Trace Analysis for Predicting the Effectiveness of Partial Evaluation*

Germán Vidal

Technical University of Valencia, Spain.
gvidal@dsic.upv.es

1 Introduction

The main goal of *partial evaluation* [3] is program specialization. Essentially, given a program and *part* of its input data—the so called *static* data—a partial evaluator returns a new, residual program which is specialized for the given data. An appropriate *residual* program for executing the remaining computations—those that depend on the so called *dynamic* data—is thus the output of the partial evaluator. Despite the fact that the main goal of partial evaluation is improving program efficiency (i.e., producing faster programs), there are very few approaches devoted to formally analyze the effects of partial evaluation, either *a priori* (prediction) or *a posteriori*. Recent approaches (e.g., [1, 5]) have considered *experimental* frameworks for estimating the best *division* (roughly speaking, a classification of program parameters into static or dynamic), so that the optimal choice is followed when specializing the source program.

Here, we introduce an alternative, *symbolic* approach for predicting the potential effects of partial evaluation (which is, in principle, computationally less expensive). Basically, we first generate a finite representation that safely describes all possible *call traces* (i.e., sequences of predicate calls) for a given program. Then, we analyze how this finite representation would change by a particular partial evaluation. By comparing the original and the transformed representations, one may in some cases predict the effects of running the partial evaluator. A more detailed description of our approach can be found in [6].

2 Trace Analysis for Logic Programs

We consider a fixed domain of predicate symbols Π. We assume that Π do not contain occurrences of the same predicate name with different arities. Furthermore, we consider a fixed computation rule for call traces, namely Prolog's leftmost computation rule, which we denote by $\mathcal{R}_{left}$. We label SLD resolution steps with the predicate symbol of the selected atom, i.e., we write $Q_0 \overset{p_0}{\leadsto} Q_1 \overset{p_1}{\leadsto} \ldots$ with $pred(\mathcal{R}_{left}(Q_i)) = p_i \in \Pi$, $i \geq 0$, where $pred(A)$ returns the predicate symbol of atom A.

* This work is partially supported by the EU (FEDER) and the Spanish Ministry MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008.

Definition 1 (call trace). *Let P be a program and Q_0 a query. We say that $\tau = p_0\, p_1 \ldots p_{n-1} \in \Pi^*$, $n \geq 1$, is a call trace for Q_0 with P iff there exists a successful SLD derivation* $Q_0 \overset{p_0}{\leadsto} Q_1 \overset{p_1}{\leadsto} \ldots \overset{p_{n-1}}{\leadsto} Q_n$.

The first step of our trace analysis consists in producing a *context-free grammar* (CFG) associated to the considered program. A CFG is a tuple $G = \langle \Sigma, N, R, S \rangle$, where Σ and N are two disjoint sets of *terminals* and *non-terminals*, respectively, $S \in N$ is the *start* symbol, and R is a set of rules. In the following, given a predicate symbol $p \in \Pi$, we denote by $\overline{p} \notin \Pi$ a fresh symbol representing the non-terminal associated to p. Furthermore, we let $\overline{pred}(A) = \overline{p}$ if $A = p(t_1, \ldots, t_n)$. Also, we let $\overline{\Pi}$ denote the set $\{\overline{p} \mid p \in \Pi\}$ of non-terminals associated to predicate symbols. In contrast, we directly use predicate symbols from Π as terminals. We let START be a fresh symbol not in $\Pi \cup \overline{\Pi}$ which we use as a generic start symbol for CFGs.

Definition 2 (trace CFG, CFG_q^P). *Let P be a program and $q \in \Pi$ a predicate symbol. The associated trace CFG is* $\mathrm{CFG}_q^P = \langle \Pi, \overline{\Pi} \cup \{\mathrm{START}\}, R, \mathrm{START} \rangle$, *where*

$$R = \{\mathrm{START} \to \overline{q}\} \cup \{\overline{pred}(A_0) \to pred(A_0)\overline{pred}(B_1) \ldots \overline{pred}(B_n) \mid A_0 \leftarrow B_1, \ldots, B_n \in P, n \geq 0\}$$

Roughly speaking, the trace CFG associated to a logic program mimics the execution of the original program by replacing queries (sequences of atoms) by sequences of non-terminals and by producing a terminal with the predicate symbol of the selected atom at each SLD-resolution step.

Example 1. Consider the following program P which defines a procedure for multiplying all elements of a list by a given value:[1]

(c_1) $mlist([\,], I, [\,])$.
(c_2) $mlist([X|R], I, L) \leftarrow ml(X, R, I, L)$.
(c_3) $ml(X, R, I, [XI|RI]) \leftarrow mult(X, I, XI),\ mlist(R, I, RI)$.
(c_4) $mult(0, Y, 0)$. (c_5) $mult(s(X), Y, Z) \leftarrow mult(X, Y, Z1),\ add(Z1, Y, Z)$.
(c_6) $add(X, 0, X)$. (c_7) $add(X, s(Y), s(Z)) \leftarrow add(X, Y, Z)$.

where natural numbers are built from 0 and $s(\cdot)$. The associated trace CFG is $\mathrm{CFG}_{mlist}^P = \langle \{mlist, ml, mult, add\}, \{\mathrm{START, MLIST, ML, MULT, ADD}\},\ R, \mathrm{START} \rangle$, where the set of rules R is as follows:

START → MLIST	ML → *ml* MULT MLIST	
MLIST → *mlist*	MULT → *mult*	ADD → *add*
MLIST → *mlist* ML	MULT → *mult* MULT ADD	ADD → *add* ADD

In [6], we prove that CFG_q^P is indeed a correct approximation of the call traces for P w.r.t. the leftmost computation rule $\mathcal{R}_{left}$.

[1] In the examples, we write non-terminals associated to predicates using capital letters.

Unfortunately, trace CFGs do not always allow us to produce a simple and compact representation of the call traces of a program (e.g., when the associated language is not regular). To overcome this drawback, we use the transformation from [4] to approximate a trace CFG with a *strongly regular* grammar (SRG). The relevance of SRGs is that they can be mapped to equivalent finite-state automata using an efficient algorithm. Moreover, the transformation of [4] guarantees that the result remains readable and mainly preserves the structure of the original CFG, which is particularly useful in our context.

A grammar is *left-linear* if every rule has either the form $(A \rightarrow t)$ or $(A \rightarrow t\,B)$, where t is a finite sequence of terminals and A, B are non-terminals.

Definition 3 (trace SRG, $\textsc{srg}_q^P$). *Let P be a program and $q \in \Pi$ a predicate symbol. The associated trace SRG, $\textsc{srg}_q^P$, is obtained from $\textsc{cfg}_q^P$ as follows. First, we compute the sets of mutually recursive non-terminals of $\textsc{cfg}_q^P$. Then, for each set M of mutually recursive non-terminals such that their rules are not all left-linear* w.r.t. the non-terminals of M *(i.e., considering non-terminals from $(\overline{\Pi} \setminus M)$ as terminals), we apply a grammar transformation as follows:*

1. *For each non-terminal $A \in M$, we introduce a fresh non-terminal A' and add the rule $A' \rightarrow \epsilon$ to the grammar (we denote by ϵ the empty sequence).*
2. *For each non-terminal $A \in M$ and each rule $A \rightarrow t_0\, B_1\, t_1\, B_2\, t_2 \ldots B_m\, t_m$ of $\textsc{cfg}_q^P$ with $m \geq 0$, $B_1, \ldots, B_m \in M$, $t_0, \ldots, t_m \in (\Pi \cup (\overline{\Pi} \setminus M))^*$, we replace this rule by the following set of rules:*

$$A \rightarrow t_0\, B_1 \qquad B_1' \rightarrow t_1\, B_1 \qquad \ldots \qquad B_{m-1}' \rightarrow t_{m-1}\, B_m \qquad B_m' \rightarrow t_m\, A'$$

   (Note that this set reduces to $A \rightarrow t_0\, A'$ when $m = 0$.)

We let $\textsc{srg}_q^P = \langle \Pi, \overline{\Pi} \cup N \cup \textsc{start}, R', \textsc{start} \rangle$, where R' are the rules obtained as described above and N are the fresh non-terminals added during this process.

Example 2. Consider the $\textsc{cfg}_q^P$ of Example 1. The sets of mutually recursive non-terminals are $\{\{\textsc{mlist}, \textsc{ml}\}, \{\textsc{mult}\}, \{\textsc{add}\}\}$. Here, the rules for both $\textsc{mlist}$ and $\textsc{ml}$ are left-linear w.r.t. $\{\textsc{mlist}, \textsc{ml}\}$. The rules for $\textsc{add}$ are clearly left-linear too. However, the second rule of $\textsc{mult}$ is not left-linear because, even if $\textsc{add}$ is treated as a terminal, it appears *to the right* of the non-terminal $\textsc{mult}$. Therefore, in $\textsc{srg}_{mlist}^P$ we replace the original rules for $\textsc{mult}$ by the following ones:

$$\{\textsc{mult}' \rightarrow \epsilon,\ \ \textsc{mult} \rightarrow \mathit{mult}\ \textsc{mult}',\ \ \textsc{mult} \rightarrow \mathit{mult}\ \textsc{mult},\ \ \textsc{mult}' \rightarrow \textsc{add}\ \textsc{mult}'\}$$

Once we have an SRG that safely approximates the call traces of a program, there are several possibilities for representing the language generated by this SRG in a compact and intuitive way. Here, we consider the generation of a *finite-state automaton* (FA) that accepts the language generated by the SRG; an alternative approach that produces *regular expressions* can be found in [6].

A finite-state automaton (FA) is specified by a tuple $\langle Q, \Sigma, \delta, s_0, F \rangle$, where Q is a set of states, Σ is an input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a set of transitions, $s_0 \in Q$ is the start state and $F \subseteq Q$ is a set of final states. For constructing a

s_0 = START, s_1 = MLIST, $s_2 = \epsilon$, s_3 = ML, s_4 = MULT MLIST, s_5 = MULT$'$ MLIST, s_6 = ADD MULT$'$ MLIST

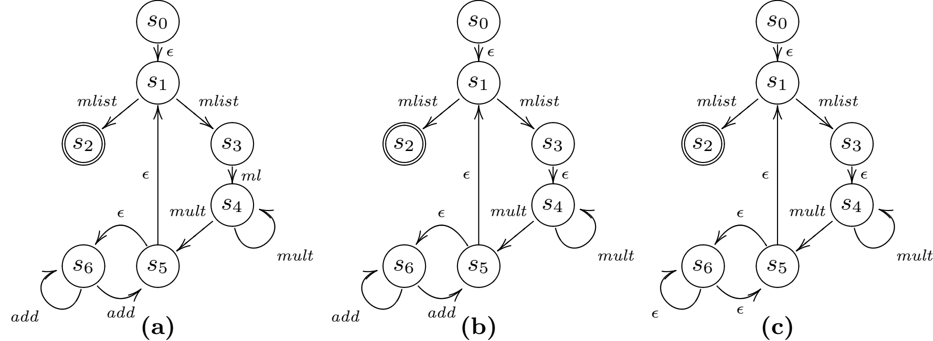

Fig. 1. Transformation of trace FAs

finite automaton $FA(G)$ from an SRG G, called *trace FA*, we follow the classical approach: there is a start state associated to the start symbol of the SRG; for each reduction $w \to w'$ with a rule $A \to t\, B$ of the SRG, we have a transition (s, α, s') in the FA, where states s, s' are associated with the sequence of non-terminals in w, w' and character α is set to the sequence t in the applied rule.

Example 3. Consider the SRG SRG^P_{mlist} of Example 2. The associated FA is shown in Fig. 1 (a), where the final state s_2 is denoted with a double circle.

3 Towards Predicting the Speedup of Partial Evaluation

The trace analysis gives us the *context* where every predicate call appears. Now, we informally describe two transformations (a more formal definition can be found in [6]) that modify the computed traces to account for the potential effects of a partial evaluation. By analyzing the traces before/after partial evaluation, one can extract useful conclusions on its effectiveness.

The first transformation is used to eliminate *intermediate* predicates. Basically, for every state with exactly one input transition and one output transition, we replace the label of the output transition by ϵ (i.e., we delete calls to predicates which are called from a single program point). Consider the trace FA of the program of Ex. 1 which is shown in Fig. 1 (a). After the elimination of intermediate states (the case of s_3), we get the trace FA shown in Fig. 1 (b).

Our second transformation is parameterized by the output of a *binding-time analysis* (BTA), which annotates each predicate with either unfold or memo, where unfold means that the predicate can be *safely* unfolded (i.e., without entering an infinite loop), and memo means that it should be specialized (i.e., a residual predicate is produced). Basically, our second transformation replaces the labels of unfoldable predicates with ϵ. Consider, e.g., that the output of a BTA annotates *mlist*, *ml*, and *mult* as memo and *add* as unfold—this is the case

when the second argument of the initial call to *mlist* is static. We then get the trace FA of Fig. 1 (c). Here, we achieve a significant improvement since, in every iteration for *mlist*, we save the (recursive) evaluation of the calls to *add*.

Clearly, we could eliminate those states whose transitions are all labeled with ϵ. However, we think that keeping the structure of the original trace FA may help the user—and automated analysis tools—to formally compare the original and transformed trace FAs.

4 Discussion

The closest approach to our trace analysis is that of [2], though we offer a different trade-off between analysis cost and accuracy. Basically, they generate trace *terms* abstracting computation trees independently of a computation rule, while we generate sequences of predicate calls for a specific computation rule; also, they do not include a technique for enumerating the (possibly infinite) set of trace terms of a program, while this is a key ingredient of our approach (though one could also apply the transformation from [4] to the CFG associated to the trace terms of [2] to obtain a finite representation). A deeper comparison with the approach of [2] is an interesting topic for further research.

A proof-of-concept implementation of our technique, called PEPE, is publicly available from `http://german.dsic.upv.es/pepe.html`. Our approach can be seen as a first step for the development of automated techniques and tools for predicting the potential speedup of partial evaluation, thus it opens a number of interesting lines for further research.

Acknowledgments. We would like to thank Elvira Albert, Sergio Antoy, Manuel Hermenegildo, Michael Leuschel, Claudio Ochoa, and Germán Puebla for many interesting discussions on the topic of this paper.

References

1. S. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.
2. J.P. Gallagher and L. Lafave. Regular Approximation of Computation Paths in Logic and Functional Languages. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, Selected Papers*, pages 115–136. Springer LNCS 1110, 1996.
3. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
4. M. Mohri and M.-J. Nederhof. *Regular Approximation of Context-Free Grammars through Transformation*, chapter 9, pages 153–163. Kluwer Academic Publishers, 2001.
5. C. Ochoa and G. Puebla. Poly-controlled Partial Evaluation in Practice. In *Proc. of PEPM'07*, pages 164–173. ACM, 2007.
6. G. Vidal. Predicting the Speedup of Partial Evaluation. Technical report, UPV, 2008. Available from `http://www.dsic.upv.es/~gvidal/german/papers.html`.