Towards Predicting the Effectiveness of Partial Evaluation

Germán Vidal
DSIC, Technical University of Valencia
Camino de Vera s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract

Recent approaches to partial evaluation—a well-known technique for program specialization—include a so called size-change analysis for ensuring the termination of the process. This paper presents a novel application of size-change analysis for predicting the effectiveness of partial evaluation. Size-change analysis is based on computing an approximation of the program loops. Here, we present an automatic transformation that takes the output of the size-change analysis and produces an approximation of the loops in the *specialized* program. This information can be used for determining—before performing the actual specialization process—when partial evaluation may produce a significant improvement and when it would be useless. An experimental evaluation demonstrates the usefulness of our approach.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—partial evaluation, program analysis; I.2.2 [*Artificial Intelligence*]: Automatic Programming—program transformation

General Terms algorithms, performance, theory

Keywords partial evaluation, size-change analysis

1. Introduction

The main goal of *partial evaluation* [7] 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*. In this paper, we present a novel approach to predict whether a concrete partial evaluation problem has the potential to achieve a significant improvement. This is essential to apply partial evaluation to real-life programming languages and applications.

We present our developments within the logic programming paradigm [14] but the main ideas can also be transferred to other programming languages. Let us illustrate the key ideas with a simple example. Consider the following Prolog program:[1]

```
power(X, 0, 1).
power(X, 1, X).
power(X, N, P) :- N > 1, M is N - 1, power(X, M, Q), P is X * Q.
```

which defines a relation `power` for computing the powers of a given number, i.e., `power(x, n, p)` holds if $x^n = p$. Here, an expression of the form "`X is Exp`" first evaluates `Exp` and then matches the computed value to `X`.

In this example, a size-change analysis [8] would infer that the program contains a single loop (associated to the recursive calls to `power`) and that the second parameter of `power` strictly decreases from one call to another. This means that the control flow is completely determined by the value of the second parameter. This information has been exploited by Glenstrup and Jones [6] to formulate a criterion—called *bounded anchoring*—to guarantee the termination of partial evaluation.

In this paper, we use a similar approach for predicting the effectiveness of partial evaluation. One may observe that the overall speedup of a program is determined by the speedups of its loops, since sufficiently long runs will consume most of the execution time inside loops. The key idea is that partial evaluation achieves significant improvements by removing loops from a program, which are then unrolled at partial evaluation time. Therefore, if we have the output of a size-change analysis available, together with the static/dynamic nature of each program's parameter, we can easily determine which program loops will likely be removed in the specialized program and which loops will definitely remain.

For instance, in the example above, our approach determines that applying partial evaluation when only the first parameter of `power` is static produces a useless specialization, while applying it when the second parameter is static has the potential to achieve a significant improvement. For example, a partial evaluation of the program above w.r.t. the query $\langle$`power(X, 3, P)`$\rangle$ would produce the following residual program:

```
power3(X, P) :- A is X * X, P is X * A.
```

which contains no loop (i.e., now `power3` is not recursive).

Our initial experiments point out the usefulness of our approach. In fact, one of its main advantages is its simplicity: it can be implemented almost for free in any partial evaluator in which termination is ensured by means of a size-change analysis (e.g., [6, 18]) or a similar termination analysis like, e.g., the abstract binary unfoldings of [4] (which are used in the partial evaluator Logen [11]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00.

[1] In the examples, as it is common in Prolog, we write variable symbols starting with upper case and all other symbols starting with lower case.

2. The Language

We assume some familiarity with the standard definitions and notations of logic programming. In the following, we briefly present some basic notions; we refer the interested reader to [14] for a detailed introduction to this programming paradigm.

We consider a first-order language with a fixed vocabulary of predicate symbols, function symbols, and variables denoted by Π, Σ and $\mathcal{V}$, respectively. We let $\mathcal{T}(\Sigma, \mathcal{V})$ denote the set of *terms* constructed using symbols from Σ and variables from $\mathcal{V}$. An *atom* has the form $p(t_1, \ldots, t_n)$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ 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 $\mathcal{V}ar(s) = \emptyset$.

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.

As it is common practice in *partial evaluation* [15], we adopt the convention that SLD derivations can be either infinite, successful, failed (e.g., $Q \leadsto^*_\theta Q'$ such that $Q' \neq true$ and no SLD resolution step can be applied to Q'), or *incomplete*, in the sense that at any point we are allowed to simply not select any query atom and terminate the derivation.

3. Size-Change Analysis

The size-change principle [8] is a recent technique originally aimed at analyzing the termination of functional programs. Intuitively speaking, it consists in tracing size changes of function arguments when going from one function call to another by means of so called *size-change graphs*. Then, assuming that the measure of size gives rise to a well-founded order, the following principle applies [8]: *If every infinite computation would give rise to an infinitely decreasing value sequence (according to the size-change graphs), then no infinite computation is possible.*

The notion of *size-change graph* is adapted to logic programs in [18]. Now, we briefly present (a simplified version of) the size-change analysis for logic programs of [18].

A *strict order* $\succ$ is an irreflexive and transitive binary relation. An order $\succ$ is *well-founded* if there are no infinite sequences of the form $t_1 \succ t_2 \succ \ldots$ An order $\succsim$ is a *quasi-order* if it is reflexive and transitive. We say that an order $\succ$ is *closed under substitutions* (or *stable*) if $s \succ t$ implies $s\sigma \succ t\sigma$ for all $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ and every substitution σ. In [18], size-change graphs are parameterized by a reduction pair [16]:

DEFINITION 1 (reduction pair). *We say that $(\succsim, \succ)$ is a reduction pair if $\succsim$ is a quasi-order and $\succ$ is a well-founded order where both $\succsim$ and $\succ$ are closed under substitutions and compatible (i.e., $\succsim \circ \succ \subseteq \succ$ or $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary).*

Symbolic norms are then considered as a basis for defining appropriate reduction pairs:

DEFINITION 2 (symbolic norm [12]). *A symbolic norm is a function $||\cdot|| : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(I\!N \cup \{+\}, \mathcal{V})$ such that*

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

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 of the variable is clear from the context).

Two popular instances of the above definition are the symbolic *term-size* norm $||\cdot||_{ts}$

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

and the symbolic *list-length norm* $||\cdot||_{ll}$

$$||t||_{ll} = \begin{cases} 1 + ||Xs|| & \text{if } t = [X|Xs] \\ t & \text{if } t \text{ is a variable} \\ 0 & \text{otherwise} \end{cases}$$

For instance, we have $||f(X, Y)||_{ts} = 2 + X + Y$, $||f(a, b)||_{ts} = 2$, $||[a|Y]||_{ll} = 1 + Y$, and $||[a, X]||_{ll} = 2$. The introduction of variables in the range of the norm provides a simple mechanism to express dependencies between the sizes of terms.

In general, given a term t, we have $||t|| = n_0 + n_1 X_1 + \ldots + n_k X_k$ for non-negative integers $n_0, n_1, \ldots, n_k$ and variables $X_1, \ldots, X_k \in \mathcal{V}ar(t)$. By abuse of notation, we say that $||s|| > ||t||$ if $||s|| = n_0 + n_1 X_1 + \ldots + n_k X_k$, $||t|| = m_0 + m_1 X_1 + \ldots + m_j X_j$, $k \geqslant j \geqslant 0$, and $n_i > m_i$ for some $i \in \{0, \ldots, j\}$ and $n_l \geqslant m_l$ for all other $l \in (\{0, \ldots, j\} \backslash \{i\})$. The definition of $||s|| \geqslant ||t||$ is perfectly analogous.

Now, a reduction pair can easily be defined from a given symbolic norm as follows:

DEFINITION 3 (induced orders). *The pair of orders $(\succsim, \succ)$ induced by a symbolic norm $||\cdot||$ are defined by $s \succ t \Leftrightarrow ||s|| > ||t||$ and $s \succsim t \Leftrightarrow ||s|| \geqslant ||t||$ for all terms s and t.*

We prove in [18, Lemma 1] that the pair of orders induced by any symbolic norm is actually a reduction pair.

The size-change graphs associated to a logic program, which are parametric w.r.t. a reduction pair, are defined as follows:

DEFINITION 4 (size-change graph). *Let P be a program and let $(\succsim, \succ)$ be a reduction pair. We define a size-change graph for every clause $p(s_1, \ldots, s_n) \leftarrow Q$ of P and every atom $q(t_1, \ldots, t_m)$ in Q (if any).*

The graph has n output nodes marked with $\{1_p, \ldots, n_p\}$ and m input nodes marked with $\{1_q, \ldots, m_q\}$. If $s_i \succ t_j$ holds, then we have a directed edge from output node i_p to input node j_q marked with $\succ$.Otherwise, if $s_i \succsim t_j$ holds, then we have an edge from output node i_p to input node j_q marked with $\succsim$.

A size-change graph is thus a bipartite graph $\mathcal{G} = (V, W, E)$ where $V = \{1_p, \ldots, n_p\}$ and $W = \{1_q, \ldots, m_q\}$ are the labels of the output and input nodes, and $E \subseteq V \times W \times \{\succsim, \succ\}$ are the edges.

EXAMPLE 5. *Consider the following program 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) $iList(X, R, I, [XI|RI]) \leftarrow nat(I),\ add(I, X, XI),$
$\qquad incList(R, I, RI).$
(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* $s(\cdot)$. *Let* $(\succsim, \succ)$ *be the reduction pair induced by the symbolic term-size norm* $||\cdot||_{ts}$. *We have six size-change graphs, depicted in Fig. 1, which are associated to clauses* c_2 *(graph* $\mathcal{G}_1$*),* c_3 *(graphs* $\mathcal{G}_2$*,* $\mathcal{G}_3$ *and* $\mathcal{G}_4$*),* c_5 *(graph* $\mathcal{G}_5$*), and* c_7 *(graph* $\mathcal{G}_6$*).*

In order to focus on program loops, the following definition introduces the notion of *maximal multigraph*:

DEFINITION 6 (maximal multigraph). *Given a program P, a multigraph of P is either a size-change graph of P or the concatenation (see below) of two multigraphs of P. Given two multigraphs:*

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1)$$

and

$$\mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

w.r.t. the same reduction pair $(\succsim, \succ)$*, then the concatenation*

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

is also a multigraph, 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* $\succsim$*.*

A multigraph $\mathcal{G}$ *of* P *is called maximal if its input and output nodes are both labeled with* $\{1_p, \ldots, n_p\}$ *for some predicate* p/n *and if it is idempotent, i.e.,* $\mathcal{G} = \mathcal{G} \bullet \mathcal{G}$*.*

Roughly speaking, given the set of size-change graphs of a program, we first compute its transitive closure under the concatenation operator, thus producing a finite set of multigraphs. Then, we only need to focus on the *maximal* multigraphs of this set because they represent the program loops.

EXAMPLE 7. *Given the size-change graphs of Example 5 (Fig. 1), we have four maximal multigraphs:* $\mathcal{G}_{14} = \mathcal{G}_1 \bullet \mathcal{G}_4$*,* $\mathcal{G}_{41} = \mathcal{G}_4 \bullet \mathcal{G}_1$*,* $\mathcal{G}_5$*, and* $\mathcal{G}_6$*, which are shown in Figure 2.*

The following auxiliary definition is slightly extended from [4] to consider an arbitrary computation rule:

DEFINITION 8 (calls). *Let P be a program,* $\mathcal{R}$ *a computation rule, and* Q_0 *a query. We say that A is a call in a derivation of* Q_0 *with P and* $\mathcal{R}$ *iff* $Q_0 \leadsto^* Q$ *and* $\mathcal{R}(Q) = A$*. We denote by* $calls_P^{\mathcal{R}}(Q_0)$ *the set of calls in the computations of* Q_0 *with P and* $\mathcal{R}$*.*

Informally speaking, $calls_P^{\mathcal{R}}(Q_0)$ contains all atoms that are reachable from Q_0 with program P and computation rule $\mathcal{R}$.

Now, we present a sufficient condition for termination which extends the notion of *size-change termination* [8] to the case of logic programs. Basically, we require the decreasing parameters of (potentially) looping predicates to be ground in the considered computations.

THEOREM 9 (termination [18]). *Let P be a logic program w.r.t. a reduction pair* $(\succsim, \succ)$*. Let* $\mathcal{A}$ *be a set of atoms. If every maximal multigraph of 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 ground, then P is terminating w.r.t.* $\mathcal{A}$*.*

Note that t_i should be ground 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 information will be available for free from the output of a standard binding-time analysis (see Sect. 4.1).

EXAMPLE 10. *Consider again the program of Example 5 and the maximal multigraphs of Fig. 2. Here, Theorem 9 guarantees the termination of SLD resolution for those computations in which the following parameters are ground:*

- *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*.*

4. Predicting the Effectiveness of PE

In this section, we introduce our approach for predicting the effectiveness of partial evaluation. First, we present a simple scheme for the partial evaluation of logic programs that follows the so called offline approach. Then, we introduce a transformation on maximal multigraphs that allows us to get an approximation of the loops in the partially evaluated program *before specializing* the original program. Finally, we show the results of an experimental evaluation that demonstrates the usefulness of our approach.

4.1 Offline Partial Evaluation

In contrast to *online* partial evaluators, *offline* partial evaluators consists of two separate stages: program annotation and proper specialization.

The program annotation stage uses appropriate static analysis for propagating static/dynamic information through the program and for ensuring the termination of the specialization process. The output of this stage is an annotated version of the original program that can be used to guide the specialization stage. Therefore, the specialization stage should only follow the program annotations.

We now informally describe a simple scheme for offline partial evaluation of logic programs (like that of [11], though considerably simplified). Our program annotation stage proceeds as follows:

1. First, a binding-time analysis[2] is applied. Here, we only consider two binding-times: `S` (static, i.e., ground at specialization time) and `D` (dynamic, i.e., possibly unknown at specialization time). For simplicity, we consider a *monovariant* binding-time analysis that only returns a single classification—a list of binding-times—for the parameters of each program's procedure.
2. Second, a size-change analysis of the program, which follows the scheme of Section 3, is performed.
3. Finally, using the output of the above analysis, the program is annotated as follows:
   - For each predicate symbol p/n of the program, we have a filtering declaration of the form $p(b_1, \ldots, b_n)$ where each binding-time b_i is either `S` or `T`.
   - Each atom in the body of a program clause is annotated either as `unfold` or `memo` (see below).

The specialization stage consists of an iterative algorithm that takes an annotated program, together with the values of the static parameters of the initial query, and produces a residual, specialized program. This stage starts with a set $\mathcal{A}_0$ of atoms initialized with the atoms of the initial query, and produces a finite sequence of sets of atoms $\mathcal{A}_0$, $\mathcal{A}_1$, $\mathcal{A}_2$, etc., as follows:

[2] We consider that binding-time analysis only propagate static/dynamic information but does not take termination issues into account.

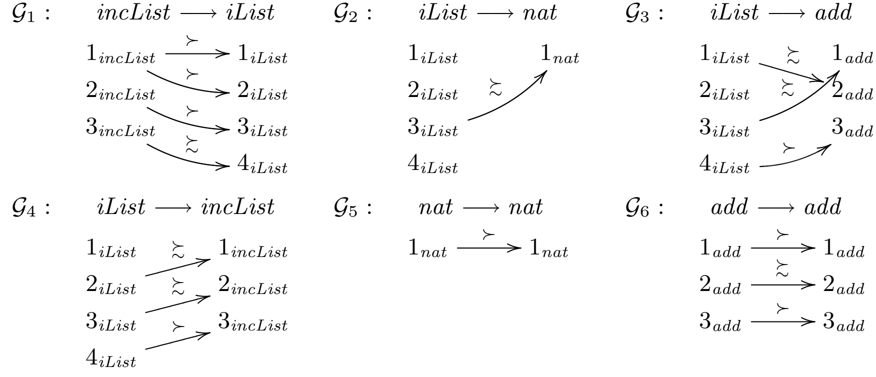

Figure 1. Size-change graphs for `incList`

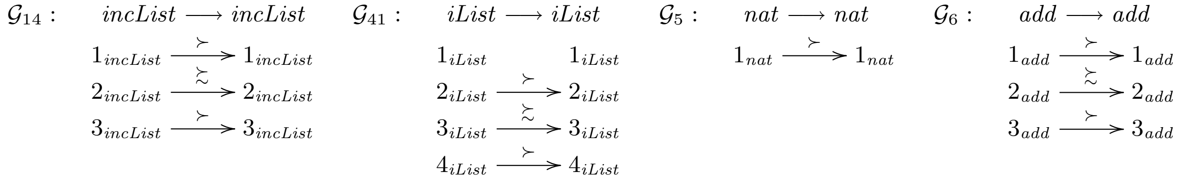

Figure 2. Maximal multigraphs for `incList`

Local level: The atoms in the current set of atoms, $\mathcal{A}_i$, are evaluated by SLD resolution and a computation rule—often called unfolding rule—that only allows the unfolding of atoms which are annotated with `unfold`. This level returns a (possibly incomplete) finite SLD resolution tree.

Global level: The atoms in the leaves of the SLD tree returned by the local level are added to the current set $\mathcal{A}_i$ of partially evaluated atoms by replacing dynamic parameters with fresh variables. In this way, a new set of atoms $\mathcal{A}_{i+1} \supseteq \mathcal{A}_i$ is produced, which is then passed to the local level, and so forth.

The process stops when $\mathcal{A}_i = \mathcal{A}_{i+1}$ (up to variable renaming). The final set of partially evaluated atoms is used to produce the residual program. Also, a post-processing of renaming is generally applied in order to ensure the correctness of the transformation. We do not describe further this stage since it is not needed for understanding the forthcoming developments.

4.2 Transforming Maximal Multigraphs

In this section, we present the main contribution of this work: a transformation on maximal multigraphs which allows us to predict the effectiveness of partial evaluation.

In the following, we consider that the binding-time analysis 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 $b_i \in \{\mathtt{S}, \mathtt{D}\}$ for $i = 1, \ldots, n$, and returns a *division*, i.e., a mapping $\mu \in (\Pi \rightarrow \{\mathtt{S}, \mathtt{D}\}^*)$ which classifies every program parameter as either static or dynamic. Also, we say that an atom $p(t_1, \ldots, t_n)$ is a concrete instance of an abstract atom $p(b_1, \ldots, b_n)$ if t_i is a ground term whenever $b_i = \mathtt{S}$ for all $i = 1, \ldots, n$.

We denote by $\mathcal{G}[i_p \xrightarrow{R} i_p]$ a multigraph of a predicate p/n that includes an edge $i_p \xrightarrow{R} i_p$ for some $i \in \{1, \ldots, n\}$ and $R \in \{\succ, \succsim\}$. Also, given a multigraph of the form $\mathcal{G}[i_p \xrightarrow{R} i_p]$, we denote by $\mathcal{G}[\,]$ the multigraph that results from $\mathcal{G}[i_p \xrightarrow{R} i_p]$ by removing the edge $i_p \xrightarrow{R} i_p$.

DEFINITION 11 (transformed multigraphs w.r.t. a division). *Let P be a program and $\mathcal{S}$ a set of maximal multigraphs of P. Let μ be a division for P w.r.t. an abstract atom A. Then, the set of transformed multigraphs $\mathcal{S}_\mu$ of $\mathcal{S}$ w.r.t. μ is obtained from $\mathcal{S}$ by applying the following rules as much as possible:*

$$\begin{array}{llll} (1) & \mathcal{S}' \cup \{\mathcal{G}[i_p \xrightarrow{\succ} i_p]\} & \Rightarrow \; \mathcal{S}' & \textit{if } b_i = \mathtt{S} \\ (2) & \mathcal{S}' \cup \{\mathcal{G}[i_p \xrightarrow{\succsim} i_p]\} & \Rightarrow \; \mathcal{S}' \cup \{\mathcal{G}[\,]\} & \textit{if } b_i = \mathtt{S} \end{array}$$

where $\mu(p) = [b_1, \ldots, b_n]$ and $i \in \{1, \ldots, n\}$.

The first rule in the transformation above allows us to completely remove a maximal multigraph when it includes a strictly decreasing static parameter. When the first rule is no longer applicable, the second rule allows us to simplify the remaining maximal multigraphs by removing those edges which are associated with non-strictly decreasing static parameters.

EXAMPLE 12. *Consider the maximal multigraphs of Fig. 2 and a binding-time analysis that computes the following division:*

$$\mu \;=\; \left\{ \begin{array}{rcl} incList & \mapsto & [\mathtt{D}, \mathtt{S}, \mathtt{D}], \\ iList & \mapsto & [\mathtt{D}, \mathtt{D}, \mathtt{S}, \mathtt{D}], \\ nat & \mapsto & [\mathtt{S}], \\ add & \mapsto & [\mathtt{S}, \mathtt{D}, \mathtt{D}] \end{array} \right\}$$

for program `incList` *(see Example 5) w.r.t. the abstract atom $incList(\mathtt{D}, \mathtt{S}, \mathtt{D})$. Then, Definition 11 returns the transformed set of maximal multigraphs which is shown in Fig. 3.*

Given a division μ for a program P, the goal of the transformed set of multigraphs is to describe the program loops of the specialized program P_μ that would be obtained by (the best possible) partial evaluation of P using the division μ.

EXAMPLE 13. *Consider again the program of Example 5, together with the division μ of Example 12. A partial evaluation of program* `incList` *w.r.t. the query $\langle incList(L, s^{10}(0), T) \rangle$,*[3] *a concrete in-*

[3] We write $s^{10}(0)$ as a shorthand for $s(s(s(s(s(s(s(s(s(s(0))))))))))$.

$$\mathcal{G}_{14}: \quad incList \longrightarrow incList \qquad \mathcal{G}_{41}: \quad iList \longrightarrow iList$$

$$1_{incList} \xrightarrow{\succ} 1_{incList} \qquad 1_{iList} \quad 1_{iList}$$

$$3_{incList} \xrightarrow{\succ} 3_{incList} \qquad 2_{iList} \xrightarrow{\succ} 2_{iList}$$

$$4_{iList} \xrightarrow{\succ} 4_{iList}$$

Figure 3. Transformed set of maximal multigraphs for `incList`

$$\mathcal{G}_{14}: incList' \longrightarrow incList' \qquad \mathcal{G}_{41}: iList10 \longrightarrow iList10$$

$$1_{incList'} \xrightarrow{\succ} 1_{incList'} \qquad 1_{iList10} \quad 1_{iList10}$$

$$2_{incList'} \xrightarrow{\succ} 2_{incList'} \qquad 2_{iList10} \xrightarrow{\succ} 2_{iList10}$$

$$3_{iList10} \xrightarrow{\succ} 3_{iList10}$$

Figure 4. Maximal multigraphs for `incList10`

stance of $incList(\mathtt{D},\mathtt{S},\mathtt{D})$*, returns the following residual program:*

(c_1) $incList10([\,],[\,])$.
(c_2) $incList10([X|R],L) \leftarrow iList10(X,R,L)$.
(c_3) $iList10(X,R,[s^{10}(X)|RI]) \leftarrow incList'(R,RI)$.
(c_4) $incList'([\,],[\,])$.
(c_5) $incList'([X|R],L) \leftarrow iList10(X,R,L)$.

The set of maximal multigraphs of this residual program is shown in Fig. 4. Observe that these graphs are equal (up to renaming of predicates and parameters) to the transformed set of multigraphs which is shown in Fig. 3.

THEOREM 14. *Let P be a logic program and $\mathcal{S}$ be a set of maximal multigraphs of P w.r.t. a reduction pair $(\succsim,\succ)$. Let μ be a division for P w.r.t. an abstract atom A. Then, for every concrete instance Q of A, there exists a partial evaluation P' of P w.r.t. Q (using μ) such that the set of maximal multigraphs of P' w.r.t. the same reduction pair $(\succsim,\succ)$ is equal (up to renaming of predicates and parameters) to the transformed set of multigraphs $\mathcal{S}_\mu$ of $\mathcal{S}$ w.r.t. μ.*

The proof of this result relies on Theorem 9 and the correctness of the binding-time analysis that computes the considered division. Basically, if the computed division is *congruent* (i.e., the values of all static parameters can be determined from the values of other static parameters only), then Theorem 9 ensures that a partial evaluator can fully unfold all atoms $p(t_1,\ldots,t_n)$ when there exists an edge $i_p \xrightarrow{\succ} i_p$, $i \in \{1,\ldots,n\}$, in the associated maximal multigraphs and the i-th parameter of p/n is classified as static. As a consequence, no loop (i.e., no recursive definition) associated with p/n will appear in the residual program. Otherwise, when there exists an edge $i_p \xrightarrow{\succsim} i_p$, $i \in \{1,\ldots,n\}$, in the associated maximal multigraphs and the i-th parameter of p/n is classified as static (i.e., it is ground in every call to p/n), we can get rid off this edge because a partial evaluator will remove the i-th parameter during the post-processing of renaming.

The relevance of our transformation lies in the fact that one can analyze the effectiveness of a partial evaluation problem by comparing the maximal multigraphs of the original program and the transformed set of multigraphs by Definition 11. Therefore, only the first stage (including binding-time and size-change analysis) is needed to determine whether the second stage—the proper specialization—will be able to achieve a significant improvement or not.

In particular, in the following we consider the simplest possible condition: if no multigraph is removed by the transformation of Definition 11 (rule 1), then the associated specialization is considered useless. Then, the user could use this information to modify the program (or give up the intended specialization). Also, we could use this information to let the partial evaluator to automatically reclassify as dynamic those static parameters that cannot be used to remove a maximal multigraph, thus producing smaller residual programs and a faster specialization process.

4.3 Experimental Evaluation

In this section, we show the results of an experimental evaluation that illustrates the usefulness of our approach. We consider a set of typical partial evaluation benchmarks (most of them appear in the DPPD collection of problems for partial deduction [9]). For each benchmark, Table 1 shows several possible partial evaluations w.r.t. different abstract atoms.

The benchmarks have been partially evaluated using the system Logen [11] (in particular, we used its web interface [10]), an offline partial evaluator for Prolog programs. Logen uses a termination analysis based on abstract binary unfoldings, a similar approach to size-change analysis. For each benchmark and abstract atom, we show the run times of the original (orig) and specialized (spec) programs in SICStus Prolog, as well as the speedup achieved (orig/spec). Input queries were chosen to give a reasonably long overall time. In general, we show the average of ten executions. In some cases (those marked with "$*$"), however, run times were so small that we show the sum of thousands of repeated executions; these benchmarks are thus less reliable.

We also show the size of the original (orig) and specialized (spec) compiled programs with SICStus Prolog, as well as the code increase (spec/orig) due to partial evaluation. This is also a useful information because there is often a trade off between the speedup achieved and the size of the residual program. We hope to be able to also infer an approximation of the size of residual programs from the maximal multigraphs and the values of static parameters.

Finally, we show the number of maximal multigraphs that are removed by the transformation of Definition 11 (but we ignore the removal of single edges). Here, k/n means that k maximal multigraphs are removed out of a total of n multigraphs.

Observe in Table 1 that there are several cases where some loop is removed but no significant speedup is achieved (e.g., rows 1, 3, 6, 15, 18). This can easily be explained by the fact that the partial evaluator does not always produce the optimal specialization. On the other hand, we can observe that all benchmarks in which a significant speedup has been achieved (i.e., 1.10 or more) at least one maximal multigraph has been removed.

The benchmarks of Table 1 are publicly available from the URL `http://www.dsic.upv.es/users/elp/german/scpre/`.

5. Related Work

We find very few works devoted to formally analyze the effectiveness of partial evaluation. For instance, Amtoft [2] 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 [3] for a flow chart language). Andersen and Gomard [3] develop 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 clearly inspired by the work of [3], but several significant differences exist: they determine the program loops statically in the source program, while we use the size-change analysis to identify the program loops; [3] 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; they compute a speedup interval as the ratio between the cost of the—both static and dynamic—sentences in a loop and the cost of the dynamic sentences,[4] while we focus only on those loops that can be completely unrolled because control flow is determined by a static parameter; finally, [3] develops an *ad-hoc* analysis, while we reuse an existing device—the size-

[4] For this purpose, [3] should measure an approximation of the cost of each sentence of the language.

Table 1. Benchmark results

Benchmark		abstract atom	run time			program size			removed
			orig (ms)	spec (ms)	speedup (%)	orig (bytes)	spec (bytes)	increase (%)	loops
applast	1	applast(D,D,D)	425	410	1.04	1082	1294	1.20	1/2
	2	∗ applast(S,D,D)	3920	2600	1.51	1630	18628	11.42	2/2
	3	applast(D,S,D)	432	417	1.04	1125	1337	1.19	1/2
contains	4	contains(D,D)	394	389	1.01	1839	1855	1.01	0/2
	5	contains(S,D)	407	280	1.96	1927	4216	2.19	1/2
	6	∗ contains(D,S)	1070	1980	0.54	1955	929401	475.40	1/2
incList	7	incList(D,D,D)	794	737	1.08	1326	1558	1.17	0/4
	8	incList(D,S,D)	739	68	10.87	1619	1031	1.57	2/4
	9	∗ incList(S,D,D)	760	640	1.13	2801	10567	3.77	2/4
	10	∗ incList(S,S,D)	3740	40	93.50	1919	15524	8.09	4/4
insertSort	11	insertSort(D,D)	339	338	1.00	1108	1120	1.01	0/2
	12	∗ insertSort(S,D)	660	20	33.00	1342	519	0.39	2/2
match	13	match(D,D)	647	623	1.08	966	976	1.01	0/2
	14	match(S,D)	644	448	1.42	1051	1918	1.82	1/2
	15	∗ match(D,S)	1640	4620	0.35	1078	27712	25.71	2/2
power	16	power(D,D,D)	465	458	1.01	1446	3003	2.08	0/3
	17	power(S,D,D)	469	452	1.04	1709	4306	2.52	0/3
	18	power(D,S,D)	476	455	1.05	1521	1230	0.81	1/3
regexp	19	generate(D,D,D)	851	882	0.97	1461	2400	1.64	0/5
	20	generate(S,D,S)	844	120	7.03	1752	1340	0.77	4/5

change analysis—that is already used by a partial evaluator, and thus almost no overhead is introduced.

Vidal [17] introduces a cost-augmented semantics that can be used to perform narrowing-driven partial evaluation [1] in such a way that residual programs also include information about the speedup achieved by each function. In contrast to our approach, an estimation of the speedup achieved is computed *a posteriori*.

Finally, Craig and Leuschel [5] present an optimized partial evaluator that finds the best possible specialization w.r.t. both run time and code size of residual programs. Their approach is purely experimental in the sense that alternative divisions are tested by producing the associated residual program and then using some input queries to analyze its run time. In contrast to ours, their approach incurs into a significant overhead.

6. Discussion

This paper presents a promising step towards predicting the effectiveness of partial evaluation. Our approach is based on a novel application of size-change analysis.

We consider several possibilities for future work. Here, we basically tried to answer the question "is a partial evaluation problem worthwhile?". The answer is thus only yes (if some loop is removed from the set of maximal multigraphs) or no (otherwise). We could refine this condition by considering a form of *weighted* size-change graph (similarly to the weighted graphs of [13]); this would allow us to quantify the expected improvement. Also, we could analyze the maximal multigraphs that cannot me removed but are "shortened" because of the static parameters; this situation could also point out a source of improvement. We also plan to investigate whether an estimate for both the size of residual programs and the specialization time could be inferred from (an appropriate extension of) the set of maximal multigraphs.

References

[1] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.

[2] 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.

[3] 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.

[4] M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[5] S. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.

[6] A.J. Glenstrup and N.D. Jones. Termination Analysis and Specialization-Point Insertion in Offline Partial Evaluation. *ACM TOPLAS*, 27(6):1147–1215, 2005.

[7] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[8] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.

[9] M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks. Available from URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`.

[10] M. Leuschel, D. Elphick, M. Varea, S. Craig, and M. Fontaine. The Ecce and Logen Partial Evaluators and Their Web Interfaces. In *Proc. of PEPM'06*, pages 88–94. IBM Press, 2006.

[11] 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.

[12] 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.

[13] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Proving Termination for Logic Programs by the Query-Mapping Pairs Approach. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, pages 453–498. Springer LNCS 3049, 2004.

[14] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[15] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[16] R. Thiemann and J. Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

[17] G. Vidal. Cost-Augmented Partial Evaluation of Functional Logic Programs. *Higher-Order and Symbolic Computation*, 17(1-2):7–46, 2004.

[18] G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. Technical report, DSIC, Technical University of Valencia, 2006. Available from URL: `http://www.dsic.upv.es/users/elp/german/papers.html`.