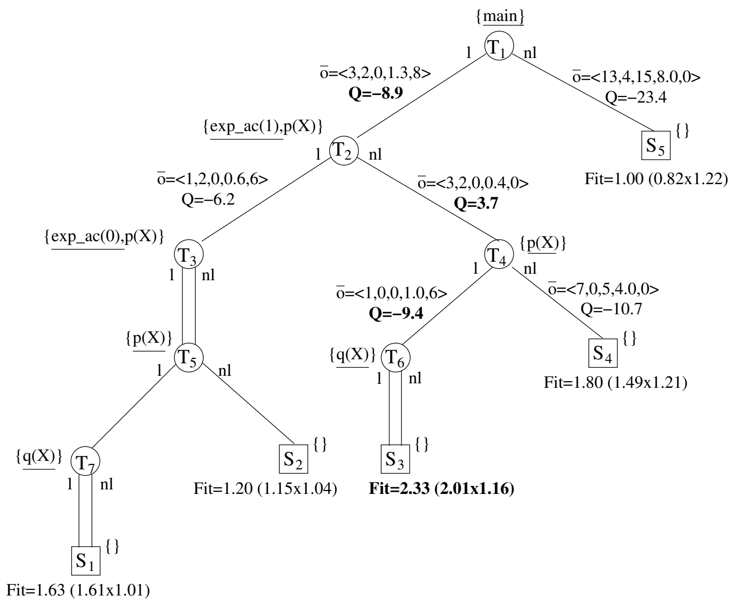

Fig. 1. A motivating example and its PCPE tree

1.1 A Motivating Example

Let us consider the program P listed in the left-hand side of Fig. 1. Consider also that we are interested in specializing this program w.r.t. the single call pattern $main(A, B, C)$, and that the set of control strategies $\mathcal{CS} = \{\langle G_1, U_1\rangle, \langle G_1, U_1^l\rangle\}$, where G_1 and U_1 are an abstraction function and an unfolding function, respectively, which are both based on homeomorphic embedding [9,8]. Both G_1 and U_1 are described in more detail in Section 6. Finally, U_1^l is a restricted version of U_1 which can only perform *leftmost* derivation steps. On the right-hand side of Fig. 1 we can see a complete PCPE-tree. Leaves in this tree, represented with a square, correspond to final states (specialized programs). Intermediate nodes, represented with circles, have two children, one for each control strategy. Left branches, annotated with l, correspond to applying $\langle G_1, U_1^l\rangle$ to the selected call pattern (underlined in the figure), and right branches, annotated with nl, to the application of $\langle G_1, U_1\rangle$. In T_3, T_6 and T_7 there is only one child because both strategies result in identical specialization. As can be seen, even for this simple example, five different specialized programs can be obtained using PCPE. Since PE applies a single control strategy to all call patterns, it can obtain S_1 using $\langle G_1, U_1^l\rangle$ or S_5 using $\langle G_1, U_1\rangle$. The three other programs can only be obtained by PCPE using *hybrid* strategies, i.e., using different control strategies for different call patterns.

1.2 Choosing a PCPE Specialized Program

Since in partial evaluation we are interested in obtaining a single specialized program, we need some way of automatically choosing the best program among those obtainable using PCPE. In our motivating example, we should choose one program among $S_1, \ldots, S_5$. For this purpose, an *evaluation step* [13] is introduced which uses a *fitness function*. This function assigns a numerical value to each specialized program, which is an estimate of how good the program is. The specialization process can be *resource-aware* by using a fitness function which takes into account factors such as the size and memory-efficiency of the specialized programs, in addition to time-efficiency. In our motivating example, the best program is S_3, which is twice as fast as the original program (2.01) and somewhat smaller (1.16), the product of which, i.e., the *balance* fitness function (see Section 5.1) is 2.33. In the path to S_3 we have performed nonleftmost unfolding in state T_2 (where we specialize the atom $exp\_ac(1, B, T, R)$), and leftmost in T_1 and T_4. In this example it can be seen that nonleftmost unfolding can slow down programs, as S_5 is slower (0.82) than the original program. On the other hand, restricting ourselves to leftmost precludes important optimizations, since S_1 is both slower and larger than S_3. This is an example where it is difficult to know *a priori* which is the best strategy to use. But there are many more situations where allowing to try out different strategies and compare the results is desirable. An important thing to note is that the fitness function can only be applied to leaves of the PCPE tree in order to decide *a posteriori* which specialized program is the best. I.e., we need to compute all PCPE specialized programs to be able to tell which one is the best. This generate+evaluate approach to PCPE [13] has been implemented and tested on several benchmarks, showing that PCPE can obtain better specialized programs than traditional partial evaluation. Unfortunately, this approach to PCPE is too costly in practice, even with the techniques for reducing the search space studied in [11], since the number of states in the search space grows exponentially with the cardinality of $\mathcal{CS}$.

In this work we investigate the possibility of using an *oracle* which decides which is the most promising control strategy for each call pattern based on the specialization results for such call patterns using the different strategies. All other branches in the tree are pruned away. In our example, the oracle should be able to tell us: in T_1, after specializing using the two control strategies, the most promising state between T_2 and S_5 is T_2. Then, between T_3 and T_4, the latter is preferable. Then, T_6 is preferable to S_4. Finally, from T_6 we can only reach S_3. The benefits of building such an oracle are twofold, since a single specialized program would be computed. First, we do not spend time generating multiple specialized programs. Second, as in the case of PE, we do not need an evaluation phase, which can be very costly. However, this approach can only be useful in practice if the oracle makes good decisions, since some of the PCPE specialized programs outperform PE (such as S_3), but others produce bad results (such as S_5 or S_2).

In the rest of this paper we present an *empirical oracle* whose parameters are approximated from a set of training data, gathered from a set of calibrating examples and converted into a constraint logic program. Our experimental results show that specialization based on our empirical oracle introduces a constant overhead factor w.r.t. PE, while obtaining significantly better specialized programs.

2 On-Line Partial Evaluation of Logic Programs

We recall some terminology on logic programming. See for example [10] for more details. An *atom* A is a syntactic construction of the form $p(t_1, \dots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \dots, t_n$ 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. In what follows, we restrict ourselves to definite programs. A *goal* (or query) is a conjunction of atoms. We denote by $\{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$ the *substitution* σ with $X_i\sigma = t_i$ for all $i = 1, \dots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $X\sigma = X$ for any other variable X, where t_i are terms. A substitution θ is a *unifier* of a finite set S of simple expressions 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 $t \approx t'$, if there exist substitutions θ and σ s.t. $t = t'\theta$ and $t' = t\sigma$. We now recall some concepts of on-line partial evaluation of logic programs (LP), which is traditionally presented in terms of SLD semantics ([10]).

Definition 2.1 [derivation step] Let G be a goal $\leftarrow A_1, \dots, A_R, \dots, A_k$. Let A_R be the *selected atom*. Let $C = H \leftarrow B_1, \dots, B_m$ be a renamed apart clause in P. Then G' is *derived* from G and C via A_R if the following conditions hold:

$$\begin{array}{r} \theta = mgu(A_R, H) \\ G' \text{ is the goal } \leftarrow (A_1, \dots, A_{R-1}, B_1, \dots, B_m, A_{R+1}, \dots, A_k)\theta \end{array}$$

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 $A_{R1} : A_{R2} : \ldots$ of selected atoms, 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} via A_{Ri} 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.

Given a program P and an atom A, an *unfolding rule* U computes a SLD tree τ for $P \cup \{\leftarrow A\}$, denoted $U(P, A) = \tau$. Given a finite SLD tree τ containing the SLD derivations $D_1, \ldots, D_n$ where $D_i = A, \ldots, G_i$ with computer answer substitution θ_i for $i = 1, \ldots, n$, the *resultants* of τ, denoted *resultants*(τ), is the set of clauses $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$. Then, an *abstraction function* G is applied before adding the atoms in the right-hand sides of resultants to the set of atoms to be partially evaluated. This abstraction function performs the *global control* and is in charge of guaranteeing that the number of atoms which are specialized remains finite. This is done by replacing atoms by more general ones, i.e., by losing precision in order to guarantee termination. Given an atom A and an set of atoms H, we use $A' = G(A, H)$ to denote that A' is the *abstraction* of A computed by G w.r.t. H. It is a correctness requirement that $A = A'\theta$.

3 Poly-Controlled Partial Evaluation

We now provide a general formalization of the PCPE process, which can be used as a basis for both the PCPE approach in [13] and the one proposed here. PCPE takes as input a program P, a set $\mathcal{A}$ of atoms describing the initial call patterns, and a set $\mathcal{CS}$ of control strategies. As output, PCPE can generate potentially multiple specialized programs. The PCPE process starts from an initial *state* and obtains from it a *child* state until a *final state* is reached. Since we allow several *control strategies*, non-final states can have several *children* states. Depending on the approach used, a different number of states will be expanded and thus, different (sets of) specialized programs will be obtained.

Definition 3.1 [state] A *state* is a pair $\langle S, H\rangle$, where S is a set of atoms and H is a set of tuples of the form $\langle A, A', U\rangle$. The set S contains the atoms to be specialized and H contains the specialization history: for each previously specialized atom A we store, in addition to A itself, the result A' of applying an abstraction function to it, and the unfolding rule U which has been applied on A'.

The atom A is stored in each tuple in H for precise predicate renaming, while U is stored in order to use exactly such unfolding rule during code generation (see Def. 3.5). A state is *initial* when it is of the form $\langle \mathcal{A}, \emptyset \rangle$. A state is *final* when it is of the form $\langle \emptyset, H \rangle$. States that are not final are called *intermediate* states. In Fig. 1, T_1 is the initial state and S_1,...,S_5 are final states. Also, each state is adorned with its S set. The H set of each state is not shown explicitly, but it can be retrieved by traversing the tree from each node upwards up to the root.

As customary in PE, we consider the existence of an arbitrary function, which we call $\mathsf{TakeOne}$, which given an intermediate state $\langle S, H \rangle$, decides the atom A to be specialized among those in S, denoted $A = \mathsf{TakeOne}(S)$. Also, a control strategy CS is a pair $\langle G, U \rangle$ s.t. G is an abstraction function and U is an unfolding rule. We assume the existence of a function *atoms* that extracts the generalized atoms out of the tuples in H. I.e. given $H = \{\langle A_1, A'_1, U_{1i} \rangle, \ldots, \langle A_n, A'_n, U_{nj} \rangle\}$, $atoms(H) = \{A'_1, \ldots, A'_n\}$. In an abuse of notation, when referring to abstraction functions we simply write $A' = G(A, H)$ instead of $A' = G(A, atoms(H))$. Finally, given a state $T = \langle S, H \rangle$ and a program P, we use $\tau = CS(T)$ to denote that $A = \mathsf{TakeOne}(S)$, $A' = G(A, H)$ and $\tau = U(P, A')$.

Definition 3.2 [PCPE-step] Let $T = \langle S, H \rangle$ be an intermediate state, and let $A = \mathsf{TakeOne}(S)$. Let $CS = \langle G, U \rangle$ be a control strategy. Then a *PCPE-step* for T using CS generates a new state $T' = \langle S', H' \rangle$, denoted $T \leadsto_{CS} T'$, s.t.

- $S' = (S - \{A\}) \cup \{B \in leaves(CS(T)) \mid \forall \ \langle C, \_, \_ \rangle \in H \ . \ B \not\approx C\}$
- $H' = H \cup \{\langle A, A', U \rangle\}$, with $A' = G(A, H)$

where the function *leaves* collects the atoms in the bodies of resultants($CS(T)$).

If $T \leadsto_{CS} T'$ then T' is a *child* of T. From PCPE-steps we get PCPE-paths.

Definition 3.3 [PCPE-path] A *PCPE-path* consists of a sequence $T_0 : T_1 : \ldots : T_p$ of states and a sequence $CS_1 : CS_2 : \ldots : CS_p$ of control strategies s.t. for $i = 1..$p, $T_i \leadsto_{CS_{i+1}} T_{i+1}$.

A PCPE-path $T_0 \leadsto_{CS_1} \ldots \leadsto_{CS_p} T_p$ is *complete* iff T_0 is an initial state and T_p is a final state. A state T' is *reachable* from a state T iff there is a path of the form $T \leadsto_{CS_1} \ldots \leadsto_{CS_p} T'$, $p \geq 0$. Paths can be organized into *PCPE-trees*.

Definition 3.4 [PCPE-tree] A *PCPE-tree* is a tree where each node of the tree corresponds to a state, and which satisfies:

- The root node is an initial state.
- Leaves are final states.
- There is an arc from node T to node T' iff there is a control strategy $CS \in \mathcal{CS}$ s.t. $T \leadsto_{CS} T'$.

The generate+evaluate algorithm of [13] traverses the complete PCPE-tree, gathering a set of final states. From each of these states we can obtain a *PCPE specialized program*. As usual in partial evaluation, during code generation we will rename apart atoms in order to avoid the independence requirement [6]. We use *rename* to refer to a procedure which assigns a fresh predicate name to each atom $A'_i \in atoms(H)$

and performs appropriate renamings (using the pairs of atoms A_i, A'_i in the tuples of H) in the head and body of resultants so that each program point uses a correct (and as optimized as possible) version.

Definition 3.5 [PCPE specialized program, SP] Let $T = \langle \emptyset, H \rangle$ be a final state. Then the *PCPE specialized program* P_T obtained from T, denoted $P_T = SP(T)$, is $P_T = \bigcup_{\langle A_i, A'_i, U_i \rangle \in H} rename(resultants(U_i(P, A'_i)), H)$.

From an (intermediate) state we can reach a set of final states, each one corresponding to a possibly different specialized program.

Definition 3.6 [solutions] Let T be a state. The set of *solutions* for T is defined as $\mathsf{solutions}\ (T) = \{SP(T') \mid T'$ is reachable from $T \wedge T'$ is final$\}$.

In order to choose the best PCPE specialized program, in [13] we apply an *evaluation* step which uses a *fitness function* F to assess how good each specialized program P_T is w.r.t. the original program P. The fitness function returns a value in $[0 \ldots \infty)$, with larger fitness values indicating better programs. Also, values smaller than one indicate that the specialized program is worse than the original one. As noted in [3,13], fitness functions can be resource-aware.

Definition 3.7 [maximal fitness value, mfv] Let T be a state. Let F be a fitness function. Then the *maximal fitness value* of T w.r.t. F, denoted $mfv_F(T)$, is defined as $max(\{F(P_1), \ldots, F(P_n)\})$, where $solutions(T) = \{P_1, \ldots, P_n\}$.

As usual, $max(R)$ returns the largest value in the set R. We can now define a PCPE-path leading to a solution of maximal fitness.

Definition 3.8 [PCPE-path of maximal fitness] A complete PCPE path $T_0 \rightsquigarrow_{CS_1} \ldots \rightsquigarrow_{CS_p} T_p$ is of *maximal fitness* w.r.t. a fitness function F iff $mfv_F(T_p) = mfv_F(T_0)$.

Note that for all pairs of states T and T', if T' is reachable from T then $mfv_F(T) \geq mfv_F(T')$, for any fitness function F. In a path of maximal fitness, we always perform PCPE-steps which preserve the maximal fitness value. Next, we study whether it is possible to *guess* which PCPE-steps lead to paths of maximal fitness, without traversing the complete PCPE-tree.

4 Oracle-Based Partial Evaluation

The central idea behind *Oracle-based PE (OBPE)* is to traverse only one complete PCPE-path. For this, given a state, we generate all of its children using each control strategy in $\mathcal{CS}$, and choose the *most promising child* according to some *oracle* that uses information from the specialization process of each child.

Definition 4.1 [oracle] Let P be a program. Let T be a state. Let $CS \in \mathcal{CS}$ be a control strategy s.t. $CS(T) = \tau$. An *oracle* is a function which receives as input T, τ and P and returns a number $Q \in \mathbb{Q}$. This is denoted $Q = \mathsf{oracle}(T, \tau, P)$.

A *perfect* oracle always obtains a solution of maximal fitness value.

Definition 4.2 [perfect oracle] Given a fitness function F, an oracle function oracle is *perfect* w.r.t. F if for any state T,

$$\begin{array}{r}(T \leadsto_{CS_i} T_i \wedge T \leadsto_{CS_j} T_j) \wedge \\ (\mathsf{oracle}(T, CS_i(T), P) \geq \mathsf{oracle}(T, CS_j(T), P)) \Rightarrow \\ mfv_F(T_i) \geq mfv_F(T_j)\end{array}$$

Finding a perfect oracle function will in general be impossible since the information available to the oracle is not quite enough in order to make perfect decisions. However, as our experimental results show, good results can be obtained without a perfect oracle function.

Since the oracle can rank two children with the same value, we impose an order on the generated children of a given state, by using a *sequence* of control strategies instead of a set. We can then use this order to break any possible tie.

We define now a function *mpchild*, which chooses a most promising child out of a sequence of children states. In case of a tie, *mpchild* selects the *first* state in the sequence having the highest Q value.

Definition 4.3 [mpchild] Let T be a state. Let $\mathcal{CS} = CS_1 : \ldots : CS_m$ be a *sequence* of control strategies. Let $\mathcal{T} = T_1 : \ldots : T_m$ with $T \leadsto_{CS_i} T_i$ be the children of T. Let $\mathcal{T}' = T_{i1} : \ldots : T_{in}$ be the maximal sub-sequence of $\mathcal{T}$ s.t. $\forall T_{ij} \in \mathcal{T}'$ oracle $(T, CS_{ij}(T), P) = Q$ and $\forall T_k \in \mathcal{T}$ oracle $(T, CS_k(T), P) \leq Q$. Then T_{i1} is the *most promising child* of T, denoted $T_{i1} = mpchild(T)$.

In OBPE, steps are deterministic: only the most promising child is expanded.

Definition 4.4 [OBPE-step] Let T be a state. Then an *OBPE-step* for T generates a new state T' s.t. $T' = mpchild(T)$.

OBPE receives as input a program P, a set $\mathcal{A}$ of atoms describing the initial call patterns, a sequence $\mathcal{CS}$ of control strategies, and a selection function $\mathsf{TakeOne}$. It starts by building an initial state $\langle \mathcal{A}, \emptyset \rangle$, and then performs a series of OBPE-steps until a final state $\langle \emptyset, H \rangle$ is reached, i.e., it traverses a complete PCPE-path, therefore generating only one specialized program $P' = SP(H)$.

5 An Empirical Oracle Function using a Linear Model

We now propose an oracle model which makes the problem of *empirically* determining an oracle function tractable. Furthermore, using this model, we obtain oracle functions which can be executed efficiently. This is important since during the specialization process the oracle is applied many times.

We propose to decompose the oracle function into two parts. The first one corresponds to computing the numerical value of a vector of *observables*, which should capture the relevant information about the specialization process. For this we use an auxiliary function $\mathsf{quantify}$, which takes as input a state T, an SLD tree τ, and a program P and extracts the numeric value corresponding to each observable, denoted $\overline{o} = \mathsf{quantify}(T, \tau, P)$. The second part corresponds to the oracle function proper, which returns a numerical value as a function of the values of the observables.

5.1 Useful Observables for Resource-Aware Specialization

Since the oracle function will make its decisions based on the values of the observables, the practical success of OBPE has as prerequisite determining the *right* set of observables for the considered fitness function. Those aspects of the specialization process which have a lot of impact on the quality of a specialized program should be considered. Otherwise, the oracle will not be able to make good decisions. In our case, as an example of a resource-aware specialization policy, we consider the fitness function balance [13,3], which takes into account both the time and space efficiency of the specialized program P_T w.r.t. the original program P:

$$\mathsf{balance}(P_T) = \mathsf{speedup}(P_T) \times \mathsf{reduction}(P_T) = \frac{Time(P)}{Time(P_T)} \times \frac{Size(P)}{Size(P_T)}$$

Fig. 1 shows the fitness value of the specialized program obtained from each final state, and also shows, in parentheses, the speedup and reduction values. Thus, the observables considered should somehow take these two factors into account. In all our experiments we consider the following observables, where the first three ones are mostly related to time efficiency, and the last two to space efficiency:

D: The number of *derivation steps* that have been performed during unfolding and thus no longer need to be performed at runtime.

E: The number of *evaluation steps* that have been performed during unfolding. This indicates the number of calls to builtins and library predicates which have been evaluated [12] at specialization-time.

N: The number of atoms whose computation is replicated in several clauses as a result of *non-deterministic non-leftmost unfolding*. It is well-known that non-leftmost unfolding can increase the amount of computation required, by replicating the computation of atoms to the left of the selected one.

C: An estimation of the *growth of the residual code*, computed as a factor between the size (using a variation of the *term size* metrics [5]) of the specialized code for the selected atom A and the size of the original definition of the predicate $pred(A)$.

S: An estimation of the code size for the atoms added to S as a result of the current PCPE-step. Since no specialized code is available for these atoms, we use their original definition in P as an estimate of their size.

In most existing control strategies, which are focused on time efficiency, observables C and S are not explicitly handled and most heuristics aim at maximizing D and E while keeping N with the value zero. Observable S is an example of information which is just partial when applying the oracle: in order to obtain a covered program, the code for the new atoms in S may in turn need including code for other atoms not yet covered. Perfect information can only be determined by actually expanding the PCPE-tree and observing it *a posteriori*.

Example 5.1 Given the tree in Fig. 1, the value of the observables $\langle D, E, N, C, S\rangle$ which correspond to $T_1 \rightsquigarrow_{\langle G_1, U_1^l\rangle} T_2$ is $\langle 3, 2, 0, 1.3, 8\rangle$. This is because 3 derivation steps have been performed during unfolding of `main(A,B,C)` with the U_1^l rule, and 2 calls to builtins have been evaluated. In this case, as well as in all states obtained by applying U_1^l, the value of N is 0, since U_1^l only performs leftmost derivation steps. The growth of the residual code w.r.t. the original definition of `main/3` is 1.3, and the estimation of the size of the code associated to the atoms `exp_ac(1)` and `p(X)` added to S is 8. Furthermore, in the case of $T_1 \rightsquigarrow_{\langle G_1,U_1\rangle} S_5$, the vector is $\langle 13, 4, 15, 8.0, 0\rangle$ because U_1 has performed 13 derivation steps and 4 evaluation steps. However, it replicates 15 atoms by doing non-deterministic non-leftmost unfolding. The growth of the residual code w.r.t. the original definition of `main/3` is 8.0 and $S = 0$ since no new atoms appear in the resultants.

5.2 A Linear Model for the Oracle

In order to simplify our oracle model as much as possible, we will restrict ourselves to *linear* oracle functions.

Definition 5.2 [linear oracle function] Let $\overline{o} = \langle o_1, \ldots, o_n\rangle$ be an observable vector. A *linear oracle function* oracle receives $\overline{o}$ as input and returns a numeric value $Q \in \mathbb{Q}$ which is defined as $Q = \mathsf{oracle}(\overline{o}) = \sum_{i\in\{1,\ldots,n\}} k_i \times o_i$, where $\overline{k} = \langle k_1, \ldots, k_n\rangle$ is a vector of *oracle constants*, $k_i \in \mathbb{Q}$.

To obtain a vector of oracle constants to be used with a given fitness function F, we build complete PCPE-trees, compute the mfv_F of all nodes, and then use this information as *training data*. For this, *O-constraints* are generated.

Definition 5.3 [O-constraint] Let F be a fitness function and T a state. Let T_1 and T_2 be two children of T s.t. $T \rightsquigarrow_{CS_1} T_1$ and $T \rightsquigarrow_{CS_2} T_2$. Let $\overline{o}_1 = \mathsf{quantify}(T, CS_1(T), P)$ and $\overline{o}_2 = \mathsf{quantify}(T, CS_2(T), P)$. Then the *O-constraint* for the pair (T_1, T_2) is $\mathsf{oracle}(\overline{o}_1)\ \mathcal{R}\ \mathsf{oracle}(\overline{o}_2)$, where $mfv_F(T_1)\ \mathcal{R}\ mfv_F(T_2)$, $\mathcal{R} \in \{<, \leq, =, \geq, >\}$.

Given a PCPE-tree *Tree*, we use $\mathcal{C}(\mathit{Tree})$ to denote the set of O-constraints which can be obtained from *Tree*. The cardinality of $\mathcal{C}(\mathit{Tree})$ is usually quite large: for each intermediate node T in *Tree* with p children we can build $\binom{p}{2}$ constraints for T. Thus, for a realistic tree *Tree* it is not possible to find a vector of oracle constants which allow satisfying all constraints in $\mathcal{C}(\mathit{Tree})$ simultaneously. There are several reasons for this. First, we have restricted ourselves to linear functions. It could be the case that there exists a non-linear oracle function which satisfies all constraints. However, the advantage of linear functions is that there exist tools capable of handling them, whereas inferring non-linear functions is a rather complicated task. Second, as already mentioned, a perfect oracle function does not exist in general, since it has to make decisions based on partial information, i.e., without expanding the complete tree below the current node.

We can formulate the process of finding a vector of oracle constants as a *Maximum Constraint Satisfaction Problem* (MAX CSP): though the set of O-constraints is unsatisfiable, the goal is to find a vector of oracle constants that maximizes the number of satisfied constraints in $\mathcal{C}(\mathit{Tree})$. Unfortunately, the cardinality of $\mathcal{C}(\mathit{Tree})$ is large in general, and finding an optimal solution to this MAX CSP problem is quite costly. A simpler model results from collecting only (some of) the O-constraints occurring in a PCPE-path of maximal fitness.

Definition 5.4 [step-constraint] Let T be a state. Let $\mathcal{CS} = CS_1 : \ldots : CS_m$. Let $T_1 : \ldots : Tm$ be the children of T with $T \rightsquigarrow_{CS_i} T_i$. Let $T_i = \mathit{mpchild}(T)$. Then a *step-constraint* for T is $\bigwedge_{j=1..m \wedge j \neq i} \mathsf{oracle}(\overline{o}_i) \geq \mathsf{oracle}(\overline{o}_j)$.

Example 5.5 In the PCPE tree of Fig. 1, we have labeled some arcs with a vector $\overline{o} = \langle D, E, N, C, S\rangle$ containing the actual values the function $\mathsf{quantify}$ would return, and with the value Q computed for each vector $\overline{o}$ by using the empirical linear oracle function oracle we have obtained in our experiments. As can be seen in the figure, the PCPE-path traversed by OBPE would be $T_1 \rightsquigarrow_{\langle G_1, U_1^l\rangle} T_2 \rightsquigarrow_{\langle G_1, U_1\rangle} T_4 \rightsquigarrow_{\langle G_1, U_1^l\rangle} T_6 \rightsquigarrow S_3$, which coincides with the solution of maximal fitness value S_3. This is because $-8.9 \geq -23.4$, $3.7 \geq -6.2$, and $-9.4 \geq -10.7$. Also, an example of O-constraint generated for the pair (T_2, S_5), using the simplified linear model defined above, would be $3 \times D + 2 \times E + 0 \times N + 1.3 \times C + 8 \times S \geq 13 \times D + 4 \times E + 15 \times N + 8 \times C + 0 \times S$, since $mfv_F(T_2) \geq mfv_F(S_5)$, $F=$ $\mathsf{balance}$. This is also a step-constraint for T_1.

By collecting only those step-constraints in a PCPE-path of maximal fitness we have an instance of a Max CSP that is more tractable than the original model that considered all O-constraints in a complete PCPE-tree. We have used as *calibration benchmarks* those used in [11], since they are a representative set of PCPE examples for which it is possible to compute the complete PCPE tree. For each benchmark we collect a set $\mathcal{C}_j$ of step-constraints. Then, we enumerate all possible subsets $\mathcal{C}'_{ji} \subseteq \mathcal{C}_j$ and input each $\mathcal{C}'_{ji}$ to a Constraint Logic Programming solver [3], larger subsets first, until we find a maximal satisfiable subset $\mathcal{C}'_{ji}$ for each benchmark and its solution $\overline{k}_j = \langle k_{j1}, \ldots, k_{jn}\rangle$.

After collecting a set $\{\overline{k}_1, \ldots, \overline{k}_p\}$ of oracle constants, one for each of the calibration benchmarks, we normalize the value of each vector $\overline{k}_j$ by forcing the absolute value of the first constant k_{j1} (in our case corresponding to the observable D) to be 1 (written $|k_{j1}| = 1$). This is done by multiplying all constants $k_{j1}, \ldots, k_{jn}$ in each vector by $1/|k_{j1}|$. Note that this is a correct transformation since by multiplying a vector by a constant greater than zero, all constraints which were satisfied are again satisfied. Finally, the calibrated oracle constants result from computing the arithmetic mean over each normalized constant k_{ji}.

6 Experimental Results

We have run a series of experiments in order to both evaluate the quality of the specialized programs obtained by means of OBPE and to compare the cost of this approach w.r.t. other specialization techniques. Three different specialization techniques have been considered: standard PE (column **PE**), the optimized generate+evaluate PCPE presented in [11], which prunes the PCPE-tree using a combination of heuristics and branch and bound techniques (column **PB-PCPE**), and Oracle-based PE (column **OBPE**).

[3] We have used the **clpq** solver available in `Ciao` [2].

Benchmark	LOC	Size	PE				PB-PCPE	OBPE
			CS_1	CS_2	CS_3	CS_4		
analysis	343	39985	0.0001	0.69	0.02	**1.03**	-	1.19
boyer	407	36619	0.33	0.52	0.59	**0.99**	1.04	1.01
browse	119	12579	0.78	1.76	2.21	**2.55**	2.65	2.57
credit	264	16932	**1.64**	1.39	0.72	1.37	-	1.81
exp_p	34	5639	0.55	0.86	0.57	**0.96**	0.95	0.86
gr_unify	78	10164	**5.76**	4.63	0.27	1.01	-	6.04
prolog_read	396	28300	0.08	0.10	0.94	**0.96**	-	5.09
qplan	397	37512	0.84	0.84	1.02	**1.04**	-	0.99
vanilla_db	110	13395	32.21	1.09	**32.39**	1.02	36.61	32.50
Geom Mean	**180.28**	**18502.74**	**0.40**	**0.88**	**0.77**	**1.15**	**-**	**2.56**

Table 1
Quality of Specialized Programs

We have used four control strategies, i.e., $\mathcal{CS} = CS_1 : CS_2 : CS_3 : CS_4$ with $CS_1 = \langle G_1, U_1 \rangle$, $CS_2 = \langle G_1, U_2 \rangle$, $CS_3 = \langle G_2, U_1 \rangle$, and $CS_4 = \langle G_2, U_2 \rangle$. G_1 is an abstraction function based on homeomorphic embedding [9,8] and flags atoms as potentially dangerous (and are thus generalized) when they homeomorphically embed any of the previously visited atoms. G_2 abstracts away the value of all arguments of the atom and replaces them with distinct variables. U_1 is an unfolding rule based on homeomorphic embedding (see [12]). It can handle external predicates safely and can perform non-leftmost unfolding as long as unfolding is safe (see [1]) and *local* (see [12]). Finally, U_2 performs deterministic unfolding.

The first phase of the experiments involves obtaining an empirical oracle function. For this, we have used the benchmarks in [11] and have executed PB-PCPE over them using the set $\mathcal{CS}$ mentioned above. We do not show results of this phase due to lack of space. We just mention that running PE with the four strategies in $\mathcal{CS}$, the one which obtains the best overall results is CS_1. During the second phase, we have used a set of benchmark programs *not* included in the set of calibrating benchmarks[4], since we are interested in knowing whether our oracle function obtains good results for arbitrary programs. Some of these programs are actual libraries from existing Prolog systems, and most of them contain several hundred lines of source code, as shown in column **LOC** of Table 1. In this table, column **Size** shows the size of the compiled bytecode of each benchmark.

Table 1 compares the quality of the specializations obtained in terms of the fitness value (using balance) of the (best) solution found by each approach. In order to be as fair as possible, we compare both **PB-PCPE** and **OBPE** using $\mathcal{CS}$ against traditional PE using *all* control strategies in $\mathcal{CS}$. For each benchmark, we specify in bold the fitness value of the winning control strategy using PE. These values are not very high in several benchmarks. This is mainly because not much static data is available for such benchmarks. By looking at this table it seems that, at least for the balance fitness function, there is no single control strategy which allows consistently obtaining good results. For instance, if we look at CS_1, which was the winning strategy for the calibration benchmarks, we see that in some cases it produces specialized programs that are considerably better than the

[4] Source code available at `http://clip.dia.fi.upm.es/Systems/pcpe`.

original program—`groundunify_simple`, `gr_unify` in the tables for short, (5.76) and `vanilla_db` (32.21)—while in most cases it obtains specialized programs that are worse than the original program (fitness values below 1). An interesting case is `analysis`. The original program has 343 lines of code, the program obtained by CS_1 has over 38000 lines of code, and its compiled bytecode is over 5Gb. This is because U_1 is an aggressive unfolding rule and it tries to unfold as much as possible, in this case resulting in code explosion, which is harmful in resource-aware program specialization. Indeed, the fitness value for this benchmark is so low that the geometric mean computed over all benchmarks but `analysis` is 1.14 (vs 0.40). By looking at the overall results (row **Geom Mean**), it seems that the best control strategy for dealing with these benchmarks is CS_4. However, this is a quite conservative control strategy, and does not benefit from the static information provided to the specializer. Thus, for many of the benchmarks, fitness results using CS_4 are close to 1, as observed in the table, with the exceptions of `browse` and `credit`.

PB-PCPE runs out of memory for several benchmarks, indicated with "–" in the table. As a result, we do not compute its geometric mean. OBPE, on the other hand, performs well in most cases, finding specialized programs that are, in average, 2.56 times better than the original one, as indicated in the **Geom Mean** row, and consistently similar or slightly better than the program obtained by the best PE, with a couple of exceptions. In the case of `exponential_peano` (`exp_p` in the tables for short), the specialized program is worse than that achieved using CS_4. This is an indication that, for this benchmark, our empirical oracle function has not made perfect decisions. The other exception is `prolog_read`, where the program obtained by OBPE is considerably better than any of the four programs

Benchmark	**PE**		**OBPE**		
	CS_1	CS_4	**States**	**Path**	**Ties**
analysis	334	54	227	77	2
boyer	83	32	44	15	7
browse	15	11	14	6	0
credit	28	25	82	32	3
exp_p	8	7	18	8	0
gr_unify	8	13	59	22	0
prolog_read	184	48	212	54	6
qplan	53	51	161	49	3
vanilla_db	7	4	19	10	0
Overall	**33.9**	**19.6**	**58.4**	**21.6**	**7.69 %**

Table 2
Number of States and Details on Specialization

obtained by PE, which in all cases have a fitness below 1. This is an indication that OBPE allows obtaining *hybrid* solutions which are not achievable using any of the control strategies in isolation, and which outperform the solutions of PE. Note that, if we decide to use PE with several control strategies, we would again need to introduce an evaluation step (as in PB-PCPE) and which is not needed in OBPE. Finally, it is worth mentioning that OBPE outperforms PE using any of the four control strategies in isolation by a factor of 2.22 or higher.

In addition to evaluating the benefits of OBPE, it is also important to evaluate its *cost*. Table 2 shows the number of states generated by PE and OBPE for each benchmark, and in the case of OBPE, some additional data. We no longer include PB-PCPE, since as already seen in Table 1, it runs out of memory in several of the benchmarks and it is not a realistic alternative to traditional PE in general purpose specialization. The row **Overall** shows the *geometric mean* computed over the different benchmarks, except for the column **Ties**, which is discussed below. For simplicity, in this figure the comparison is against PE using CS_1 and CS_4 only, denoted PE_{CS_1} and PE_{CS_4} respectively, since they are the two most interesting cases. The former is the most aggressive control strategy (and the winning strategy in the calibration bencharks), while the latter is the most conservative one (and also the one which obtains the best fitness values in Table 1). It is important to note that the number of states generated by OBPE is bounded by the length of the PCPE path chosen by the oracle multiplied by the cardinality of $\mathcal{CS}$ (in our case, four). However, it can be seen in the table that, at least in our experiments, the number of states generated by OBPE (58.4), as shown in column **States** under **OBPE**, is slightly less than twice as many states as PE_{CS_1} (33.9), and three times as many states as PE_{CS_4} (19.6). There are several reasons for this. One is that the best solution under balance tends to have fewer predicates in the residual code, which implies that the path traversed is shorter. This can be observed in the column **Path**, which indicates the length of the PCPE-path whose overall is smaller (21.6) than that of PE_{CS_1} (33.9) and quite similar to that of PE_{CS_4} (19.6).

Another reason for this is that, for efficiency, in the implementation abstraction functions are applied first, and then those generalized atoms which are different are unfolded, i.e., if after abstraction we obtain two identical generalized atoms, only two children states are generated, instead of four. Column **Ties** shows the number

Bench.	**PE (CS_1)**			**PE (CS_4)**			**OBPE**			
	Spec	**CG**	**Tot**	**Spec**	**CG**	**Tot**	**Spec**	**Ora**	**CG**	**Tot**
analysis	7571	30648	**38219**	462	260	**721**	13612	1254	339	**13951**
boyer	11077	296	**11374**	14494	254	**14748**	11037	25	203	**11240**
browse	587	71	**658**	854	62	**915**	623	4	50	**673**
credit	238	161	**399**	285	144	**429**	492	26	161	**653**
exp_p	109	30	**139**	153	29	**181**	110	2	23	**133**
gr_unify	122	36	**158**	177	61	**238**	208	10	52	**260**
prolog_read	880	1847	**2728**	405	306	**711**	1839	438	616	**2455**
qplan	373	438	**811**	799	429	**1228**	642	81	447	**1089**
vanilla_db	9700	2546	**12246**	186	41	**227**	15675	7	3714	**19389**
G. Mean	**925**	**394**	**1598**	**517**	**121**	**690**	**1392**	**29**	**206**	**1709**

Table 3
Specialization Time

of times the oracle returns the same value for two children. If this number were too high, it would probably indicate that the set of observables chosen does not convey enough information, and the possibility of choosing the wrong path would increase. However, this happens only 7.69% of the total number of decisions taken.

Finally, Table 3 shows the specialization times (in msecs) of both PE (using CS_1 and CS_4) and OBPE. In all cases, total specialization times (in columns **Tot**) are split into time spent doing partial evaluation (columns **Spec**), and code generation (columns **CG**). In the case of OBPE, we also add a column **Ora** to indicate the time spent by the oracle function when selecting the most promising child state. Experiments have been run using `Ciao` 1.13 over a 2.6 Linux kernel, on a Pentium IV 3.4GHz CPU, with 512Mb of RAM. We can see that the total specialization time of OBPE (1708.5) is quite similar to that of PE_{CS_1} (1597.5), and 2.47 times higher than that of PE_{CS_4} (690.0). These times are consistent with the number of states which need to be generated in the different approaches, plus the cost of code generation. An important point to mention is that the cost of OBPE represents a constant overhead factor w.r.t. PE. Such factor is directly proportional to the cardinality of $\mathcal{CS}$. For aggressive strategies, such as CS_1, the cost of OBPE is quite close to that of PE. Also, we can see that the time spent by the oracle function is negligible when compared to the total specialization time.

7 Discussion

Control of PE has received considerable attention, but there is still plenty of room for improvement, especially in the context of resource aware specialization. Many decisions have to be taken during PE and it is often not obvious which is the right choice. The main advantage of PCPE is that we do not need to restrict ourselves to a single control strategy, but rather we can use several ones. This opens up the door to obtaining hybrid specializations which often outperform pure ones. However, the main problem of the generate+evaluate approach to PCPE [13] is that, even with the optimizations proposed in [11], it is too expensive in practice: it is an alternative only when the quality of the specialized program is of much importance, and the PCPE tree has a moderate size.

In this paper we have presented Oracle-based PE. This approach, in contrast to previous work [13,11], introduces a constant overhead factor, instead of an exponential one, to the complexity of standard PE. At least in our experiments, OBPE obtains specialized programs which are significantly better than those generated by standard PE and the constant overhead factor is quite reasonable.

Acknowledgments

The authors would like to thank Samir Genaim for useful discussions on the use of constraint logic programming for calibrating the oracle.

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] Albert, E., G. Puebla and J. Gallagher, *Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates*, in: *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in LNCS (2006), pp. 115–132.

[2] Bueno, F., D. Cabeza, M. Carro, M. Hermenegildo, P. López-García and G. P. (Eds.), *The Ciao System. Ref. Manual (v1.13)*, Technical report, C. S. School (UPM) (2006), available at `http://www.ciaohome.org`.

[3] Craig, S.-J. and M. 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* (2005), pp. 23–34.

[4] Debray, S. K., *Resource-Bounded Partial Evaluation*, in: *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1997), pp. 179–192.

[5] Debray, S. K. and N. W. Lin, *Cost analysis of logic programs*, ACM Transactions on Programming Languages and Systems **15** (1993), pp. 826–875.

[6] Gallagher, J., *Tutorial on specialisation of logic programs*, in: *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1993), pp. 88–98.

[7] Jones, N., C. Gomard and P. Sestoft, "Partial Evaluation and Automatic Program Generation," Prentice Hall, New York, 1993.

[8] Leuschel, M., *On the power of homeomorphic embedding for online termination*, in: G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503 (1998), pp. 230–245.

[9] Leuschel, M. and M. Bruynooghe, *Logic program specialisation through partial deduction: Control issues*, Theory and Practice of Logic Programming **2** (2002), pp. 461–515.

[10] Lloyd, J., "Foundations of Logic Programming," Springer, second, extended edition, 1987.

[11] Ochoa, C. and G. Puebla, *Poly-Controlled Partial Evaluation in Practice*, in: *ACM Partial Evaluation and Program Manipulation (PEPM'07)* (2007), pp. 164–173.

[12] Puebla, G., 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 (2005), pp. 149–165.

[13] Puebla, G. and C. Ochoa, *Poly-Controlled Partial Evaluation*, in: *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)* (2006), pp. 261–271.

[14] Venken, R. and B. Demoen, *A partial evaluation system for prolog: some practical considerations*, New Generation Computing **6** (1988), pp. 279–290.