Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Programs [*]

J. Guadalupe Ramos [†]
I.T. La Piedad, Av. Tecnológico 2000,
La Piedad, Mich., México
guadalupe@dsic.upv.es

Josep Silva Germán Vidal
DSIC, Tech. University of Valencia,
Camino de Vera s/n, E-46022 Valencia, Spain
{jsilva,gvidal}@dsic.upv.es

Abstract

Narrowing-driven partial evaluation is a powerful technique for the specialization of (first-order) functional and functional logic programs. However, 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 programs which are *quasi-terminating*, i.e., the computations performed with needed narrowing—the symbolic computation mechanism of narrowing-driven partial evaluation—only contain finitely many different terms (and, thus, partial evaluation terminates). Since this class is quite restrictive, we also introduce an annotation algorithm for a broader class of programs so that they behave like quasi-terminating programs w.r.t. an extension of needed narrowing. Preliminary experiments are encouraging and demonstrate the usefulness of our approach.

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

General Terms algorithms, performance, theory

Keywords narrowing, quasi-termination, offline partial evaluation

1. Introduction

Given a program and an initial call (usually containing some known data), the aim of a partial evaluator is the construction of a new, residual program specialized for this call. The essential component of many partial evaluators is a technique to compute a *finite* representation of the—generally *infinite*—computation space for the initial call, so that a (hopefully more efficient) residual program can be extracted from this representation. For instance, given a program $\mathcal{P}$ and an initial function call, $f(t, x)$, where t is a known input data and x is a free variable, a trivial partial evaluator may simply return the residual program $\mathcal{P}' = \mathcal{P} \cup \{f_t(x) = f(t, x)\}$ containing a specialized version f_t of function f. While the correctness of this trivial partial evaluator is obvious, it is also clear that no efficiency improvement can be achieved. A challenge in partial evaluation is the definition of techniques for constructing nontrivial yet finite representations of the computation space of a program so that efficient residual programs can be extracted.

Narrowing-driven partial evaluation (NPE) is a powerful specialization technique for rewrite systems [2], i.e., for the first-order component of many functional (logic) languages like Haskell [40] or Curry [21]. Higher-order features can still be modeled by using an explicit application operator, i.e., by defunctionalization [42]; this strategy is used in several implementations of lazy functional logic languages, like the Portland-Aachen-Kiel Curry System (PAKCS [27]) and the Münster Curry Compiler (MCC [38]). Although NPE can be seen as a traditional partial evaluation scheme for program specialization, it can also achieve more powerful optimizations like deforestation [46], elimination of higher-order functions (represented in a first-order setting by defunctionalization), etc. A narrowing-driven partial evaluator is currently integrated into the PAKCS environment for Curry (an experimental evaluation can be found in [1]).

At the core of the NPE scheme we find a method to construct a finite representation of a (usually) infinite computation space. To be precise, given a rewrite system $\mathcal{R}$ and a term t, NPE constructs a *finite* representation of all possible reductions of t—and any of its instances if it contains variables—in $\mathcal{R}$, and then extracts a new, often simpler and more efficient, rewrite system. Since t may contain variables, some form of *symbolic computation* is required. In NPE, a refinement of *narrowing* [43] is used to perform symbolic computations, being needed narrowing [9] the strategy that presents better properties (as shown in [5]). 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* [19] 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).

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 more information is available. For instance, the original NPE scheme (which follows the online approach) considers a variant of the Kruskal

[*] This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-00231 and HU 2003-0003, by the *Generalitat Valenciana* GRUPOS03/025, and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

[†] This research was done while visiting the T.U. Valencia (Spain), funded by grants SEIT-ANUIES 20020V and DGEST "beca-comisión" (México).

tree condition called "homeomorphic embedding" [34] to ensure the termination of the process [4]: if a term embeds some previous term in the same narrowing computation, some form of generalization—usually the *most specific generalization* operator—is applied and partial evaluation is restarted with the generalized terms. However, this extra precision comes at a cost: the homeomorphic embedding tests, together with the associated generalizations, make NPE very expensive and, thus, it does not scale up well to realistic problems like interpreter specialization [30] or compiler generation by self-application [22].

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 evaluators. Let us remark that, in the NPE framework, the so-called static/dynamic distinction is hardly present. Indeed, in a functional logic setting, one can require the (nondeterministic) evaluation of terms containing free variables at runtime. Therefore, in contrast to traditional binding-time analysis, the first stage of our offline partial evaluation scheme ensures termination even if all arguments are dynamic (i.e., unknown).

Contributions. The main contributions of this work are the following. First, 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 introduce an algorithm that takes an *inductively sequential* program—a much broader class—and returns an *annotated* program. Then, we define an extended needed narrowing relation, *generalizing needed narrowing*, in which annotated subterms are generalized. We prove that computations with this relation are quasi-terminating for annotated inductively sequential programs and, thus, it forms an appropriate basis for ensuring termination of NPE offline. Finally, we explain how our new developments can be integrated into the NPE scheme and prove its correctness and termination. Preliminary experiments with a prototype implementation of the new partial evaluation method are encouraging and demonstrate the usefulness of our approach.

Plan of the paper. This paper is structured as follows. Section 2 presents an informal overview of our approach to the partial evaluation of functional (logic) programs. Then, after providing some preliminary definitions in Sect. 3, we introduce the characterization of nonincreasing programs in Sect. 4. Section 5 presents an algorithm for annotating inductively sequential programs, together with an extension of needed narrowing that exploits program annotations to ensure quasi-termination. Section 6 describes the scheme of the complete offline NPE method and includes a summary of the experimental evaluation. Finally, Sect. 7 includes a comparison with related work and concludes. Proofs of technical results can be found in [41].

2. Partial Evaluation

In this section, we present an informal overview of our approach to the partial evaluation of functional (logic) programs.

In our setting, the input for the partial evaluator are a rewrite system—a typical first-order functional program—and an initial function call, which usually contains some known data (the so-called *static* data). Consider, for instance, the following rewrite system:

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

where natural numbers are built from $zero$ and $succ$. We can partially evaluate this program w.r.t. the initial term $inc(x)$ in order to obtain a direct definition for function inc (i.e., by specializing function add to have a fixed first argument $succ(zero)$).

Both online and offline partial evaluators should construct some form of *symbolic execution tree*. It is *symbolic* because terms may contain free variables and, thus, a non-standard, symbolic execution mechanism if often required. Furthermore, we get a *tree* structure since the evaluation of function calls containing free variables generally require nondeterministic evaluation steps.

The construction of such a symbolic execution tree is explicit in some partial evaluation techniques (like, e.g., positive supercompilation [44] or narrowing-driven partial evaluation [2]). In some other techniques, the construction of a symbolic execution tree is only implicit. For instance, many partial evaluators for functional programs (see, e.g., [32]) include an algorithm that iteratively (1) takes a function call, (2) performs some symbolic 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. Observe that, if we add an arrow from each term to its set of successors, we would also obtain a sort of symbolic execution tree.

In order to perform symbolic computations in a functional context, an extension of the standard semantics is required for evaluating terms with free variables. Here, the choice of *narrowing* [43] as symbolic computation mechanism arise naturally since it combines functional reductions with the instantiation of free variables (see the next section for a formal definition). Moreover, in the setting of functional logic programming, the same operational principle can be used for performing both standard and symbolic computations [2] (similarly to the partial evaluation of logic programs, where SLD-resolution is used for both standard and symbolic computations [36]).

For instance, the symbolic execution tree for the initial call $inc(x)$ w.r.t. the program above is as follows (the selected function call is underlined):

$$\begin{tikzcd} \underline{inc(x)} \arrow[d] \\ \underline{add(succ(zero), x)} \arrow[d] \\ succ(\underline{add(zero, x)}) \arrow[d] \\ succ(x) \end{tikzcd}$$

Here, no instantiation of free variables was necessary; therefore, we get a deterministic evaluation. The associated residual program can easily be extracted from the root-to-leaf computations of the symbolic execution tree. In the example above, we get the single rule

$$inc(x) \;\to\; succ(x)$$

In practice, partial evaluators often include a sort memoization technique to avoid the repeated evaluation of the same term (modulo variable renaming). Consider the following definition:

$$inc'(x) \;\to\; add(x, succ(zero))$$

Although the symbolic execution tree for $inc'(x)$ is infinite:

$$\begin{tikzcd}
 & \underline{inc'(x)} \arrow[d] & \\
 & \underline{add(x, succ(zero))} \arrow[dl, "\{x\mapsto zero\}"'] \arrow[dr, "\{x\mapsto succ(y)\}"] & \\
succ(zero) & & succ(\underline{add(y, succ(zero))}) \arrow[d, dotted, no head] \\
 & & \infty
\end{tikzcd}$$

a partial evaluator would terminate in this example since the function call $add(y, succ(zero))$ is a variant of $add(x, succ(zero))$. In the tree above, the arrows issuing from $add(x, succ(zero))$ are labeled with the computed substitution by narrowing, i.e., a substitution such that, when applied to $add(x, succ(zero))$, allows a reduction step with the standard semantics. Here, the associated residual program is the following:

$$\begin{array}{rcl}
inc'(x) & \to & add(x, succ(zero)) \\
add(zero, succ(zero)) & \to & succ(zero) \\
add(succ(y), succ(zero)) & \to & succ(add(y, succ(zero)))
\end{array}$$

In this case, we have a residual rule associated to the first evaluation step and two residual rules associated to each nondeterministic step (here, the associated bindings are applied to the left-hand sides of the rules).

The finiteness of the symbolic execution tree can be guaranteed when symbolic computations are *quasi-terminating*, i.e., when only finitely many different terms—modulo variable renaming—are obtained. Note that, even if the considered program is terminating w.r.t. the standard semantics, the symbolic execution mechanism may give rise to both non-terminating and non-quasi-terminating computations. Consider the following function definition:

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

Given the initial call $double(x)$, the associated symbolic tree is infinite (we use $succ^2(zero)$ as a shorthand for $succ(succ(zero))$):

$$\begin{tikzcd}
 & \underline{double(x)} \arrow[d] & \\
 & \underline{prod(x, succ^2(zero))} \arrow[dl, "\{x\mapsto zero\}"'] \arrow[dr, "\{x\mapsto succ(y)\}"] & \\
zero & & add(\underline{prod(y, succ^2(zero))}, succ^2(zero)) \arrow[dl, "\{y\mapsto zero\}"'] \arrow[dd, "\{y\mapsto succ(z)\}"] \\
 & \underline{add(zero, succ^2(zero))} \arrow[dl] & \\
succ^2(zero) & & add(add(\underline{prod(z, succ^2(zero))}, succ^2(zero))) \arrow[d, dotted, no head] \\
 & & \infty
\end{tikzcd}$$

In order to always ensure the finiteness of symbolic execution trees, one should consider a *generalization* operation on terms. The decision on which terms should be generalized can be taken in a pre-processing stage (the case of offline partial evaluation) or during partial evaluation itself (as in online partial evaluation). Online partial evaluators are usually more precise since more information is available for deciding whether generalization is necessary or not. In contrast, offline partial evaluators are less precise but generally much faster since the partial evaluation stage should only follow the annotations given by a pre-processing analysis (the so-called binding-time analysis).

In the example above, termination can be guaranteed by generalizing the second call to function $prod$ as follows:

$$\begin{tikzcd}
 & \underline{double(x)} \arrow[d] & \\
 & \underline{prod(x, succ^2(zero))} \arrow[dl, "\{x\mapsto zero\}"'] \arrow[dr, "\{x\mapsto succ(y)\}"] & \\
zero & & add(\boxed{prod(y, succ^2(zero))}, succ^2(zero)) \arrow[dl] \arrow[dr] & \\
 & \underline{add(w, succ^2(zero))} \arrow[dl, "\{w\mapsto zero\}"'] \arrow[dr, "\{w\mapsto succ(z)\}"] & & \underline{prod(y, succ^2(zero))} \\
succ^2(zero) & & succ(\underline{add(z, succ^2(zero))}) &
\end{tikzcd}$$

Now, the symbolic execution tree is kept finite since all the leaves are values (i.e., they do not contain function calls, like $zero$ and $succ^2(zero)$) or contain a function call that is a variant of a previous function call in the tree (the case of $prod(y, succ^2(zero))$ and $add(z, succ^2(zero))$, which are variants of $prod(x, succ^2(zero))$ and $add(w, succ^2(zero))$, respectively).

From this symbolic execution tree, the following residual program can be extracted:

$$\begin{array}{rcl}
double(x) & \to & prod(x, succ^2(zero)) \\
prod(zero, succ^2(zero)) & \to & zero \\
prod(succ(y), succ^2(zero)) & \to & add(prod(y, succ^2(zero))) \\
add(zero, succ^2(zero)) & \to & succ^2(zero) \\
add(succ(z), succ^2(zero)) & \to & succ(add(z, succ^2(zero)))
\end{array}$$

In the remainder of this paper, we present a systematic approach to the *offline* partial evaluation of inductively sequential systems.

3. Foundations

Term rewriting [11] offers an appropriate framework to model the first-order component of many functional (logic) programming languages.[1] Therefore, in the remainder of this paper we follow the standard framework of term rewriting for developing our results.

3.1 The Source Language

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 $\mathcal{F} \cap \mathcal{V} = \varnothing$.

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 $\mathcal{V}ar(t)$. 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

[1] Nevertheless, higher-order features can be modeled by using an explicit application operator, i.e., by defunctionalization [42].

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 $root(t)$. A term t is *operation-rooted* (resp. *constructor-rooted*) if $root(t) \in \mathcal{D}$ (resp. $root(t) \in \mathcal{C}$). 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 $\mathcal{V}ar(t) = \varnothing$. 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 *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. In the following, we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

Inductively sequential TRSs [6] 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.[2] Also, the class of inductively sequential programs provides for optimal computations both in functional and functional logic programming [6, 9].

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

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

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

$$\boxed{n} \leqslant m \Longrightarrow \left\{ \begin{array}{l} zero \leqslant m \\ succ(x) \leqslant \boxed{m} \Longrightarrow \left\{ \begin{array}{l} succ(x) \leqslant zero \\ succ(x) \leqslant succ(y) \end{array} \right. \end{array} \right.$$

where arguments in a box denote a case distinction (this is similar to the notion of definitional tree in [6]).

3.2 Semantics

The evaluation of terms w.r.t. a TRS is formalized 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 [25]. 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 very often the *most general unifier*[3] 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 $\sigma = id$).

Due to the presence of free variables, a term may be reduced to different values after instantiating these variables to different terms. Given a narrowing derivation $t_0 \leadsto^*_\sigma t_n$, we say that t_n is a computed *value* and σ is a computed *answer* for t_0.

EXAMPLE 2. *Consider the following definition of function "+":*

$$\begin{array}{rcll} zero + y & \to & y & (R_1) \\ succ(x) + y & \to & succ(x + y) & (R_2) \end{array}$$

Given the term $x + succ(zero)$, narrowing nondeterministically performs the following derivations:

$$\begin{array}{ll} x + succ(zero) & \\ \quad \leadsto_{\epsilon,R_1,\{x \mapsto zero\}} & succ(zero) \\ x + succ(zero) & \\ \quad \leadsto_{\epsilon,R_2,\{x \mapsto succ(y_1)\}} & succ(y_1 + succ(zero)) \\ \quad \leadsto_{1,R_1,\{y_1 \mapsto zero\}} & succ(succ(zero)) \\ x + succ(zero) & \\ \quad \leadsto_{\epsilon,R_2,\{x \mapsto succ(y_1)\}} & succ(y_1 + succ(zero)) \\ \quad \leadsto_{1,R_2,\{y_1 \mapsto succ(y_2)\}} & succ(succ(y_2 + succ(zero))) \\ \quad \leadsto_{1.1,R_1,\{y_2 \mapsto zero\}} & succ(succ(succ(zero))) \\ \ldots & \end{array}$$

Therefore, $x + succ(zero)$ nondeterministically computes the following values (here, we use $succ^n$ as a shorthand for n applications of function $succ$):

- *$succ(zero)$ with answer $\{x \mapsto zero\}$,*
- *$succ^2(zero)$ with answer $\{x \mapsto succ(zero)\}$,*
- *$succ^3(zero)$ with answer $\{x \mapsto succ^2(zero)\}$, etc.*

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).

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 [23, 37, 39]. Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* [9] is currently the best lazy narrowing strategy.

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 [29], 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 [9, Def. 13], which is based on the notion of a *definitional tree* [6] (a hierarchical structure containing the rules of a function definition, which is used to guide the needed narrowing steps). This

[2] Curry also accepts *overlapping* inductively sequential systems. This class extends inductively sequential systems with a disjunction operator which introduces additional don't-know nondeterminism. Nevertheless, the nice properties of inductive sequentiality carry over to overlapping systems too.

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

strategy is basically equivalent to *lazy narrowing* [39] 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., [9]) and, thus, needed narrowing generally computes a smaller search space.

EXAMPLE 3. *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.*

Let us note that needed narrowing is only defined on operation-rooted terms, i.e., a needed narrowing derivation stops when a head normal form (a *value* in our context) is obtained. This is not a restriction since the evaluation to normal form can be reduced to a sequence of head normal form computations (see [26]).

A precise definition of inductively sequential TRSs and needed narrowing is not necessary in this work (the interested reader can find detailed definitions in [6, 9]). In the following, we use *needed narrowing* to refer to the particular strategy λ in [9, Def. 13].

4. Ensuring Quasi-Termination w.r.t. Needed Narrowing

In the NPE framework [2], narrowing is used as a symbolic computation mechanism to perform partial computations. Roughly speaking, given a program $\mathcal{R}$ and an initial term t, partial evaluation proceeds by constructing a narrowing tree for t in $\mathcal{R}$ with the additional constraint of not evaluating terms that are variants of previous terms in the computation. Therefore, the termination of the NPE process can be ensured when all narrowing computations are quasi-terminating. Analogously to Holst [28], we say that a computation is *quasi-terminating* when it only contains finitely many different terms (modulo variable renaming).

The most recent instance of the NPE scheme is based on needed narrowing [5]. However, while the original NPE scheme guarantees that computations are quasi-terminating *online* (by applying appropriate termination tests and generalization operators), in this work we introduce a sufficient condition for TRSs so that needed narrowing computations are always quasi-terminating. First, we need the following preparatory definitions:

DEFINITION 4 (graph of 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 5 (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 6. *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 $f, g, h, i \in \mathcal{D}$ are defined functions and $0, s, c \in \mathcal{C}$ are constructor. The associated graph of functional dependencies, $\mathcal{G}(\mathcal{R})$, is as follows:

$$\begin{tikzcd} f \arrow[r] & g \arrow[r] & h \arrow[loop above] \arrow[r] & i \end{tikzcd}$$

Thus, functions f, g, and i are noncyclic, while h is cyclic.

Clearly, noncyclic functions cannot introduce nonterminating (nor non-quasi-terminating) computations as long as the cyclic functions do not introduce them. Thus, we turn our attention to cyclic functions. Following [13], the *depth of a variable x in a constructor term t*, in symbols $dv(t,x)$, is defined as follows:

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

where $c \in \mathcal{C}$ is a constructor term with arity $n \geqslant 0$.

The following definition introduces the notion of *nonincreasing* function, i.e., a function that always *consume* its parameters or leave them unchanged:

DEFINITION 7 (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 (i.e., defined function symbols that occur inside other 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 $j \in \{1,\ldots,m\}$.*

EXAMPLE 8. *A function defined by the single 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:*

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

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

Analogously to [19], 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 9 (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 (see below) but also for ensuring that no repeated computations are introduced by function unfolding.

THEOREM 10. *If $\mathcal{R}$ is a nonincreasing TRS, then $\mathcal{R}$ is quasi-terminating for any linear term w.r.t. needed narrowing.*

We note that there is no clear relation between quasi-termination w.r.t. needed narrowing and related conditions in term rewriting. Consider, for instance, the following TRS:

$$f(0,y) \;\to\; y \qquad f(s(x),y) \;\to\; f(x,s(y))$$

which is not nonincreasing, where $0, s \in \mathcal{C}$ and $f \in \mathcal{D}$. This TRS is trivially terminating w.r.t. rewriting since the first parameter of function f strictly decreases with each recursive call. However, it is not quasi-terminating w.r.t. needed narrowing as witnessed by the following (infinite) computation:

$$\begin{array}{lll} f(x,y) & \leadsto_{\{x\mapsto s(x')\}} & f(x',s(y)) \\ & \leadsto_{\{x'\mapsto s(x'')\}} & f(x'',s(s(y))) \\ & \leadsto & \ldots \end{array}$$

Related notions like size-change termination [33] (adapted to TRSs in [45]) are equally not useful for ensuring (quasi-)termination w.r.t. needed narrowing, since they only ensure that *some* parameter decreases (but not all of them), which is not useful in our context where all parameters may be unknown (i.e., free variables). Right-linearity is an essential requirement even for the simplest functions. Consider, for example, the following nonincreasing functions:

$$f(0,y) \to y \qquad f(s(x),y) \to f(x,y) \qquad g(x) \to f(x,x)$$

where $0, s \in \mathcal{C}$ and $f, g \in \mathcal{D}$. This is not a nonincreasing TRS since function g is not right-linear. Thus, quasi-termination w.r.t. needed narrowing is not ensured:

$$\begin{array}{lllll} g(x) & \leadsto_{id} & f(x,x) & \leadsto_{\{x\mapsto s(x')\}} & f(x',s(x')) \\ & & & \leadsto_{\{x'\mapsto s(x'')\}} & f(x'',s(s(x''))) \\ & & & \leadsto & \ldots \end{array}$$

Clearly, the use of *needed* narrowing is also crucial, i.e., quasi-termination for other narrowing strategies (e.g., *innermost* narrowing) is not guaranteed. For instance, given the following quasi-terminating TRS:

$$\begin{array}{lllllll} f(x) & \to & g(h(x)) & \qquad & h(0) & \to & 0 \\ g(x) & \to & x & & h(s(x)) & \to & s(h(x)) \end{array}$$

needed narrowing is quasi-terminating while an *innermost* strategy would produce the following non-quasi-terminating derivation:

$$\begin{array}{lllll} f(x) & \leadsto_{id} & g(h(x)) & \leadsto_{\{x\mapsto s(x')\}} & g(s(h(x'))) \\ & & & \leadsto_{\{x'\mapsto s(x'')\}} & g(s(s(h(x'')))) \\ & & & \leadsto & \ldots \end{array}$$

The closest characterizations to ours have been presented by Wadler [46] and Chin and Khoo [13]. Wadler introduced the notion of *treeless* functions in order to ensure the termination of *deforestation* [46]. Treeless functions are a subclass of our nonincreasing functions where, additionally, all function calls in the right-hand sides of the rules can only have variable arguments. Chin and Khoo [13] 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 in two points. Firstly, Chin and Khoo only require linear function calls in the right-hand sides of the rules (rather than being linear the entire right-hand sides, as we impose). This relaxed definition, however, is not safe in our context. Consider, e.g., the following nonincreasing consumers according to Chin and Khoo [13]:

$$f(x) \to c(g(x),x) \qquad g(s(x)) \to g(x) \qquad h(c(s(x),y)) \to x$$

where $c, s \in \mathcal{C}$ and $f, g, h \in \mathcal{D}$. Here, given the initial term $h(f(x))$, needed narrowing has an infinite derivation which is not quasi-terminating:

$$\begin{array}{lll} h(f(x)) & \leadsto_{id} & h(c(g(x),x)) \\ & \leadsto_{\{x\mapsto s(x')\}} & h(c(g(x'),s(x'))) \\ & \leadsto_{\{x'\mapsto s(x'')\}} & h(c(g(x''),s(s(x'')))) \\ & \leadsto & \ldots \end{array}$$

And, secondly, Chin and Khoo do not accept nested function calls in the right-hand side of any program rule. In contrast, we accept arbitrary (linear) terms in the right-hand sides of noncyclic functions, which allows us to cope with a wider range of functions.

5. From Online to Offline NPE

The current formulation of the NPE scheme ensures termination *online* (see, e.g., [1, 2, 4]), 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. As mentioned before, this scheme achieves significant optimizations but it is also very expensive (in terms of both time and space consumption) and, thus, it does not scale up well to realistic problems. In order to remedy this situation, in this section we introduce a faster NPE method which ensures termination *offline* by including a pre-processing stage based on the notion of nonincreasing TRS.

In principle, a naive NPE method could restrict the source programs to nonincreasing TRSs and, then, apply neither termination tests nor generalizations during partial evaluation since needed narrowing computations would be quasi-terminating (cf. Theorem 10). This would give rise to a very fast NPE tool—only equality tests modulo variable renaming would be required—but, unfortunately, the class of acceptable programs would be too restrictive.

Therefore, we now consider the class of programs for which NPE is originally defined: inductively sequential programs—a much broader class of TRSs—and define an algorithm that *annotates* the expressions which may cause the non-quasi-termination of needed narrowing computations. These annotations will be later used by an extended needed narrowing relation in order to *generalize* problematic subterms. We let $\mathcal{F}_\bullet = \mathcal{F} \cup \{\bullet\}$, where $\bullet \notin \mathcal{F}$ is a fresh symbol. Given a TRS $\mathcal{R}$, a term t is annotated by replacing t by $\bullet(t)$. The following auxiliary functions will be useful to manipulate annotated terms:

$$\begin{array}{rcll} gen(x) & = & x & \text{if } x \in \mathcal{V} \\ gen(h(\overline{t_n})) & = & h(\overline{gen(t_n)}) & \text{if } h \in \mathcal{F}, n \geqslant 0 \\ gen(\bullet(t)) & = & y & \text{where } y \in \mathcal{V} \text{ is a fresh variable} \end{array}$$

i.e., given an annotated term $t \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$, the expression $gen(t) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ returns a generalization of t by replacing annotated subterms by fresh variables.

$$\begin{array}{rcll} aterms(x) & = & \varnothing & \text{if } x \in \mathcal{V} \\ aterms(h(\overline{t_n})) & = & \bigcup_{i=1}^{n} aterms(t_i) & \text{if } h \in \mathcal{F}, n \geqslant 0 \\ aterms(\bullet(t)) & = & \{t\} \cup aterms(t) & \end{array}$$

Here, the expression $aterms(t) \subseteq \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$ returns the set of annotated subterms (possibly containing annotations) in $t \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$.

Let us illustrate the use of functions gen and $aterms$ with some simple examples:

$$\begin{array}{lcl} gen(f(x,g(h(y)))) & = & f(x,g(h(y))) \\ gen(f(x,\bullet(g(h(y))))) & = & f(x,w) \\ gen(f(x,\bullet(g(\bullet(h(y)))))) & = & f(x,w) \end{array}$$

$$\begin{array}{lcl} aterms(f(x,g(h(y)))) & = & \{\,\} \\ aterms(f(x,\bullet(g(h(y))))) & = & \{g(h(y))\} \\ aterms(f(x,\bullet(g(\bullet(h(y)))))) & = & \{g(\bullet(h(y))),\ h(y)\} \end{array}$$

The following definition introduces our transformation to annotate inductively sequential programs. Intuitively speaking, we annotate those arguments of the topmost operation-rooted subterms in the right-hand sides of the rules that either contain defined function symbols or break the nonincreasing property; then, every annotated subterm is treated similarly to the original right-hand side (thus, nested annotations are possible); finally, all repeated occurrences but one of the same variable are annotated. Formally,

DEFINITION 11 ($ann(\mathcal{R})$). *Let $\mathcal{R} = \{l_i \to r_i \mid i = 1,\ldots,k\}$ be an inductively sequential TRS over $\mathcal{F}$. The annotated TRS, $ann(\mathcal{R})$, over $\mathcal{F}_\bullet$ is given by the set of rules $\{l_i \to r'_i \mid i = 1,\ldots,k\}$ where r'_i, $i = 1,\ldots,k$, is computed as follows:*

1. *If $root(l_i)$ is a noncyclic function, then r'_i is obtained from r_i by annotating all occurrences of the same variable but one (e.g., the leftmost one), so that $gen(r'_i)$ is a linear term.*
2. *If $root(l_i)$ is cyclic, then r'_i is obtained from $qs(l_i,r_i)$ by annotating the least number of variables such that $gen(t)$ becomes linear for all $t \in \{qs(l_i,r_i)\} \cup aterms(qs(l_i,r_i))$. The definition of auxiliary function qs is shown in Fig. 1.*

Intuitively speaking, auxiliary function qs ignores constructor symbols until an operation-rooted subterm $f(t_1,\ldots,t_n)$ is found. Then, for each argument t_i, it proceeds (by calling qs') as follows:

- if t_i is a constructor term and all variables fulfill the nonincreasing property, then t_i remains unchanged;
- otherwise, the considered subterm, t_i, is annotated and the process is restarted for t_i.

Trivially, for any nonincreasing TRS $\mathcal{R}$, we have $ann(\mathcal{R}) = \mathcal{R}$. Furthermore, if $\mathcal{R}$ is an inductively sequential TRS so is $ann(\mathcal{R})$, since the left-hand sides of the rules are not modified.

Note that Definition 11 is nondeterministic since it does not fix which variable occurrence should not be annotated when there are repeated occurrences of the same variable. In some cases, this decision can dramatically affect the result of a partial evaluation (see Sect 6.2). This situation could be improved in some cases by allowing the programmer to choose the (static) variable that should not be annotated.

EXAMPLE 12. *Consider the following inductively sequential program $\mathcal{R}$:*

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

where $f,g \in \mathcal{D}$ and $0,s \in \mathcal{C}$. The annotated TRS, $ann(\mathcal{R})$, is as follows:

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

Observe that repeated occurrences of x in the second rule should not be annotated since

$$aterms(g(x,\bullet(f(x,\bullet(s(y)))))) = \{f(x,\bullet(s(y))), s(y)\}$$

and, hence, $gen(t)$ is linear for all

$$t \in \{g(x,\bullet(f(x,\bullet(s(y))))), f(x,\bullet(s(y))), s(y)\}$$

i.e., $g(x,w_1)$, $f(x,w_2)$, and $s(y)$ are linear terms, where w_1 and w_2 are fresh variables.

Since partial computations are computed in the NPE scheme by means of needed narrowing, we now extend this relation in order to generalize annotated subterms (thus ensuring the termination of the partial evaluation process).

DEFINITION 13 (generalizing needed narrowing). *Let $\mathcal{R}$ be an annotated inductively sequential TRS over $\mathcal{F}_\bullet$. The generalizing needed narrowing relation, in symbols $\rightsquigarrow$, is defined as the least relation satisfying*

(needed narrowing)
$$\frac{s \rightsquigarrow_{p,R,\sigma} t}{s \rightsquigarrow_\sigma t} \qquad \text{if } root(s) \in \mathcal{D} \text{ and } s \in \mathcal{T}(\mathcal{F},\mathcal{V})$$

(generalization)
$$\frac{t \in \{s\} \cup aterms(s)}{s \rightsquigarrow_\bullet gen(t)} \qquad \text{if } root(s) \in \mathcal{D} \text{ and } s \notin \mathcal{T}(\mathcal{F},\mathcal{V})$$

(decomposition)
$$\frac{s = c(t_1,\ldots,t_n) \ \wedge\ i \in \{1,\ldots,n\}}{s \rightsquigarrow_\mathcal{C} t_i} \qquad \text{if } root(s) \in \mathcal{C}$$

A generalizing needed narrowing derivation $s \rightsquigarrow^*_\sigma t$ is thus composed of *proper* needed narrowing steps (for operation-rooted terms with no annotations), generalizations (for annotated terms), and constructor decompositions (for constructor-rooted terms with no annotations), where σ is the composition of the substitutions labeling the proper needed narrowing steps. Note that, since needed narrowing only computes a *head normal form* (i.e., a variable or a constructor-rooted term), the decomposition rule is required to ensure that all inner functions (if any) are eventually partially evaluated. Some examples of generalizing needed narrowing computations can be found in the next section.

We also note that our generalization step is somehow equivalent to the splitting operation of *conjunctive partial deduction* (CPD) of logic programs [17]. 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).

The next result shows the correctness of the annotation algorithm.

THEOREM 14. *Let $\mathcal{R}$ be an inductively sequential TRS and t a linear term. Every generalizing needed narrowing derivation for t in $ann(\mathcal{R})$ is quasi-terminating.*

6. The Offline NPE Method in Practice

In this section, we first describe the complete offline NPE method based on annotated programs and generalizing needed narrowing. Then, we illustrate the new scheme by means of some selected examples. Finally, we provide a summary of the experiments conducted with a prototype implementation of the method which show the advantages of our approach in comparison with the previous online NPE method.

6.1 Overview of the Offline NPE Method

In our offline approach to NPE, given an inductively sequential TRS $\mathcal{R}$, the *first stage* consists in computing the annotated TRS: $ann(\mathcal{R})$. Then, the *second stage*—the proper partial evaluation—takes the annotated TRS, $ann(\mathcal{R})$, together with an initial

$$qs(l,t) = \begin{cases} t & \text{if } t \in \mathcal{V} \text{ is a variable} \\ c(\overline{qs(l,t_n)}) & \text{if } t = c(\overline{t_n}),\, c \in \mathcal{C}, \text{ and } n \geqslant 0 \\ f(\overline{t'_n}) & \text{if } t = f(\overline{t_n}),\, f \in \mathcal{D}, \text{ and } t'_i = qs'(l,t_i) \;\text{ for all } i = 1,\ldots,n,\, n \geqslant 0 \end{cases}$$

$$qs'(f(\overline{p_n}),t) = \begin{cases} t & \text{if } t \in \mathcal{T}(\mathcal{C},\mathcal{V}) \text{ is a constructor term and } dv(p_i,x) \geqslant dv(t,x) \text{ for all } x \in \mathcal{V}ar(p_i),\, i = 1,\ldots,n \\ \bullet(qs(f(\overline{p_n}),t)) & \text{otherwise} \end{cases}$$

Figure 1. Auxiliary functions qs and qs'

(linear) term, t, constructs the (finite) generalizing needed narrowing tree for t in $ann(\mathcal{R})$, and extracts the residual—partially evaluated—program.

Essentially, residual programs are extracted by producing a so-called *resultant*, $\sigma(s) \to rann(t)$, for each proper needed narrowing step $s \leadsto_\sigma t$ in the considered generalizing needed narrowing tree, where function $rann$ simply removes the occurrences of "$\bullet$" in a term. In general, however, the left-hand sides of these rules need not be of the form $f(s_1,\ldots,s_n)$, where s_i are constructor terms, since they may contain nested defined function symbols. Therefore, a renaming of terms is often mandatory. The following definitions from [5] formalize the notion of renaming:

DEFINITION 15 (independent renaming [5]). *An independent renaming ρ for a set of terms T is a mapping from terms to terms defined as follows: for all term $t \in T$,*

- *$\rho(t) = t$ if $t = f(\overline{x_n})$, where $f \in \mathcal{D}$ and $\overline{x_n}$ are different variables, and*
- *$\rho(t) = f_t(\overline{x_n})$, otherwise, where $\overline{x_n}$ are the distinct variables of t in the order of their first occurrence and $f_t \notin \mathcal{D}$ is a fresh function symbol.*

Observe that function calls whose arguments are different variables are not renamed since it is not necessary.

EXAMPLE 16. *Consider the set of terms*

$$T = \{f(x,y),\; g(h(x),y),\; s(c(x),x)\}$$

where $f,g,h,s \in \mathcal{D}$ are defined functions and $c \in \mathcal{C}$ is a constructor symbol. Then, the following mapping ρ is an independent renaming for T:

$$\rho = \{ \begin{array}{rcl} f(x,y) & \mapsto & f(x,y), \\ g(h(x),y) & \mapsto & g'(x,y), \\ s(c(x),x) & \mapsto & s'(x) \end{array} \}$$

Basically, given the annotated program $ann(\mathcal{R})$ and a linear term t, the partial evaluation stage proceeds by constructing the generalizing needed narrowing tree for t in $ann(\mathcal{R})$, 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. The quasi-termination of generalizing needed narrowing computations (Theorem 14) guarantees that the tree thus constructed is finite. Once the tree is built, we compute an independent renaming ρ for the set of terms $\{s \mid s \leadsto_\sigma t\}$, i.e., for the terms to which a proper needed narrowing step is applied. While the mapping ρ suffices to rename the left-hand sides of resultants, the right-hand sides require a more elaborated mapping, ren_ρ, that *recursively* replaces each call in the term by a call to the corresponding renamed function. Formally,

DEFINITION 17 (renaming mapping [5]). *Let T be a finite set of terms and ρ an independent renaming of T. Given a term s, the (nondeterministic) mapping ren_ρ is defined as follows:*

$$ren_\rho(s) = \begin{cases} s & \text{if } s \in \mathcal{V} \\ c(\overline{ren_\rho(t_n)}) & \text{if } s = c(\overline{t_n}),\; c \in \mathcal{C},\; \text{and } n \geqslant 0 \\ \theta'(\rho(t)) & \text{if there exists a term } t \in T \\ & \text{such that } s = \theta(t) \text{ and } \theta' = \\ & \{x \mapsto ren_\rho(\theta(x)) \mid x \in \mathcal{D}om(\theta)\} \end{cases}$$

EXAMPLE 18. *Consider the set of terms T and the independent renaming of Example 16. Given the term $g(h(x), f(a, s(c(b), b)))$, where $a, b \in \mathcal{C}$ are constructor symbols, function ren_ρ returns the renamed term $g'(x, f(a, s'(b)))$.*

Now, the offline NPE method can be formalized as follows:

DEFINITION 19 (offline NPE). *Let $\mathcal{R}$ be an inductively sequential TRS and $f(\overline{x_n})$ a linear term*[4] *with $f \in \mathcal{D}$. The offline NPE of $\mathcal{R}$ w.r.t. $f(\overline{x_n})$ is obtained as follows:*

1. *First, we compute the annotated TRS $ann(\mathcal{R})$.*
2. *Then, we construct a (finite) generalizing needed narrowing tree, τ, for $f(\overline{x_n})$ in $ann(\mathcal{R})$, where each derivation stops whenever it reaches a constructor term or an operation-rooted term that is a variable renaming of some previous term in the same (or a previous) derivation.*
3. *Finally, the residual TRS contains a (renamed) rule*

   $$\sigma(\rho(s)) \to ren_\rho(rann(s'))$$

   for each proper needed narrowing step $s \leadsto_\sigma s'$ in τ. Here, ρ is an independent renaming of $\{s \mid s \leadsto_\sigma s' \in \tau\}$.

For simplicity, in the definition above, we extract a resultant from each single needed narrowing step. Clearly, more refined algorithms for extracting resultants from a generalizing needed narrowing tree are possible; e.g., in many cases, one can extract a single resultant associated to a *sequence* of narrowing steps rather than to a single narrowing step. In fact, the implemented system follows such a refined strategy.

Now we state the correctness and termination of this partial evaluation method.

THEOREM 20. *Let $\mathcal{R}$ be an inductively sequential TRS and $f(\overline{x_n})$ a linear term with $f \in \mathcal{D}$. The algorithm of Def. 19 always terminates computing an inductively sequential TRS $\mathcal{R}'$ such that needed narrowing computes the same results for $f(\overline{x_n})$ in $\mathcal{R}$ and in $\mathcal{R}'$.*

6.2 Selected Examples

In this section, we illustrate the offline NPE method presented so far by means of some selected examples.

[4] This is not a restriction since one can consider an arbitrary term t by simply adding a new function definition $f(\overline{x_n}) \to t$ to $\mathcal{R}$, where $\overline{x_n}$ are the different variables of t.

$$\begin{tikzcd}
 & & & main(x) \arrow[d, "id"'] & & & \\
 & & & pow(x,s(s(0))) \arrow[d, "id"'] & & & \\
 & & & x \times \bullet(pow(x,s(0))) \arrow[dll, "\bullet"'] \arrow[drr, "\bullet"] & & & \\
 & x \times w_1 \arrow[dl, "\{x\mapsto 0\}"'] \arrow[dr, "\{x\mapsto s(w_2)\}"] & & & & pow(x,s(0)) \arrow[d, "id"] & \\
0 & & w_1 + \bullet(w_2 \times w_1) \arrow[dl, "\bullet"'] \arrow[dr, "\bullet"] & & & x \times \bullet(pow(x,0)) \arrow[dl, "\bullet"'] \arrow[dr, "\bullet"] & \\
 & w_1 + w_3 \arrow[dl, "\{w_1\mapsto 0\}"'] \arrow[dr, "\{w_1\mapsto s(w_4)\}"] & & w_2 \times w_1 & x \times w_5 & & pow(x,0) \arrow[d, "id"] \\
w_3 & & s(w_4 + w_3) \arrow[d, "\mathcal{C}"] & & & & s(0) \\
 & & w_4 + w_3 & & & &
\end{tikzcd}$$

Figure 2. Generalizing needed narrowing tree for $main(x)$

Program specialization. Our first example illustrates the use of the offline NPE method for program specialization. Consider the following TRS which has been annotated according to Def. 11:

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

Given the initial term $main(x)$, we construct the generalizing needed narrowing tree depicted in Fig. 2. Then, the associated residual TRS contains the following rules:

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

together with the original definitions of "$\times$" and "$+$". The considered independent renaming is as follows:

$$\begin{array}{lrcll}
\rho = \{ & main(x) & \mapsto & main(x), & \\
 & pow(x, s(s(0))) & \mapsto & pow_2(x), & \\
 & pow(x, s(0)) & \mapsto & pow_1(x), & \\
 & pow(x, 0) & \mapsto & pow_0(x), & \\
 & x \times y & \mapsto & x \times y, & \\
 & x + y & \mapsto & x + y & \}
\end{array}$$

Furthermore, these four rules can easily be simplified by using a standard post-unfolding *transition compression* [32] as follows:

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

since functions pow_2, pow_1, and pow_0 are only intermediate functions (i.e., there is only one call to any of them). This simple example shows that, despite the annotation of some subterms, the specialization power of the original (online) NPE is not lost in our offline approach.

Deforestation. Our second example is concerned with Wadler's *deforestation* to eliminate intermediate data structures [46]. Here,

$$\begin{tikzcd}
 & & lenapp(x,y) \arrow[d, "id"'] & \\
 & & len(app(x,y)) \arrow[dl, "\{x\mapsto[\,]\}"'] \arrow[dr, "\{x\mapsto z:zs\}"] & \\
 & len(y) \arrow[dl, "\{y\mapsto[\,]\}"'] \arrow[dr, "\{y\mapsto w:ws\}"] & & len(z : app(zs,y)) \arrow[d, "id"] \\
0 & & s(len(ws)) \arrow[d, "\mathcal{C}"] & s(len(app(zs,y))) \arrow[d, "\mathcal{C}"] \\
 & & len(ws) & len(app(zs,y))
\end{tikzcd}$$

Figure 3. Generalizing needed narrowing tree for $lenapp(x, y)$

we consider the following TRS $\mathcal{R}$:

$$\begin{array}{rcl}
lenapp(x, y) & \to & len(app(x, y)) \\
len([\,]) & \to & 0 \\
len(x : xs) & \to & s(len(xs)) \\
app([\,], y) & \to & y \\
app(x : xs, y) & \to & x : app(xs, y)
\end{array}$$

where $lenapp(x, y)$ computes the length of the concatenation of lists x and y. This function is not efficient since an intermediate data structure (the concatenation of x and y) is built. Since $\mathcal{R}$ is already nonincreasing, we have that $ann(\mathcal{R}) = \mathcal{R}$. Given the initial term $lenapp(x, y)$, we construct the generalizing needed narrowing tree depicted in Fig. 3. By using the following independent renaming:

$$\begin{array}{lrcll}
\rho = \{ & lenapp(x, y) & \mapsto & lenapp(x, y), & \\
 & len(app(x, y)) & \mapsto & la(x, y), & \\
 & len(y) & \mapsto & len(y), & \\
 & len(z : app(zs, y)) & \mapsto & la2(z, zs, y) & \}
\end{array}$$

the associated residual TRS is as follows:

$$\begin{array}{rcl}
lenapp(x, y) & \to & la(x, y) \\
la([\,], y) & \to & len(y) \\
la(z : zs, y) & \to & la2(z, zs, y) \\
la2(z, zs, y) & \to & s(la(zs, y))
\end{array}$$

together with the original definition of function len. As in the previous example, a simple post-unfolding transformation would eliminate the intermediate function $la2$. Note that the residual TRS is completely deforested (i.e., no intermediate list is built).

Higher-order removal. Our last example consist in the elimination of higher-order functions. In some programming languages, higher-order features are *defunctionalized* [42, 47], i.e., they are expressed by means of a first-order program with an explicit application operator.[5] For instance, the following TRS, which has already been annotated according to Def. 11, includes the definition of the well-known higher-order function map:

$$\begin{array}{rcl}
minc(x) & \to & map(inc_0, x) \\
map(f, [\,]) & \to & [\,] \\
map(f, x : xs) & \to & apply(\bullet(f), x) : map(f, xs) \\
inc(x) & \to & s(x) \\
apply(inc_0, x) & \to & inc(x)
\end{array}$$

[5] As in the language Curry, we do not allow the evaluation of higher-order calls containing free variables as functions (i.e., such calls are *suspended* to avoid the use of higher-order unification). A more flexible strategy can be found in [10].

$$\begin{tikzcd}
& minc(x) \arrow[d, "id"'] & & \\
& map(inc_0,x) \arrow[dl, "\{x\mapsto[\,]\}"'] \arrow[dr, "\{x\mapsto y:ys\}"] & & \\
[\,] & & apply(\bullet(inc_0),y):map(inc_0,ys) \arrow[dl, "c"'] \arrow[dr, "c"] & \\
& apply(\bullet(inc_0),y) \arrow[dl, "\bullet"'] \arrow[dr, "\bullet"] & & map(inc_0,ys) \\
apply(w,y) & & inc_0 &
\end{tikzcd}$$

Figure 4. Generalizing needed narrowing tree for $minc(x)$

Here, we used the explicit application operator $apply$ together with the partial function application inc_0 (a constructor symbol).

Observe that, in this example, we have annotated the leftmost occurrence of variable f in the third program rule. This is essential to obtain a first-order definition for $map(inc_0, x)$. Indeed, by annotating the second occurrence of variable f, the original program is basically returned by the partial evaluator.

Given the initial term $minc(x)$, the generalizing needed narrowing tree of Fig. 4 is built. Note that $apply(w, y)$ is not further reduced because, as mentioned before, this higher-order call contains a free functional variable and, thus, its evaluation suspends (which means that the original definition of $apply$ should also be included in the residual program).

Given the following independent renaming:

$$\begin{array}{rrcll}
\rho = \{ & minc(x) & \mapsto & minc(x), & \\
& map(inc_0, ys) & \mapsto & mapinc(ys), & \\
& inc(y) & \mapsto & inc(y), & \\
& apply(w, y) & \mapsto & apply(w, y) & \}
\end{array}$$

the residual TRS computed by offline NPE is as follows:

$$\begin{array}{rcl}
minc(x) & \to & mapinc(x) \\
mapinc([\,]) & \to & [\,] \\
mapinc(y : ys) & \to & apply(inc_0, y) : mapinc(ys) \\
inc(y) & \to & s(y) \\
apply(inc_0, y) & \to & inc(y)
\end{array}$$

Finally, by using a simple post-unfolding simplification we get the following TRS:

$$\begin{array}{rcl}
minc(x) & \to & mapinc(x) \\
mapinc([\,]) & \to & [\,] \\
mapinc(y : ys) & \to & s(y) : mapinc(ys)
\end{array}$$

where the explicit application operator $apply$ is no longer needed. We note that this transformation often achieves significant speedups in practice (see, e.g., [1]).

6.3 Experimental Evaluation

The offline NPE method outlined in Sect. 6.1 has been implemented in the declarative multi-paradigm language Curry [21]. The sources of the partial evaluator and a detailed explanation of the benchmarks considered below are publicly available from `http://www.dsic.upv.es/users/elp/german/offpeval/`.

The offline NPE tool is purely declarative and accepts Curry programs containing additional features like higher-order functions, several built-in functions, etc.

Table 1 shows the results of some benchmarks:

`ackermann`: This is the well-known Ackermann's function specialized for an input argument greater than or equal to 10.

`allones`: The aim of this benchmark is to automatically produce a new function that transforms all elements of a list into "1" by first computing the length of the original list and, then, constructing a new list of the same length whose elements are 1. This is a typical deforestation example [46].

`fliptree`: Another typical deforestation example. Here, the aim is to flip a tree structure twice so that the original tree is obtained; no static values are provided.

`foldr.allones`: The goal of this benchmark is the specialization of a function that concatenates a number of lists and, then, transforms all elements into 1. The original function is defined by means of the higher-order combinator $foldr$. The specialization considers that one of the lists is known.

`foldr.sum`: In this benchmark, we produce a specialized function to sum the elements of a list (with a given prefix) by using the higher-order function $foldr$.

`fun_inter`: This benchmark consists in the specialization of simple functional interpreter for a given program.

`gauss`: Our goal in this benchmark is the specialization of the well-known Gauss' function to consider natural numbers greater than or equal to 5.

`kmp_matcher`: A naive pattern matcher specialized for a given pattern. This benchmark is known as the "KMP-test" [15].

`power`: The specialization shown in Section 6.2 for a fixed exponent of 6.

For each benchmark, we show the size (in bytes) of each program (`codesize`), the time for executing the previous *online* NPE tool (`onlineNPE`), the time for executing the new *offline* NPE tool described so far (`offlineNPE`), where we show both the time for analyzing and annotating the original program (`ann`) and for performing partial computations and extracting the residual program (`mix`), as well as the speedups achieved by the programs specialized with each technique (`speedup1` and `speedup2`); speedups are given by *orig*/*spec*, where *orig* and *spec* are the absolute run times of the original and specialized 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). Runtime input goals were chosen to give a reasonably long overall time. The programs were executed with the Curry to Prolog compiler of PAKCS [27].

As it can be seen in Table 1, we have reduced the partial evaluation time to approximately 25% of the original NPE tool, which means that our main goal has been achieved. As for the speedups, we note that most of the benchmarks were *specialization* problems (rather than *optimization* problems), which explains the good results achieved by our offline NPE tool. Let us remark, however, that the new method is not able to pass the so-called "KMP-test" [15] (see benchmark `kmp_matcher`). There are two main requirements for passing the KMP test: a good propagation of information and a powerful termination analysis that avoids too much generalization. While our offline scheme propagates information as well as the previous online approach (which does pass the KMP test), our (implicit) termination analysis is much simpler. It would be interesting to check whether a mixed online/offline approach could be useful here. Our partial evaluator deals well with arithmetic functions (benchmark `ackermann`), with the simplification of higher-order calls (benchmarks `foldr.allones` and `foldr.sum`), and with a simple functional interpreter (benchmark `fun_inter`), where speedups are not shown since the execution time of the specialized programs is zero (i.e., the input program to the interpreter has been fully evaluated).

Table 1. Benchmark results

benchmark	codesize (bytes)	onlineNPE (ms.)	speedup1 (online)	offlineNPE ann (ms.)	offlineNPE mix (ms.)	speedup2 (offline)
ackermann	1496	20290	1.006	100	590	4.750
allones	1191	180	1.065	50	200	1.050
fliptree	1861	1940	0.985	100	240	0.977
foldr.allones	2910	3633	1.024	120	430	2.034
foldr.sum	3734	6797	1.311	170	3340	1.293
fun_inter	4266	28955	—	160	5190	—
gauss	1241	11090	1.040	100	757	1.013
kmp_matcher	3222	11670	5.346	157	9410	1.219
power	1693	160	3.087	110	280	1.012
Average	**2402**	**9413**	**1.858**	**119**	**2271**	**1.668**

7. Related Work and Discussion

Despite the relevance of narrowing as a symbolic computation mechanism, we find in the literature very few works devoted to analyze its termination. For instance, Dershowitz and Sivakumar [20] defined a narrowing procedure that incorporates pruning of some unsatisfiable goals. Similar approaches have been presented by Chabin and Réty [12], where narrowing is directed by a graph of terms, and by Alpuente et al. [3], where the notion of *loop-check* is introduced. Also, Antoy and Ariola [8] introduced a sort of memoization technique for functional logic languages so that, in some cases, a finite representation of an infinite narrowing space is achieved. All these techniques are *online*, since they use information about the term being narrowed. On the other hand, Christian [14] introduced a characterization of TRSs for which narrowing terminates. Basically, it requires the left-hand sides to be *flat*, i.e., all arguments are either variables or ground terms. None of these works considered *quasi-termination* nor presented a method to annotate TRSs so that termination is enforced.

Other related works come from the extensive literature on partial evaluation. Within the logic programming paradigm, Decorte et al. [18] studied the quasi-termination of *tabled* logic programs in order to port specialization techniques from “standard” logic programs to tabled ones. They introduced the characterization of *quasi-acceptable* programs and proved that this class of programs guarantees quasi-termination. However, determining whether a program is quasi-acceptable is not easy to check (the authors sketched how standard termination analysis could be extended).

Within the functional setting, Holst [28] introduced a sufficient condition for quasi-termination in order to ensure the termination of partial evaluation (which was then used by Glenstrup and Jones [24] to define a BTA algorithm ensuring the termination of offline partial evaluation). Holst also presented a static analysis based on abstract interpretation in order to check the sufficient condition for quasi-termination. Similarly to [18], the presented conditions are based on the semantics and, thus, are generally difficult to analyze.

In contrast, our approach relies on a simple *syntactic* characterization which is generally less precise but very easy to check. In fact, the closest approaches to our work are the syntactic characterizations given by Wadler [46] and Chin and Khoo [13], which have been already discussed in Sect. 4.

In summary, we have 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 introduced an algorithm that annotates those subterms which may cause the non-quasi-termination of needed narrowing. We also introduced a generalizing extension of needed narrowing which is guided by program annotations. Finally, we described how our new developments can be used to define a correct and terminating NPE scheme that ensures termination offline. Preliminary experiments conducted on a wide variety of programs are encouraging and demonstrate the usefulness of our approach.

Although we considered inductively sequential systems as programs and *needed narrowing* [9] as operational semantics, our developments could easily be extended to *overlapping* inductively sequential systems and inductively sequential narrowing [7]. The main difference is that overlapping systems allow the use of an explicit disjunction operator which introduces additional don't-know nondeterminism. In this context, introducing a function with a disjunction in the right-hand side, e.g., $f(x) \to t_1 \; or \; t_2$, is basically equivalent to writing the following single rules:

$$f(x) \to t_1 \qquad\qquad f(x) \to t_2$$

Since our termination characterization mainly depends on how the function parameters change from the left- to the right-hand side of a rule, the treatment of disjunctions in overlapping systems presents no additional problems; basically, a disjunction operator could be considered as a constructor symbol.

Positive supercompilation [44] shares many similarities with NPE since *driving*, the symbolic computation mechanism of positive supercompilation, is equivalent to needed narrowing on comparable programs. Therefore, our results could easily be transferred to the setting of positive supercompilation.

Regarding future work, one of the most recent approaches to ensure the (quasi-)termination of functional programs is based on *size-change graphs* [33] (which have been already used in the context of partial evaluation in [31]). An interesting topic for future work is thus the use of size-change graphs for defining more precise—though computationally more expensive—annotation algorithms. On the other hand, our algorithm for annotating TRSs is independent of the term considered for partial evaluation. This means that a TRS only needs to be annotated once and, then, it can be partially evaluated w.r.t. different terms without computing new annotations. However, it also means that we are not exploiting the known structure of the term considered for partial evaluation. Hence, it would be interesting to study the combination of our first stage with traditional binding-time analysis. Here, our functional *logic* setting poses new demands for binding-time analysis due to the use of logical variables and nondeterministic functions. For this purpose, we plan to investigate techniques for the binding-time analysis of logic programs within the *partial deduction* technique (like, e.g., [16, 35]).

Acknowledgments

We would like to thank the anonymous referees for many helpful comments and suggestions.

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, M.J. Ramis, and G. Vidal. Narrowing Approximations as an Optimization for Equational Logic Programs. In *Proc. of PLILP'93*, pages 391–409. Springer LNCS 714, 1993.

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

[5] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *Theory and Practice of Logic Programming*, 5(3):273–303, 2005.

[6] 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.

[7] S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

[8] S. Antoy and Z.M. Ariola. Narrowing the Narrowing Space. In *In Proc. of PLILP'97*, pages 1–15. Springer LNCS 1292, 1997.

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

[10] S. Antoy and A. Tolmach. Typed Higher-Order Narrowing without Higher-Order Strategies. In *Proc. of FLOPS'99*, pages 335–352. Springer LNCS 1722, 1999.

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

[12] J. Chabin and P. Réty. Narrowing directed by a graph of terms. In *Proc. of the 4th Int'l Conf. on Rewriting Techniques and Applications (RTA'91)*, pages 112–123. Springer LNCS 488, 1991.

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

[14] J. Christian. Some termination criteria for narrowing and E-narrowing. In *Proc. of CADE-11*, pages 582–588. Springer LNCS 607, 1992.

[15] C. Consel and O. Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30:79–86, 1989.

[16] S.J. Craig, J. Gallagher, M. Leuschel, and K.S Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of LOPSTR'04*. Springer LNCS, 2005. To appear.

[17] 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.

[18] 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, 1997.

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

[20] N. Dershowitz and G. Sivakumar. Goal-Directed Equation Solving. In *Proc. of 7th National Conf. on Artificial Intelligence*, pages 166–170. Morgan Kaufmann, 1988.

[21] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.

[22] Yoshihiko Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprint of article in *Systems, Computers, Controls* 1971.

[23] 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.

[24] A.J. Glenstrup and N.D. Jones. BTA Algorithms to Ensure Termination of Off-Line Partial Evaluation. In *Proc. of the 2nd Int'l Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 273–284. Springer LNCS 1181, 1996.

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

[26] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[27] 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.

[28] C.K. Holst. Finiteness Analysis. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 473–495. Springer LNCS 523, 1991.

[29] 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.

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

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

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

[33] 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.

[34] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, pages 379–403. Springer LNCS 2566, 2002.

[35] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *TPLP*, 4(1-2):139–191, 2004.

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

[37] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.

[38] W. Lux. Münster Curry v0.9.8: User's Guide. Technical report, University of Münster, Germany, 2004.

[39] 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.

[40] Simon Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.

[41] J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. Technical report, Technical University of Valencia, 2005. Available at `http://www.dsic.upv.es/~gvidal`.

[42] J.C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–297, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[43] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[44] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[45] R. Thiemann and J. Giesl. Size-Change Termination for Term Rewriting. In *Proc. of RTA'03*, pages 264–278. Springer LNCS 2706, 2003.

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

[47] D.H.D. Warren. Higher-Order Extensions to Prolog —Are they needed? *Machine Intelligence*, volume 10. Ellis Horwood, 1982.