Predicting the Effectiveness of Partial Evaluation

Germán Vidal

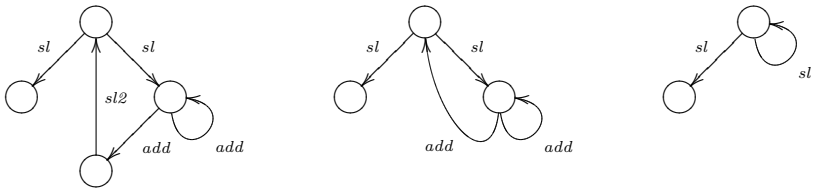
Technical University of Valencia, Spain.
gvidal@dsic.upv.es

Abstract. Partial evaluation aims at improving programs by specializing them w.r.t. part of their input data. In general, however, the effectiveness of the partial evaluation process is hard to measure, even *a posteriori*. Recent approaches have introduced experimental (often computationally expensive) frameworks for this purpose.
In this paper, we present an alternative, *symbolic* approach for predicting the effectiveness of partial evaluation by combining a *trace* analysis with a termination analysis. The termination analysis—namely, a *size-change* analysis—is used to determine which procedures are potentially removable by partial evaluation (i.e., can be fully unfolded at specialization time). Then, the trace analysis helps us to put this information into context by producing a compact representation of the call sequences of the program. By inspecting the output of the combined analysis, the user may determine the impact of a partial evaluation *before* it is performed.

1 Introduction

The main goal of *partial evaluation* [18] 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. In the optimal case, all operations that depend only on the static data are performed once and for all during partial evaluation. 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.

Among the different techniques for program specialization, partial evaluation is likely the one which has achieved a higher level of automation. However, 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., [11, 27]) 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. The main drawback of these approaches, however, is that they are often computationally expensive since a number of testing partial evaluations (though for *simpler* test cases) should be made in order to determine the best division.



(a) (b) (c)

Fig. 1. Graphical representations of call sequences

In this paper, we present an alternative, *symbolic* approach for predicting the potential effects of a partial evaluation (which is, in principle, computationally less expensive). Consider, e.g., the following simple program for adding the elements of a list using an accumulating parameter:

$$\begin{array}{ll} sl([\,],N,N). & add(0,Y,Y). \\ sl([X|R],N,S) \leftarrow add(N,X,M), & add(s(X),Y,s(Z)) \leftarrow add(X,Y,Z). \\ \qquad\qquad sl2(R,M,S). & sl2(R,M,S) \leftarrow sl(R,M,S). \end{array}$$

where natural numbers are built from 0 and $s(\cdot)$.

Let us now analyze what could be the effects of partially evaluating this program. First, given a flat call of the form $sl(A,B,C)$, the SLD search space (using Prolog's leftmost computation rule) has the shape depicted in Fig. 1 (a).

Here, one can observe—just by looking at this representation—that the call to *sl2* is not really needed: it acts simply as an *intermediate* call and could be safely removed by unfolding. The resulting graph is depicted in Fig. 1 (b).

Now, let us assume that the second parameter, the accumulating parameter, is static and thus known at partial evaluation time. Therefore, all calls to *add* have a static first argument and, hence, the number of recursive calls is bounded by the value of this argument. As a consequence, all calls to *add* can be fully unfolded at partial evaluation time, so that we get the graph of Fig. 1 (c). If we compare graphs (a) and (c), it is easy to conclude that graph (c) represents a significantly less expensive computation than graph (a) since every iteration of the loop for *sl* has changed from $1+(n+1)+1$ steps (i.e., a call to *sl*, $n+1$ calls to *add*, where n depends on its first argument, and a call to *sl2*) to simply 1 step. If this loop is traversed many times (e.g., when the first argument of *sl* is a large list), then the improvement achieved would be quite significant.

Although the graphical representation may be more intuitive for the user, other textual representations could be more amenable to automation. For this purpose, we also introduce the use of *regular expressions*. The following table shows the regular expression that represents the call traces of the original pro-

gram (first row labeled with *none*), together with the regular expressions associated to partial evaluations using different divisions:[1]

division	regular expression	
none	`(sl add* add sl2)* sl`	Fig. 1 (a)
$sl(\mathsf{d},\mathsf{d},\mathsf{d})$	`(sl add* add ___)* sl`	Fig. 1 (b)
$sl(\mathsf{s},\mathsf{d},\mathsf{d})$	`(__ add* add ___)* __`	
$sl(\mathsf{d},\mathsf{s},\mathsf{d})$	`(sl ___* ___ ___)* sl`	Fig. 1 (c)
$sl(\mathsf{s},\mathsf{s},\mathsf{d})$	`(__ ___* ___ ___)* __`	

Here, as it is common practice, we use p^* to denote zero or more occurrences of p. Observe the equivalence between the regular expressions of rows 1, 2 and 4 and the graphs depicted in Fig. 1. Intuitively speaking, the meaning of each row is as follows:

- For division $sl(\mathsf{d},\mathsf{d},\mathsf{d})$, only intermediate calls are removed by partial evaluation.
- For division $sl(\mathsf{s},\mathsf{d},\mathsf{d})$, the initial call to *sl* can be fully unfolded, but a number of calls to `add` remains in the partially evaluated program. Actually, this is not a good specialization since we are just moving a constant number of steps from run time to partial evaluation time.
- For division $sl(\mathsf{d},\mathsf{s},\mathsf{d})$, every call to *add* can be fully unfolded but the outer loop for *sl* remains in the partially evaluated program. A significant speedup can be achieved here for large input lists as discussed above.
- Finally, for division $sl(\mathsf{s},\mathsf{s},\mathsf{d})$, the call to `sl` can be fully unfolded. As in the second case above, this is not a good specialization since only a constant number of steps would be saved.

It is the aim of this work to present a symbolic framework to formally discuss the effects achieved by partial evaluation.

Constributions. Our first contribution is a systematic method to approximate the call traces of a given program. Basically, given a logic program, we first construct a context-free grammar that safely approximates the call sequences of the program. For instance, for procedure *add* above we get

$$\begin{array}{lcl} \textsc{add} & \rightarrow & \mathit{add} \\ \textsc{add} & \rightarrow & \mathit{add}\ \textsc{add} \end{array}$$

where ADD is a non-terminal associated to predicate *add* (which is used as a terminal of the grammar). In this case, the grammar is already regular and, thus, a regular expression ($add^*\ add$) can be obtained. When it is not regular, we apply the transformation of [24] to get a regular approximation while keeping the structure of the original grammar as far as possible.

[1] For the moment, divisions are just expressed as atoms with `s` (static) or `d` (dynamic) arguments.

Our second contribution is a method for determining how a given division may affect the program loops. For this purpose, we consider the *size-change* analysis of [31]. The relevance of this analysis (in contrast to other termination analysis) is that it is independent of the computation rule (which may allow much faster partial evaluation, as shown in [20] in the context of the partial evaluator Logen [19]). This is a requirement in our setting since partial evaluation often considers liberal selection policies that depend on the available information (e.g., only calls which are instantiated enough to ensure finite unfolding are unfolded).

Once size-change analysis has identified the (potential) loops of the program, we use the information provided by a division in order to determine which loops can be safely unfolded. As a consequence, the output of the trace analysis (e.g., a regular expression denoting the possible sequences of calls) is modified in order to reflect the elimination of these loops. In this paper, we focus on providing simple and useful information for the user in order to analyze the effects of different partial evaluations. Nonetheless, an automated analysis of the associated regular expressions is also possible, though it is outside the scope of this paper.

The paper is organized as follows. After introducing some preliminaries in the next section, we present our stepwise transformation for approximating the call traces of logic programs in Sect. 3. Then, in Sect. 4, we recall the fundamentals of size-change analysis and use the output of this analysis for estimating the effects of partial evaluation w.r.t. a given division; we also present some details of a prototype implemention as well as a number of experimental results. Finally, Sect. 5 discusses some related work and Sect. 6 concludes and presents several possibilities for future work.

2 The Language

We consider a first-order language with a fixed vocabulary of predicate symbols, function symbols, and variables denoted by Π, $\mathcal{F}$ and $\mathcal{V}$, respectively. We let $\mathcal{T}(\mathcal{F},\mathcal{V})$ denote the set of *terms* constructed using symbols from $\mathcal{F}$ and variables from $\mathcal{V}$. An *atom* has the form $p(t_1,\ldots,t_n)$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$ for $i = 1,\ldots,n$. A *query* is a finite sequence of atoms $\langle A_1,\ldots,A_n\rangle$, where the *empty query* is denoted by *true*. A *clause* has the form $H \leftarrow B_1,\ldots,B_n$ where $H, B_1,\ldots,B_n$, $n \geqslant 0$, are atoms (i.e., we only consider *definite* programs). A logic *program* is a finite sequence of clauses. $\mathcal{V}ar(s)$ denotes the set of variables in the syntactic object s (i.e., s can be either a term, an atom, a query, or a clause). A syntactic object s is *ground* if $\mathcal{V}ar(s) = \varnothing$.

Substitutions and their operations are defined as usual. In particular, the set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of a substitution σ. A syntactic object s_1 is *more general* than a syntactic object s_2, denoted $s_1 \leqslant s_2$, if there exists a substitution θ such that $s_2 = s_1\theta$. The *most general unifier* of two syntactic objects, s_1 and s_2, denoted by $mgu(s_1,s_2)$, is a unifier of s_1 and s_2 which is more general than any other unifier of s_1 and s_2.

Computations in logic programming are formalized by means of SLD resolution. The notion of *computation rule* $\mathcal{R}$ is used to select an atom within

a query for its evaluation. Given a program P, a query $Q = \langle A_1, \ldots, A_n \rangle$, and a computation rule $\mathcal{R}$, we say that $Q \leadsto_{P,\mathcal{R},\sigma} Q'$ is an *SLD resolution step* for Q with P and $\mathcal{R}$ if $\mathcal{R}(Q) = A_i$, $1 \leqslant i \leqslant n$, is the selected atom, $H \leftarrow B_1, \ldots, B_m$ is a renamed apart clause of P, $\sigma = mgu(A, H)$, and $Q' = (\langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n \rangle)\sigma$; we often omit P, $\mathcal{R}$ and/or σ in the notation of an SLD resolution step when they are clear from the context. An *SLD derivation* is a (finite or infinite) sequence of SLD resolution steps. We often use $Q_0 \leadsto_\theta^* Q_n$ as a shorthand for $Q_0 \leadsto_{\theta_1} Q_1 \leadsto_{\theta_2} \ldots \leadsto_{\theta_n} Q_n$ with $\theta = \theta_1 \circ \cdots \circ \theta_n$ (where $\theta = \{\}$ if $n = 0$). An SLD derivation $Q \leadsto_\theta^* Q'$ is *successful* when $Q' = true$; in this case, we say that θ is the *computed answer substitution*. SLD derivations are represented by a (possibly infinite) finitely branching tree.

3 Trace Analysis for Logic Programs

In this section, we aim at capturing the *shape* of a computation by producing a finite representation of all possible sequences of predicate calls.

For this purpose, we introduce a stepwise method that starts with a context-free grammar (CFG) that approximates the call sequences of a logic program (LP), which is then approximated by a strongly regular grammar (SRG), if needed, and transformed into a finite automaton (FA) whose accepted language can be represented by means of a regular expression (ER). Graphically:

$$\boxed{\text{LP} \;\Rightarrow\; \text{CFG} \;\Rightarrow\; \text{SRG} \;\Rightarrow\; \text{FA} \;\Rightarrow\; \text{RE}}$$

The next sections formalize this process.

3.1 From Logic Programs to Context-Free Grammars

Let us first formalize our notion of *call trace*. For the sake of clarity, we assume in the following that programs do not contain occurrences of the same predicate name with different arities.[2]

Furthermore, we consider a fixed computation rule for call traces, namely Prolog's leftmost computation rule, which we denote by $\mathcal{R}_{left}$.[3] In what follows, we often 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 $pred(\mathcal{R}_{left}(Q_i)) = p_i$, $i \geq 0$, where $pred(A)$ returns the predicate symbol of atom A.

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 $Q_0 \overset{p_0}{\leadsto} Q_1 \overset{p_1}{\leadsto} \ldots \overset{p_{n-1}}{\leadsto} Q_n$.*

[2] This is not a real restriction and, indeed, it is not required in the implemented tool (where predicate names are simply suffixed with their arity).

[3] Note that we assume $\mathcal{R}_{left}$ only at run time, but allow arbitrary computation rules at partial evaluation time.

The first step of our trace analysis consists in producing a *context-free grammar* (CFG) associated to the considered program. A CFG G is a tuple $G = \langle \Sigma, N, R, S \rangle$, where Σ and N are two finite disjoint set of *terminals* and *non-terminals*, respectively, $S \in N$ is the *start* symbol, and R is a finite set of rules. Each rule has the form $A \to \alpha$ with $A \in N$ and $\alpha \in V^*$, where V denotes $\Sigma \cup N$. The relation $\to$ on $N \times V^*$ is extended to a relation on $V^* \times V^*$ in the usual way. The transitive and reflexive closure of $\to$ is denoted by $\to^*$. The *context-free* language generated by G is given by $L(G) = \{w \mid \Sigma^* \mid S \to^* w\}$. See, e.g., [17] for more details on CFGs.

In the following, given a predicate symbol $p \in \Pi$, we denote by $\textsc{p} \notin \Pi$ a fresh symbol representing the non-terminal associated to p. Furthermore, we denote by $\textsc{pred}(A)$ the non-terminal associated to the predicate symbol of atom A, i.e., $\textsc{pred}(A) = \textsc{p}$ if $A = p(t_1, \ldots, t_n)$. Also, we let $\overline{\Pi}$ denote the set $\{\textsc{p} \mid p \in \Pi\}$ of non-terminals associated to predicate symbols. In contrast, we directly use predicate symbols from Π as terminals.

We let $\textsc{start}$ be a fresh symbol not in $\Pi \cup \overline{\Pi}$ which we use as a generic start symbol for CFGs. Given a program and a predicate symbol, we construct an associated CFG, called *trace CFG*, as follows:

Definition 2 (trace CFG, $\textsc{cfg}_q^P$). *Let P be a program and $q \in \Pi$ a predicate symbol with associated non-terminal* $\textsc{q}$*. The associated trace CFG is $\textsc{cfg}_q^P = \langle \Pi, \overline{\Pi} \cup \{\textsc{start}\}, R, \textsc{start} \rangle$, where the set of rules R is defined as follows:*

$$\{\textsc{start} \to \textsc{q}\}$$
$$\cup$$
$$\{\textsc{pred}(A_0) \to \mathit{pred}(A_0)\textsc{pred}(B_1) \ldots \textsc{pred}(B_n) \mid A_0 \leftarrow B_1, \ldots, B_n \in P,\ n \geq 0\}$$

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.

Clearly, the trace CFG may produce call traces that are not possible in the associated logic program because the "propositional" approximation that forms the basis of trace CFGs clearly involves a loss of accuracy (consider, e.g., that not all atoms with the same predicate symbol unify).

As a counterpart, the generation of the trace CFG can be done very efficiently since only a single pass over the associated logic program is required.

Example 1. Consider the following program P which defines a procedure to increase all elements of a list by a given value:

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

where natural numbers are built from 0 and $s(\cdot)$. The associated trace CFG, $\text{CFG}^P_{incList}$, is given by

$$\langle \{incList, iList, nat, add\}, \{\text{START}, \text{INCLIST}, \text{ILIST}, \text{NAT}, \text{ADD}\}, R, \text{START}\rangle$$

where the set of rules R is as follows:

$$\begin{array}{rl@{\qquad}rl}
\text{START} & \to \text{INCLIST} & \text{NAT} & \to nat \\
\text{INCLIST} & \to incList & \text{NAT} & \to nat\ \text{NAT} \\
\text{INCLIST} & \to incList\ \text{ILIST} & \text{ADD} & \to add \\
\text{ILIST} & \to iList\ \text{NAT ADD INCLIST} & \text{ADD} & \to add\ \text{ADD}
\end{array}$$

Our next result states that CFG^P_q is indeed a correct approximation of the call traces for P w.r.t. the leftmost computation rule $\mathcal{R}_{left}$.

Theorem 1. *Let P be a program and $Q_0 = \langle A_1, A_2, \ldots, A_m\rangle$ a query with $\text{PRED}(A_i) = \text{A}_i$ for all $i = 1, \ldots, m$. Let CFG^P_q be the trace CFG associated to P for some predicate symbol $q \in \Pi$. If τ is a call trace for Q_0 with P, then there exists a sequence $\text{A}_1 \ldots \text{A}_m \to^* \tau$ in CFG^P_q.*

Proof. We prove the claim by induction on the length n of the SLD derivation that produced the call trace τ.

Base case $n = 1$. Then, $Q_0 = \langle A_1\rangle$ is an atomic query and, moreover, it is solved using a fact of P, i.e., $Q_0 = \langle A_1\rangle \overset{pred(A_i)}{\rightsquigarrow} Q_1 = true$ using some clause $H \leftarrow$ of P. By definition of CFG^P_q, there is a rule $\text{PRED}(A_1) \to pred(A_1)$ in CFG^P_q (no matter the value of q). Therefore, the sequence $\text{PRED}(A_1) \to pred(A_1)$ can be done in CFG^P_q and the claim follows.

Inductive case $n > 1$. Then, there exists a successful SLD derivation $Q_0 \overset{q_0}{\rightsquigarrow} Q_1 \overset{q_1}{\rightsquigarrow} \ldots \overset{q_{n-1}}{\rightsquigarrow} Q_n$ with $\tau = q_0\, q_1 \ldots q_{n-1}$ and $q_0 = pred(A_1) = a_1$. Assume that the first SLD resolution step is performed with clause $H \leftarrow B_1, \ldots, B_k$, $k \geq 0$, so that $Q_1 = \langle B_1\sigma_1, \ldots, B_k\sigma_1, A_2\sigma_2, \ldots, A_n\sigma_1\rangle$ and $\sigma_1 = mgu(H, A_1)$. Therefore, there exists a rule of the form $\text{PRED}(H) \to pred(H)\text{B}_1 \ldots \text{B}_k$ in CFG^P_q, with $\text{PRED}(B_i) = \text{B}_i$ for all $i = 1, \ldots, k$. Since $mgu(H, A_1) \neq fail$, we have both $pred(H) = pred(A_1) = a_1$ and $\text{PRED}(H) = \text{PRED}(A_1) = \text{A}_1$. Hence the sequence $\text{A}_1\ \text{A}_2 \ldots \text{A}_n \to a_1\ \text{B}_1 \ldots \text{B}_k\ \text{A}_2 \ldots \text{A}_n$ can be performed in CFG^P_q using

rule $\text{A}_1 \rightarrow a_1\ \text{B}_1 \ldots \text{B}_k$. Now, consider the SLD derivation $Q_1 \rightsquigarrow \ldots \rightsquigarrow Q_n$ with associated call trace τ'. By the inductive hypothesis, we have that the sequence $\text{PRED}(B_1\sigma_1) \ldots \text{PRED}(B_k\sigma_1)\text{PRED}(A_2\sigma_2) \ldots \text{PRED}(A_n\sigma_1) \rightarrow^* \tau'$ can be performed with the rules of CFG_q^P. Finally, since $pred(A) = pred(A\sigma)$ and $\text{PRED}(A) = \text{PRED}(A\sigma)$ for all atom A and substitution σ, the claim follows from $\text{A}_1\ \text{A}_2 \ldots \text{A}_n \rightarrow a_1\ \text{B}_1 \ldots \text{B}_k\ \text{A}_2 \ldots \text{A}_n$ and $\text{B}_1 \ldots \text{B}_k\ \text{A}_2 \ldots \text{A}_n \rightarrow^* \tau'$.

The next corollary is an straightforward consequence of Theorem 1:

Corollary 1. *Let P be a program and $Q_0 = \langle q(t_1, \ldots, t_n)\rangle$ an atomic query. Let Ω be the (possibly infinite) set of call traces for Q_0 with P. Then $\Omega \subseteq L(\text{CFG}_q^P)$.*

3.2 Approximating Trace CFGs

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 languages are not regular). In this section, we use the transformation from [24] to approximate a trace CFG with a *strongly regular* grammar (SRG).[4] The relevance of SRGs is that they can be mapped to equivalent finite-state automata using an efficient algorithm. Moreover, the transformation of [24] guarantees that the result remains readable and mainly preserves the structure of the original CFG.

Given a CFG, the first step of the transformation consists in computing the sets of *mutually recursive non-terminals*. This can be done in linear time in the size of the CFG by, e.g., computing the strongly connected components of the graph of the grammar.[5]

A grammar is called *left-linear* if every rule has either the form

$$A \rightarrow t \qquad \text{or} \qquad A \rightarrow t\ B$$

where t is a finite sequence of terminals and A, B are non-terminals. The following definition is slightly adapted from [24] to the case of trace CFGs:

Definition 3 (trace SRG, SRG_q^P). *Let P be a program, $q \in \Pi$ a predicate symbol, and $\text{CFG}_q^P = \langle \Pi, \overline{\Pi} \cup \text{START}, R, \text{START}\rangle$ the trace CFG for P and q. The associated trace SRG, in symbols SRG_q^P, is obtained from CFG_q^P as follows:*

- *First, we compute the set of mutually recursive non-terminals of CFG_q^P.*
- *Then, for each set M of mutually recursive non-terminals such that the rules defining these non-terminals are not all left-linear* w.r.t. the non-terminals of M,[6] *we apply a grammar transformation as follows:*

[4] SRGs coincide with the class of grammars without self-embedding [8].

[5] The graph of a grammar contains one node for each non-terminal and an edge from node A to node B if non-terminal B appears in the right-hand side of a rule with left-hand side A.

[6] This condition relaxes the standard notion of left-linear grammar by considering non-terminals from $(\overline{\Pi} \setminus M)$ as terminals.

1. *For each non-terminal $A \in M$, we introduce a fresh non-terminal A' and add the following rule to the grammar:*[7]

$$A' \to \epsilon$$

2. *For each non-terminal $A \in M$ and each rule*

$$A \to t_0\ B_1\ t_1\ B_2\ t_2 \ldots B_m\ t_m$$

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:

$$\begin{aligned} A &\to t_0\ B_1 \\ B'_1 &\to t_1\ B_1 \\ B'_2 &\to t_2\ B_3 \\ &\ldots \\ B'_{m-1} &\to t_{m_1}\ B_m \\ B'_m &\to t_m\ A' \end{aligned}$$

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

– *Finally, we let $\text{SRG}_q^P = \langle \Pi, \overline{\Pi} \cup N \cup \text{START}, R', \text{START} \rangle$, where R' is the set of rules that results from R by applying the process above and N are the fresh non-terminals added during this process.*

According to [24], the transformed grammar SRG_q^P is strongly regular and can be compiled into a finite automaton in linear time by existing algorithms. Furthermore, the language generated by the transformed grammar is a superset of that of the original grammar [24], i.e., $L(\text{SRG}_q^P) \supseteq L(\text{CFG}_q^P)$ for all program P and predicate symbol q, which means (by Corollary 1) that SRG_q^P is a complete approximation of all call traces for P.

Example 2. Consider the CFG_q^P of Example 1. The sets of mutually recursive non-terminals are

$$\{\{\text{INCLIST}, \text{ILIST}\}, \{\text{NAT}\}, \{\text{ADD}\}\}$$

Since the rules for NAT and ADD are left-linear, we focus on the set $M = \{\text{INCLIST}, \text{ILIST}\}$. Here, the only potentially non-linear rule is

$$\text{ILIST} \to \textit{iList}\ \text{NAT}\ \text{ADD}\ \text{INCLIST}$$

However, since $\text{NAT}, \text{ADD} \notin M$, this rule is considered left-linear w.r.t. M and needs not be replaced. Therefore, in this case, we have $\text{CFG}_{incList}^P = \text{SRG}_{incList}^P$.

Example 3. Consider the following program P defining multiplication and addition on natural numbers:

$$\begin{aligned} &\textit{mult}(0, Y, 0). \\ &\textit{mult}(s(X), Y, Z) \leftarrow \textit{mult}(X, Y, Z1),\ \textit{add}(Z1, Y, Z). \\ &\textit{add}(0, Y, Y). \\ &\textit{add}(s(X), Y, s(Z)) \leftarrow \textit{add}(X, Y, Z). \end{aligned}$$

[7] We denote by ϵ the empty sequence.

The trace grammar CFG^P_{mult} contains the following rules:

$$\begin{array}{lll} \text{START} \rightarrow \text{MULT} & \text{MULT} \rightarrow \mathit{mult} & \text{ADD} \rightarrow \mathit{add} \\ & \text{MULT} \rightarrow \mathit{mult}\ \text{MULT}\ \text{ADD} & \text{ADD} \rightarrow \mathit{add}\ \text{ADD} \end{array}$$

The sets of mutually recursive non-terminals are $\{\{\text{MULT}\}, \{\text{ADD}\}\}$. While the rules for ADD are clearly left-linear, the second rule of MULT is not left-linear because, even if ADD is treated as a terminal, it appears *to the right* of the non-terminal MULT. Here, SRG^P_{mult} will contain the following set of rules:

$$\begin{array}{lll} \text{START} \rightarrow \text{MULT} & \text{MULT} \rightarrow \mathit{mult}\ \text{MULT}' & \text{ADD} \rightarrow \mathit{add} \\ \text{MULT}' \rightarrow \epsilon & \text{MULT} \rightarrow \mathit{mult}\ \text{MULT} & \text{ADD} \rightarrow \mathit{add}\ \text{ADD} \\ & \text{MULT}' \rightarrow \text{ADD}\ \text{MULT}' & \end{array}$$

3.3 A Compact Representation for Call Traces

Once we have a 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 as well as a *regular expression* (RE) that represents this language.

Trace Finite Automata. A finite-state automaton (FA) is specified by a tuple $\langle Q, \Sigma, \delta, s_0, F\rangle$, where Q is a finite set of states, Σ is an input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a (finite) set of transitions, $s_0 \in Q$ is the start state and $F \subseteq Q$ is a set of final states. For constructing a finite automaton $FA(G)$ from a SRG G, we follow the classical approach from [1] (though more refined methods exist, see, e.g., [26, 25]). Intuitively speaking, we proceed as follows:

- There is a start state in the FA associated to the start symbol of the SRG.
- Then, for each reduction $w \rightarrow w'$ using a rule $A \rightarrow t\,B$ of the SRG, we have a transition (s, α, s') in the FA. States s, s' are associated with the sequence of non-terminals in w, w' (so that if the same sequence occurs more than once, the same state is used). The character α is set to the sequence t in the applied rule (see [1] for a detailed description).

The next example illustrates the construction of a FA from a SRG:

Example 4. Consider the CFG $\text{CFG}^P_{incList}$ of Example 1 (which, as shown in Example 2, is already a SRG). The associated FA is

$$FA(\text{CFG}^P_{incList}) = \langle Q, \{\mathit{incList}, \mathit{iList}, \mathit{nat}, \mathit{add}, \epsilon\}, \delta, s_0, \{s_2\}\rangle$$

where

$$\begin{array}{rl} Q = & \{\ s_0, s_1, s_2, s_3, s_4, s_5\} \\ \delta\ = & \{\ (s_0, \epsilon, s_1), (s_1, \mathit{incList}, s_2), (s_1, \mathit{incList}, s_3), (s_3, \mathit{iList}, s_4), \\ & \quad (s_4, \mathit{nat}, s_5), (s_4, \mathit{nat}, s_4), (s_5, \mathit{add}, s_1), (s_5, \mathit{add}, s_5) \qquad \} \end{array}$$

The FA is graphically shown in the leftmost, topmost corner of Fig. 2, where the final state s_2 is denoted with a double circle.

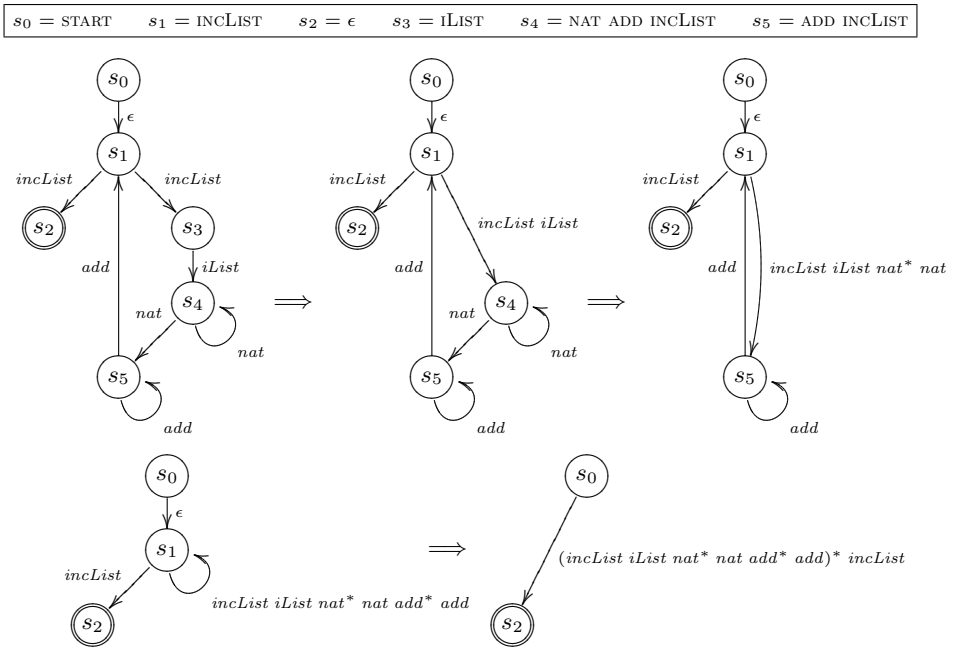


Fig. 2. Graphical representation of $FA(\text{CFG}^{P}_{incList})$ and state elimination sequence

Trace Regular Expressions Although not so well studied as the construction of a FA from a regular expression (RE), there exists several methods for computing a RE that denotes the language accepted by a FA. In this work, we consider a method called *state elimination* [7, 32]: given a FA, we keep removing states—except for the start and final states—while preserving the transition information until there are no more states to eliminate. Once only transitions from the initial to the final states remain, say $r_1, \ldots, r_n$, the RE $(r_1 + \ldots + r_n)$ will denote the language accepted by the FA. Here, as it is common practice, we use p^* to denote zero or more occurrences of p and $(p+q)$ to denote a choice between p and q. Note that during the state elimination process we label transitions with REs rather than characters (formally, they are *expression* automata [7, 15]).

Basically, given a FA $A = \langle Q, \Sigma, \delta, s_0, F \rangle$, the state elimination of state $s \in Q \setminus \{\{s_0\} \cup F\}$ proceeds as follows:

- for each pair of transitions (s_i, α_i, s) and (s, α_j, s_j) in δ, we produce a new transition $(s_i, \alpha_i\, \alpha_j, s_j)$;
- if there exists a self-looping transition (s, α, s) in δ, then the new transition is $(s_i, \alpha_i\, \alpha^*\, \alpha_j, s_j)$ instead;
- if there is already a transition (s_i, α_{ij}, s_j) in δ, then both the old and the new transitions are merged to $(s_i, (\alpha_i\, \alpha^*\, \alpha_j) \;+\; \alpha_{ij}, s_j)$;
- finally, we remove the state s and all transitions $(\_, \_, s)$ and $(s, \_, \_)$ from δ.

Different REs can be obtained depending on the order in which states are eliminated. Clearly, the choice of the next state to be removed is crucial to obtain simpler REs (see Sect. 4.3 for a particular heuristics).

Example 5. Consider the FA of Example 4, i.e., $FA(\text{CFG}^{P}_{incList})$. The sequence of state eliminations, using the heuristics of Sect. 4.3, is shown in Fig. 2. Therefore, the associated regular expression is $(incList\ iList\ nat^*\ nat\ add^*\ add)^*\ incList$.

To summarize, the *correctness* of our trace analysis, i.e., the fact that the set of call sequences in a program belong to the regular language accepted by the generated FA or represented by the associated RE, is a straightforward consequence of Corollary 1 and results from [24] (for approximating CFGs with SRGs) and [1, 7, 32] (for constructing FAs and REs associated to a SRG).

As for its computational cost, all steps involved in the process are linear in the size of the source program and, thus, reasonable run times can be expected.

4 Predicting the Effectiveness of Partial Evaluation

The output of the trace analysis gives us the *context* where every predicate call appears. In this section, we determine (with the help of a termination analysis, namely a size-change analysis) which predicate calls could be deleted by partial evaluation from the computed traces. By analyzing the traces before/after partial evaluation, one can extract useful conclusions on its effectiveness.

4.1 Size-Change Analysis

In this section, we present an informal account of the size-change analysis for logic programs introduced in [31].

In contrast to other termination analysis for logic programs (e.g., [9, 21]), the size-change analysis of [31] considers *strong* termination [5], i.e., termination of all SLD derivations *w.r.t. any computation rule*. Although this is a strong requirement, it is quite useful in the context of partial evaluation since it allows us to use rather liberal selection policies at specialization time without recomputing the termination analysis every time a body atom is marked as "non-unfoldable" (a detailed discussion on this topic can be found in [20]).

Size-change analysis proceeds in two steps: first, size-change graphs are built with information on how the size of predicate arguments changes from one call to another; then, a sort of transitive closure is computed in order to identify the (potential) program loops. Size-change graphs are parametric w.r.t. a *reduction pair* $(\succsim, \succ)$ which is induced from a symbolic norm $||\cdot||$:

Definition 4 (symbolic norm [21]). *Given a term t,*

$$||t|| = \begin{cases} m + \sum_{i=1}^{n} k_i ||t_i|| & \text{if } t = f(t_1, \ldots, t_n),\ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

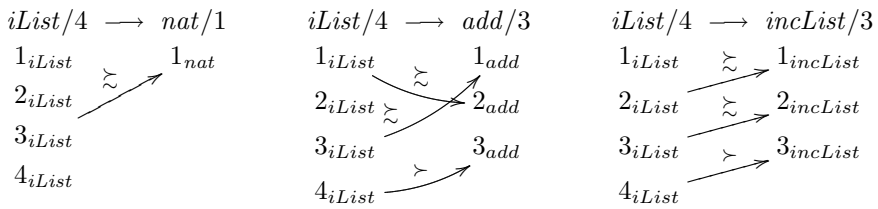
where m and $k_1, \ldots, k_n$ are non-negative integer constants depending only on f/n. Note that we associate a variable over integers to each logical variable (we use the same name for both since the meaning is clear from the context).

Then, we have $t_1 \succ t_2$ (resp. $t_1 \succsim t_2$) if $\forall ||t_1|| > ||t_2||$ (resp. if $\forall ||t_1|| \geqslant ||t_2||$). Here, the use of variables in the range of symbolic norms provides a simple mechanism to express dependencies between the sizes of terms. Two popular instances of symbolic norms are the symbolic *term-size* norm $||\cdot||_{ts}$ (which counts the arities of the term symbols) and the symbolic *list-length norm* $||\cdot||_{ll}$ (which counts the number of elements of a list), e.g.,

$$\begin{array}{rcl} f(X,Y) \succ f(X,a) & \text{since} & ||f(X,Y)||_{ts} = X + Y + 2 > X + 2 = ||f(X,a)||_{ts} \\ [X|R] \succsim [a|R] & \text{since} & ||[X|R]||_{ll} = R + 1 \geqslant R + 1 = ||[a|R]||_{ll} \end{array}$$

For every pair of atoms (H, B_i) associated to a clause $H \leftarrow B_1, \ldots, B_n$ with $n > 0$ (i.e., there are no size-change graphs associated with facts), we construct a size-change graph with edges between the arguments of H and B_i when the size of the corresponding terms *decrease* w.r.t. a given reduction pair $(\succsim, \succ)$.

Example 6. Consider the program of Example 1. In this case, the size-change graphs associated to, e.g., clause c_3 are as follows:



using a reduction pair induced from the symbolic term-size norm.

Now, in order to identify the program *loops*, we should compute roughly a transitive closure of the size-change graphs by composing them in all possible forms. Basically, given two size-change graphs:

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1) \quad \mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

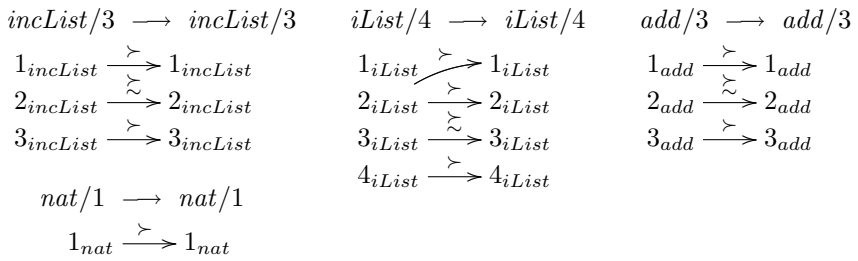
w.r.t. the same reduction pair $(\succsim, \succ)$, their concatenation is defined by

$$\mathcal{G} \bullet \mathcal{H} = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E)$$

where E contains an edge from i_p to k_r iff E_1 contains an edge from i_p to some j_q and E_2 contains an edge from j_q to k_r. Furthermore, if some of the edges are labeled with $\succ$, then so is the edge in E; otherwise, it is labeled with $\succsim$.

Among all computed concatenations of size-change graphs, we only need to consider the *idempotent* graphs (i.e., those graphs $\mathcal{G}$ with $\mathcal{G} \bullet \mathcal{G} = \mathcal{G}$), because they represent the (potential) program loops.

Example 7. For the program of Example 1, we compute the following idempotent size-change graphs:



These graphs represent how the sizes of the arguments of the three potentially looping predicates decrease from one iteration to the next.

In the following, we denote by $calls_P^{\mathcal{R}}(Q_0)$ the set of calls in the computations of a goal Q_0 with program P and a computation rule $\mathcal{R}$. Also, we say that a query Q is *strongly terminating* w.r.t. a program P if every SLD derivation for Q with P and $\mathcal{R}$ is finite for any computation rule $\mathcal{R}$.

Once the idempotent size-change graphs of a program are computed, the following result characterizes its strong termination. Basically, we require the strictly decreasing parameters of (potentially) looping predicates to be *instantiated enough*[8] w.r.t. a given symbolic norm in the considered computations:

Theorem 2 (strong termination [31]**).** *Let P be a program and $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $||\cdot||$. Let $\mathcal{A}$ be a set of atoms. If every idempotent size-change graph for P contains at least one edge $i_p \xrightarrow{\succ} i_p$ such that, for every atom $A \in \mathcal{A}$, computation rule $\mathcal{R}$, and atom $p(t_1, \ldots, t_n) \in calls_P^{\mathcal{R}}(A)$, t_i is instantiated enough w.r.t. $||\cdot||$, then P is strongly terminating w.r.t. $\mathcal{A}$.*

Note that t_i should be instantiated enough in every possible derivation for the considered set of atoms w.r.t. any computation rule. Obviously, this is an undecidable condition because the set $calls_P^{\mathcal{R}}(A)$ is generally infinite. Luckily, in the context of partial evaluation this condition can be simply approximated by using the information provided by a standard binding-time analysis (which is already available in many partial evaluators).

Example 8. Consider the program of Example 1 and the idempotent size-change graphs of Example 7. Here, Theorem 2 guarantees the termination of SLD resolution (with any computation rule) for those computations in which the following parameters are instantiated enough w.r.t. the symbolic term-size norm $||\cdot||_{ts}$:

- either the first or the third parameter of *incList*,
- either the second or the fourth parameter of *iList*,
- the first parameter of *nat*, and
- either the first or the third parameter of *add*.

[8] A term t is instantiated enough w.r.t. a symbolic norm $||\cdot||$ if $||t||$ is an integer constant [21]. A closely related notion is that of *rigidity* [6], where a term t is rigid w.r.t. a norm $||\cdot||$ if, for any substitution σ, $||t\sigma|| = ||t||$.

4.2 Transforming Call Traces

In this section, we consider that the call traces of a program P are safely approximated by a finite automaton FA_P constructed as in Sect. 3 (extending the forthcoming transformation to regular expressions is straightforward). We also consider that the output of a binding-time analysis (BTA) is available. This is not a limitation since current *offline* partial evaluators for logic programs include a BTA (e.g., Logen [19]).

Here, for simplicity, we consider that the BTA takes a program and an *abstract atom*, i.e., an atom of the form $p(b_1, \ldots, b_n)$ with $p/n \in \Pi$ and $b_i \in \{\mathsf{s}, \mathsf{d}\}$ for $i = 1, \ldots, n$, and returns a *division* that classifies every program parameter as either static or dynamic.[9] We denote a division by a set of abstract atoms; furthermore, we consider that divisions contain one (and only one) abstract atom for each predicate (i.e., we consider a *monovariant* BTA).

Our first transformation deals with *intermediate* predicates, i.e., non-recursive predicates that can be safely unfolded at specialization time. This transformation is related to the *transition compression* of traditional partial evaluation [18] and is independent of the computed division.

In what follows, given a state s, each transition $(s, \_, s')$, $s \neq s'$, is called an *out-transition* of s, each transition $(s', \_, s)$, $s \neq s'$, is called an *in-transition* of s, and each transition $(s, \_, s)$ is called a *self-looping transition* of s.

Definition 5 (elimination of intermediate states).
Let $FA_P = \langle Q, \Sigma, \delta, s_0, F\rangle$ be the trace FA associated to program P. Let $s \in Q \setminus \{\{s_0\} \cup F\}$ be a state. We say that $s \in Q$ is an intermediate state if δ contains exactly one in-transition (s', q', s), one out transition (s, q'', s''), and no self-looping transition for s. In this case, FA_P can be transformed into

$$FA'_P = \langle Q, \Sigma, \delta \setminus \{(s, q'', s'')\} \cup \{(s, \epsilon, s'')\}, s_0, F\rangle$$

The transformation is applied iteratively as long as FA'_P differs from FA_P.

As an example, one can consider the FA shown in the leftmost, topmost corner of Fig. 2. Here, state s_3 is an intermediate state and can thus be eliminated. Observe that, in contrast to the state elimination of Sect. 3.3 (which returns the second FA in Fig. 2), we do keep the "eliminated" state and just replace the terminal symbol in the out-transition with ϵ (see Fig. 3 (a)). This will simplify the comparison between the original and the transformed FAs.

The rationale for our transformation is as follows: the labels of the out-transitions for intermediate states correspond to predicates that are called from a single program point. Therefore, we can safely unfold these calls during specialization and, hence, they will not appear in the partially evaluated program.

Our second, and most important, transformation deals with the output of the size-change analysis and is parameterized by a given division. Roughly speaking,

[9] In practice, we would get more accurate results by considering a BTA that computes binding *types* as in [12], which suffices for checking the "instantiated enough" condition of Theorem 2.

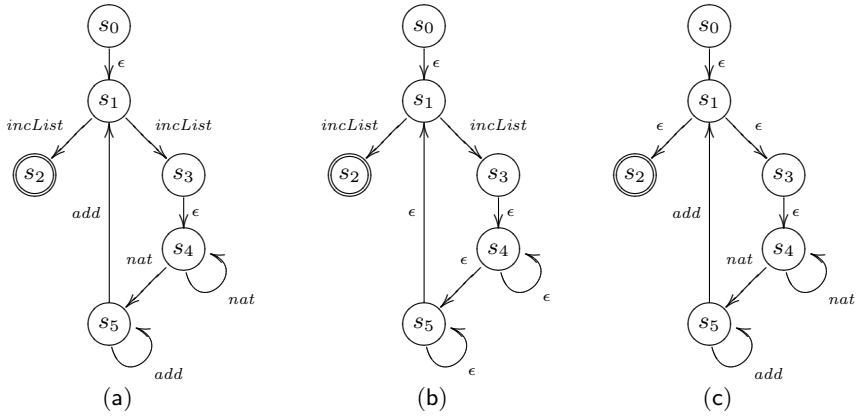


Fig. 3. Transformation of trace FAs

the information in the division is used to determine which predicate arguments are statically controlled (i.e., they fulfill the termination condition of Theorem 2).

Definition 6 (elimination of static loops).
Let $FA_P = \langle Q, \Sigma, \delta, s_0, F\rangle$ be the trace FA associated to program P. Let $\mathcal{G}$ be the set of idempotent size-change graphs of P and μ a division.

Then, we say that a predicate $q \in \Sigma$ is statically controlled if, for each idempotent size-change graph $G \in \mathcal{G}$ for q, there exists at least one edge $i_q \xrightarrow{\succ} i_q$ such that $q(b_1, \ldots, b_i, \ldots, b_n) \in \mu$ and $b_i = \mathsf{s}$.[10]

Now, for each statically controlled predicate q, we transform FA_P into $FA'_P = \langle Q, \Sigma \setminus \{q\}, \delta', s_0, F\rangle$, where δ' is obtained from δ by replacing each transition (s, q, s') with (s, ϵ, s').

The rationale for this transformation is as follows: if the condition of Theorem 2 hold for a given predicate, all calls to this predicate can be finitely unfolded with the available information (denoted by a division) and, thus, one can expect that any reasonable partial evaluator will remove it during specialization.

Example 9. Consider the program of Example 1 and the idempotent size-change graphs shown in Example 7. The original trace FA for this program is shown in the leftmost, topmost corner of Fig. 2. After the elimination of intermediate states (the case of s_3), we get the trace FA shown in Fig. 3 (a).

Now, if we consider the following division:

$$\mu_1 = \{incList(\mathsf{d}, \mathsf{s}, \mathsf{d}),\ iList(\mathsf{d}, \mathsf{d}, \mathsf{s}, \mathsf{d}),\ nat(\mathsf{s}),\ add(\mathsf{s}, \mathsf{d}, \mathsf{d})\}$$

[10] With a more accurate BTA, this condition could be relaxed to require a parameter which is instantiated enough w.r.t. the symbolic norm of the size-change analysis.

then *nat* and *add* become statically controlled and, hence, we get the transformed trace FA of Fig. 3 (b). Finally, if we consider the following division:

$$\mu_2 = \{incList(\mathsf{s},\mathsf{d},\mathsf{d}),\ iList(\mathsf{s},\mathsf{s},\mathsf{d},\mathsf{d}),\ nat(\mathsf{d}),\ add(\mathsf{d},\mathsf{s},\mathsf{d})\}$$

then *incList* and *iList* are now statically controlled and hence we get the transformed trace FA of Fig. 3 (c).

Clearly, we could eliminate those states whose transitions are all labeled with ϵ similarly to the standard state elimination process of Sect. 3.3. 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.

For instance, if we look at the trace FA of Fig. 3 (a), we can conclude that even if no static information is provided, a significant optimization can still be achieved by partial evaluation: in every iteration for *incList* an unfolding is saved (the call to *iList*).

Consider now the trace FA of Fig. 3 (b). Here, we achieve even a more significant improvement since, in every iteration for *incList*, we save not only the unfolding of *iList* but also the complete evaluation of the recursive calls to *nat* and *add*.

Finally, consider the trace FA of Fig. 3 (c). Here, we could expect a similar run time for the specialized program as in the case of Fig. 3 (a) since the elimination of the outer loop (predicates *incList* and *iList*) will only imply saving a constant number of steps (that are moved from run time to partial evaluation time).

4.3 The Approach in Practice

A prototype tool, called Pepe, for estimating the speedup of partial evaluation has been developed. It is implemented in Prolog and includes the trace analysis of Sect. 3, the size-change analysis of Sect. 4.1, and a simple (monovariant) BTA.

Given a program and an abstract atom, the tool returns two regular expressions that represent the call traces of the original and partially evaluated programs. The tool is publicly available through a simple web interface from `http://german.dsic.upv.es/pepe.html`. This is mainly a proof-of-concept implementation and much work can still be done to improve it, e.g., depicting graphically the finite automata rather than their associated textual regular expressions, allowing the user to focus on how a given part of the program would change by the partial evaluation, adding automated tools for determining the best division (or the best one from a number of alternatives), etc.

Actually, a challenge of the current implementation is showing the shortest possible regular expression. A drawback of the state elimination method is that different sequences of state removals may give rise to different regular expressions for the same language. There exists in the literature several approaches that allow one to produce shorter regular expressions, e.g., [13, 16]. In particular, we have implemented a slight variant of the technique in [13], which proposes a heuristics based on assigning *weights* to the states of the FA and then choosing the state

Table 1. Experimental results

Benchmark	Trace Regular Expressions (original/specialized)
applast(d, s, d) run time speedup: 2.5	applast app* app (last last_)* last _______ app* app (last _____)* last
incList(s, d, d) run time speedup: 0.98	(incList iList nat* nat add* add)* incList (_______ _____ nat* nat add* add)* _______
incList(d, s, d) run time speedup: 5.25	(incList iList nat* nat add* add)* incList (incList _____ ___* ___ ___* ___)* incList
match(s, d) run time speedup: 6.72	match (loop eq)* loop + match (loop eq + loop neq next)* loop _____ (loop __)* loop + _____ (loop __ + loop ___ next)* loop
power(d, s, d) run time speedup: 1.21	power* power (mult* mult (add* add)*)* _____* _____ (mult* mult (add* add)*)*

with the lightest weight. Given a transition (s, α, s'), its weight is the number of characters in α. Then, the weight of a state is obtained as the sum of the weights of its in-transitions, its out-transitions, and its self-looping transitions.

Consider, e.g., the FA in the leftmost, topmost corner of Fig. 2. Then, the weight of the states—which are not a start or a final state—is as follows:[11]

$$s_1 = 3 \quad s_3 = 2 \quad s_4 = 3 \quad s_5 = 3$$

Thus the first state to be removed is s_3. In the next FA of the sequence, we have the following weights:

$$s_1 = 4 \quad s_4 = 4 \quad s_5 = 3$$

Therefore, either s_1 or s_4 could be removed. Here, we consider first those states with exactly one in-transition and one out-transition (this is a refinement over the standard technique of [13] that gives good results in our setting). Hence state s_4 is chosen. The new weights are as follows:

$$s_1 = 6 \quad s_5 = 6$$

Again, both states have the same weight, but only s_5 fulfill the above condition. Thus we first remove s_5 and, finally, s_1.

Now, we show the results of a preliminary experimental evaluation with some typical benchmarks. Table 1 shows, for each benchmark, the abstract atom used for the partial evaluation, the experimental speedup (obtained by partially evaluating them with Proff [28], a simple *offline* partial evaluator for pure Prolog), the original regular expression and the transformed one according to Sect. 4.2.

Observe, for instance, the case of `incList` (the program of Example 1): having a static first argument, as in `incList(s, d, d)`, has no (positive) impact on the associated partial evaluation since no call in the main loop is removed; in contrast, it is the main loop that is fully unrolled. Here, the slight slowdown could be explained by the likely increase in code size and memory consumption

[11] Note that every predicate symbol is considered as a single character.

due to loop unrolling. In contrast, if we consider `incList(d, s, d)`, the outer loop for `incList` remains but all calls to `iList`, `nat`, and `add` are fully unfolded, which explains the significant performance improvement. Benchmarks `applast` and `match` are classical examples where partial evaluation may get a significant improvement. As can be seen in the associated regular expressions, some steps inside a loop are reduced in each example. Finally, benchmark `power` shows an example where only some calls to `power` are unfolded (hence a constant improvement) and, thus, no significant speedup is achieved.

Clearly, the finite automata or regular expressions produced by our technique are not always easy to analyze. For small examples, they can help the user to understand why adding more static information has no effects in some cases, why significant improvements can be achieved even with no static data, etc. For larger programs, however, the analysis becomes rather complex and the development of analysis techniques and tools will be required, an interesting topic for further research.

5 Related Work

We find very few works devoted to formally analyze the effectiveness of partial evaluation. For instance, [3] establishes several properties of program transformations based on folding/unfolding in the context of logic programming. In particular, he proves that *superlinear* speedup cannot be accomplished by partial evaluation (this result can also be found in [4] for a flow chart language). [4] develops a *speedup analysis* that, for any binding-time annotated program, computes a relative speedup interval such that the specialization of this program will result in a speedup within the predicted interval. Our approach is partly inspired by the work of [4], but several significant differences exist: they determine the program loops statically in the source program, while we use a combination of size-change analysis and trace analysis to identify the program loops and the context where they appear; [4] is formalized in the context of a simple flow chart language, while we consider a logic programming language; they do not distinguish whether the static parameters decrease strictly or non-strictly from one call to another, while this is essential in our approach.

Regarding our trace analysis, we share the aims of previous work by Gallagher and Lafave [14]. There are, however, a number of important differences: 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, their approximation technique is based on abstract interpretation [10], while ours is based on (simpler) techniques from formal languages and automata theory; the main difference, though, is that 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, where call traces are elegantly represented by means of finite automata or regular expressions (since they form a regular language, in contrast to the trace terms of [14]).

As mentioned in Sect. 1, there are some recent approaches (e.g., [11, 27]) where *experimental* frameworks for estimating the *best* division are introduced. Although their aim is similar to ours, we put the emphasis on developing a *symbolic* framework and thus our goals are different. Indeed, both approaches can be seen as complementary.

To summarize, this work constitutes our last contribution from a long-term research on formally measuring and estimating the effects of partial evaluation (see, [2, 29, 30]). These works, however, never considered *prediction* (i.e., speedups were measured *a posteriori*).

6 Conclusions and Future Work

Predicting the potential speedup that can be achieved by partial evaluation is a challenge that has received little attention so far. In this work, we introduced a symbolic framework for analyzing the effects of a partial evaluation given a program and an initial call. Basically, we use a size-change analysis to determine which recursive predicates could be safely unfolded because their control flow is statically determined by the available information. Then, a trace analysis has been introduced in order to get the context of each procedure call, so that the impact of its elimination can be better estimated.

The techniques introduced in this paper can be seen as a first step for the development of automated *quantitative* techniques and tools for predicting the potential speedup, thus it opens a number of interesting lines for further research. For instance, we could define formal techniques for comparing finite automata or regular expressions (with the same structure), simple cost models based on the output of the trace analysis, tools for easing the graphical inspection of large finite automata and regular expressions, etc.

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. A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, 1973.
2. E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042, 2001.
3. T. Amtoft. Properties of Unfolding-based Meta-level Systems. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-based Program Transformation (PEPM'91)*, 1991.

4. L.O. Andersen and C.K. Gomard. Speedup Analysis in Partial Evaluation: Preliminary Results. In *Proc. of the ACM Workshop on Partial Evaluation and Semantics-based Program Transformation (PEPM'92)*, pages 1–7. Yale University, New Haven, CT, 1992.
5. M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.
6. A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT'91*, pages 153–180. Springer LNCS 494, 1991.
7. J.A. Brzozowski and E.J. McCluskey. Signal Flow Graph Techniques for Sequential Circuit Diagrams. *IEEE Trans. on Electronic Computers*, EC-13:67–76, 1963.
8. N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.
9. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
10. 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 Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
11. S. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.
12. S.-J. Craig, J. Gallagher, M. Leuschel, and K.S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of the Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 53–68. Springer LNCS 3573, 2005.
13. M. Delgado and J. Morais. Approximation to the Smallest Regular Expression for a Given Regular Language. In *Proc. of the 9th Int'l Conf. on Implementation and Application of Automata (CIAA'04)*, pages 312–314. Springer LNCS 3317, 2004.
14. J.P. Gallagher and L. Lafave. Regular Approximation of Computation Paths in Logic and Functional Languages. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, pages 115–136. Springer LNCS 1110, 1996.
15. Y.-S. Han and D. Wood. The generalization of generalized automata: Expression automata. *International Journal of Foundations of Computer Science*, 16:499–510, 2005.
16. Y.-S. Han and D. Wood. Obtaining shorter regular expressions from finite-state automata. *Theoretical Computer Science*, 370(1-3):110–120, 2007.
17. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
18. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
19. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline Specialisation in Prolog using a Hand-Written Compiler Generator. *Theory and Practice of Logic Programming*, 4(1-2):139–191, 2004.
20. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Proc. of the 18th Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2008)*. Technical University of Valencia, 2008. Available from `http://www.dsic.upv.es/~gvidal/german/papers.html`. To appear.
21. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. of Int'l Conf. on Logic Programming (ICLP'97)*, pages 63–77. The MIT Press, 1997.

22. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
23. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
24. M. Mohri and M.-J. Nederhof. *Regular Approximation of Context-Free Grammars through Transformation*, chapter 9, pages 153–163. Kluwer Academic Publishers, The Netherlands, 2001.
25. M. Mohri and F.C.N. Pereira. Dynamic Compilation of Weighted Context-Free Grammars. In *Proc. of the 36th Annual Meeting of the Association for Computational Linguistics and the 17th Int'l Conf. on Computational Linguistics (COLING-ACL'98)*, pages 891–897, 1998.
26. M.-J. Nederhof. *Regular approximation of CFLs: a grammatical view*, chapter 12, pages 221–241. Kluwer Academic Publishers, The Netherlands, 2000.
27. C. Ochoa and G. Puebla. Poly-controlled Partial Evaluation in Practice. In *Proc. of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'07)*, pages 164–173. ACM, 2007.
28. S. Tamarit and G. Vidal. PROFF - A PRolog OFFline partial evaluator. URL: `http://german.dsic.upv.es/proff.html`, 2007.
29. G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62. ACM Press, 2002.
30. G. Vidal. Cost-Augmented Partial Evaluation of Functional Logic Programs. *Higher-Order and Symbolic Computation*, 17(1-2):7–46, 2004.
31. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM'07)*, pages 51–60. ACM Press, 2007.
32. D. Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, 1987.