Specialization of Functional Logic Programs Based on Needed Narrowing[*]

MARÍA ALPUENTE, SALVADOR LUCAS, GERMÁN VIDAL

DSIC, Technical University of Valencia
Camino de Vera s/n, E-46020 Valencia, Spain
(*e-mail:* {alpuente,slucas,gvidal}@dsic.upv.es)

MICHAEL HANUS

Institut für Informatik
CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany
(*e-mail:* mh@informatik.uni-kiel.de)

submitted 1 February 2000; revised 25 November 2002, 3 February 2004 ; accepted 2 March 2004

Abstract

Many functional logic languages are based on narrowing, a unification-based goal-solving mechanism which subsumes the reduction mechanism of functional languages and the resolution principle of logic languages. Needed narrowing is an optimal evaluation strategy which constitutes the basis of modern (narrowing-based) lazy functional logic languages. In this work, we present the fundamentals of partial evaluation in such languages. We provide correctness results for partial evaluation based on needed narrowing and show that the nice properties of this strategy are essential for the specialization process. In particular, the structure of the original program is preserved by partial evaluation and, thus, the same evaluation strategy can be applied for the execution of specialized programs. This is in contrast to other partial evaluation schemes for lazy functional logic programs which may change the program structure in a negative way. Recent proposals for the partial evaluation of declarative multi-paradigm programs use (some form of) needed narrowing to perform computations at partial evaluation time. Therefore, our results constitute the basis for the correctness of such partial evaluators.

KEYWORDS: partial evaluation, functional logic programming, needed narrowing

1 Introduction

Functional logic languages combine the operational principles of the most important declarative programming paradigms, namely functional and logic programming. Efficient demand-driven functional computations are amalgamated with the flexible use of logical variables providing for function inversion and search for solutions. The

[*] A preliminary short version of this paper appeared in the Proceedings of the International Conference on Functional Programming (ICFP'99), pp. 273–283, Paris, 1999. This work has been partially supported by CICYT TIC2001-2705-C03-01, by MCYT under grant HA2001-0059, and by the German Research Council (DFG) under grant Ha 2457/1-2.

operational semantics of such languages is usually based on narrowing, a generalization of term rewriting which combines reduction and variable instantiation. A *narrowing step* instantiates variables of an expression and applies a reduction step to a *redex* (*red*ucible *ex*pression) of the instantiated expression. The instantiation of variables is usually computed by unifying a subterm of the entire expression with the left-hand side of some rule.

Example 1
Consider the following rules which define the less-or-equal predicate "$\leqslant$" on natural numbers which are represented by terms built from data constructors `0` and `s` (note that variable names always start with an uppercase letter):

$$\begin{array}{rcl@{\quad}c@{\quad}l}
\mathtt{0} & \leqslant & \mathtt{N} & \to & \mathtt{true} \\
\mathtt{s(M)} & \leqslant & \mathtt{0} & \to & \mathtt{false} \\
\mathtt{s(M)} & \leqslant & \mathtt{s(N)} & \to & \mathtt{M} \leqslant \mathtt{N}
\end{array}$$

The goal $\mathtt{s(X)} \leqslant \mathtt{Y}$ can be solved (i.e., reduced to `true`) by instantiating `Y` to `s(Y1)` to apply the third rule followed by the instantiation of `X` to `0` to apply the first rule:

$$\mathtt{s(X)} \leqslant \mathtt{Y} \leadsto_{\{\mathtt{Y} \mapsto \mathtt{s(Y1)}\}} \mathtt{X} \leqslant \mathtt{Y1} \leadsto_{\{\mathtt{X} \mapsto \mathtt{0}\}} \mathtt{true}$$

Narrowing provides completeness in the sense of logic programming (computation of all solutions) as well as functional programming (computation of values). Since simple narrowing can have a huge search space, great effort has been made to develop sophisticated narrowing strategies without losing completeness; see (Hanus 1994) for a survey. To avoid unnecessary computations and to provide computations with infinite data structures as well as a demand-driven generation of the search space, most recent work has advocated *lazy* narrowing strategies, e.g., (Antoy et al. 2000; Giovannetti et al. 1991; Loogen et al. 1993; Moreno-Navarro and Rodríguez-Artalejo 1992). Many lazy evaluation strategies are based on the notions of *demanded* or *needed* computations. The following example informally explains the difference between these two notions:

Example 2
Consider the rules for "$\leqslant$" in Example 1 together with the following rules defining the addition on natural numbers:

$$\begin{array}{rcl}
\mathtt{0} + \mathtt{N} & \to & \mathtt{N} \\
\mathtt{s(M)} + \mathtt{N} & \to & \mathtt{s(M} + \mathtt{N)}
\end{array}$$

The initial term is $\mathtt{X} \leqslant \mathtt{X} + \mathtt{X}$. The evaluation of subterm $\mathtt{X} + \mathtt{X}$ is *demanded* by the second and third rules for "$\leqslant$", since these rules cannot be applied to $\mathtt{X} \leqslant \mathtt{X} + \mathtt{X}$ until the subterm $\mathtt{X} + \mathtt{X}$ is reduced to a term rooted by a data constructor symbol. However, evaluating this subterm is not *needed* since, if we instantiate `X` to `0`, we directly obtain `true` by using the first rule for "$\leqslant$".

On the other hand, if the initial term is $\mathtt{X} + (\mathtt{0} + \mathtt{0})$, the evaluation of $\mathtt{0} + \mathtt{0}$ is *needed* to compute its value whereas it is not *demanded* by any rule for "+".

Needed narrowing (Antoy et al. 2000) is based on the idea of evaluating only subterms which are *needed* in order to compute a result. For instance, in a term like

$t_1 \leqslant t_2$, it is always necessary to evaluate t_1 (to some *head normal form*, i.e., either a variable or a constructor-rooted term) since all three rules in Example 1 have left-hand sides whose first argument is not a variable. On the other hand, the evaluation of t_2 is only needed if t_1 is of the form $\mathtt{s}(\cdots)$. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor, here $\mathtt{0}$ or $\mathtt{s}(\cdots)$. Then, depending on this instantiation, either the first rule is applied or the second argument t_2 is evaluated. Needed narrowing is currently the best narrowing strategy for first-order functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions (Antoy et al. 2000). Informally speaking, needed narrowing derivations are the shortest possible narrowing derivations if common subterms are shared (as it is usually done in implementations of functional languages), and the set of all solutions computed by needed narrowing is minimal since needed narrowing computes only independent solutions (see also Theorem 1 below). Furthermore, it can be efficiently implemented by pattern matching and unification (Hanus 1995; Loogen et al. 1993). For instance, the operational semantics of the declarative multi-paradigm language Curry (Hanus (ed.) 2003) is based on needed narrowing. Needed narrowing has also been extended to higher-order functions and λ-terms as data structures and proved optimal w.r.t. the independence of computed solutions (Hanus and Prehofer 1999).

Partial evaluation (PE) is a semantics-preserving performance optimization technique for computer programs which consists of the specialization of the program w.r.t. parts of its input. PE has been widely applied in the fields of term rewriting systems (Bellegarde 1995; Bondorf 1988; Dershowitz and Reddy 1993; Lafave and Gallagher 1997), functional programming (Consel and Danvy 1993; Jones et al. 1993), and logic programming (Gallagher 1993; Lloyd and Shepherdson 1991; De Schreye et al. 1999). Although the objectives are similar, the general methods are often different due to the distinct underlying models and the different perspectives (Alpuente et al. 1998a). This separation has the negative consequence of duplicated work since developments are not shared and many similarities are overlooked. A unified treatment can bring the different methodologies closer and lays the ground for new insights in all three fields (Alpuente et al. 1998a; Alpuente et al. 1998b; Glück and Sørensen 1994; Pettorossi and Proietti 1996a; Sørensen et al. 1996).

In order to perform reductions at specialization time, *online* partial evaluators normally include an interpreter (Consel and Danvy 1993). This implies that the power of the transformation is highly influenced by the properties of the evaluation strategy in the underlying interpreter. Narrowing-driven PE (Alpuente et al. 1998a; Albert and Vidal 2002) is the first generic algorithm for the specialization of functional logic programs. The method is parametric w.r.t. the narrowing strategy which is used for the automatic construction of the search trees. The method is formalized within the theoretical framework established by Lloyd and Shepherdson (1991) for the PE of logic programs (also known as *partial deduction*), although a number of concepts have been generalized to deal with the functional component of the language (e.g., nested function calls in expressions, different evaluation strategies, etc). This approach has better opportunities for optimization thanks to the functional dimension (e.g., by the inclusion of deterministic evaluation steps). Also, since unification is embedded into narrowing, it is able to automatically propagate syntactic information on the partial input (term structure) and not only constant values, similar to partial deduction. Using the terminology of Glück and Sørensen (1996), narrowing-driven PE is able to produce both *polyvariant* and *polygenetic* specializations, i.e., it can produce different specializations for the same function definition and can also combine distinct original function definitions into a comprehensive specialized function. This means that narrowing-driven PE has the same potential for specialization as *positive supercompilation* of functional programs (Sørensen et al. 1996) and *conjunctive partial deduction* of logic programs (De Schreye et al. 1999); more detailed comparisons can be found in (Alpuente et al. 1998a; Alpuente et al. 1998b; Albert and Vidal 2002).

The main contribution of this work is the proof of the basic computational properties of PE based on needed narrowing. The most recent approaches for the PE of multi-paradigm functional logic languages (Albert et al. 1999; Albert et al. 2002; Albert et al. 2003) use (a form of) needed narrowing to perform computations at PE time (see also Section 6). Therefore, our results constitute the basis for the correctness of such partial evaluators. To be more precise, we provide the following results for PE based on needed narrowing:

- We prove the strong correctness of the PE scheme: the answers computed by needed narrowing in the original and the partially evaluated programs coincide.
- We establish the relation between PE based on needed narrowing and PE based on a different lazy evaluation mechanism—which is the basis of previous partial evaluators (Alpuente et al. 1997). We formally prove the superiority of needed narrowing to perform partial computations. In particular, we prove that the structure of the original program is preserved by PE based on needed narrowing and, thus, the same optimal evaluation strategy can be applied for the execution of specialized programs. This is in contrast to previous PE schemes (Alpuente et al. 1997) for lazy functional logic programs which may change the program structure in a negative way.
- We show that specialized programs preserve deterministic evaluations, i.e., if the source program can evaluate a goal without any choice, then the partially evaluated program does just the same. This is important from an implementation point of view and it is not obtained by PE based on other operational models, like lazy narrowing.

Providing experimental evidence of the practical advantages of using needed narrowing to perform PE is outside the scope of this paper. We refer, e.g., to (Albert et al. 2002) where this topic has been extensively addressed for a practical partial evaluator based on the foundations presented in this paper.

The structure of the paper is as follows. After some basic definitions in the next section, we recall in Section 3 the formal definition of inductively sequential programs and needed narrowing. Section 4 recalls the lazy narrowing strategy and relates it to needed narrowing. The definition of partial evaluation based on needed narrowing is provided in Section 5 together with results about the structure of specialized programs and the (strong) correctness of the transformation. Section 6 outlines several recent extensions of PE based on needed narrowing. Finally, Section 7 concludes. Detailed proofs of selected results can be found in (Alpuente et al. 2004).

2 Preliminaries

Term rewriting systems (TRSs) provide an adequate computational model for functional languages which allow the definition of functions by means of patterns (e.g., Haskell, Hope or Miranda). Within this framework, the class of *inductively sequential* programs, which we consider in this paper, has been defined, studied, and used for the implementation of programming languages which provide for optimal computations both in functional and functional logic programming (Antoy 1992; Antoy et al. 2000; Hanus 1997; Hanus et al. 1998; Loogen et al. 1993). Inductively sequential programs can be thought of as constructor-based TRSs with discriminating left-hand sides, i.e., typical functional programs where at most one rule is used to reduce a particular subterm (without variables). Thus, in the remainder of the paper we follow the standard framework of term rewriting (Dershowitz and Jouannaud 1990) for developing our results.

We consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. There is at least one sort *Bool* containing the Boolean constructors *true* and *false*. Given a set of variables $\mathcal{X}$, the set of *terms* and *constructor terms* are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$. A term is *linear* if it does not contain multiple occurrences of one variable. We write $\overline{o_n}$ for the *sequence* of objects $o_1, \ldots, o_n$.

A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A term is *operation-rooted* if it has an operation symbol at the root. *root*(t) denotes the symbol at the root of the term t. A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). They are used to address the nodes of a term viewed as a tree (Dewey notation). For instance, if $t = f(t_1, \ldots, t_n)$, positions $1, \ldots, n$ refer to arguments $t_1, \ldots, t_n$ respectively; thus, given a position p_i of a subterm of t_i, position $i \cdot p_i$ denotes the corresponding subterm of t. Positions are ordered by the *prefix* ordering: $u \leq v$, if there exists w such that $u \cdot w = v$. Given a term t, $\mathcal{P}os(t)$ and $\mathcal{NVP}os(t)$ denote the set of positions and the set of non-variable positions of t, respectively. $t|_p$ 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 (see (Dershowitz and Jouannaud 1990) for details).

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is called the *domain* of σ. A substitution σ is *constructor* (*ground constructor*), if $\sigma(x)$ is constructor (ground constructor) for all $x \in \mathcal{D}om(\sigma)$. The identity substitution is denoted by *id*. Substitutions are extended to morphisms on terms by $\sigma(f(\overline{t_n})) = f(\overline{\sigma(t_n)})$ for every term $f(\overline{t_n})$. Given a substitution θ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta_{\upharpoonright V}$ the substitution obtained from θ by restricting its domain to V. We write $(\theta = \sigma)[V]$ if $\theta_{\upharpoonright V} = \sigma_{\upharpoonright V}$, and $(\theta \leq \sigma)[V]$ denotes the existence of a substitution γ such that $(\gamma \circ \theta = \sigma)[V]$.

Term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$. This implies a (relative generality) *subsumption ordering* on terms which is defined by $t \leq t'$ iff t' is an instance of t. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. The unifier σ is *most general* if $(\sigma \leq \sigma')[\mathcal{X}]$ for each other unifier σ'.

A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A set of rewrite rules is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A TRS $\mathcal{R}$ is *left-linear* if l is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor-based (CB) if each lhs l is a pattern. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position $p \in \mathcal{NVP}os(l)$ and a most general unifier σ such that $\sigma(l|_p) = \sigma(l')$. A left-linear TRS without overlapping rules is called *orthogonal*. In the remainder of this paper, a *functional logic program* is a finite left-linear CB-TRS. Conditions in program rules are treated by using the predefined functions `and`, `if_then_else`, `case_of` which are reduced by standard defining rules (Moreno-Navarro and Rodríguez-Artalejo 1992).

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there is a position p in t, a rewrite rule R of the form $l \to r$ 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 rewrite step). The instantiated lhs $\sigma(l)$ is called a *redex*. $\mathcal{P}os_{\mathcal{R}}(t)$ denotes the set of *redex positions* of the term t in the TRS $\mathcal{R}$. $\to^+$ ($\to^*$) denotes the transitive (reflexive and transitive) closure of $\to$. If $t \to^* s$, we say that t is rewritten to s. A term t is root-stable (often called a *head-normal form*) if it cannot be rewritten to a redex. A *constructor root-stable* term is either a variable or a *constructor-rooted* term, i.e., a term rooted by a constructor symbol. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$.

In order to evaluate terms containing variables, narrowing non-deterministically instantiates its variables such that a rewrite step is possible—usually by computing most general unifiers between a subterm and some lhs (Hanus 1994), but this requirement is relaxed in needed narrowing steps in order to obtain an optimal evaluation strategy (Antoy et al. 2000). Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to_{p,R} t'$. 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 $\sigma = id$). Since we are interested in computing *values* (constructor terms) as well as *answers* (substitutions), we say that the narrowing derivation $t \leadsto^*_\sigma c$ *computes the result* c *with answer* σ if c is a constructor term. The evaluation to ground constructor terms is the most common semantics of functional (logic) languages. In lazy functional (logic) languages, the equality predicate $\approx$ used in some examples is defined as the *strict equality* on terms (note that we do not require terminating rewrite systems and, thus, reflexivity is not desired), i.e., the equation $t_1 \approx t_2$ is satisfied if and only if t_1 and t_2 are reducible to the same ground constructor term. Furthermore, a substitution σ is a *solution* for an equation $t_1 \approx t_2$ if $\sigma(t_1) \approx \sigma(t_2)$

is satisfied. The strict equality can be defined as a binary Boolean function by the following set of orthogonal rewrite rules:

$$\begin{array}{rcll} \mathtt{c} \approx \mathtt{c} & \rightarrow & \mathtt{true} & \mathtt{c}/\mathtt{0} \in \mathcal{C} \\ \mathtt{c}(\mathtt{X}_1,\ldots,\mathtt{X}_\mathtt{n}) \approx \mathtt{c}(\mathtt{Y}_1,\ldots,\mathtt{Y}_\mathtt{n}) & \rightarrow & (\mathtt{X}_1 \approx \mathtt{Y}_1) \wedge \ldots \wedge (\mathtt{X}_\mathtt{n} \approx \mathtt{Y}_\mathtt{n}) & \mathtt{c}/\mathtt{n} \in \mathcal{C}, \mathtt{n} > \mathtt{0} \\ \mathtt{true} \wedge \mathtt{X} & \rightarrow & \mathtt{X} & \end{array}$$

Thus, we do not treat strict equality in any special way and it is sufficient to consider it as a Boolean function. We say that σ is a *computed answer substitution* for an equation e if there is a narrowing derivation $e \leadsto^*_\sigma \mathtt{true}$. More details about strict equality can be found in (Antoy et al. 2000; Giovannetti et al. 1991; Moreno-Navarro and Rodríguez-Artalejo 1992).

As in logic programming, narrowing derivations can be represented by a (possibly infinite) finitely branching *tree*. Formally, given a program $\mathcal{R}$ 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). A *failing leaf* contains a term which is not a constructor term and which cannot be further narrowed. Following (Lloyd and Shepherdson 1991), in this work we adopt the convention that a derivation can be *incomplete* (thus, a branch can be failed, incomplete, successful, or infinite).

3 Needed Narrowing

Since functional logic languages are intended to extend (pure) logic languages, completeness of the operational semantics is an important issue. Similarly to logic programming, completeness means the ability to compute representatives of all solutions for one or more equations (this will be formalized in Theorem 1). Narrowing, as defined in the previous section, is complete but highly (don't-know) non-deterministic: if t is a term, we have to apply at all non-variable subterms all possible rules with all possible substitutions in order to compute all solutions. Clearly, this would be too inefficient for a realistic functional logic language. Thus, a challenge in the design of functional logic languages is the definition of a "good" narrowing strategy, i.e., a restriction on the narrowing steps issuing from a given term t, without losing completeness. (Hanus 1994) contains a survey of various attempts to define reasonable narrowing strategies.

Needed narrowing (Antoy et al. 2000) is currently the best known narrowing strategy due to its optimality properties (see the discussion in Section 1 and Theorem 1). Needed narrowing is defined on *inductively sequential programs*, a class of CB-TRSs where the left-hand sides do not overlap (in particular, they are not unifiable). To provide a definition of this class of programs and the needed narrowing strategy, we introduce definitional trees (Antoy 1992). Here we use the definition of (Antoy 1997) which is more appropriate for our purposes.

A *definitional tree* of a finite set S of linear patterns is a non-empty set $\mathcal{P}$ of linear patterns partially ordered by subsumption having the following properties:

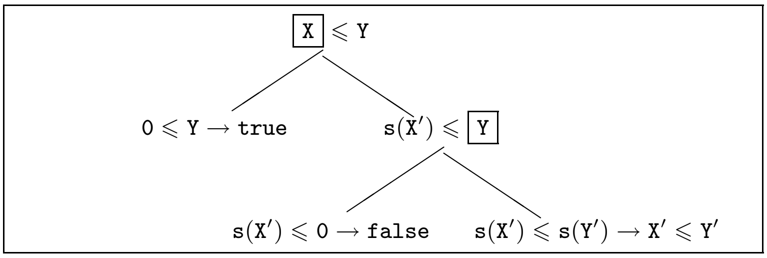
$\boxed{\texttt{X}} \leqslant \texttt{Y}$
$\texttt{0} \leqslant \texttt{Y} \to \texttt{true}$
$\texttt{s(X')} \leqslant \boxed{\texttt{Y}}$
$\texttt{s(X')} \leqslant \texttt{0} \to \texttt{false}$
$\texttt{s(X')} \leqslant \texttt{s(Y')} \to \texttt{X'} \leqslant \texttt{Y'}$

Fig. 1. Definitional tree for the function "$\leqslant$"

Root property: $\mathcal{P}$ has a minimum element (that we denote as *pattern*($\mathcal{P}$)), also called the *pattern* of the definitional tree.

Leaves property: The maximal elements of $\mathcal{P}$, called the *leaves* of the definitional tree, are the elements of S. Non-maximal elements are also called *branch* nodes.

Parent property: If $\pi \in \mathcal{P}$, $\pi \neq \mathit{pattern}(\mathcal{P})$, there exists a unique $\pi' \in \mathcal{P}$, called the *parent* of π (and π is called a *child* of π'), such that $\pi' < \pi$ and there is no other pattern $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ with $\pi' < \pi'' < \pi$.

Induction property: Given $\pi \in \mathcal{P} \backslash S$, there is a position o in π with $\pi|_o \in \mathcal{X}$ (called the *inductive position*), and constructors $c_1/k_1, \ldots, c_n/k_n \in \mathcal{C}$ with $c_i \neq c_j$ for $i \neq j$, such that, for all $\pi_1, \ldots, \pi_n$ which have the parent π, $\pi_i = \pi[c_i(\overline{x_{k_i}})]_o$ (where $\overline{x_{k_i}}$ are new distinct variables) for all $1 \leq i \leq n$.[1]

If $\mathcal{R}$ is an orthogonal TRS and f/n a defined function, we call $\mathcal{P}$ a *definitional tree of f* if $\mathit{pattern}(\mathcal{P}) = f(\overline{x_n})$ for distinct variables $\overline{x_n}$ and the leaves of $\mathcal{P}$ are all (and only) variants of the left-hand sides of the rules in $\mathcal{R}$ defining f (i.e., rules $l \to r$ such that $\mathit{root}(t) = f, f \in \mathcal{F}$). Due to the orthogonality of $\mathcal{R}$, we can assign a unique rule defining f to each leaf. A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential. An inductively sequential TRS can be viewed as a set of definitional trees, each defining a function symbol. There can be more than one definitional tree for an inductively sequential function. In the following, we assume that there is a fixed definitional tree for each defined function.

It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position in branch nodes is surrounded by a box, and the leaves contain the corresponding rules. For instance, the definitional tree of the function "$\leqslant$" in Example 1 is illustrated in Figure 1.

The following auxiliary proposition shows that functions defined by a single rule are always inductively sequential.

Proposition 1

[1] There might be more than one potential inductive position when constructing a definitional tree. In this case one can select any of them since the results about needed narrowing do not depend on the selected definitional tree.

If $f(\overline{t_n})$ is a linear pattern, then there exists a definitional tree for the set $\{f(\overline{t_n})\}$ with pattern $f(\overline{x_n})$.

Proof
By induction on the number of constructor symbols occurring in t, where each constructor symbol is introduced in a child of a branch node and each branch node has only one child. □

For the definition of needed narrowing, we assume that t is an operation-rooted term and $\mathcal{P}$ is a definitional tree with $\mathit{pattern}(\mathcal{P}) = \pi$ such that $\pi \leq t$. We define a function λ from terms and definitional trees to sets of tuples (position, rule, substitution) as the least set satisfying the following properties. We consider two cases for $\mathcal{P}$:[2]

1. If π is a leaf, i.e., $\mathcal{P} = \{\pi\}$, and $\pi \to r$ is a variant of a rewrite rule, then

$$\lambda(t, \mathcal{P}) = \{(\Lambda, \pi \to r, id)\}\cdot$$

2. If π is a branch node, consider the inductive position o of π and a child $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. Let $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ be the definitional tree where all patterns are instances of π_i. Then we consider the following cases for the subterm $t|_o$:

$$\lambda(t,\mathcal{P}) \ni \begin{cases} (p, R, \sigma \circ \tau) & \text{if } t|_o = x \in \mathcal{X},\ \tau = \{x \mapsto c_i(\overline{x_n})\}, \\ & \quad \text{and } (p, R, \sigma) \in \lambda(\tau(t), \mathcal{P}_i); \\ (p, R, \sigma \circ id) & \text{if } t|_o = c_i(\overline{t_n}) \text{ and } (p, R, \sigma) \in \lambda(t, \mathcal{P}_i); \\ (o \cdot p, R, \sigma \circ id) & \text{if } t|_o = f(\overline{t_n}),\ f \in \mathcal{F}, \text{ and } (p, R, \sigma) \in \lambda(t|_o, \mathcal{P}') \\ & \quad \text{where } \mathcal{P}' \text{ is a definitional tree for } f. \end{cases}$$

Informally speaking, needed narrowing applies a rule, if the definitional tree does not require further pattern matching (case 1), or checks the subterm corresponding to the inductive position of the branch node (case 2): if it is a variable, it is instantiated to the constructor of a child; if it is already a constructor, we proceed with the corresponding child (note that we do not actually need substitution *id* but we include it to provide a normalized representation of a needed narrowing step, see below); if it is a function, we evaluate it by recursively applying needed narrowing. Thus, the strategy differs from typical lazy functional languages only in the instantiation of free variables.

Note that, in each recursive step during the computation of λ, we compose the current substitution with the local substitution of this step (which can be the identity). Thus, each needed narrowing step can be represented as $(p, R, \varphi_k \circ \cdots \circ \varphi_1)$, where each φ_j is either the identity or the replacement of a single variable computed in each recursive step (see the following proposition). This is also called the *canonical representation* of a needed narrowing step. As in proof procedures for logic programming, we assume that the definitional trees always contain new variables

[2] This description of a needed narrowing step is slightly different from (Antoy et al. 2000) but it results in the same needed narrowing steps.

if they are used in a narrowing step. This implies that all computed substitutions are idempotent (we will implicitly assume this property in the following).

To compute needed narrowing steps for an operation-rooted term t, we take the definitional tree $\mathcal{P}$ for the root of t and compute $\lambda(t,\mathcal{P})$. Then, for all $(p,R,\sigma) \in \lambda(t,\mathcal{P})$, $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*. We call this step *deterministic* if $\lambda(t,\mathcal{P})$ contains exactly one element.

Example 3

Consider the rules in Example 2. Then the function λ computes the following set for the initial term $\mathtt{X} \leqslant \mathtt{X}+\mathtt{X}$:

$$\{(\Lambda, \mathtt{0} \leqslant \mathtt{N} \to \mathtt{true}, \{\mathtt{X} \mapsto \mathtt{0}\}),\ (2, \mathtt{s(M)}+\mathtt{N} \to \mathtt{s(M+N)}, \{\mathtt{X} \mapsto \mathtt{s(M)}\})\}$$

This corresponds to the following narrowing steps:

$$\begin{array}{lll} \mathtt{X} \leqslant \mathtt{X}+\mathtt{X} & \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} & \mathtt{true} \\ \mathtt{X} \leqslant \mathtt{X}+\mathtt{X} & \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(M)}\}} & \mathtt{s(M)} \leqslant \mathtt{s(M+s(M))} \end{array}$$

In the following we state some interesting properties of needed narrowing which are useful for our later results. The first proposition shows that each substitution in a needed narrowing step instantiates only variables occurring in the initial term.

Proposition 2

If $(p, R, \varphi_k \circ \cdots \circ \varphi_1) \in \lambda(t,\mathcal{P})$ is a needed narrowing step, then, for $i = 1,\ldots,k$, $\varphi_i = id$ or $\varphi_i = \{x \mapsto c(\overline{x_n})\}$ (where $\overline{x_n}$ are pairwise different variables) with $x \in \mathcal{V}ar(\varphi_{i-1} \circ \cdots \circ \varphi_1(t))$.

Proof

By induction on k. □

The next lemma shows that for different narrowing steps (computing different substitutions) there is always a variable which is instantiated to different constructors:

Lemma 1

Let t be an operation-rooted term, $\mathcal{P}$ a definitional tree with *pattern*$(\mathcal{P}) \leq t$ and $(p,R,\varphi_k\circ\cdots\circ\varphi_1), (p',R',\varphi'_{k'}\circ\cdots\circ\varphi'_1) \in \lambda(t,\mathcal{P})$, $k \leq k'$. Then, for all $i \in \{1,\ldots,k\}$,

- either $\varphi_i \circ \cdots \circ \varphi_1 = \varphi'_i \circ \cdots \circ \varphi'_1$, or
- there exists some $j < i$ with
  1. $\varphi_j \circ \cdots \circ \varphi_1 = \varphi'_j \circ \cdots \circ \varphi'_1$, and
  2. $\varphi_{j+1} = \{x \mapsto c(\cdots)\}$ and $\varphi'_{j+1} = \{x \mapsto c'(\cdots)\}$ with $c \neq c'$.

Proof

By induction on k (the number of recursive steps performed by λ to compute $(p, R, \varphi_k \circ \cdots \circ \varphi_1)$):

$k = 1$: Then $\mathcal{P} = \{\pi\}$ and $\lambda(t,\mathcal{P}) = \{(\Lambda, R, id)\}$. Thus, the proposition trivially holds.

$k > 1$**:** Then $\pi = \mathit{pattern}(\mathcal{P})$ is a branch node and there is an inductive position o of π such that all children of π have the form $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. Let $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ be the definitional tree where all patterns are instances of π_i, for $i = 1, \ldots, n$. We prove the induction step by a case distinction on the form of the subterm $t|_o$:

$t|_o = x \in \mathcal{X}$**:** Then $\varphi_1 = \{x \mapsto c_i(\overline{x_n})\}$ and $(p, R, \varphi_k \circ \cdots \circ \varphi_2) \in \lambda(\varphi_1(t), \mathcal{P}_i)$ for some i. If $\varphi'_1 = \{x \mapsto c(\cdots)\}$ with $c \neq c_i$, then the proposition directly holds. Otherwise, if $\varphi_1 = \varphi'_1$, the proposition follows from the induction hypothesis applied to $(p, R, \varphi_k \circ \cdots \circ \varphi_2), (p', R', \varphi'_{k'} \circ \cdots \circ \varphi'_2) \in \lambda(\varphi_1(t), \mathcal{P}_i)$.

$t|_o = c_i(\overline{t_n})$**:** Then $\varphi_1 = \mathit{id}$ and $(p, R, \varphi_k \circ \cdots \circ \varphi_2) \in \lambda(t, \mathcal{P}_i)$. Clearly, $\varphi'_1 = \mathit{id}$ by definition of λ. Hence the proposition follows from the induction hypothesis applied to $(p, R, \varphi_k \circ \cdots \circ \varphi_2), (p', R', \varphi'_{k'} \circ \cdots \circ \varphi'_2) \in \lambda(t, \mathcal{P}_i)$.

$t|_o = f(\overline{t_n})$**:** Then $\varphi_1 = \mathit{id}$ and $(p, R, \varphi_k \circ \cdots \circ \varphi_2) \in \lambda(t|_o, \mathcal{P}')$ where $\mathcal{P}'$ is a definitional tree for f. By definition of λ, $\varphi'_1 = \mathit{id}$. Then the proposition follows from the induction hypothesis applied to $(p, R, \varphi_k \circ \cdots \circ \varphi_2), (p', R', \varphi'_{k'} \circ \cdots \circ \varphi'_2) \in \lambda(t|_o, \mathcal{P}')$.

□

For inductively sequential programs, needed narrowing is sound and complete w.r.t. strict equality when we consider constructor substitutions as solutions (note that constructor substitutions are sufficient in practice since a broader class of solutions would contain unevaluated or undefined expressions for the considered programs). Moreover, needed narrowing does not compute redundant solutions. These properties are formalized as follows, where we say that two substitutions σ and σ' are *independent* (on a set of variables $V \subseteq \mathcal{X}$) iff there is some $x \in V$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable.[3]

Theorem 1 (*Antoy et al. 2000*)
Let $\mathcal{R}$ be an inductively sequential program and e an equation.

1. (Soundness) If $e \leadsto^*_\sigma \mathit{true}$ is a needed narrowing derivation, then σ is a solution for e.
2. (Completeness) For each constructor substitution σ that is a solution of e, there exists a needed narrowing derivation $e \leadsto^*_{\sigma'} \mathit{true}$ with $\sigma' \leq \sigma[\mathcal{V}ar(e)]$.
3. (Minimality) If $e \leadsto^*_\sigma \mathit{true}$ and $e \leadsto^*_{\sigma'} \mathit{true}$ are two distinct needed narrowing derivations, then σ and σ' are independent on $\mathcal{V}ar(e)$.

An important advantage of functional logic languages in comparison to pure logic languages is their improved operational behavior by avoiding non-deterministic computation steps. One reason for that is a demand-driven computation strategy which can avoid the evaluation of potential non-deterministic expressions. For instance, consider the rules in Examples 1 and 3 and the term $\mathtt{0} \leqslant \mathtt{X} + \mathtt{X}$. Needed narrowing evaluates this term by one deterministic step to `true`. In an equivalent

[3] Actually, (Antoy et al. 2000) prove a stronger property (disjointness of solutions) but this is not necessary here.

logic program, this nested term must be flattened into a conjunction of two predicate calls, like $+(\mathtt{X},\mathtt{X},\mathtt{Z}) \wedge \leqslant(\mathtt{0},\mathtt{Z})$, which causes a non-deterministic computation due to the predicate call $+(\mathtt{X},\mathtt{X},\mathtt{Z})$.[4] Another reason for the improved operational behavior of functional logic languages is the ability of particular evaluation strategies (like needed narrowing or parallel narrowing (Antoy et al. 1997)) to evaluate ground terms in a completely deterministic way, which is important to ensure an efficient implementation of purely functional evaluations. This property, which is obvious by the definition of needed narrowing, is formally stated in the following proposition. For this purpose, we call a term t *deterministically evaluable* (w.r.t. needed narrowing) if each step in a narrowing derivation issuing from t is deterministic. A term t *deterministically normalizes* to a constructor term c (w.r.t. needed narrowing) if t is deterministically evaluable and there is a needed narrowing derivation $t \leadsto^*_{id} c$ (i.e., c is the normal form of t).

Proposition 3
Let $\mathcal{R}$ be an inductively sequential program and t be a term.

1. If $t \leadsto^*_{id} c$ is a needed narrowing derivation, then t deterministically normalizes to c.
2. If t is ground, then t is deterministically evaluable.

4 Lazy Narrowing and Uniform Programs

One of the main objectives of this work is to clarify the relation between the definition of a PE scheme based on needed narrowing and a previous PE method based on lazy narrowing (Alpuente et al. 1997). In order to show the improvements obtained by using needed narrowing to perform partial computations, we first provide a brief review of the lazy narrowing strategy in this section.

Lazy narrowing reduces expressions at outermost narrowable positions. Narrowing at inner positions is performed only if it is demanded (by the pattern in the lhs of some rule). In the following, we specify a lazy narrowing strategy which is similar to (Moreno-Navarro and Rodríguez-Artalejo 1992).

The following definitions are necessary for our formalization of lazy narrowing. A *linear unification problem* is a pair of terms: $\delta = \langle f(\overline{d_n}), f(\overline{t_n})\rangle$, where $f(\overline{d_n})$ and $f(\overline{t_n})$ do not share variables, and $f(\overline{d_n})$ is a linear pattern. Linear unification $\mathtt{LU}(\delta)$ can either succeed, fail or suspend, delivering $(\mathtt{Succ},\sigma)$, $(\mathtt{Fail},\varnothing)$ or $(\mathtt{Demand},P)$, respectively, where P is the set of *demanded positions* which require further evaluation; details can be found in (Alpuente et al. 1997).

We define the lazy narrowing strategy in the following definition. Roughly speaking, the set-valued function $\lambda_{lazy}(t)$ returns the set of triples (p,R,σ) such that p is a demanded position of t which can be narrowed by the rule R with substitution σ (where σ is a most general unifier of $t|_p$ and the left-hand side of R). We assume the rules of $\mathcal{R}$ to be numbered with $R_1,\ldots,R_m$.

[4] Such non-deterministic computations could be avoided using Prolog systems with coroutining which allow the suspension of some non-deterministic computations, but then we are faced with the problem of floundering and incompleteness.

Definition 1 (*lazy narrowing strategy*)

$$\begin{array}{lcl} \lambda_{lazy}(t) & = & \bigcup_{k=1}^{m} \lambda\_(t, \Lambda, k) \\ \lambda\_(t, p, k) & = & \text{if } root(l_k) = root(t|_p) \text{ then} \\ & & \quad \text{case } \mathsf{LU}(\langle l_k, t_{|p} \rangle) \text{ of } \left\{ \begin{array}{ll} (\texttt{Succ}, \sigma): & \{(p, R_k, \sigma)\} \\ (\texttt{Fail}, \varnothing): & \varnothing \\ (\texttt{Demand}, P): & \bigcup_{q \in P} \bigcup_{k=1}^{m} \lambda\_(t, p \cdot q, k) \end{array} \right. \\ & & \text{else } \varnothing \end{array}$$

where $R_k = (l_k \to r_k)$ is a (renamed apart) rule of $\mathcal{R}$.

Example 4
Consider the rules for "$\leqslant$" and "+" in Examples 1 and 3. Then lazy narrowing evaluates the term $\mathtt{X} \leqslant \mathtt{X} + \mathtt{X}$ by applying a narrowing step at the top (with the first rule for "$\leqslant$") or by applying a narrowing step to the second argument $\mathtt{X} + \mathtt{X}$ since this is demanded by the second and third rule for "$\leqslant$". Thus, there are three lazy narrowing steps:

$$\begin{array}{lll} \mathtt{X} \leqslant \mathtt{X} + \mathtt{X} & \leadsto_{\{\mathtt{X} \mapsto \mathtt{0}\}} & \mathtt{true} \\ \mathtt{X} \leqslant \mathtt{X} + \mathtt{X} & \leadsto_{\{\mathtt{X} \mapsto \mathtt{0}\}} & \mathtt{0} \leqslant \mathtt{0} \\ \mathtt{X} \leqslant \mathtt{X} + \mathtt{X} & \leadsto_{\{\mathtt{X} \mapsto \mathtt{s(M)}\}} & \mathtt{s(M)} \leqslant \mathtt{s(M + s(M))} \end{array}$$

Note that the second lazy narrowing step is in some sense superfluous since it also yields the final value `true` with the same binding as the first step. The avoidance of such superfluous steps by using needed narrowing will have a positive impact on the PE process, as we will see later.

In orthogonal programs, lazy narrowing is complete w.r.t. strict equality and constructor substitutions:

Proposition 4 (*Moreno-Navarro and Rodríguez-Artalejo 1992*)
Let $\mathcal{R}$ be an orthogonal program, e an equation, and σ a constructor substitution that is a solution for e. Then there is a lazy narrowing derivation $e \leadsto_{\sigma'}^{*} true$ such that $\sigma' \leq \sigma[\mathcal{V}ar(e)]$.

Thus, lazy narrowing is complete for a larger class of programs than needed narrowing (since inductively sequential programs are always orthogonal), but it may have a worse behavior than needed narrowing (see Example 4). Nevertheless, the idea of needed narrowing can also be extended to almost orthogonal programs (Antoy et al. 1997), but then the optimality properties are lost. There exists a class of programs where the superfluous steps of lazy narrowing are avoided, since lazy narrowing and needed narrowing coincide on this class. These are the *uniform* programs (Zartmann 1997) which are inductively sequential programs where at most one constructor occurs in the left-hand side of each rule. A program is *uniform* if each function f is defined by one rule $f(\overline{x_n}) \to r$ or the left-hand side of every rule R_i defining f is left-linear and has the form $f(\overline{x_k}, c_i(\overline{y_{n_i}}), \overline{z_m})$, where the constructors c_i are distinct in different rules. Note that uniform programs are orthogonal.

In the latter case, an evaluation of a call to f demands its $(k+1)$-th argument. A different definition of uniform programs can be found in (Kuchen et al. 1990).

There is a simple mapping $\mathcal{U}$ from inductively sequential into uniform programs which is based on flattening nested patterns, see (Zartmann 1997). For instance, if $\mathcal{R}$ is the program in Example 1, then $\mathcal{U}(\mathcal{R})$ consists of the rules

$$\begin{array}{rcl@{\qquad}lcl}
\mathtt{0} \leqslant \mathtt{N} & \rightarrow & \mathtt{true} & \mathtt{M} \leqslant' \mathtt{0} & \rightarrow & \mathtt{false} \\
\mathtt{s(M)} \leqslant \mathtt{N} & \rightarrow & \mathtt{M} \leqslant' \mathtt{N} & \mathtt{M} \leqslant' \mathtt{s(N1)} & \rightarrow & \mathtt{M} \leqslant \mathtt{N1}
\end{array}$$

where $\leqslant'$ is a new function symbol.

The following theorem states a correspondence between needed narrowing derivations using the original program and lazy narrowing derivations in the transformed uniform program. For a more detailed comparison between needed narrowing and lazy narrowing, we refer to (Alpuente et al. 2003).

Theorem 2 (*Zartmann 1997*)
Let $\mathcal{R}$ be an inductively sequential program, $\mathcal{U}(\mathcal{R})$ the transformed uniform program, and t an operation-rooted term. Then there exists a needed narrowing derivation $t \leadsto^*_\sigma s$ w.r.t. $\mathcal{R}$ to a constructor root-stable form s iff there exists a lazy narrowing derivation $t \leadsto^*_\sigma s$ w.r.t. $\mathcal{U}(\mathcal{R})$.

5 Partial Evaluation with Needed Narrowing

In this section, we introduce the basic notions of PE in (lazy) functional logic programming. Then, we analyze the fundamental properties of PE based on needed narrowing and establish the relation with PE based on lazy narrowing.

Partial evaluation is a semantics-based program optimization technique which has been investigated within different programming paradigms and applied to a wide variety of languages. The first PE framework for functional logic programs has been defined by (Alpuente et al. 1998a). In this framework, narrowing (the standard operational semantics of integrated languages) is used to drive the PE process; similarly to partial deduction, specialized program rules are constructed from narrowing derivations using the notion of *resultant*. In the following, $s \leadsto^+_\sigma t$ denotes a narrowing derivation with at least one narrowing step.

Definition 2 (*resultant*)
Let $\mathcal{R}$ be a TRS and s be a term. Given a narrowing derivation $s \leadsto^+_\sigma t$, its associated resultant is the rewrite rule $\sigma(s) \rightarrow t$.

Note that, whenever the specialized call s is not a linear pattern, the left-hand sides of resultants may not be linear patterns either and hence resultants may not be program rules:

Example 5
Consider the following inductively sequential program:

$$\begin{array}{rcl}
\mathtt{double(X)} & \rightarrow & \mathtt{X}+\mathtt{X} \\
\mathtt{0}+\mathtt{N} & \rightarrow & \mathtt{N} \\
\mathtt{s(M)}+\mathtt{N} & \rightarrow & \mathtt{s(M}+\mathtt{N)}
\end{array}$$

Given the term $\mathtt{double(W)} + \mathtt{W}$ and the following needed narrowing derivation (the selected redex is underlined at each narrowing step):

$$\underline{\mathtt{double(W)}} + \mathtt{W} \leadsto_{\mathtt{id}} (\underline{\mathtt{W} + \mathtt{W}}) + \mathtt{W} \leadsto_{\{\mathtt{W} \mapsto \mathtt{s(M)}\}} \mathtt{s(M} + \mathtt{s(M))} + \mathtt{s(M)}$$

we compute the associated resultant:

$$\mathtt{double(s(M))} + \mathtt{s(M)} \to \mathtt{s(M} + \mathtt{s(M))} + \mathtt{s(M)}$$

This resultant is not a legal program rule since its left-hand side contains nested defined function symbols ("+" and "double") as well as multiple occurrences of the same variable.

In order to produce legal program rules, we introduce a post-processing of renaming which not only eliminates redundant structures but also obtains *independent* specializations in the sense of (Lloyd and Shepherdson 1991). Furthermore, it is also necessary for the correctness of the PE transformation. Roughly speaking, independence ensures that the different specializations for the same function definition are correctly distinguished, which is crucial for polyvariant specialization.

The *(pre–)partial evaluation* of a term s is obtained by constructing a (possibly incomplete) narrowing tree for s and then extracting the specialized definitions (the resultants) from the non–failing, root–to–leaf paths of the tree.

Definition 3 (*pre–partial evaluation*)
Let $\mathcal{R}$ be a TRS and s a term. Let T be a finite (possibly incomplete) narrowing tree for s in $\mathcal{R}$ such that no constructor root-stable term in the tree has been narrowed. Let $\overline{t_n}$ be the terms in the non-failing leaves of T. Then, the set of resultants $\{\sigma_i(s) \to t_i \mid i = 1, \ldots, n\}$ for the narrowing sequences $\{s \leadsto^{+}_{\sigma_i} t_i \mid i = 1, \ldots, n\}$ is called a pre–partial evaluation of s in $\mathcal{R}$.

The pre–partial evaluation of a set of terms S in $\mathcal{R}$ is defined as the union of the pre–partial evaluations for the terms of S in $\mathcal{R}$.

Example 6
Consider the following function append to concatenate two lists (here we use "nil" and ":" as constructors of lists):

$$\mathtt{append(nil}, \mathtt{Y_s}) \to \mathtt{Y_s}$$
$$\mathtt{append(X : X_s}, \mathtt{Y_s}) \to \mathtt{X : append(X_s}, \mathtt{Y_s})$$

together with the set of calls $S = \{\mathtt{append(append(X_s, Y_s), Z_s)}, \mathtt{append(X_s, Y_s)}\}$. Given the needed narrowing trees of Figure 2, the associated pre–partial evaluation of S in $\mathcal{R}$ is as follows:

$$\mathtt{append(append(nil, Y_s), Z_s)} \to \mathtt{append(Y_s, Z_s)}$$
$$\mathtt{append(append(X : X_s, Y_s), Z_s)} \to \mathtt{X : append(append(X_s, Y_s), Z_s)}$$
$$\mathtt{append(nil, Z_s)} \to \mathtt{Z_s}$$
$$\mathtt{append(Y : Y_s, Z_s)} \to \mathtt{Y : append(Y_s, Z_s)}$$

The following example illustrates that the restriction not to evaluate terms in constructor root-stable form cannot be dropped.

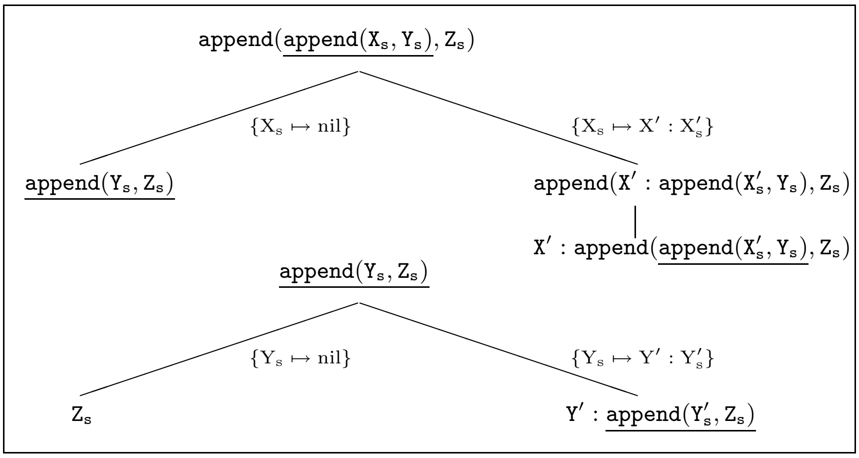

Fig. 2. Needed Narrowing trees for $\mathtt{append}(\mathtt{append}(\mathtt{X_s},\mathtt{Y_s}),\mathtt{Z_s})$ and $\mathtt{append}(\mathtt{X_s},\mathtt{Y_s})$.

Example 7
Consider the following program $\mathcal{R}$:

$$\begin{array}{rcl} \mathtt{f(0)} & \rightarrow & \mathtt{0} \\ \mathtt{g(X)} & \rightarrow & \mathtt{s(f(X))} \\ \mathtt{h(s(X))} & \rightarrow & \mathtt{s(0)} \end{array}$$

together with the set of calls $S = \{\mathtt{g(X)}, \mathtt{h(X)}\}$. Given the needed narrowing derivations:

$$\begin{array}{l} \underline{\mathtt{g(X)}} \leadsto_{\mathtt{id}} \mathtt{s(}\underline{\mathtt{f(X)}}\mathtt{)} \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} \mathtt{s(0)} \\ \underline{\mathtt{h(X)}} \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(Y)}\}} \mathtt{s(0)} \end{array}$$

a pre–partial evaluation of S in $\mathcal{R}$ is the following program $\mathcal{R}'$:

$$\begin{array}{rcl} \mathtt{g(0)} & \rightarrow & \mathtt{s(0)} \\ \mathtt{h(s(X))} & \rightarrow & \mathtt{s(0)} \end{array}$$

Now, the equation $\mathtt{h(g(s(0)))} \approx \mathtt{X}$ has the following successful needed narrowing derivation in $\mathcal{R}$:

$$\mathtt{h(}\underline{\mathtt{g(s(0))}}\mathtt{)} \approx \mathtt{X} \leadsto_{\mathtt{id}} \underline{\mathtt{h(s(f(s(0))))}} \approx \mathtt{X} \leadsto_{\mathtt{id}} \mathtt{s(0)} \approx \mathtt{X} \leadsto^{*}_{\{\mathtt{X}\mapsto\mathtt{s(0)}\}} \mathtt{true}$$

whereas it fails in the specialized program $\mathcal{R}'$.

The problem shown in the above example is due to the *backpropagation* of bindings to the left-hand sides of resultants: within a lazy context, the instantiation of the left-hand sides of resultants with bindings which come from the evaluation of terms in constructor root-stable form may incorrectly restrict the domain of functions (e.g., function "g" above).

A recursive *closedness* condition, which guarantees that each call which might occur during the execution of the resulting program is covered by some program rule, is formalized by inductively checking that the different calls in the rules are sufficiently covered by the specialized functions. For instance, a function call like $\mathtt{s(X)} + \mathtt{Y}$ cannot be considered *closed* w.r.t. the set of calls $\{\mathtt{0} + \mathtt{Y}, \mathtt{s(0)} + \mathtt{Y}\}$.

Informally, a term t rooted by a defined function symbol is closed w.r.t. a set of calls S, if it is an instance of a term of S and the terms in the matching substitution are recursively closed by S.

Definition 4 (*closedness*)
Let S be a finite set of terms. We say that a term t is S-closed if *closed*(S,t) holds, where the predicate *closed* is defined inductively as follows:

$$closed(S,t) \;\Leftrightarrow \begin{cases} true & \text{if } t \in \mathcal{X} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C}^*,\ n \geq 0 \\ \bigwedge_{x \mapsto t' \in \theta} closed(S,t') & \text{if } \exists s \in S \text{ such that } \theta(s) = t \\ & \;\text{for some substitution } \theta \end{cases}$$

where $\mathcal{C}^* = (\mathcal{C} \cup \{\approx, \wedge\})$.

We say that a set of terms T is S-closed, written *closed*(S,T), if *closed*(S,t) holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is S-closed if *closed*$(S,\mathcal{R}_{calls})$ holds. Here we denote by $\mathcal{R}_{calls}$ the set of the right-hand sides of the rules in $\mathcal{R}$.

For instance, the pre–partial evaluation of Example 6 is closed w.r.t. the set of partially evaluated calls $\{\mathtt{append(append(X_s,Y_s),Z_s)}, \mathtt{append(X_s,Y_s)}\}$.

According to the (non-deterministic) definition above, an expression rooted by a "primitive" function symbol, such as a conjunction $t_1 \wedge t_2$ or an equation $t_1 \approx t_2$, can be proved closed w.r.t. S either by checking that t_1 and t_2 are S-closed or by testing whether the conjunction (equation) is an instance of a call in S (followed by an inductive test of the subterms). This is useful when we are not interested in specializing complex expressions (like conjunctions or equations) but we still want to run them after specialization. Note that this is safe since we consider that the rules which define the primitive functions "$\approx$" and "$\wedge$" are automatically added to each program by existing programming environments, hence calls to these symbols are steadily covered in the specialized program. A general technique for dealing with primitive symbols which deterministically splits terms before testing them for closedness can be found in (Albert et al. 1998).

In general, given a call s and a program $\mathcal{R}$, there exists an infinite number of different pre–partial evaluations of s in $\mathcal{R}$. A fixed rule for generating resultants called an *unfolding rule* is assumed, which determines the expressions to be narrowed (by using a fixed narrowing strategy) and which decides how to stop the construction of narrowing trees; see (Albert et al. 1998; Alpuente et al. 1998a; Albert et al. 2002) for the definition of concrete unfolding rules.

In the following, we denote by pre–NN–PE and pre–LN–PE the sets of resultants computed for S in $\mathcal{R}$ by considering an unfolding rule which constructs finite needed and lazy narrowing trees, respectively. We will use the acronyms NN–PE and LN–PE for the renamed rules which will result from the corresponding post-processing of *renaming*. The idea behind this transformation is that, for any call (which is closed w.r.t. the considered set of calls), the answers computed for this call in the original program and the answers computed for the renamed call in the specialized, renamed program do coincide. In particular, in order to define a partial evaluator based on needed narrowing and to ensure that the resulting program is inductively sequential whenever the source program is, we have to make sure that the set of specialized terms (after renaming) contains only linear patterns with distinct root symbols. This can be ensured by introducing a new function symbol for each specialized term and then replacing each call in the specialized program by a call to the corresponding renamed function. In particular, the left-hand sides of the specialized program (which are constructor instances of the specialized terms) are replaced by instances of the corresponding new linear patterns through renaming.

Definition 5 (*independent renaming*)
An independent renaming ρ for a set of terms S is a mapping from terms to terms defined as follows: for $s \in S$, $\rho(s) = f_s(\overline{x_n})$, where $\overline{x_n}$ are the distinct variables in s in the left-to-right ordering and f_s is a new function symbol, which does not occur in $\mathcal{R}$ or S and is different from the root symbol of any other $\rho(s')$, with $s' \in S$ and $s' \neq s$. We also denote by $\rho(S)$ the set $S' = \{\rho(s) \mid s \in S\}$.

Example 8
Consider the set $S = \{\mathtt{append}(\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{Z_s}), \mathtt{append}(\mathtt{X_s}, \mathtt{Y_s})\}$. The following mapping:

$$\rho = \{\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}) \mapsto \mathtt{app}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{append}(\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{Z_s}) \mapsto \mathtt{dapp}(\mathtt{X_s}, \mathtt{Y_s}, \mathtt{Z_s})\}$$

is an independent renaming for S.

While independent renamings suffice to rename the left-hand sides of resultants (since they are constructor instances of the specialized calls), the right-hand sides are renamed by means of the auxiliary function ren_ρ, which *recursively* replaces each call in the given expression by a call to the corresponding renamed function (according to ρ).

Definition 6 (*renaming function*)
Let S be a finite set of terms and ρ an independent renaming of S. Given a term t, the non-deterministic function ren_ρ is defined as follows:

$$ren_\rho(t) = \begin{cases} t & \text{if } t \in \mathcal{X} \\ c(\overline{ren_\rho(t_n)}) & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C}^*, \text{ and } n \geq 0 \\ \theta'(\rho(s)) & \text{if } \exists \theta, \exists s \in S \text{ such that } t = \theta(s) \text{ and} \\ & \quad \theta' = \{x \mapsto ren_\rho(\theta(x)) \mid x \in \mathcal{D}om(\theta)\} \\ t & \text{otherwise} \end{cases}$$

where $\mathcal{C}^* = (\mathcal{C} \cup \{\approx, \wedge\})$.

Similarly to the test for closedness, an equation $s \approx t$ can be (non-deterministically) renamed either by independently renaming s and t or by replacing the considered equation by a call to the corresponding new, renamed function (when the equation is an instance of some specialized call in S). Note also that the renaming function is a *total* function: if an operation-rooted term t is not an instance of any term in S (which can occur if t is not S-closed), the function $ren_\rho(t)$ returns t itself (i.e., term t is not renamed).

The notion of partial evaluation can be formally defined as follows.

Definition 7 (*partial evaluation*)
Let $\mathcal{R}$ be a TRS, S a finite set of terms and $\mathcal{R}'$ a pre–partial evaluation of $\mathcal{R}$ w.r.t. S. Let ρ be an independent renaming of S. We define the partial evaluation $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. S (under ρ) as follows:

$$\mathcal{R}'' = \bigcup_{s \in S} \{\theta(\rho(s)) \to \mathit{ren}_\rho(r) \mid \theta(s) \to r \in \mathcal{R}' \text{ is a resultant for } s \text{ in } \mathcal{R}\}$$

We now illustrate these definitions with an example.

Example 9
Let us consider the program `append` and the set of terms S of Example 6, together with the independent renaming ρ of Example 8. A partial evaluation $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. S (under ρ) is:

$$\begin{array}{rcl}
\mathtt{dapp(nil, Y_s, Z_s)} & \to & \mathtt{app(Y_s, Z_s)} \\
\mathtt{dapp(X : X_s, Y_s, Z_s)} & \to & \mathtt{X : dapp(X_s, Y_s, Z_s)} \\
\mathtt{app(nil, Y_s)} & \to & \mathtt{Y_s} \\
\mathtt{app(X : X_s, Y_s)} & \to & \mathtt{X : app(X_s, Y_s)}
\end{array}$$

Note that, for a given renaming ρ, the renamed form of a program $\mathcal{R}$ may depend on the strategy which selects the term from $\rho(S)$ which is used to rename a given call t in $\mathcal{R}$ (e.g., $\mathtt{append(append(X_s, Y_s), Z_s)}$), since there may exist, in general, more than one term in S that covers the call t. Some potential specialization might be lost due to an inconvenient choice. Appropriate heuristics which are able to produce the best potential specialization have been introduced in the implementation of the partial evaluator described in (Albert et al. 2002).

The correctness of LN-PE is stated in (Albert et al. 1998; Alpuente et al. 1997). It is important to clarify that, even if the methodology for narrowing-driven PE in (Alpuente et al. 1998a) is parametric w.r.t. the narrowing strategy, this framework only ensures that:

- partially evaluated programs are *closed* w.r.t. the set of partially evaluated calls—which is necessary, although does not suffice, to guarantee the completeness of the transformation—, and
- the PE process always terminates.

In particular, the correctness of the PE transformation cannot be proved in a way independent of the narrowing strategy. These results are by their nature highly dependent on the concrete strategy which is considered, as it is known that different narrowing strategies have quite different semantic properties. In fact, the use of a lazy evaluation strategy imposes some additional restrictions on PE, such as the use of “strict equality”, the requirement not to evaluate terms in constructor root-stable form during PE, or the need for an additional post-processing of renaming. All these additional requirements are essential to ensure the correctness of the transformation and were not present in the original framework of (Alpuente et al. 1998a; Alpuente et al. 1998b), where correctness is only proved for an eager narrowing strategy. Therefore, it was necessary to develop a new theory for PE based on lazy narrowing as a separate work (Alpuente et al. 1997), which is now overcome by the needed narrowing methodology formalized in this article.

The following lemma shows that any PE based on needed narrowing can also be obtained (but possibly with more steps) by PE of the transformed uniform program based on lazy narrowing. This means that, in some sense, the specializations computed by a partial evaluator based on needed narrowing cannot be worse than the specializations computed by a partial evaluator based on lazy narrowing. On the other hand, we will also show later that there are cases where a LN-PE is worse than a NN-PE for the same original program.

Lemma 2
Let $\mathcal{R}$ be an inductively sequential program, $\mathcal{R}_u = \mathcal{U}(\mathcal{R})$ the corresponding uniform program, and S a finite set of operation-rooted terms. If $\mathcal{R}'$ is an NN-PE of S in $\mathcal{R}$, then $\mathcal{R}'$ is also an LN-PE of S in $\mathcal{R}_u$.

Proof
Since the final renaming applied in the partial evaluation of a program does not depend on the narrowing strategy used during the pre-partial evaluation, it suffices to show that each resultant w.r.t. needed narrowing in $\mathcal{R}$ corresponds to a resultant w.r.t. lazy narrowing in $\mathcal{R}_u$. Due to the definition of a resultant, each rule in the pre-partial evaluation w.r.t. needed narrowing in $\mathcal{R}$ has the form

$$\sigma(t) \to s$$

where $t \in S$ and $t \leadsto_\sigma^+ s$ is a needed narrowing derivation w.r.t. $\mathcal{R}$. By Theorem 2, there exists a lazy narrowing derivation $t \leadsto_\sigma^+ s$ w.r.t. $\mathcal{R}_u$ which has the same answer and result (note that Theorem 2 states this property only for derivations into constructor-rooted terms, but it also holds in the direction used here for arbitrary needed narrowing derivations since each needed narrowing step corresponds to a sequence of lazy narrowing steps w.r.t. the transformed uniform programs, which can be seen by the proof of this theorem). Thus, $\sigma(t) \to s$ is a resultant of this lazy narrowing derivation w.r.t. $\mathcal{R}_u$. $\square$

The following theorem states an important property of PE based on needed narrowing: if the input program is inductively sequential, then the partially evaluated program is also inductively sequential and, thus, we can also apply the needed narrowing strategy to evaluate calls in the specialized program. The proof of this theorem can be found in (Alpuente et al. 2004). An extension of this theorem—although it relies on the result below regarding the unfolding transformation—in the context of a more general fold/unfold framework can be found in (Alpuente et al. 2004).

Theorem 3
Let $\mathcal{R}$ be an inductively sequential program and S a finite set of operation-rooted terms. Then each NN-PE of $\mathcal{R}$ w.r.t. S is inductively sequential.

The following example reveals that, when we consider lazy narrowing, the LN-PE of a uniform program w.r.t. a linear pattern may not be uniform.

Example 10
Let $\mathcal{R}$ be the following uniform program:

$$\begin{array}{rcl} \mathtt{f(X,b)} & \to & \mathtt{g(X)} \\ \mathtt{g(a)} & \to & \mathtt{a} \end{array}$$

and $t = \mathtt{f(X,Y)}$ and $\rho(t) = \mathtt{f2(X,Y)}$. Then a LN-PE $\mathcal{R}'$ of t in $\mathcal{R}$ (under ρ) is

$$\mathtt{f2(a,b)} \quad \to \quad \mathtt{a}$$

which is not uniform.

The residual program $\mathcal{R}'$ in the example above is inductively sequential. This raises the question whether the LN-PE of a uniform program is always inductively sequential. Corollary 1 will positively answer this question.

Corollary 1
Let $\mathcal{R}$ be a uniform program and S a finite set of operation-rooted terms. If $\mathcal{R}'$ is a LN-PE of S in $\mathcal{R}$, then $\mathcal{R}'$ is inductively sequential.

Proof
Since a uniform program is inductively sequential and lazy narrowing steps w.r.t. uniform programs are also needed narrowing steps (cf. proof of Theorem 2), the proposition is a direct consequence of Theorem 3. □

The uniformity condition in Corollary 1 cannot be weakened to inductive sequentiality when LN-PEs are considered, as demonstrated by the following counterexample.

Example 11
Let $\mathcal{R}$ be the following inductively sequential program:

$$\begin{array}{rclcrcl} \mathtt{f(a,a,a)} & \to & \mathtt{b} & \quad & \mathtt{h(a,b,X)} & \to & \mathtt{b} \\ \mathtt{f(b,b,X)} & \to & \mathtt{b} & & \mathtt{h(e,X,k)} & \to & \mathtt{b} \\ \mathtt{g(a,b,X)} & \to & \mathtt{b} & & \mathtt{i(X,c,d)} & \to & \mathtt{b} \\ \mathtt{g(X,c,d)} & \to & \mathtt{b} & & \mathtt{i(e,X,k)} & \to & \mathtt{b} \end{array}$$

Let $t = \mathtt{f(g(X,Y,Z),h(X,Y,Z),i(X,Y,Z))} \in S$ and ρ be a renaming such that $\rho(t) = \mathtt{f3(X,Y,Z)}$. Then, every LN-PE $\mathcal{R}'$ of S in $\mathcal{R}$ (considering depth-2 lazy narrowing trees to construct the resultants) contains the rules:

$$\begin{array}{rcl} \mathtt{f3(a,b,X)} & \to & \cdots \\ \mathtt{f3(e,X,k)} & \to & \cdots \\ \mathtt{f3(X,c,d)} & \to & \cdots \end{array}$$

and thus $\mathcal{R}'$ is not inductively sequential.

One of the main factors affecting the quality of a PE is the treatment of choice points (Leuschel and Bruynooghe 2002; Gallagher 1993). The following examples illustrate the different way in which NN-PE and LN-PE "compile-in" choice points during unfolding, which is crucial to performance since a poor control choice during the construction of the computation trees can inadvertently introduce extra computation into a program.

Example 12
Consider again the rules of Example 3 and the input term $\mathtt{X} \leqslant \mathtt{X}+\mathtt{Y}$. The computed LN-PE is as follows:

$$\begin{array}{rcl} \mathtt{leq2(0,N)} & \rightarrow & \mathtt{true} \\ \mathtt{leq2(0,N')} & \rightarrow & \mathtt{true} \\ \mathtt{leq2(s(M),N)} & \rightarrow & \mathtt{leq2(M,N)} \end{array}$$

where the renamed initial term is leq2(X, Y). The redundancy of lazy narrowing has the effect that the first two rules of the specialized program are identical (up to renaming). In contrast, a better specialization—without generating redundant rules—is obtained by PE based on needed narrowing, since the NN-PE consists of the following rules:

$$\begin{array}{rcl} \mathtt{leq2(0,N)} & \rightarrow & \mathtt{true} \\ \mathtt{leq2(s(M),N)} & \rightarrow & \mathtt{leq2(M,N)} \end{array}$$

Note that a call-by-value partial evaluator based on innermost narrowing (Alpuente et al. 1998a) has an even worse behavior in this example since it does not specialize the program at all.

In the example above, the superfluous rule in the LN-PE can be avoided by removing duplicates in a post-processing step. The next example shows that this is not always possible.

Example 13
Lazy evaluation strategies are necessary if one wants to deal with infinite data structures and possibly non-terminating function calls. The following orthogonal program makes use of these features:

$$\begin{array}{rclcrcl} \mathtt{f(0,0)} & \rightarrow & \mathtt{s(f(0,0))} & \quad & \mathtt{g(0)} & \rightarrow & \mathtt{g(0)} \\ \mathtt{f(s(N),X)} & \rightarrow & \mathtt{s(f(N,X))} & & \mathtt{h(s(X))} & \rightarrow & \mathtt{0} \end{array}$$

The specialization is initiated with the term h(f(X, g(Y))). Note that this term reduces to 0 if X is bound to $\mathtt{s}(\cdots)$, and it does not terminate if X is bound to 0 due to the nonterminating evaluation of the second argument. The NN-PE of this program perfectly reflects this behavior (the renamed initial term is h2(X, Y)):

$$\begin{array}{rclcrcl} \mathtt{h0} & \rightarrow & \mathtt{h0} & \quad & \mathtt{h2(0,0)} & \rightarrow & \mathtt{h0} \\ & & & & \mathtt{h2(s(X),Y)} & \rightarrow & \mathtt{0} \end{array}$$

On the other hand, the LN-PE of this program has a worse structure:

$$\begin{array}{rclcrcl} \mathtt{h1(X)} & \rightarrow & \mathtt{h1(X)} & \quad & \mathtt{h2(X,0)} & \rightarrow & \mathtt{h1(X)} \\ \mathtt{h1(s(X))} & \rightarrow & \mathtt{0} & & \mathtt{h2(s(X),Y)} & \rightarrow & \mathtt{0} \\ & & & & \mathtt{h2(s(X),0)} & \rightarrow & \mathtt{0} \end{array}$$

The program specialized by LN-PE in the example above is not inductively sequential (nor orthogonal), in contrast to the original one. This does not only mean that lazy and needed narrowing are not applicable to the specialized program but also that the specialized program has a worse termination behavior than the original one. For instance, consider the term h(f(s(0), g(0))). The evaluation of this term has a finite derivation tree w.r.t. lazy narrowing as well as needed narrowing in the original program. However, the renamed term $\texttt{h2}(\texttt{s}(\texttt{0}),\texttt{0})$ has a finite derivation tree w.r.t. the NN-PE but an infinite derivation tree w.r.t. the LN-PE (using lazy narrowing); the infinite branch is caused by the application of the rules $\texttt{h2}(\texttt{X},\texttt{0}) \rightarrow \texttt{h1}(\texttt{X})$ and $\texttt{h1}(\texttt{X}) \rightarrow \texttt{h1}(\texttt{X})$.

This last example also shows that LN-PE can destroy the advantages of deterministic reduction of functional logic programs, which is not possible using NN-PE. This is ensured by the following theorem, which guarantees that a term which is deterministically normalizable w.r.t. the original program cannot cause a non-deterministic evaluation w.r.t. the specialized program obtained by NN-PE.

Theorem 4
Let $\mathcal{R}$ be an inductively sequential program, S a finite set of operation-rooted terms, ρ an independent renaming of S, and e an equation. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If e deterministically normalizes to *true* w.r.t. $\mathcal{R}$, then e' deterministically normalizes to *true* w.r.t. $\mathcal{R}'$.

Proof
Since e deterministically normalizes to *true* w.r.t. $\mathcal{R}$, there is a needed narrowing derivation $e \leadsto^*_{id}$ *true* in $\mathcal{R}$. By Theorem 5 (see below), there is a needed narrowing derivation $e' \leadsto^*_{\sigma}$ *true* in $\mathcal{R}'$ with $\sigma = id[\mathcal{V}ar(e)]$. This implies $\sigma = id$ by definition of needed narrowing. Therefore, e' deterministically normalizes to *true* w.r.t. $\mathcal{R}'$ by Proposition 3. □

This property of specialized programs is desirable and important from an implementation point of view, since the implementation of non-deterministic steps is an expensive operation in logic-oriented languages. Moreover, additional non-determinism in the specialized programs can result in additional infinite derivations, as shown in Example 13. This might have the effect that solutions are no longer computable in a sequential implementation based on backtracking. Essentially, deterministic computations are preserved thanks to the use of needed narrowing over inductively sequential programs to perform partial computations. For instance, consider the function "`leq`" of Example 1 together with the simple function "`foo`":

$$\texttt{foo}(\texttt{0}) \quad \rightarrow \quad \texttt{0}$$

Given a function call of the form $\texttt{X} \leqslant \texttt{foo}(\texttt{Y})$, many narrowing strategies (e.g., lazy narrowing) have two ways to proceed: either by reducing the call to function "$\leqslant$" using the first rule

$$\texttt{X} \leqslant \texttt{foo}(\texttt{Y}) \leadsto_{\{\texttt{X}\mapsto\texttt{0}\}} \texttt{true}$$

and by reducing the call to function "`foo`" (which is demanded by the second and third rules of "$\leqslant$")

$$\texttt{X} \leqslant \texttt{foo}(\texttt{Y}) \leadsto_{\{\texttt{Y}\mapsto\texttt{0}\}} \texttt{X} \leqslant \texttt{0}$$

Thus, their associated resultants are as follows:

$$\begin{array}{rcl}\texttt{0} \leqslant \texttt{foo(Y)} & \rightarrow & \texttt{true}\\ \texttt{X} \leqslant \texttt{foo(0)} & \rightarrow & \texttt{X} \leqslant \texttt{0}\end{array}$$

Now, given a call of the form $\texttt{0} \leqslant \texttt{foo(Z)}$, both resultants are applicable but the second one is clearly redundant. Actually, the second resultant is only meaningful to evaluate those calls whose first argument is of the form $\texttt{s}(\cdots)$, since only the second and third rules of "$\leqslant$" demanded the evaluation of call foo(0) that gave rise to this resultant. The advantage of using needed narrowing is that it applies some additional bindings so that this information is made explicit in the computed resultants, e.g., the resultants obtained by needed narrowing are

$$\begin{array}{rcl}\texttt{0} \leqslant \texttt{foo(Y)} & \rightarrow & \texttt{true}\\ \underline{\texttt{s(Z)}} \leqslant \texttt{foo(0)} & \rightarrow & \texttt{s(Z)} \leqslant \texttt{0}\end{array}$$

thus avoiding the creation of additional non-determinism. This property is somehow related to the notion of *perfect splits* used in (Abramov and Glück 2000; Abramov and Glück 2002; Glück and Klimov 1993) to guarantee that no computations are neither lost nor added when constructing—by driving (Turchin 1986), a symbolic execution mechanism which shares many similarities with lazy narrowing—the perfect process trees of (positive) supercompilation (Sørensen et al. 1996).

Note that there is no counterpart of this property in the partial deduction of logic programs, since the considered execution mechanism (some variant of SLD-resolution) never demands—in a *don't-know* non-deterministic way—the evaluation of different atoms of the same goal.

Finally, we state the strong correctness of NN-PE, which amounts to the computational equivalence between the original and the specialized programs (i.e., the fact that the two programs compute exactly the same answers) for the considered goals. The proof of this theorem can be found in (Alpuente et al. 2004).

Theorem 5 (*strong correctness*)
Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. Then, $e \leadsto^*_\sigma$ *true* is a needed narrowing derivation for e in $\mathcal{R}$ iff there exists a needed narrowing derivation $e' \leadsto^*_{\sigma'}$ *true* in $\mathcal{R}'$ such that $(\sigma' = \sigma)[V]$ (up to renaming).

It is worthwhile to note that the correctness of NN-PE cannot be derived from the correctness of LN-PE (Alpuente et al. 1997), since the preservation of inductive sequentiality (cf. Theorem 3) is a crucial point in our proof scheme, and this property does not hold for LN-PE.

On the other hand, it is well-known that partial evaluation can be defined within the fold/unfold framework (Pettorossi and Proietti 1996b) by using only unfolding and a restricted form of folding. Hence the correctness of NN-PE could be derived from the correctness of a fold/unfold framework for the transformation of functional logic programs based on needed narrowing. However, the only framework of this kind in the literature is (Alpuente et al. 1999; Alpuente et al. 2004) and their proofs of correctness—regarding the unfolding transformation—rely on the results in this article. The precise relation between partial evaluation and the fold/unfold transformation—for lazy functional logic programs—can be found in (Alpuente et al. 2000).

6 Further Developments

In the previous sections, we introduced the theoretical basis for PE in the context of lazy functional logic programming. Since the preliminary publication of these results, several extensions as well as concrete partial evaluators have been developed. In this section, we review some of these subsequent developments.

The computational model of modern declarative multi-paradigm languages, which integrate the most important features of functional, logic and concurrent programming, is based on a combination of two different operational principles: needed narrowing and residuation (Hanus 1997). The *residuation* principle is based on the idea of delaying function calls until they are sufficiently instantiated for a deterministic evaluation by rewriting. The particular mechanism (narrowing or residuation) is specified by *evaluation annotations*: deterministic functions are annotated as *rigid* (which forces a delayed evaluation by rewriting), while non-deterministic functions are annotated as *flexible* (which enables narrowing steps).

Although NN-PE is originally formulated for functional logic languages based uniquely on needed narrowing, it is still possible to adapt it to the use of distinct operational mechanisms. In fact, NN-PE has been already adjusted to perform partial computations using the combined operational semantics described above (Albert 2001; Albert et al. 1999).

On the other hand, NN-PE has also been extended (Albert et al. 2002) in order to make it viable for defining partial evaluators for *practical* multi-paradigm functional logic languages like Curry (Hanus (ed.) 2003) or Toy (López-Fraguas and Sánchez-Hernández 1999). When one considers a practical language, several extensions have to be considered, e.g., higher-order functions, concurrent constraints, calls to external functions, etc. In order to deal with these additional features, the underlying operational calculus becomes usually more complex. As we mentioned earlier, an on-line partial evaluator normally includes an interpreter of the language (Consel and Danvy 1993). Then, as the operational semantics becomes more elaborated, the associated PE techniques become (more powerful but) also increasingly more complex. To avoid this problem, an approach successfully tested in other contexts (Bondorf 1989; Glück and Klimov 1993; Nemytykh et al. 1996) is to consider the PE of programs written in a maximally simplified programming language.

Hanus and Prehofer (1999) have introduced a *flat* representation for functional logic programs in which definitional trees are embedded in the rewrite rules by means of case expressions:

Example 14

Function “$\leqslant$” of Example 1 can be written in the flat representation as follows:

$$\begin{array}{llllll}
\mathtt{X} \leqslant \mathtt{Y} = \mathtt{case}\ \mathtt{X}\ \mathtt{of} & \{ \ \mathtt{0} & \rightarrow & \mathtt{true}; & & \\
 & \ \ \ \mathtt{s(X_1)} & \rightarrow & \mathtt{case}\ \mathtt{Y}\ \mathtt{of} & \{ & \mathtt{0} \rightarrow \mathtt{false}; \\
 & & & & & \mathtt{s(Y_1)} \rightarrow \mathtt{X_1} \leqslant \mathtt{Y_1} \quad \} \quad \}
\end{array}$$

Two nice properties of the flat representation are that it provides more explicit control—hence the associated calculus is simpler than needed narrowing—and source programs can be automatically translated to the new representation. Moreover, it constitutes the basis of a recent proposal for an intermediate language, FlatCurry, used during the compilation of Curry programs (Antoy and Hanus 2000; Antoy et al. 2001). A new PE scheme (Albert 2001; Albert et al. 2002) has been designed by considering such a *flat* representation for functional logic programs.

However, the use of the standard semantics for flat programs—the LNT calculus (Hanus and Prehofer 1999), which is equivalent to needed narrowing—at PE time does not avoid the backpropagation of bindings when evaluating terms in constructor root-stable form, which can be problematic within a lazy context (see Example 7). In order to overcome this problem, a *residualizing* version of the standard semantics is introduced: the RLNT calculus (Albert 2001; Albert et al. 2003). Finally, since modern lazy functional logic languages can be automatically translated into this flat representation—which still contains all the necessary information about programs—the resulting technique is widely applicable.

All these results laid the ground for the development of a partial evaluation tool for Curry programs, which has been distributed with the Portland Aachen Kiel Curry System (Hanus (ed.) et al. 2003) since April 2001. Our partial evaluator constructs optimized, residual versions for selected calls of the input program. These calls are annotated by means of the function `PEVAL` which is equivalent to the identity function. Let us show a typical session with the partial evaluator. Here we consider the optimization of a program containing several calls to higher-order functions (since it is common to use higher-order combinators such as `map`, `foldr`, etc. in Curry programs). Although the use of such functions makes programs concise, some overhead is introduced at run time. Hence, we apply our partial evaluator to optimize calls to these functions. As a concrete example, consider the following (annotated) Curry program:[5]

```
main xs ys = (PEVAL (map (iter (+1) 2) xs)) ++ ys

iter f n = if n==0 then f else iter (comp f f) (n-1)
comp f g x = f (g x)

bench = main [1..20000] []
```

stored in the file `map_iter.curry`. Function `comp` is a higher-order function to compose two input functions, while `iter` composes a given function $2^{\mathtt{n}}$ times. Thus, given two input lists, `xs` and `ys`, function `main` adds 4 to each element of `xs`—the annotated expression—and then concatenates the result with the second list `ys`.

[5] Here we follow the Curry syntax: both variables and functions (except for `PEVAL`) start with lower case letters and function application is denoted by juxtaposition.

The built-in function "++" denotes list concatenation in Curry (more details can be found in (Hanus (ed.) 2003)). In order to measure the improvement achieved by the process, we have also included the function `bench` with a simple call to function `main`, where `[1..20000]` represents a list from `1` to `20000`. First, we load the program into PAKCS, turn on the time mode (to obtain the run time of computations), and execute function `bench`:

```
prelude> :l map_iter
...
compiled /tmp/map_iter.pl in module user, 620 msec 9888 bytes
map_iter> :set +time
map_iter> bench
Runtime: 750 msec.
Result: [5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,...]
```

Now, we run the partial evaluation tool and show the result of the process:

```
map_iter> :peval
...
Writing specialized program into "map_iter_pe.flc"...
Loading partially evaluated program "map_iter_pe"...
map_iter_pe> :show
main xs ys = (map_pe0 xs) ++ ys

iter f n = if n==0 then f else iter (comp f f) (n-1)
comp f g x = f (g x)

bench = main [1..20000] []

map_pe0 [] = []
map_pe0 (x : xs) = ((((x + 1) + 1) + 1) + 1) : map_pe0 xs
```

Only two modifications have been performed over the original program: the annotated expression has been replaced by a call to the new function `map_pe0` and the residual (first-order) definition of `map_pe0` has been added. In order to check the improvement achieved, we can run function `bench` again:

```
map_iter_pe> bench
Runtime: 170 msec.
Result: [5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,...]
```

Thus, the new program runs approximately 4.5 times faster than the original one. The reason is that it has a first-order definition and is completely "deforested" (Wadler 1990) in contrast to the original definition. In fact, the most successful experiences were achieved by specializing calls involving higher-order functions (obtaining speedups up to a factor of 9) and generic functions with some static data, like a string pattern matcher where a speedup of 14 was obtained; experimental results can be found in (Albert et al. 2002).

Note that all aforementioned proposals rely on the theoretical foundations presented in this work. Therefore, our results constitute the basis for the correctness of all these developments.

7 Conclusions

Few attempts have been made to investigate powerful and effective PE techniques which can be applied to term rewriting systems, logic programs and functional programs. In this work, we have introduced the theoretical basis for the PE of functional logic programs based on needed narrowing. We have proved its strong correctness, i.e., that the answers computed by needed narrowing in the original and specialized programs for the considered goals are identical (up to renaming). Furthermore, we have proved that the PE process keeps the inductively sequential structure of programs so that the needed narrowing strategy can also be used for the execution of specialized programs. As a consequence, our PE process preserves the following desirable property for functional logic programs: deterministic evaluations w.r.t. the original program are still deterministic in the specialized program. This property is nontrivial as witnessed by counterexamples for the case of lazy narrowing. This allows us to conclude that PE based on needed narrowing provides the best known basis for specializing functional logic programs.

To summarize, the notions presented in this article seem to be the most promising approach for the PE of modern functional logic languages based on a lazy semantics:

- We have shown that a partial evaluator based on lazy narrowing may lead from orthogonal programs to programs outside this class. This is clearly improved by PE based on needed narrowing as it preserves the original (inductively sequential) structure of programs, which is the only requirement for the completeness of the method.
- On the other hand, modern functional logic languages are based on (some form of) needed narrowing and, thus, this article is intended to be the foundational work in this area.

Finally, as we mentioned before, current approaches to the PE of multi-paradigm functional logic languages (Albert et al. 1999; Albert et al. 2002) rely on the theoretical foundations presented in this work. Therefore, our results provide the necessary basis for the correctness of all these subsequent developments.

References

Abramov, S. and Glück, R. 2000. The Universal Resolving Algorithm: Inverse Computation in a Functional Language. In *Proc. of the 5th Int'l Conf. on Mathematics of Program Construction (MPC 2000).* Springer LNCS 1837, 187–212.

Abramov, S. and Glück, R. 2002. The Universal Resolving Algorithm and its Correctness: Inverse Computation in a Functional Language. *Science of Computer Programming 43,* 2-3, 193–229.

Albert, E. 2001. Partial Evaluation of Multi-Paradigm Declarative Languages: Foundations, Control, Algorithms and Efficiency. Ph.D. thesis, DSIC, Technical University of Valencia. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Albert, E., Alpuente, M., Falaschi, M., Julián, P., and Vidal, G. 1998. Improving Control in Functional Logic Program Specialization. In *Proc. of the Int'l Static Analysis Symposium (SAS'98).* Springer LNCS 1503, 262–277.

Albert, E., Alpuente, M., Hanus, M., and Vidal, G. 1999. A Partial Evaluation Framework for Curry Programs. In *Proc. of the 6th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*. Springer LNCS 1705, 376–395.

Albert, E., Hanus, M., and Vidal, G. 2002. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming 2002,* 1, 1–34.

Albert, E., Hanus, M., and Vidal, G. 2003. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters 85,* 1, 19–25.

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

Alpuente, M., Falaschi, M., Julián, P., and Vidal, G. 1997. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*. Sigplan Notices 32(12). ACM Press, New York, 151–162.

Alpuente, M., Falaschi, M., Julián, P., and Vidal, G. 2003. Uniform Lazy Narrowing. *Journal of Logic and Computation 13,* 2, 287–312.

Alpuente, M., Falaschi, M., Moreno, G., and Vidal, G. 1999. A Transformation System for Lazy Functional Logic Programs. In *Proc. of the 4th Fuji Int'l Symposium on Functional and Logic Programming (FLOPS'99)*. Springer LNCS 1722, 147–162.

Alpuente, M., Falaschi, M., Moreno, G., and Vidal, G. 2000. An Automatic Composition Algorithm for Functional Logic Programs. In *Proc. of the 27th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2000)*. Springer LNCS 1963, 289–297.

Alpuente, M., Falaschi, M., Moreno, G., and Vidal, G. 2004. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science*. To appear.

Alpuente, M., Falaschi, M., and Vidal, G. 1998a. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems 20,* 4, 768–844.

Alpuente, M., Falaschi, M., and Vidal, G. 1998b. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys 30,* 3es, 9es.

Alpuente, M., Hanus, M., Lucas, S., and Vidal, G. 2004. Specialization of functional logic programs based on needed narrowing. *Computing Research Repository (CoRR) at* `http://arXiv.org/abs/cs.PL/0403011`.

Antoy, S. 1992. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*. Springer LNCS 632, 143–157.

Antoy, S. 1997. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming (ALP'97)*. Springer LNCS 1298, 16–30.

Antoy, S., Echahed, R., and Hanus, M. 1997. Parallel Evaluation Strategies for Functional Logic Languages. In *Proc. of the 14th Int'l Conference on Logic Programming (ICLP'97)*. MIT Press, Cambridge, Mass., 138–152.

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

Antoy, S. and Hanus, M. 2000. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS 2000)*. Springer LNCS 1794, 171–185.

Antoy, S., Hanus, M., Massey, B., and Steiner, F. 2001. An Implementation of Narrowing Strategies. In *Proc. of the 3rd Int'l ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*. ACM Press, 207–217.

BELLEGARDE, F. 1995. ASTRE: Towards a Fully Automated Program Transformation System. In *Proceedings of the Int'l Conf. on Rewriting Techniques and Applications (RTA'95)*. Springer LNCS 914, 403–407.

BONDORF, A. 1988. Towards a Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, D. Bjørner, A. Ershov, and N. Jones, Eds. North-Holland, Amsterdam, 27–50.

BONDORF, A. 1989. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of Int'l Conf. on Theory and Practice of Software Development, Barcelona, Spain*, J. Diaz and F. Orejas, Eds. Springer LNCS 352, 81–95.

CONSEL, C. AND DANVY, O. 1993. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages (POPL'93)*. ACM, New York, 493–501.

DE SCHREYE, D., GLÜCK, R., JØRGENSEN, J., LEUSCHEL, M., MARTENS, B., AND SØRENSEN, M. 1999. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming 41,* 2&3, 231–277.

DERSHOWITZ, N. AND JOUANNAUD, J.-P. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B: Formal Models and Semantics. Elsevier, Amsterdam, 243–320.

DERSHOWITZ, N. AND REDDY, U. 1993. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation 15*, 467–494.

GALLAGHER, J. 1993. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*. ACM, New York, 88–98.

GIOVANNETTI, E., LEVI, G., MOISO, C., AND PALAMIDESSI, C. 1991. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences 42*, 363–377.

GLÜCK, R. AND KLIMOV, A. 1993. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In *Proc. of the 3rd Int'l Workshop on Static Analysis (WSA'93)*. Springer LNCS 724, 112–123.

GLÜCK, R. AND SØRENSEN, M. 1994. Partial Deduction and Driving are Equivalent. In *Proc. Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*. Springer LNCS 844, 165–181.

GLÜCK, R. AND SØRENSEN, M. 1996. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, O. Danvy, R. Glück, and P. Thiemann, Eds. Springer LNCS 1110, 137–160.

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

HANUS, M. 1995. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. of the 5th Int'l Workshop on Logic Program Synthesis and Transformation (LOPSTR'95)*. Springer LNCS 1048, 252–266.

HANUS, M. 1997. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*. ACM, New York, 80–93.

HANUS, M., LUCAS, S., AND MIDDELDORP, A. 1998. Strongly Sequential and Inductively Sequential Term Rewriting Systems. *Information Processing Letters 67,* 1, 1–8.

HANUS, M. AND PREHOFER, C. 1999. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming 9,* 1, 33–75.

HANUS (ED.), M. 2003. Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry/`.

HANUS (ED.), M., ANTOY, S., ENGELKE, M., HÖPPNER, K., KOJ, J., NIEDERAU, P., SADRE, R., AND STEINER, F. 2003. PAKCS 1.5.0: The Portland Aachen Kiel Curry System User Manual. Tech. rep., University of Kiel, Germany.

HUET, G. AND LÉVY, J. 1992. Computations in Orthogonal Rewriting Systems, Part I + II. In *Computational Logic – Essays in Honor of Alan Robinson*, J. Lassez and G. Plotkin, Eds. The MIT Press, Cambridge, MA, 395–443.

JONES, N., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ.

KUCHEN, H., LOOGEN, R., MORENO-NAVARRO, J., AND RODRÍGUEZ-ARTALEJO, M. 1990. Lazy Narrowing in a Graph Machine. In *Proc. of the Int'l Conf. on Algebraic and Logic Programming (ALP'90)*. Springer LNCS 463, 298–317.

LAFAVE, L. AND GALLAGHER, J. 1997. Constraint-based Partial Evaluation of Rewriting-based Functional Logic Programs. In *Proc. of the Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'97)*. Springer LNCS 1463, 168–188.

LEUSCHEL, M. AND BRUYNOOGHE, M. 2002. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming 2,* 4 & 5, 461–515.

LLOYD, J. AND SHEPHERDSON, J. 1991. Partial Evaluation in Logic Programming. *Journal of Logic Programming 11*, 217–242.

LOOGEN, R., LÓPEZ-FRAGUAS, F., AND RODRÍGUEZ-ARTALEJO, M. 1993. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*. Springer LNCS 714, 184–200.

LÓPEZ-FRAGUAS, F. AND SÁNCHEZ-HERNÁNDEZ, J. 1999. TOY: A Multiparadigm Declarative System. In *Proc. of the Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*. Springer LNCS 1631, 244–247.

LUCAS, S. 1998. Root-Neededness and Approximations of Neededness. *Information Processing Letters 67,* 5, 245–254.

MIDDELDORP, A. 1997. Call by Need Computations to Root-Stable Form. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM, New York, 94–105.

MORENO-NAVARRO, J. AND RODRÍGUEZ-ARTALEJO, M. 1992. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming 12,* 3, 191–224.

NEMYTYKH, A., PINCHUK, V., AND TURCHIN, V. 1996. A Self-Applicable Supercompiler. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, O. Danvy, R. Glück, and P. Thiemann, Eds. Springer LNCS 1110, 322–337.

OYAMAGUCHI, M. 1993. NV-Sequentiality: a Decidable Condition for Call-by-Need Computations in Term-Rewriting Systems. *SIAM Journal of Computation 22,* 1, 114–135.

PETTOROSSI, A. AND PROIETTI, M. 1996a. A Comparative Revisitation of Some Program Transformation Techniques. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, O. Danvy, R. Glück, and P. Thiemann, Eds. Springer LNCS 1110, 355–385.

PETTOROSSI, A. AND PROIETTI, M. 1996b. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys 28,* 2, 360–414.

SØRENSEN, M., GLÜCK, R., AND JONES, N. 1996. A Positive Supercompiler. *Journal of Functional Programming 6,* 6, 811–838.

TURCHIN, V. 1986. Program Transformation by Supercompilation. In *Programs as Data Objects, 1985*, H. Ganzinger and N. Jones, Eds. Springer LNCS 217, 257–281.

WADLER, P. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73*, 231–248.

ZARTMANN, F. 1997. Denotational Abstract Interpretation of Functional Logic Programs. In *Proc. of the 4th Int'l Static Analysis Symposium (SAS'97)*, P. V. Hentenryck, Ed. Springer LNCS 1302, 141–159.