Offline narrowing-driven partial evaluation

J. Guadalupe Ramos, Josep Silva, Germán Vidal
DSIC, Technical University of Valencia.
Camino de Vera S/N, 46022, Valencia.
{guadalupe,jsilva,gvidal}@dsic.upv.es

Abstract

Narrowing-driven partial evaluation (NPE) is a powerful technique for the specialization of rewrite systems. Although it gives good results on small programs, it does not scale up well to realistic problems (e.g., interpreter specialization). In this work, we introduce a faster partial evaluation scheme by ensuring the termination of the process *offline*. For this purpose, we first characterize a class of rewrite systems which are *quasi-terminating*, i.e., the computations performed with needed narrowing—the symbolic computation mechanism of NPE—only contain finitely many different terms (and, thus, partial evaluation terminates). Since this class is quite restrictive, we introduce an annotation algorithm for a broader class of systems so that they behave like quasi-terminating rewrite systems w.r.t. a proposed extension of needed narrowing.

1 Introduction

Narrowing-driven partial evaluation (NPE) is a powerful transformation technique for rewrite systems [2], i.e., for the first-order component of many functional (logic) languages like Haskell or Curry. In general, the narrowing space of a term may be infinite. However, even in this case, NPE may still terminate when the original program is *quasi-terminating* [9] w.r.t. the considered narrowing strategy, i.e., when only finitely many different terms—modulo variable renaming—are computed. The reason is that a finite representation of the infinite computations can be built by replacing repeated terms in a computation by implicit calls to the previously encountered variants (a technique known as *specialization-point insertion* in the partial evaluation literature).

Partial evaluators fall in two main categories, *online* and *offline*, according to the time when termination issues are addressed. Online partial evaluators are usually more precise since they have more information available. However, this extra precision comes at a cost: ensuring termination online is very expensive and, thus, it does not scale up well to realistic problems like interpreter specialization [12].

In this work, we propose a faster NPE scheme by ensuring termination *offline*. Offline partial evaluators usually proceed in two stages: the first stage returns a program that includes annotations to guide the partial computations (e.g., to identify those function calls that can be safely unfolded); then, the second stage (the proper partial evaluation) only needs to obey the annotations and, thus, it is generally much faster than online partial evaluation.

In this work we identify a class of quasi-terminating rewrite systems, called *nonincreasing*, by providing a sufficient condition. This is an interesting result on its own since no previous characterization appears in the literature. Unfortunately, this class is too restrictive and, thus, we also sketch an algorithm that takes an *inductively sequential* system—a much broader class—and returns an *annotated* system. Then, by using a slightly extended needed narrowing relation, in order to consider

an annotated program, we get an appropriate basis for ensuring termination of NPE offline.

2 Preliminaries

We assume familiarity with term rewriting (see, e.g., [7]). For a term rewriting system (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. $\mathcal{R}$ is a *constructor system* 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., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, where $\mathcal{V}$ is a set variables with $\mathcal{F} \cap \mathcal{V} = \varnothing$). The set of variables appearing in a term t is denoted by $\mathcal{V}ar(t)$. 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 rule $l \to r \in \mathcal{R}$. 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 $root(t)$. A term t is *operation-rooted* (resp. *constructor-rooted*) if $root(t) \in \mathcal{D}$ (resp. $root(t) \in \mathcal{C}$). $t|_p$ denotes the *subterm* of t at position p (represented by a sequence of natural numbers, where ϵ is the root position), and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$. 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 *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$.

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 $s = t[\sigma(r)]_p$. The instantiated left-hand side $\sigma(l)$ is called a *redex*. To evaluate terms containing variables, narrowing nondeterministically instantiates the variables such that a rewrite step is possible [10]. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* iff p is a nonvariable position of t and $\sigma(t) \to_{p,R} t'$ (we sometimes omit p, R and/or σ when they are clear from the context). σ is usually the *most general unifier* of $t|_p$ and the left-hand side 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 $\sigma = id$).

As in logic programming, narrowing derivations can be represented by a (possibly infinite) finitely branching *tree*. Formally, given a TRS $\mathcal{R}$ and an operation-rooted term t, a *narrowing tree* for t in $\mathcal{R}$ is a tree satisfying the following conditions: (a) each node of the tree is a term, (b) the root node is t, and (c) if s is a node of the tree then, for each narrowing step $s \leadsto_{p,R,\sigma} s'$, the node has a child s' and the corresponding arc in the tree is labeled with (p, R, σ).

3 Quasi-terminating TRSs

The most recent instance of the NPE scheme [2] is based on needed narrowing (see [4]). *Needed narrowing* [6] is defined on *inductively sequential* TRSs [5], a subclass of left-linear constructor 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 components of many functional (logic) programs written in, e.g., Haskell, ML or Curry, are inductively sequential.

We say that $s \leadsto_{p,R,\sigma} t$ is a *needed narrowing step* iff $\sigma(s) \to_{p,R} t$ is a *needed rewrite* step in the sense of Huet and Lévy [11], 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 partic-

ular needed narrowing strategy, denoted by λ in [6, Def. 13], which is based on the notion of a *definitional tree* [5] (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* [15] 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 outermost 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., [6]). In the following, we use *needed narrowing* to refer to the particular strategy λ in [6, Def. 13].

Since the termination of NPE is ensured when the (possibly partial) computations are quasi-terminating, we introduce a sufficient condition for quasi-termination. First, we need the following preparatory definitions:

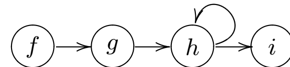
Definition 1 (functional dependencies) *Given a TRS $\mathcal{R}$, its graph of functional dependencies, in symbols $\mathcal{G}(\mathcal{R})$, contains nodes labeled with the function symbols in $\mathcal{D}$ and there is an arrow from node f to node g iff there is a call to g from the right-hand side of some rule in the definition of f.*

Definition 2 (cyclic, noncyclic function) *Let $\mathcal{R}$ be a TRS. A function $f \in \mathcal{D}$ is cyclic if node f belongs to a cycle in $\mathcal{G}(\mathcal{R})$ and it is noncyclic otherwise.*

Example 1 *Consider the following TRS $\mathcal{R}$:*

$$\begin{array}{rcl} f(s(x),y) & \to & g(x,y) \\ g(x,s(y)) & \to & h(x,y) \\ h(0,y) & \to & y \\ h(s(x),y) & \to & c(i(x),h(x,y)) \\ i(x) & \to & x \end{array}$$

where $f,g,h,i \in \mathcal{D}$ and $0,s,c \in \mathcal{C}$. The associated graph of functional dependencies, $\mathcal{G}(\mathcal{R})$, is shown below. Thus, functions f,g and i are noncyclic, while h is cyclic.



Noncyclic functions cannot introduce nonterminating, nor non-quasi-terminating computations. Therefore, we turn our attention to cyclic functions. Following [8], the *depth of a variable x in a constructor term t*, in symbols $dv(t,x)$, is defined as follows:

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

where $c \in \mathcal{C}$.

Definition 3 (nonincreasing function) *Let $\mathcal{R}$ be a left-linear, constructor TRS. A function $f \in \mathcal{D}$ is nonincreasing iff each rule $f(\overline{s_n}) \to r$ in the definition of f fulfills the following conditions:*

1. *the right-hand side does not contain nested defined function symbols, and*
2. *$dv(s_i,x) \geqslant dv(t_j,x)$ for all operation-rooted subterms $g(\overline{t_m})$ in r, where $i \in \{1,\ldots,n\}$, $x \in \mathcal{V}ar(s_i)$, and $j \in \{1,\ldots,m\}$.*

Informally speaking, a nonincreasing function must always *consume* its parameters or leave them unchanged:

Example 2 *A function defined by the rule $f(x,y,s(z)) \to c(g(x),h(z))$, with $s,c \in \mathcal{C}$ and $f,g,h \in \mathcal{D}$, is nonincreasing since the following relations hold:*

$$\begin{array}{rcl} dv(x,x)=0 & \geqslant & 0=dv(x,x) \\ dv(y,y)=0 & \geqslant & -1=dv(x,y) \\ dv(s(z),z)=1 & \geqslant & -1=dv(x,z) \\ dv(x,x)=0 & \geqslant & -1=dv(z,x) \\ dv(y,y)=0 & \geqslant & -1=dv(z,y) \\ dv(s(z),z)=1 & \geqslant & 0=dv(z,z) \end{array}$$

i.e., variable x is just copied, variable y vanishes, and (the depth of) variable z decreases.

We say that a computation is quasi-terminating when it only contains finitely many different terms (modulo variable renaming). Analogously to [9], we say that a TRS is *quasi-terminating for a set of terms T w.r.t. needed narrowing* iff all needed narrowing derivations issuing from the terms in T are quasi-terminating. Now, we give a sufficient condition for quasi-termination:

Definition 4 (nonincreasing TRS) *Let $\mathcal{R}$ be an inductively sequential TRS. $\mathcal{R}$ is nonincreasing iff all functions $f \in \mathcal{D}$ are right-linear and either noncyclic or nonincreasing.*

The restriction to inductively sequential TRSs is not really necessary (i.e., left-linear, constructor TRSs would suffice) but we impose this condition because needed narrowing is only defined for this class of TRSs. On the other hand, right-linearity is not only necessary to guarantee quasi-termination but also for ensuring that no repeated computations are introduced by function unfolding.

Theorem 5 *If $\mathcal{R}$ is a nonincreasing TRS, then $\mathcal{R}$ is quasi-terminating for any linear term w.r.t. needed narrowing.*

4 An offline NPE scheme

The current formulation of the NPE scheme ensures termination *online* (see, e.g., [1, 2, 3]), i.e., appropriate termination tests and generalization operators are used during partial evaluation to guarantee that only a finite number of distinct terms (modulo variable renaming) are computed. This scheme achieves significant optimizations but it is also very expensive and, thus, it does not scale up well to realistic problems. In order to remedy this situation, a faster offline NPE method can be introduced by including a pre-processing stage which *annotates* the expressions that may cause the non-quasi-termination of needed narrowing computations. Our method ensure quasi-termination for the class of programs for which NPE is originally defined: inductively sequential systems. It proceeds in two steps:

Pre-processing stage

Given a TRS $\mathcal{R}$, a term t is annotated by replacing t by $\bullet(t)$. Right hand sides of functions are annotated depending on if they are cyclic or noncyclic, as follows:

noncyclic All occurrences of the same variable, but one (e.g. the leftmost one), is annotated in order to get a linear term.

cyclic Nested function calls and those terms that break the nonincreasing properties are annotated.

For instance, the following inductively sequential system $\mathcal{R}$:

$$\begin{array}{rcl}
main(x) & \to & pow(x, s(s(0))) \\
pow(x, 0) & \to & s(0) \\
pow(x, s(n)) & \to & x \times pow(x, n) \\
0 \times m & \to & 0 \\
s(n) \times m & \to & m + (n \times m) \\
0 + m & \to & m \\
s(n) + m & \to & s(n + m)
\end{array}$$

contains two cyclic functions with nested function calls which are annotated:

$$\begin{array}{rcl}
main(x) & \to & pow(x, s(s(0))) \\
pow(x, 0) & \to & s(0) \\
pow(x, s(n)) & \to & x \times \bullet(pow(x, n)) \\
0 \times m & \to & 0 \\
s(n) \times m & \to & m + \bullet(n \times m) \\
0 + m & \to & m \\
s(n) + m & \to & s(n + m)
\end{array}$$

Observe that it is not necessary to annotate variables x and m (rules 3 and 5) because they are inside of an already annotated term.

Partial evaluation stage

Since partial computations are performed in the NPE scheme by means of needed narrowing, we require an extension of this relation in order to generalize annotated subterms. We call this extension *generalizing needed narrowing*. Roughly speaking, it proceeds as follows:

1. *Generalization*: Here, a set of expressions is computed for further specialization from annotated terms. First,

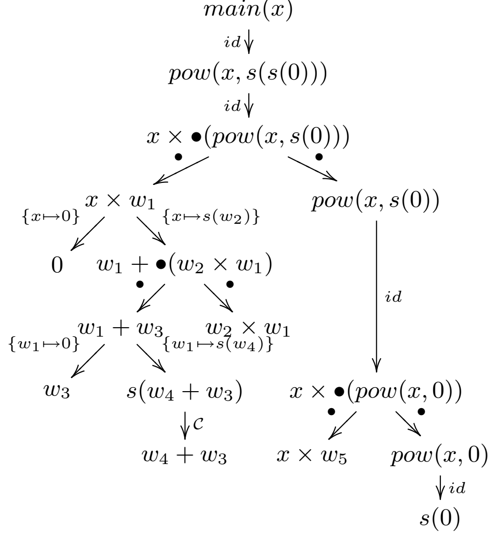


Figure 1: Generalizing needed narrowing tree for $main(x)$

each annotated term is generalized to a free variable; and then, both the term that contained the annotated term and the proper annotated term are added to the set of terms to specialize. For instance, $g(x, \bullet(f(x, \bullet(s(y)))))$ is generalized to $\{g(x, w_1),\ f(x, w_2), s(y)\}$.

2. *Decomposition*: Each constructor root term is reduced to its arguments.

3. *Narrowing*: For operation rooted terms with no annotatios a proper needed-narrowing step is applied.

A generalizing needed narrowing derivation $s \leadsto^*_\sigma t$ is thus composed of *proper* needed narrowing steps (for operation-rooted terms with no annotations), generalizations, and constructor decompositions, where σ is the composition of the substitutions labeling the proper needed narrowing steps.

Given the initial term $main(x)$, generalizing needed narrowing constructs the tree shown in Fig. 1, analogously to narrowing trees.

In order to construct the residual (specialized) program we extract —the so called— *resultants*. Intuitively we extract a resultant $\sigma(s) \to t$ for each proper needed narrowing step $s \leadsto_\sigma t$ in the considered tree.

For instance the resultants from the tree of Fig. 1 are as follows:

$$\begin{array}{rcl} main(x) & \to & pow(x, s(s(0))) \\ pow(x, s(s(0))) & \to & x \times pow(x, s(0)) \\ pow(x, s(0)) & \to & x \times pow(x, 0) \\ pow(x, 0) & \to & s(0) \end{array}$$

together with the original definitions of "$\times$" and "+". Finally, the rules of residual programs usually require a postprocessing of renaming (see [4]). In the example we get:

$$\begin{array}{rcl} main(x) & \to & pow_2(x) \\ pow_2(x) & \to & x \times pow_1(x) \\ pow_1(x) & \to & x \times pow_0(x) \\ pow_0(x) & \to & s(0) \end{array}$$

The original definitions of "$\times$" and "+" need not be renamed. Also, these four rules can easily be simplified by using a standard post-unfolding *transition compression* [14] as follows:

$$main(x) \to x \times (x \times s(0))$$

This example also shows that, despite the annotation of some subterms, the specialization power of NPE is not lost.

5 Related work and conclusion

Despite its relevance, we find in the literature very few works devoted to analyze the termination of narrowing.

One of the most recent approaches to ensure the termination of partial evaluation is based on *size-change graphs* [13]. However, similarly to traditional binding-time analysis, it relies on a clear distinction between *static* (i.e., known) and *dynamic* (i.e., unknown) data, which is hardly present in our NPE scheme. The closest characterizations to ours have been presented by Wadler [16] and Chin and Khoo [8]. Wadler introduced the notion of *treeless* functions in order to ensure the termination of *deforestation* [16]. Chin and Khoo [8], on the other hand, introduced the class of *nonincreasing consumers* and proved that any

set of mutually recursive functions that are nonincreasing consumers can be transformed into an equivalent set of treeless functions, so that deforestation can be applied. This characterization differs from ours mainly because Chin and Khoo do not accept nested function calls nor nonlinear calls in the right-hand side of any program rule. In contrast, we accept arbitrary terms in the right-hand sides of noncyclic funcions, which allows us to cope with a wider range of functions.

In summary, we introduced a novel characterization for TRSs that ensures the quasi-termination of needed narrowing computations. This is a difficult problem of independent interest that has not been tackled before. Since the considered class of TRSs is too restrictive, we then considered inductively sequential programs (a much broader class) and sketched an algorithm that annotates those subterms which may cause the non-quasi-termination of needed narrowing. Finally we discussed a generalizing extension of needed narrowing which is guided by program annotations.

References

[1] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

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

[3] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.

[4] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *Theory and Practice of Logic Programming*, 2005. To appear (http://www.dsic.upv.es/users/elp/german/papers.html).

[5] 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.

[6] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[7] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[8] W.N. Chin and S.C. Khoo. Better Consumers for Program Specializations. *Journal of Functional and Logic Programming*, 1996(4), 1996.

[9] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

[10] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[11] 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.

[12] N.D. Jones. Transformation by Interpreter Specialisation. *Science of Computer Programming*, 52:307–339, 2004.

[13] N.D. Jones and A. Glenstrup. Partial Evaluation Termination Analysis and Specialization-Point Insertion. *ACM TOPLAS*, 2005. To appear.

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

[15] 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.

[16] P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.