

Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs^{*}

Gustavo Arroyo, J.Guadalupe Ramos, Josep Silva, and Germán Vidal

Technical University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain.
{garroyo,guadalupe,jsilva,gvidal}@dsic.upv.es

Abstract. Recently, an offline approach to narrowing-driven partial evaluation—a partial evaluation scheme for first-order functional and functional logic programs—has been introduced. In this approach, program annotations (i.e., the expressions that should be generalized at partial evaluation time to ensure termination) are based on a simple syntactic characterization of quasi-terminating programs. This work extends the previous offline scheme by introducing a new annotation strategy which is based on a combination of size-change graphs and binding-time analysis. Preliminary experiments point out that the number of program annotations is drastically reduced compared to the previous approach.

1 Introduction

Narrowing [9] extends the reduction principle of functional languages by replacing matching with unification (as in logic programming). Narrowing-driven partial evaluation (NPE) [1] is a powerful specialization technique for the first-order component of many functional and functional logic languages like Haskell or Curry. In NPE, some refinement of narrowing [9] is used to perform symbolic computations. Currently, *needed narrowing* [3], a narrowing strategy which only selects a function call if its reduction is necessary to compute a value, is the strategy that presents better properties. In general, the narrowing space (i.e., the counterpart of the SLD search space in logic programming) of a term may be infinite. However, even in this case, NPE may still terminate when the original program is *quasi-terminating* w.r.t. the considered narrowing strategy, i.e., when only finitely many different terms—modulo variable renaming—are computed. The reason is that the (partial) evaluation of multiple occurrences of the same term (modulo variable renaming) in a computation can be avoided by inserting a call to some previously encountered variant (a technique known as *specialization-point insertion* in the partial evaluation literature).

Recently, [8] identified a class of quasi-terminating rewrite systems (w.r.t. needed narrowing) that are called *non-increasing*. This characterization is purely syntactic and very easy to check, though too restrictive to be useful in practice.

^{*} This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02 and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

Therefore, [8] introduces an offline scheme for NPE by 1) annotating the program expressions *that violate the non-increasingness property*, and 2) considering a slight extension of needed narrowing to perform partial computations so that annotated subterms are *generalized* at specialization time (which ensures the termination of the process).

In this work, however, we improve on the simpler characterization of non-increasing rewrite systems by the use of *size-change graphs* [7], which approximate the changes in parameter sizes at function calls. In particular, we use the information in the size-change graphs to identify a particular form of quasi-termination, i.e., that only finitely many different *function calls* (modulo variable renaming) can be produced in a computation. For this purpose, the output of a standard binding-time analysis is also used in order to have information on which function arguments are *static* (and thus ground) or *dynamic*. When the information gathered from the combined use of size-change graphs and binding-time analysis does not allow us to infer that the rewrite system quasi-terminates, we proceed as in [8] and annotate the problematic subterms to be generalized at partial evaluation time. Finally we present some benchmarks on the implementation of the new analysis and conclude.

Our work shares many similarities with [6], where a quasi-termination analysis based on size-change graphs is used to ensure the termination of an offline partial evaluator for first-order functional programs. However, transferring Glenstrup and Jones' scheme to functional logic programs and NPE is not easy. For instance, NPE propagates bindings forward in the partial computations and, thus, some additional requirements (compared to [6]) are necessary to still ensure quasi-termination.

2 Preliminaries

In this section, we introduce some basic notions of term rewriting (further details in [4]). A *term rewriting system* (TRS for short) is a set of rewrite rules $l \rightarrow r$ such that l is a nonvariable term and r is a term whose variables appear in l ; terms l and r are called the left-hand side and the right-hand side of the rule, respectively. Given a TRS \mathcal{R} over a signature \mathcal{F} , the *defined* symbols \mathcal{D} are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. We restrict ourselves to finite signatures and TRSs. We denote the domain of terms and *constructor terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively, where \mathcal{V} is a set of variables with $\mathcal{F} \cap \mathcal{V} = \emptyset$.

A TRS \mathcal{R} is *constructor-based* if the left-hand sides of its rules have the form $f(s_1, \dots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \dots, n$. In the following, we write $\overline{o_n}$ for the *sequence of objects* o_1, \dots, o_n . The set of variables appearing in a term t is denoted by $\text{Var}(t)$. A term t is *linear* if every variable of \mathcal{V} occurs at most once in t . \mathcal{R} is left-linear (resp. right-linear) if l (resp. r) is linear for all rules $l \rightarrow r \in \mathcal{R}$. The *definition* of f in \mathcal{R} is the set of rules in \mathcal{R} whose root symbol in the left-hand side is f . A function $f \in \mathcal{D}$

is left-linear (resp. right-linear) if the rules in its definition are left-linear (resp. right-linear).

The root symbol of a term t is denoted by $root(t)$. A term t is *operation-rooted* (resp. *constructor-rooted*) if $root(t) \in \mathcal{D}$ (resp. $root(t) \in \mathcal{C}$). A term t is *ground* if $Var(t) = \emptyset$. A term t is a *variant* of term t' if they are equal modulo variable renaming. Inductively sequential TRSs [2] are a subclass of left-linear constructor-based TRSs. Essentially, a TRS is *inductively sequential* when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction.

3 A Simple Offline NPE Scheme

In this section, we briefly present the offline approach to NPE from [8]. Given an inductively sequential TRS \mathcal{R} , the *first stage* of the process consists in computing the annotated TRS. In [8], annotations were added to those subterms that violate the non-increasingness condition, a simple syntactic characterization of programs that guarantees the quasi-termination of computations. Nevertheless, annotations can be based on other, more refined analysis—the goal of this paper—as long as the annotated program still ensures the termination of the specialization process.

For the annotation stage, the signature \mathcal{F} of a program is extended with a fresh symbol: “ \bullet ”. A term t is then annotated by replacing t by $\bullet(t)$.

Then, the *second stage*—the proper partial evaluation—takes the annotated TRS, together with an initial term, t , and constructs its associated (finite) *generalizing* needed narrowing tree (see below) where, additionally, a test is included to check whether a variant of the current term has already been computed and, if so, stop the derivation; finally, a residual—partially evaluated—program is extracted from the generalizing needed narrowing tree. Essentially, a *generalizing needed narrowing derivation* $s \rightsquigarrow_{\sigma}^* t$ is composed of

- a) *proper needed narrowing steps*, for operation-rooted terms with no annotations,
- b) *generalizations*, for annotated terms, e.g., $f(\bullet(g(y)), x)$ is reduced to both $f(w, x)$ and $g(y)$, where w is a fresh variable, and
- c) *constructor decompositions*, for constructor-rooted terms with no annotations, e.g., $c(f(x), g(y))$ is reduced to $f(x)$ and $g(x)$ when $c \in \mathcal{C}$ and $f, g \in \mathcal{D}$,

where σ is the composition of the substitutions labeling the proper needed narrowing steps. Consider, for instance, the following definitions of the addition and product on natural numbers built from *zero* and *succ*:

$$\begin{array}{ll} add(zero, y) & \rightarrow y & prod(zero, y) & \rightarrow zero \\ add(succ(x), y) & \rightarrow succ(add(x, y)) & prod(succ(x), y) & \rightarrow add(prod(x, y), y) \end{array}$$

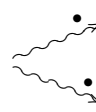
According to [8], this program is not non-increasing. Then, it is annotated by

$$\begin{array}{ll} add(zero, y) & \rightarrow y & prod(zero, y) & \rightarrow zero \\ add(succ(x), y) & \rightarrow succ(add(x, y)) & prod(succ(x), y) & \rightarrow add(\bullet(prod(x, y)), y) \end{array}$$

For example, the following needed narrowing computation is not quasi-terminating w.r.t. the original program (the selected function call is underlined):

$$\begin{aligned} \underline{\text{prod}}(x, y) &\rightsquigarrow_{\{x \mapsto \text{succ}(x')\}} \text{add}(\underline{\text{prod}}(x', y), y) \\ &\rightsquigarrow_{\{x' \mapsto \text{succ}(x'')\}} \text{add}(\text{add}(\underline{\text{prod}}(x'', y), y), y) \rightsquigarrow \dots \end{aligned}$$

In contrast, the corresponding computation by *generalizing* needed narrowing is quasi-terminating (generalization steps are denoted by “ $\rightsquigarrow \bullet$ ”):

$$\begin{aligned} \underline{\text{prod}}(x, y) &\rightsquigarrow_{\{x \mapsto \text{succ}(x')\}} \text{add}(\bullet(\text{prod}(x', y)), y) \rightsquigarrow \dots \\ &\rightsquigarrow \dots \end{aligned}$$


We skip the details of the extraction of residual programs from generalizing needed narrowing trees since it is orthogonal to the topic of this paper (see [8]).

4 Ensuring Quasi-Termination with Size-Change Graphs

In this section, we first recall some basic notions on size-change graphs from [7, 10] and, then, introduce our new approach for ensuring quasi-termination.

In the following, we say that a given order “ \succ ” is *closed under substitutions* (or *stable*) if $s \succ t$ implies $\sigma(s) \succ \sigma(t)$ for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and substitution σ .

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 on terms 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).

Definition 2 (size-change graph). Let (\succsim, \succ) be a reduction pair. For every rule $f(\overline{s_n}) \rightarrow r$ of a TRS \mathcal{R} and every subterm $g(\overline{t_m})$ of r where $g \in \mathcal{D}$, we define a size-change graph as follows. The graph has n output nodes marked with $\{1_f, \dots, n_f\}$ and m input nodes marked with $\{1_g, \dots, m_g\}$. If $s_i \succ t_j$, then there is a directed edge marked with \succ from i_f to j_g . Otherwise, if $s_i \succsim t_j$, then there is an edge marked with \succsim from i_f to j_g .

A size-change graph is thus a bipartite graph $G = (V, W, E)$ where $V = \{1_f, \dots, n_f\}$ and $W = \{1_g, \dots, m_g\}$ are the labels of the output and input nodes, respectively, and we have edges $E \subseteq V \times W \times \{\succsim, \succ\}$.

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

Definition 3 (Concatenation, Maximal Multigraphs). Every size-change graph of \mathcal{R} is a multigraph of \mathcal{R} and if $G = (\{1_f, \dots, n_f\}, \{1_g, \dots, m_g\}, E_1)$ and $H = (\{1_g, \dots, m_g\}, \{1_h, \dots, p_h\}, E_2)$ are multigraphs of \mathcal{R} w.r.t. the same reduction pair (\succsim, \succ) , then the concatenation $G \cdot H = (\{1_f, \dots, n_f\}, \{1_h, \dots, p_h\}, E)$ is also a multigraph of \mathcal{R} . For $1 \leq i \leq n$ and $1 \leq k \leq p$, E contains an edge

from i_f to k_h iff E_1 contains an edge from i_f to some j_g and E_2 contains an edge from j_g to k_h . If there is such a j_g where the edge of E_1 or E_2 is labeled with “ \succ ”, then the edge in E is labeled with “ \succ ” as well. Otherwise, it is labeled with “ \succsim ”.

A multigraph G of \mathcal{R} is called a maximal multigraph of \mathcal{R} if its input and output nodes are both labeled with $\{1_f, \dots, n_f\}$ for some f and if it is idempotent, i.e., $G = G \cdot G$.

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 1. Consider the following example which computes the reverse of a list:

$$\begin{array}{ll} rev([]) & \rightarrow [] \\ rev(x : xs) & \rightarrow app(rev(xs), x : []) \end{array} \quad \begin{array}{ll} app([], y) & \rightarrow y \\ app(x : xs, y) & \rightarrow x : app(xs, y) \end{array}$$

where “ $[]$ ” and “ $..$ ” are the list constructors. In this example, we consider a particular reduction pair (\succsim, \succ) defined as follows:

- $s \succsim t$ iff $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ and for all $x \in \mathcal{V}ar(t)$, $dv(t, x) \leq dv(s, x)$;
- $s \succ t$ iff $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ and for all $x \in \mathcal{V}ar(t)$, $dv(t, x) < dv(s, x)$.

where the *depth of a variable x in a constructor term t* [5], $dv(t, x)$, is defined as follows:

$$\begin{array}{ll} dv(c(\overline{t_n}), x) = 1 + \max(\overline{dv(t_n, x)}) & \text{if } x \in \mathcal{V}ar(c(\overline{t_n})) \\ dv(c(\overline{t_n}), x) = -1 & \text{if } x \notin \mathcal{V}ar(c(\overline{t_n})) \end{array} \quad \begin{array}{ll} dv(y, x) = 0 & \text{if } x = y \\ dv(y, x) = -1 & \text{if } x \neq y \end{array}$$

with $c \in \mathcal{C}$ a constructor term with arity $n \geq 0$. Now, the corresponding size-change graphs for this program are the following:

$$\begin{array}{lll} G_1 : 1_{rev} & \xrightarrow{\succ} & 1_{rev} \\ G_2 : 1_{rev} & \xrightarrow{\succsim} & \begin{array}{l} 1_{app} \\ 2_{app} \end{array} \\ G_3 : 1_{app} & \xrightarrow{\succ} & 1_{app} \\ & & \xrightarrow{\succsim} 2_{app} \end{array}$$

where G_1 and G_3 are also the maximal multigraphs of the program.

Definition 4 (PE-termination, PE-terminating TRS). A needed narrowing computation is PE-terminating if only a finite number of different function calls (i.e., redexes) have been unfolded modulo variable renaming. A TRS is PE-terminating if every possible needed narrowing computation is PE-terminating.

Observe that a PE-terminating TRS does not ensure the quasi-termination of its computations. For instance, given the TRS of Example 1 and the initial call $rev(xs)$, we have the following needed narrowing derivation:

$$\begin{array}{l} rev(xs) \rightsquigarrow_{\{xs \mapsto y:ys\}} app(rev(ys), y : []) \\ \rightsquigarrow_{\{ys \mapsto z:zs\}} app(app(rev(zs), z : []), y : []) \rightsquigarrow_{\{zs \mapsto w:ws\}} \dots \end{array}$$

Although this derivation contains an infinite number of different terms, there is only a finite number of different function calls modulo variable renaming. Fortunately, this is sufficient to ensure the termination in many partial evaluation schemes.

In the following, we consider that the output of a simple (monovariant) binding-time analysis (BTA) is available. Informally speaking, given a TRS and the information on which parameters of the initial function call are static and which are dynamic, a BTA maps each program function to a list of static/dynamic values. Here, we consider that a static parameter is definitely known at specialization time (hence it is ground), while a dynamic parameter is possibly unknown at specialization time.

In the following, we will require the component \succsim of a reduction pair (\succsim, \succ) to be *bounded*, i.e., the set $\{s \mid t \succsim s\}$ must contain a finite number of nonvariant terms for any term t .

The following theorem states sufficient conditions to ensure PE-termination:

Theorem 1. *Let \mathcal{R} be a TRS and (\succsim, \succ) a reduction pair. \mathcal{R} is PE-terminating if every maximal multigraph associated to some function f/n contains either*

- (i) *at least one edge $i_f \xrightarrow{\succ} i_f$ for some $i \in \{1, \dots, n\}$ such that i_f is static, or*
- (ii) *an edge $i_f \xrightarrow{R} i_f$, $R \in \{\succsim, \succ\}$, for all $i = 1, \dots, n$, such that \succsim is bounded.*

Also, we require \mathcal{R} to be right-linear w.r.t. the dynamic variables, i.e., no repeated occurrence of the same dynamic variable may occur in a right-hand side.

The last condition on right-linearity is required in order to avoid situations like the following one: given the TRS

$$\begin{array}{ll} \text{double}(x) \rightarrow \text{add}(x, x) & \text{add}(\text{zero}, y) \rightarrow y \\ & \text{add}(\text{succ}(x), y) \rightarrow \text{succ}(\text{add}(x, y)) \end{array}$$

although *double* and *add* seem clearly terminating (and thus quasi-terminating), the following infinite computation is possible:

$$\begin{aligned} \underline{\text{double}(x)} &\rightsquigarrow \underline{\text{add}(x, x)} \rightsquigarrow_{\{x \mapsto \text{succ}(x')\}} \text{succ}(\underline{\text{add}(x', \text{succ}(x'))}) \\ &\rightsquigarrow_{\{x' \mapsto \text{succ}(x'')\}} \text{succ}(\underline{\text{succ}(\underline{\text{add}(x'', \text{succ}(\text{succ}(x''))))})} \\ &\rightsquigarrow_{\{x'' \mapsto \text{succ}(x''')\}} \dots \end{aligned}$$

which is not quasi-terminating nor PE-terminating.

Rather than requiring source programs to fulfill the conditions of the theorem above, we use this result in order to define a new program annotation for offline NPE which ensures PE-termination.

Basically, it takes every function symbol f/n such that f has a maximal multigraph, and performs one of the following actions:

- 1) if the conditions of Theorem 1 hold, no annotation is added;
- 2) otherwise, we have two possibilities:

- if function f has a static parameter (which does not decrease in the maximal multigraph or the first condition of Theorem 1 would hold), say the i -th parameter, then the i -th argument of every function call to f in the program is annotated;
 - otherwise, if all the parameters of f are dynamic, then the j -th argument of every function call to f in the program is annotated, where j ranges over the parameters of f that do not have an edge $j_f \xrightarrow{R} j_f$, with $R \in \{\succ, \succ\}$;
- 3) finally, if there is more than one occurrence of the same dynamic variable (not yet annotated) in the right-hand side of a program rule, then all occurrences but one (e.g., the leftmost one) are annotated.

5 Discussion

We have undertaken a prototype implementation of the improved offline NPE scheme, which is publicly available at: <http://www.dsic.upv.es/~jsilva/peval>.

In order to further improve the precision of the partial evaluator, we are currently implementing a *polyvariant* version of the program annotation stage. In this case, every function call is treated separately according to the information gathered from the associated maximal multigraph. The resulting algorithm would be more expensive but also more precise.

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. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
3. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
4. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
5. W.N. Chin and S.C. Khoo. Better Consumers for Program Specializations. *Journal of Functional and Logic Programming*, 1996(4), 1996.
6. A.J. Glenstrup and N.D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Trans. Program. Lang. Syst.*, 27(6):1147–1215, 2005.
7. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *ACM Symposium on Principles of Programming Languages (POPL'01)*, volume 28, pages 81–92. ACM press, 2001.
8. J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. *ACM SIGPLAN Notices (Proc. of ICFP'05)*, 40(9):228–239, 2005.
9. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.
10. R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Appl. Algebra Eng. Commun. Comput.*, 16(4):229–270, 2005.