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. An offline approach to narrowing-driven partial evaluation (a partial evaluation scheme for first-order functional and functional logic programs) has recently been introduced. In this approach, program annotations (i.e., the expressions that should be generalised at partial evaluation time to ensure termination) are based on a simple syntactic characterisation 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 significantly reduced compared to the previous approach, which means that faster residual programs are often produced.

1 Introduction

Narrowing [30] 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 specialisation technique for the first-order component of many functional and functional logic languages like Haskell [28] or Curry [18]. In NPE, some refinement of narrowing is used to perform symbolic computations. Currently, *needed narrowing* [4], a narrowing strategy that 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 *specialisation-point insertion* in the partial evaluation literature).

[⋆] This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02, by the Mexican SEIT-ANUIES and DGEST *beca-comisión* and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

Recently, [29] identified a class of quasi-terminating rewrite systems (w.r.t. needed narrowing) that are called *non-increasing*. This characterisation is purely syntactic and very easy to check, though too restrictive to be useful in practice. Therefore, [29] introduces an offline scheme for NPE by

– annotating the program expressions *that violate the non-increasingness property* and
– considering a slight extension of needed narrowing to perform partial computations so that annotated subterms are *generalised* at specialisation time (which ensures the termination of the process).

In this work, we improve on the simpler characterisation of non-increasing rewrite systems by using *size-change graphs* [24] to approximate the changes in parameter sizes from one function call to another. In particular, we use the information in the size-change graphs to identify a particular form of quasi-termination, called PE-termination, which implies 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) and which are *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 [29] and annotate the problematic subterms to be generalised at partial evaluation time.

Related Work

Regarding quasi-termination, we find relatively few works devoted to quasi-termination analysis of functional or logic programs (and no previous work on quasi-termination of functional logic programs). The notion of quasi-termination was originally introduced in term rewriting by Dershowitz [12], where a rewrite derivation is called quasi-terminating when it only contains finitely many different terms. Within logic programming, one of the first approaches is [11], where the authors introduce the notion of *quasi-acceptability*, a sufficient and necessary condition for quasi-termination. This work has been extended in [32].

As for size-change analysis, this approach was originally introduced in [24] in the context of functional programming. The scheme was later adapted to term rewriting in [31].

Finally, regarding the use of quasi-termination analysis for ensuring the termination of offline partial evaluation, there are a few related approaches. Quasi-termination was soon recognised as an essential property to guarantee the termination of partial evaluation (see, e.g., the pioneering work of Holst [20]). In particular, we share many similarities with the approach introduced by Glenstrup and Jones [16], 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 function logic programming is not straightforward because narrowing computations propagate bindings forward in the computations (as logic programming does). As a consequence, several additional conditions should be introduced in order to preserve the termination of partial evaluation. Furthermore, we consider simpler size-change graphs (i.e., the "may-increase" relation of [16] is not used in this work). This may somewhat weaken the power of our size-change analysis, but it could be straightforwardly extended along the lines of [16].

Plan of the Paper

This paper is structured as follows. After providing some preliminary definitions in Sect. 2, we recall the original approach to offline narrowing-driven partial evaluation in Sect. 3. Then, Sect. 4 introduces a quasi-termination analysis based on size-change graphs and states the main result of the paper. Section 5 presents the new annotation procedure and illustrates it with an example. Section 6 describes an experimental evaluation of our approach by using a prototype implementation of the offline partial evaluator. Finally, Sect. 7 concludes and points out some directions for future work. More details and missing proofs can be found in [5].

2 Preliminaries

Term rewriting [6] offers an appropriate framework to model the first-order component of many functional and functional logic programming languages. Therefore, we follow the standard framework of term rewriting for developing our results.

A set of rewrite rules (or oriented equations) $l \to r$ such that l is a nonvariable term and r is a term whose variables appear in l is called a *term rewriting system* (TRS for short); 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 $f(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$. The set of variables appearing in a term t is denoted by $\mathit{Var}(t)$. 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 $l \to r \in \mathcal{R}$. 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 $\mathit{root}(t)$. A term t is *operation-rooted* (resp. *constructor-rooted*) if $\mathit{root}(t) \in \mathcal{D}$ (resp. $\mathit{root}(t) \in \mathcal{C}$). As it is common practice, a *position* p in a term t is represented by a sequence of natural numbers, where ϵ denotes the root position. Positions are used to address the nodes of a term viewed as a tree: $t|_p$ denotes the *subterm* of t at position p and

$t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. 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. A *substitution* σ is a mapping from variables to terms such that its domain $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is finite. The identity substitution is denoted by *id*. A substitution σ is *constructor*, if $\sigma(x)$ is a constructor term for all $x \in \mathcal{D}om(\sigma)$. Term t' is an *instance* of term t if there is a substitution σ with $t' = \sigma(t)$. 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 $s_2 = s_1\theta$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$; furthermore, σ is the *most general unifier* of s and t, denoted by $mgu(s,t)$ if, for every other unifier θ of s and t, we have that $\sigma \leqslant \theta$. In the following, we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

Inductively sequential TRSs [3] 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. Inductive sequentiality is not a limiting condition for programming. In fact, the first-order component of many functional (logic) programs written in, e.g., Haskell, ML or Curry, are inductively sequential.

Example 1. Consider the following rules which define the less-or-equal function on natural numbers (built from *zero* and *succ*):

$$\begin{array}{rcllcl} zero & \leqslant & y & \to & true \\ succ(x) & \leqslant & zero & \to & false \\ succ(x) & \leqslant & succ(y) & \to & x \leqslant y \end{array}$$

This function is inductively sequential since its left-hand sides can be hierarchically organised as follows:

$$\boxed{\mathrm{n}} \leqslant m \Longrightarrow \begin{cases} zero \leqslant m \\ succ(x) \leqslant \boxed{\mathrm{m}} \Longrightarrow \begin{cases} succ(x) \leqslant zero \\ succ(x) \leqslant succ(y) \end{cases} \end{cases}$$

where arguments in a box denote a case distinction (this is similar to the notion of definitional tree in [3]).

The evaluation of terms w.r.t. a TRS is formalised with the notion of *rewriting*. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l \to r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$ (p and R will often be omitted in the notation of a reduction step). The instantiated left-hand side $\sigma(l)$ is called a *redex*. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$. We denote by $\to^+$ the transitive closure of $\to$ and by $\to^*$ its reflexive and transitive closure. Given a TRS $\mathcal{R}$ and a term t, we say that t *evaluates* to s iff $t \to^* s$ and s is in normal form.

Functional *logic* programs mainly differ from purely functional programs in that function calls may contain *free* variables. In order to evaluate such terms containing variables, narrowing nondeterministically instantiates the variables such that a rewrite step is possible [17]. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* iff p is a nonvariable position of t and $\sigma(t) \rightarrow_{p,R} t'$ (we sometimes omit p, R and/or σ when they are clear from the context). The substitution σ is very often the *most general unifier*[1] of $t|_p$ and the left-hand side of (a variant of) R, restricting its domain to $\mathcal{V}ar(t)$. As in proof procedures for logic programming, we assume that the rules of the TRS always contain fresh variables if they are used in a narrowing step. We denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$).

In order to avoid unnecessary computations and to deal with infinite data structures, a demand-driven generation of the search space has been advocated by a number of *lazy* narrowing strategies [15, 26, 27]. Because of its optimality properties w.r.t. the length of derivations and the number of computed solutions, we consider *needed narrowing* [4] in the following.

We say that $s \leadsto_{p,R,\sigma} t$ is a *needed narrowing step* iff $\sigma(s) \rightarrow_{p,R} t$ is a *needed rewrite* step in the sense of Huet and Lévy [21], i.e., in every computation from $\sigma(s)$ to a normal form, either $\sigma(s)|_p$ or one of its *descendants* must be reduced. Here, we are interested in a particular needed narrowing strategy, denoted by λ in [4, Def. 13], which is based on the notion of a *definitional tree* [3] (a hierarchical structure containing the rules of a function definition, which is used to guide the needed narrowing steps). This strategy is basically equivalent to *lazy narrowing* [27] where narrowing steps are applied to the outermost function, if possible, and inner functions are only narrowed if their evaluation is *demanded* by a constructor symbol in the left-hand side of some rule (i.e., a typical call-by-name evaluation strategy). The main difference is that needed narrowing does not compute the *most general unifier* between the selected redex and the left-hand side of the rule but only a unifier. The additional bindings are required to ensure that only "needed" computations are performed (see, e.g., [4]) and, thus, needed narrowing generally computes a smaller search space.

Example 2. Consider again the rules defining function "$\leqslant$" of Example 1. In a term like $t_1 \leqslant t_2$, needed narrowing proceeds as follows: First, t_1 should be evaluated to some *head normal form* (i.e., a free variable or a constructor-rooted term) since all three rules defining "$\leqslant$" have a non-variable first argument. Then,

1. If t_1 evaluates to *zero* then the first rule is applied.
2. If t_1 evaluates to $succ(t_1')$ then t_2 is evaluated to head normal form:
   (a) If t_2 evaluates to *zero* then the second rule is applied.
   (b) If t_2 evaluates to $succ(t_2')$ then the third rule is applied.
   (c) If t_2 evaluates to a free variable, then it is instantiated to a constructor-rooted term, here *zero* or $succ(x)$ and, depending on this instantiation, we proceed as in cases (a) or (b) above.
3. Finally, if t_1 evaluates to a free variable, needed narrowing instantiates it to a constructor-rooted term (*zero* or $succ(x)$). Depending on this instantiation, we proceed as in cases (1) or (2) above.

[1] Some narrowing strategies (e.g., needed narrowing) compute unifiers that are not the most general, see below.

A precise definition of inductively sequential TRSs and needed narrowing is not necessary in this work (the interested reader can find detailed definitions in [3, 4]). In the following, we use *needed narrowing* to refer to the particular strategy λ in [4, Def. 13].

3 A Simple Offline NPE Scheme

In this section, we briefly present the offline approach to NPE from [29]. Given an inductively sequential TRS $\mathcal{R}$, the *first stage* of the process consists in computing the annotated TRS. In [29], annotations were added to those subterms that violate the non-increasingness condition, a simple syntactic characterisation of programs that guarantees the quasi-termination of computations. Nevertheless, annotations can be based on other, more refined, analyses—the goal of this paper—as long as the annotated program still ensures the termination of the specialisation 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*, i.e., the proper partial evaluation, proceeds as follows:

- it takes the annotated TRS, together with an initial term t,
- and constructs its associated (finite) *generalising* 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 generalising needed narrowing tree. Essentially, a *generalising needed narrowing derivation* $s \leadsto_\sigma^* t$ is composed of

a) *proper needed narrowing steps*, for operation-rooted terms with no annotations,
b) *generalisations*, 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 $g(y)$ when $c \in \mathcal{C}$ and $f, g \in \mathcal{D}$.

The substitution in $s \leadsto_\sigma^* t$ is the composition of the substitutions labelling the proper needed narrowing steps of $s \leadsto_\sigma^* t$. Consider, for instance, the following definitions of the addition and product on natural numbers built from *zero* and *succ*:

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

According to [29], this program is not non-increasing because of the nested functions in the right-hand side of the second rule of function *prod*. Therefore, it is annotated as follows:

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

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

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

In contrast, the corresponding computation by *generalising* needed narrowing is quasi-terminating (generalisation steps are denoted by "$\rightsquigarrow^{\bullet}$"):

$$\begin{tikzcd}
 & \underline{add(w,y)} \rightsquigarrow \ldots \\
\underline{prod(x,y)} \rightsquigarrow_{\{x\mapsto succ(x')\}} add(\bullet(prod(x',y)),y) \arrow[ur, squiggly, "\bullet"] \arrow[dr, squiggly, "\bullet"'] & \\
 & \underline{prod(x',y)} \rightsquigarrow \ldots
\end{tikzcd}$$

Our generalisation step is somehow equivalent to the splitting operation of *conjunctive partial deduction* (CPD) of logic programs [10]. While CPD considers conjunctions of atoms, we deal with terms possibly containing nested function symbols. Therefore, flattening a nested function call is basically equivalent to splitting a conjunction (in both cases some information is lost). A similar relation between term generalisation and CPD is also pointed out in [2, 23].

We skip the details of the extraction of residual programs from generalising needed narrowing trees since it is orthogonal to the topic of this paper (a more detailed description can be found in [29]).

4 Ensuring Quasi-Termination with Size-Change Graphs

In this section, we first recall some basic notions on size-change graphs from [31], where the original scheme of [24] is adapted to term rewriting, and, then, we introduce our new approach for ensuring quasi-termination.

A transitive and antisymmetric binary relation $\succ$ is an *order* and a transitive and reflexive binary relation $\succsim$ is a *quasi-order*. A binary relation $\succ$ is *well founded* iff there exist no infinite decreasing sequence $t_0 \succ t_1 \succ t_2 \succ \ldots$ 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.

Size-change graphs are parameterized by a so called reduction pair:

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., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary, where "$\circ$" is defined on binary relations R and R' as follows: $R \circ R' = \{(a,c) \mid (a,b) \in R$ and $(b,c) \in R'\}$). We also require that $s\,R\,t$ implies $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ for all $R \in \{\succsim, \succ\}$ and terms s and t.*

Informally speaking, the restriction $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ above is necessary in order to correctly propagate groundness information through narrowing steps.

Definition 2 (size-change graph). *Let $(\succsim, \succ)$ be a reduction pair. For every rule $f(\overline{s_n}) \to r$ of a TRS $\mathcal{R}$ and every subterm $g(\overline{t_m})$ of r where $g \in \mathcal{D}$, we have a size-change graph as follows:*

- *The graph has n output nodes marked with $\{1_f, \ldots, n_f\}$ and m input nodes marked with $\{1_g, \ldots, m_g\}$.*
- *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 labelled graph $G = (V, W, E)$ where $V = \{1_f, \ldots, n_f\}$ and $W = \{1_g, \ldots, m_g\}$ are the labels of the output and input nodes, respectively, and we have edges $E \subseteq V \times W \times \{\succsim, \succ\}$.

Size-change graphs are used to represent the way each function parameter changes from one call to another, according to a given reduction pair. In order to analyse the termination (or quasi-termination) of a program, it suffices to focus on its loops. For this purpose, we now compute the transitive closure of the size-change relations as follows:

Definition 3 (multigraph, concatenation). *Every size-change graph of $\mathcal{R}$ is a multigraph of $\mathcal{R}$ and if*

$$G = (\{1_f, \ldots, n_f\}, \{1_g, \ldots, m_g\}, E_1)$$

and

$$H = (\{1_g, \ldots, m_g\}, \{1_h, \ldots, p_h\}, E_2)$$

are multigraphs of $\mathcal{R}$ w.r.t. the same reduction pair $(\succsim, \succ)$, then the concatenation

$$G \cdot H = (\{1_f, \ldots, n_f\}, \{1_h, \ldots, p_h\}, E)$$

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. Furthermore, if some of the edges are labelled with "$\succ$", then the edge in E is labelled with "$\succ$" as well. Otherwise, it is labelled with "$\succsim$".

A multigraph G is idempotent if $G = G \cdot G$ (which implies that its input and output nodes are both labelled with $\{1_f, \ldots, n_f\}$ for some f). In the following, we will only focus on the idempotent multigraphs of a program, since they represent its (potential) loops.

Example 3. Consider the following example which computes the reverse of a given list:

$$\begin{array}{llll} \mathit{rev}([\,]) & \to [\,] & \mathit{app}([\,], y) & \to y \\ \mathit{rev}(x : xs) & \to \mathit{app}(\mathit{rev}(xs), x : [\,]) & \mathit{app}(x : xs, y) & \to x : \mathit{app}(xs, y) \end{array}$$

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 $x \in \mathcal{V}ar(t)$, $dv(t,x) \leqslant dv(s,x)$;
– $s \succ t$ 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 [8], $dv(t,x)$, is defined as follows:

$$
\begin{array}{rcll}
dv(c(\overline{t_n}),x) & = & 1 + max(\overline{dv(t_n,x)}) & \text{if } x \in \mathcal{V}ar(c(\overline{t_n})) \\
dv(c(\overline{t_n}),x) & = & -1 & \text{if } x \notin \mathcal{V}ar(c(\overline{t_n})) \\
dv(y,x) & = & 0 & \text{if } x = y \\
dv(y,x) & = & -1 & \text{if } x \neq y \\
dv(t,x) & = & -1 & \text{if } t \text{ is not a constructor term}
\end{array}
$$

with $c \in \mathcal{C}$ a constructor symbol of arity $n \geqslant 0$. The corresponding size-change graphs of this program are the following:

$$\begin{tikzcd}
G_1 : 1_{rev} \arrow[r, "\succ"] & 1_{rev}
\end{tikzcd}
\qquad
\begin{tikzcd}
G_2 : 1_{rev} \arrow[dr, "\succsim"] & 1_{app} \\
 & 2_{app}
\end{tikzcd}
\qquad
\begin{tikzcd}
G_3 : 1_{app} \arrow[r, "\succ"] & 1_{app} \\
2_{app} \arrow[r, "\succsim"] & 2_{app}
\end{tikzcd}$$

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

Definition 4 (PE-termination, PE-terminating TRS). *A needed narrowing computation is PE-terminating if only a finite number of nonvariant function calls (i.e., redexes) have been unfolded. 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 3 and the initial call *rev(xs)*, we have the following needed narrowing derivation:

$$
\begin{array}{ll}
\underline{rev(xs)} \leadsto_{\{xs \mapsto y:ys\}} & app(\underline{rev(ys)}, y : [\,]) \\
 \leadsto_{\{ys \mapsto z:zs\}} & app(app(\underline{rev(zs)}, z : [\,]), y : [\,]) \\
 \leadsto_{\{zs \mapsto w:ws\}} & \cdots
\end{array}
$$

Although this derivation contains an infinite number of different terms, there is only a finite number of nonvariant function calls. Fortunately, this is sufficient to ensure termination in many partial evaluation schemes because they often include some form of memoisation.

Online methods for partial evaluation usually consider a distinction between the so called *local* and *global* control levels. The *local* control should ensure that function (or procedure) calls are not unfolded infinitely, while the *global* control should take care of not unfolding infinitely many function (or procedure) calls. In fact, this distinction can be applied to both online or offline partial evaluators. In some cases, the distinction is made explicit (e.g., in the online partial evaluation scheme for logic programs of [13]) and in some other cases it is left implicit.[2] The

[2] For instance, many partial evaluators for functional programs (see, e.g., [22]) include an algorithm that iteratively (1) takes a function call, (2) performs some symbolic

main difference between these partial evaluators is that, in the online case, both the local and the global control take decisions on-the-fly, while in the offline case all decisions are taken before the actual specialisation starts (i.e., offline partial evaluators mainly follow the program annotations).

In this work, we consider a simple offline partial evaluation procedure as follows:

– *Local control*: here, we stop generalising needed narrowing derivations (i.e., needed narrowing derivations where annotated subterms are replaced by fresh variables) when the selected function call is a variant of a previously reduced function call in the same derivation. Observe that our local control examines the previous function calls in order to determine if a given function call should be unfolded or not. This should not be considered an online strategy but a simple memoisation technique. Furthermore, one could also consider cheaper (though less precise) strategies like, e.g., a depth-k unfolding strategy where narrowing computations stop after k function unfoldings and no variant checking is necessary.
– *Global control*: once the unfolding of a function call stops, the non-constructor terms in the leaves of the generalising needed narrowing tree are fully flattened before adding them to the set of (to be) partially evaluated calls. For instance, given the term $f(g(x), h(y))$, the function calls $f(w_1, w_2)$, $g(x)$ and $h(y)$ are added to the current set of (to be) partially evaluated calls, where w_1, w_2 are fresh variables. This flattening step is required in order for PE-termination to imply the termination of the partial evaluation process.

Now, 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 specialisation time (hence it is ground), while a dynamic parameter is possibly unknown at specialisation time. The output of the BTA must be *congruent* [22]: the value of every static parameter is determined by the values of other static parameters (and thus ultimately by the available input).

In the following, we will also 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. Some closely related notions are that of *rigidity* [7] and *instantiated enough* [25], both defined w.r.t. a so called *norm*. These notions are used in many termination analyses for logic programs (e.g., [9, 14, 25]).

The following theorem states sufficient conditions to ensure PE-termination. The proof of correctness is based on Ramsey's Theorem (see [5]).

evaluations, and (3) extracts from the partially evaluated expression the set of pending function calls—the so-called *successors* of the initial function call—to be processed in the next iteration of the algorithm. Steps (1) and (3) would correspond to the global control while step (2) would correspond to the local control.

Theorem 1. *Let $\mathcal{R}$ be a TRS and $(\succsim, \succ)$ a reduction pair. $\mathcal{R}$ is PE-terminating w.r.t. any linear term if every idempotent multigraph associated to a function f/n contains either*

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

Boundedness of "$\succsim$" in the second case (ii) above is necessary to ensure that no infinite sequences of nonvariant function calls with arguments of the same "size" according to $\succsim$ are allowed. Consider, for instance, an order $\succsim$ which is based on the length of a list, i.e., $t_1 \succsim t_2$ if t_1 and t_2 are lists and the number of elements of t_2 is less than or equal to the number of elements of t_1. In this case, $\succsim$ is not bounded: consider, e.g, the term $[x]$ so that the set $\{s \mid [x] \succsim s\}$ contains infinitely many nonvariant terms. Therefore, one can have infinite sequences of calls with nonvariant arguments where each argument is less than or equal to the previous one in the sequence:

$$f([x]) \leadsto f([succ(x)]) \leadsto f([succ(succ(x))]) \leadsto \ldots$$

with $[x] \succsim [succ(x)] \succsim [succ(succ(x))] \succsim \ldots$.

Example 4. The last condition of Theorem 1 on right-linearity of dynamic variables is required in order to avoid situations like the following one: given the TRS

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

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

$$\begin{array}{lll} \underline{double(x)} & \leadsto_{\{\,\}} & \underline{add(x,x)} \\ & \leadsto_{\{x \mapsto succ(x')\}} & succ(\underline{add(x', succ(x'))}) \\ & \leadsto_{\{x' \mapsto succ(x'')\}} & succ(succ(\underline{add(x'', succ(succ(x'')))})) \\ & \leadsto_{\{x'' \mapsto succ(x''')\}} & \cdots \end{array}$$

which is not quasi-terminating nor PE-terminating.

5 Annotation Procedure

In this section, we introduce our new annotation procedure for offline narrowing-driven partial evaluation. Analogously to [29], rather than requiring source programs to fulfil the conditions of Theorem 1, we use this result to determine which subterms (if any) violate the conditions of this theorem.

The annotation procedure proceeds as follows: it considers every function symbol f/n of the program such that f has an associated idempotent multigraph (i.e., there is a potential loop that involves function f), and performs one of the following actions:

1. if the conditions of Theorem 1 hold, no annotation is added to the program;
2. otherwise, each argument t_j of every function call $f(t_1, \ldots, t_j, \ldots, t_n)$ with no edge $j_f \xrightarrow{R} j_f$, $R \in \{\succsim, \succ\}$, is annotated as follows: $f(t_1, \ldots, \bullet(t_j), \ldots, t_n)$;[3]
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.

Roughly speaking, the correctness of the annotation procedure follows from the following facts:

– Let us consider a function call f/n with an associated idempotent multigraph (note that, by Theorem 1, termination can be ensured by focusing only on those program functions that have an associated idempotent multigraph).
– If the conditions of Theorem 1 hold, we have that from every call $f(t_1, \ldots, t_n)$ to the next call $f(s_1, \ldots, s_n)$ in a computation the following conditions hold:
  - there exists some $i \in \{1, \ldots, n\}$ such that $t_i \succ s_i$ and the i-th argument of f is static (i.e., both t_i and s_i are ground), which means that only finitely many different calls to f can be produced;[4]
  - otherwise, we have that either $t_i \succsim s_i$ or s_i is annotated (and thus generalising needed narrowing replaces this argument by a fresh variable) for all $i = 1, \ldots, n$, which means that only finitely many nonvariant calls to function f can be produced since $\succsim$ is bounded.
– Finally, the only exception to the above reasoning comes from the possible non right-linearity of the program w.r.t. dynamic variables, which is avoided by also annotating all but one such variables, so that situations like the one illustrated by Example 4 are no longer possible.

Let us illustrate the complete process with an example.

Example 5. Consider the well known Ackermann function:

$$\begin{array}{ll} ack(zero, n) & \to succ(n) \\ ack(succ(m), zero) & \to ack(m, succ(zero)) \\ ack(succ(m), succ(n)) & \to ack(m, ack(succ(m), n)) \end{array}$$

First, we compute the size-change graphs of this program (here, we consider the same reduction pair of Example 3):

$$\begin{tikzcd} G_1 : 1_{ack} \arrow[r, "\succ"] & 1_{ack} & G_2 : 1_{ack} \arrow[r, "\succ"] & 1_{ack} & G_3 : 1_{ack} \arrow[r, "\succsim"] & 1_{ack} \\ 2_{ack} & 2_{ack} & 1_{ack} & 2_{ack} & 2_{ack} \arrow[r, "\succ"] & 2_{ack} \end{tikzcd}$$

[3] Analogously to [29], we use a fresh symbol, denoted by $\bullet$, to annotate problematic subterms that should be generalised at partial evaluation time.

[4] This case is similar to the bounded anchoring principle of [16].

where graph G_1 is associated to the second rule and graphs G_2 and G_3 are associated to the third rule. In this example, these graphs coincide with the idempotent multigraphs of the program.

Assume that we want to specialise this program w.r.t. the initial function call $ack(succ(succ(succ(zero))), y)$, i.e., the first argument is static (ground). Clearly, the binding-time analysis returns the division $\{ack \mapsto [S, D]\}$, which means that the first argument of every call to ack is static and the second argument is dynamic. In this case, we have that

– the first condition of Theorem 1 holds for G_1 and G_2 since the first argument of ack is static and there is an edge $1_{ack} \xrightarrow{\succ} 1_{ack}$, and
– the second condition of Theorem 1 holds for G_3 since there is an edge associated to each argument (and $\succsim$ is bounded).

Furthermore, the right-linearity condition also holds since the only repeated occurrences of the same variable are the repeated occurrences of variable m in the third rule. However, no annotation is required in this case since variable m is static according to the output of the binding-time analysis. Therefore, the annotated program coincides with the original one.

Consider now that we want to specialise function ackermann w.r.t. the initial call $ack(x, succ(succ(succ(zero))))$, i.e., the second argument is static (ground). Here, the binding-time analysis returns the division $\{ack \mapsto [D, D]\}$ (because of the nested calls in the third rule). In this case, we have that

– the second condition of Theorem 1 holds for G_3 since there is an edge associated to each argument,
– but, since no condition holds for both G_1 and G_2, we should annotate the second argument of every call to function ack.

The annotated program is thus as follows:

$$\begin{array}{ll}
ack(zero, n) & \to succ(n) \\
ack(succ(m), zero) & \to ack(m, \bullet(succ(zero))) \\
ack(succ(m), succ(n)) & \to ack(m, \bullet(ack(succ(m), \bullet(n))))
\end{array}$$

Observe that there is no violation of the right-linearity condition since one of the repeated occurrences of variable m in the third rule is already inside a $\bullet$.

6 Experimental Evaluation

We have undertaken an implementation of the improved annotation procedure. In particular, we have included the new annotation procedure into an offline partial evaluator for Curry programs [29]. This partial evaluator has been implemented in Curry itself [18]. In its current form, only a subset of Curry is considered. The extension to the remaining features of Curry (e.g., constraints, higher-order functions, built-ins, etc) is planned. The sources of the partial evaluator and a detailed explanation of the benchmarks considered below are publicly

Table 1. Benchmark results

benchmark	codesize	original	simple peval	speedup1	improved peval	speedup2
`ackermann`	739	3363	1077	3.12	688	4.89
`allones`	662	1522	1444	1.05	1452	1.05
`dec_list`	825	589	587	1.00	525	1.12
`gauss`	2904	308	320	0.96	252	1.22
`inc_list`	817	937	834	1.12	730	1.28
`insert_sort`	1005	1953	1280	1.53	1322	1.48
`kmpA*B`	30580	428	298	1.44	227	1.89
`kmpB*A`	30582	86	80	1.08	72	1.21
`power`	794	591	602	0.98	571	1.03
Average	**7656**	**1086**	**725**	**1.36**	**649**	**1.68**

available from

`http://www.dsic.upv.es/users/elp/german/offpeval/`

Table 1 shows the results of some benchmarks. For each benchmark, we show the size (in bytes) of each program (`codesize`), the run time of the original program (`original`), the run time for executing the residual program specialised with the previous offline partial evaluator which uses the simpler annotation procedure (`simple peval`), the run time for executing the residual program produced with the partial evaluator which includes the new annotation procedure (`improved peval`), and the speedup achieved by each partial evaluator; speedups are given by *orig*/*spec*, where *orig* and *spec* are the absolute run times of the original and specialised programs, respectively. Times are expressed in milliseconds and are the average of 10 executions on a 2.4 GHz Linux-PC (Intel Pentium IV with 512 KB cache). Run time input goals were chosen to give a reasonably long overall time. All programs (including the partial evaluators) were executed with the Curry to Prolog compiler of PAKCS [19].

As it can be seen in Table 1, residual programs obtained with the improved partial evaluator run (in the average) 7% faster than the residual programs obtained with the previous partial evaluator. This is not an impressive improvement but demonstrates that the novel annotation procedure is able to produce faster specialised programs. Analysis and specialisation times are not shown because they are generally very small. We note that the current partial evaluator is rather simple (i.e., it follows the simple strategy mentioned in Sect. 4). We expect to produce faster residual programs by improving the control procedures involved in the specialisation phase.

7 Conclusions and Future Work

This work introduced a new annotation procedure for the offline partial evaluation of functional logic programs. This procedure combines the information gathered from a simple binding-time analysis and a size-change analysis [24]. In contrast to previous approaches like [16], several extensions were necessary to cope with the logic component of the considered functional logic language (e.g., the conditions of boundedness and right-linearity in Theorem 1 were not needed in [16]). Preliminary experiments point out the improved performance of a partial evaluator which included the new annotation procedure.

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 idempotent multigraph. The resulting algorithm would be more expensive but also more precise.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2006 for many useful comments and suggestions.

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. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.
3. 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.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation Using Size-Change Graphs. Technical report, Technical University of Valencia, 2006. Available from the following URL: `http://www.dsic.upv.es/users/elp/german/papers.html`.
6. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
7. 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.
8. W.N. Chin and S.C. Khoo. Better Consumers for Program Specializations. *Journal of Functional and Logic Programming*, 1996(4), 1996.
9. Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.
10. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
11. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K.F. Sagonas. Termination Analysis for Tabled Logic Programming. In *Proc. of LOPSTR'97*, pages 111–127. Springer LNCS 1463, 1998.
12. N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

13. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.
14. S. Genaim, M. Codish, J.P. Gallagher, and V. Lagoon. Combining Norms to Prove Termination. In *Proc. of 3rd Int'l Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'02)*, pages 126–138. Springer LNCS 2294, 2002.
15. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.
16. 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.
17. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
18. M. Hanus. Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.
19. M. Hanus (ed.), S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.6.0: The Portland Aachen Kiel Curry System—User Manual. Technical report, University of Kiel, Germany, 2004.
20. C.K. Holst. Finiteness Analysis. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 473–495. Springer LNCS 523, 1991.
21. G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.
22. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
23. L. Lafave and J.P. Gallagher. Constraint-based Partial Evaluation of Rewriting-based Functional Logic Programs. In *Proc. of the 7th Int'l Workshop on Logic Programming Synthesis and Transformation (LOPSTR'97)*, pages 168–188. Springer LNCS 1463, 1997.
24. 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.
25. Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs. In *ICLP*, pages 63–77, 1997.
26. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of 5th Int'l Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.
27. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *J. Logic Programming*, 12(3):191–224, 1992.
28. S. Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.
29. 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.
30. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

31. 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.
32. S. Verbaeten, K. Sagonas, and D. De Schreye. Termination Proofs for Logic Programs with Tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, 2001.