Narrowing-driven Specialization of Functional Logic Programs *

M. Alpuente[§] M. Falaschi [¶] G. Vidal[§]

Abstract

Languages that integrate functional and logic programming with a complete operational semantics are based on narrowing, a unification-based goal-solving mechanism which subsumes the reduction principle of functional languages and the resolution principle of logic languages. Formal methods of transformation of functional logic programs can be based on this well-established operational semantics. In this paper, we present a partial evaluation scheme for functional logic languages based on an automatic unfolding algorithm which builds narrowing trees. We study the semantic properties of the transformation and the conditions under which the technique terminates, is sound and complete, and is also generally applicable to a wide class of programs. To the best of our knowledge this is the first formal approach to partial evaluation of functional logic programs.

1 Introduction

Narrowing is the computation mechanism of languages that integrate functional and logic programming [54]. Narrowing solves equations by computing unifiers w.r.t. an equational theory usually described by means of a (conditional) term rewriting system. Function definition and evaluation are thus embedded within a logical framework and features such as existentially quantified variables, unification and program inversion become available.

Program transformation aims to derive better semantically equivalent programs. Partial evaluation (PE) is a program transformation technique which consists of the specialization of a program w.r.t. parts of their input [21]. To perform reductions at specialization time, a partial evaluator must include an interpreter [13]. The main issues with automatic partial evaluation (specialization) concern the choice of the basic transformation techniques, termination of the process, preserving the operational semantics of the original program, and effectiveness of the transformation, i.e. execution speedup for a large class of programs [60]. Two basic transformation techniques used in partial evaluation are the folding and unfolding transformations, first introduced by Burstall and Darlington in [11] for functional programs. Unfolding is essentially the replacement of a call by its body, with appropriate substitutions. Folding is the inverse transformation, the replacement of some piece of code by an equivalent function call. For functional programs, folding and unfolding steps involve only pattern matching. The fold/unfold transformation approach was first adapted to logic programs by Tamaki and Sato [61] by replacing matching with unification in the transformation rules. Because of the unification mechanism, partial evaluation of logic programs is also able to propagate syntactic information on the partial input, such as term structure, and not only constant values. Thus the mechanism of PE for logic programs results in general more powerful than for functional programs. Partial evaluation has been extensively studied both in functional [13, 37, 57] and in logic programming [22, 42, 46].

*This work has been partially supported by CICYT under grant TIC 95-0433-C03-03.

[§]DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain.

[¶]Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze 206, 33100 Udine, Italy.

In this paper, we show that, in the context of languages that integrate functional and logic programming [31], execution and specialization can be based on the unique computation mechanism of narrowing. This unified view of execution and transformation allows us to develop a simple and powerful framework for the PE of functional logic programs which improves the original program w.r.t. the ability of computing the set of answer substitutions. Moreover, we show that several optimizations are possible which are unique to the execution mechanism of functional logic programs (as it is the inclusion of a deterministic simplification process), and have the effect that functional logic programs are more efficiently specializable than equivalent logic programs. We demonstrate the usefulness of our framework by presenting an instance for innermost narrowing, which resembles the call-by-value case in functional programming. The effectiveness of the method is illustrated by various representative examples: specialization, in particular generation of efficient pattern matchers, and elimination of intermediate data structures.

Our PE procedure follows a structure similar to the framework developed for Logic Programming in [47]. Starting with the set of calls (terms) which appear in the initial goal, we partially evaluate them by using an unfolding strategy, and recursively specialize the terms which are introduced dinamically during this process. We introduce an appropriate abstract operator which guarantees termination and allows us to tune the specialization of the method.

We prove that our method passes the so-called Knuth-Morris-Pratt test [26, 36], i.e. specializing a naïve pattern matcher w.r.t. a fixed pattern obtains the efficiency of the Knuth, Morris and Pratt matching algorithm [41]. This seems to contradict the conjecture in [26] that call-by-value transformers cannot propagate information in a way similar to that in supercompilation [63]. The relation between our method and supercompilation is discussed in the following section. As a final remark, we emphasize that our PE procedure maintains the correctness and completeness of the transformed program w.r.t. a strong observable such as the set of computed answer substitutions of the original program.

1.1 Related work

Very little work has been done in the area of functional logic program specialization. In the literature we found only two noteworthy exceptions. In [45], Levi and Sirovich defined a PE procedure for the functional programming language TEL that uses a unification-based symbolic execution mechanism which can be understood as (a form of lazy) narrowing. In [14], Darlington and Pull showed how unification can enable instantiation and unfolding steps to be combined to get the ability (of narrowing) to deal with logical variables. A partial evaluator for the functional language HOPE (extended with unification) was also outlined. No actual procedure was included and no control issues were considered. The problems of ensuring termination and preserving semantics were not addressed in any of these papers. The use of unification with unfolding is also found in [17], where a rewrite-based approach to the synthesis of functional programs is formulated.

The work on supercompilation [62, 63] is, among the huge literature on program transformation, the closest to our work. Supercompilation (supervised compilation) is a transformation technique for functional programs which consists of three core constituents: *driving, generalization and generation of residual programs*. Supercompilation does not specialize the original program, but constructs a program for the (specialization of the) initial call by *driving* [26]. Driving can be understood as a unification-based function transformation mechanism, which uses some kind of evaluation machinery similar to (lazy) narrowing[1] to build (possibly infinite) 'trees of states' for a program with a given term. By virtue of driving, the supercompiler is able to get the same amount of (unification-based) information propagation and program specialization as in PE of logic programs.

[1]Turchin's papers use the following terminology [56]: 'generalized pattern matching' stands for unification, 'contractions' stand for narrowing substitutions and, very often in the texts, 'driving' stands for the narrowing itself, i.e., the operation of instantiation of a function call for all possible value cases of the arguments, followed by unfolding of the different branches.

Turchin's papers describe the supercompiler for Refal (Recursive Function Algorithmic Language), a pattern-matching functional language with a non-standard notion of patterns. The semantics of Refal is given in terms of a rewriting interpreter (with the inside-out order of evaluation), but driving is embedded into the supercompiler – an outside-in metaevaluator over Refal which subsumes deforestation [65], PE and other standard transformations of functional programming languages [60]. For example, supercompilation is able to support certain forms of theorem proving, program synthesis and program inversion. Supercompilation can also improve a program even if all the actual parameters in the function calls are variables, by removing some redundancies coming from nested loops, repeated variables, etc. Recent work by Glück and Klimov has expressed the essentials of driving in the context of a more traditional functional language with lists [25]. In [36], Jones put Turchin's driving methodology on a solid semantic foundation which is not tied to any particular programming language or data structure.

The driving process does not always terminate, and it does not preserve the semantics, as it can extend the domain of functions [60]. Techniques to ensure termination of driving are studied in [59, 64]. The idea of [64] is to supervise the construction of the tree and, at certain moments, loop back, i.e. fold a configuration to one of the previous states, and in this way construct a finite graph. The *generalization* operation which makes it possible to loop back the current configuration is often necessary. In [59], termination is guaranteed following a method which is comparable to the Martens-Gallagher general approach for ensuring global termination of PE for logic programs [47].

In [26], Glück and Sørensen focus on the correspondence between PE of logic programs (partial deduction) and driving, stating the similarities between driving of a functional program and the building of an SLD-tree for a similar Prolog program. The authors did not point out the close relationship between the driving and narrowing mechanisms. We think that exploiting this correspondence leads to a better understanding of how driving achieves its effects and makes it easier to answer many questions concerning correctness and termination of the transformation. Our results can be seen as a new formulation of the essential principle of driving in simpler and more familiar terms to the logic programming community. They also liberate the language of the strong syntactic restrictions imposed in [26, 60] in order not to encumber the formulation of driving algorithms. Our framework defines the first semantics-based PE scheme for functional logic programs.

1.2 Plan of the paper

This paper is organized as follows. In Section 2, basic definitions are given. Section 3 presents a general algorithm for PE based on narrowing and describes its properties. Partial correctness of the method is proved. In Section 4, we present our solution to the PE termination problem and make use of a deterministic simplification process which brings up further possibilities for specialization. The choice of an innermost narrowing strategy allows us to formalize in Section 5 a call-by-value partial evaluator for functional logic programs which we illustrate on the well-known string matching example. Section 6 concludes the paper with some remarks about the direction of future research. Proofs of selected theorems are given in the Appendix.

2 Preliminaries

We briefly recall some known results about rewrite systems and functional logic programming [16, 31, 32, 40]. The definitions below are given in the one-sorted case. The extension to many-sorted signatures is straightforward [51]. Throughout this paper, $\mathbf{V}$ will denote a countably infinite set of variables and Σ denotes a set of function symbols, each with a fixed associated arity. We also assume that Σ contains at least one 0-ary function symbol. $\tau(\Sigma \cup \mathbf{V})$ and $\tau(\Sigma)$ denote the sets of terms and ground terms built on Σ and $\mathbf{V}$, respectively. A Σ-equation $\mathbf{s} = \mathbf{t}$ is a pair of terms $\mathbf{s}, \mathbf{t} \in \tau(\Sigma \cup \mathbf{V})$. Terms are viewed as labelled trees in the usual way. The depth of $\mathbf{t}$, written $\mathbf{depth(t)}$, is defined recursively as follows. If $\mathbf{t}$ is a constant or a variable, then $\mathbf{depth(t)}$ is 1. Also, the depth of $\mathbf{f(t_1, \ldots, t_n)}$ is $1 + \mathbf{max}(\{\mathbf{depth(t_1)}, \ldots, \mathbf{depth(t_n)}\})$. Occurrences are represented by sequences, possibly empty, of natural numbers used to address subterms of $\mathbf{t}$, and they are ordered by the prefix ordering $\mathbf{u} \leq \mathbf{v}$, if there exists $\mathbf{w}$ such that $\mathbf{uw} = \mathbf{v}$. We let Λ denote the empty sequence. $\bar{\mathbf{O}}(\mathbf{t})$ denotes the set of nonvariable ocurrences of a term $\mathbf{t}$. $\mathbf{t}_{|\mathbf{u}}$ is the subterm at the occurrence $\mathbf{u}$ of $\mathbf{t}$. $\mathbf{t[r]_u}$ is the term $\mathbf{t}$ with the subterm at the occurrence $\mathbf{u}$ replaced with $\mathbf{r}$. These notions extend to equations and sequences of equations in a natural way. Identity of syntactic objects is denoted by $\equiv$. $\mathbf{Var(s)}$ is the set of distinct variables occurring in the syntactic object $\mathbf{s}$.

We restrict our interest to the set of idempotent substitutions over $\tau(\Sigma \cup \mathbf{V})$, which is denoted by $\mathbf{Sub}$. We consider the usual preorder on substitutions $\leq$: $\theta \leq \sigma$ iff $\exists \gamma.\ \sigma \equiv \theta\gamma$. This preorder induces a partial (pre-)ordering on terms given by: $\mathbf{t} \leq \mathbf{t'}$ iff $\exists \gamma.\ \mathbf{t'} \equiv \mathbf{t}\gamma$. The equational representation of a substitution $\theta = \{\mathbf{x_1/t_1}, \ldots, \mathbf{x_n/t_n}\}$ is the set of equations $\widehat{\theta} = \{\mathbf{x_1 = t_1}, \ldots, \mathbf{x_n = t_n}\}$. The identity function on $\mathbf{V}$ is called the empty substitution and denoted ϵ. In abuse of notation, $\mathbf{Dom}(\sigma) = \{\mathbf{x} \in \mathbf{V} \mid \mathbf{x}\sigma \not\equiv \mathbf{x}\}$ is called the domain of σ. A renaming is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} \equiv \rho^{-1}\rho \equiv \epsilon$. Two terms $\mathbf{t}$ and $\mathbf{t'}$ are variants (of each other), if there exists a renaming ρ such that $\mathbf{t}\rho \equiv \mathbf{t'}$. Given a substitution θ and a set of variables $\mathbf{W} \subseteq \mathbf{V}$, we denote by $\theta_{\restriction \mathbf{W}}$ the substitution obtained from θ by restricting its domain, $\mathbf{Dom}(\theta)$, to $\mathbf{W}$. A generalization of the nonempty set of terms $\{\mathbf{t_1}, \ldots, \mathbf{t_n}\}$ is a pair $\langle \mathbf{t}, \{\theta_1, \ldots, \theta_\mathbf{n}\}\rangle$ such that, for all $\mathbf{i} = 1, \ldots, \mathbf{n}$, $\mathbf{t}\theta_\mathbf{i} = \mathbf{t_i}$. The pair $\langle \mathbf{t}, \Theta\rangle$ is the *most specific generalization (msg)* of a set of terms $\mathbf{S}$, written $\langle \mathbf{t}, \Theta\rangle = \mathbf{msg(S)}$, if 1) $\langle \mathbf{t}, \Theta\rangle$ is a generalization of $\mathbf{S}$, and 2) any generalization of $\mathbf{S}$ is also a generalization of $\{\mathbf{t}\}$. The $\mathbf{msg}$ of a set of terms is unique up to variable renaming. A set of equations $\mathbf{E}$ is unifiable, if there exists $\vartheta \in \mathbf{Sub}$ such that for all $\mathbf{s = t}$ in $\mathbf{E}$. $\mathbf{s}\vartheta \equiv \mathbf{t}\vartheta$. ϑ is called a unifier of $\mathbf{E}$. We let $\mathbf{mgu(E)}$ denote the *most general unifier* of the equation set $\mathbf{E}$ (see, e.g., [43]).

An equational Horn theory $\mathcal{E}$ consists of a finite set of equational Horn clauses of the form $(\lambda = \rho) \Leftarrow \mathbf{C}$. The condition $\mathbf{C}$ is a (possibly empty) sequence $\mathbf{e_1}, \ldots, \mathbf{e_n}$, $\mathbf{n} \geq 0$, of equations. Variables in $\mathbf{C}$ or ρ that do not occur in λ are called extra-variables. An equational goal is an equational Horn clause with no head. We let *Goal* denote the set of equational goals. We often leave out the $\Leftarrow$ symbol when we write goals.

A Conditional Term Rewriting System (CTRS for short) is a pair $(\Sigma, \mathcal{R})$, where $\mathcal{R}$ is a finite set of reduction (or rewrite) rule schemes of the form $(\lambda \to \rho \Leftarrow \mathbf{C})$, $\lambda, \rho \in \tau(\Sigma \cup \mathbf{V})$, $\lambda \notin \mathbf{V}$ and $\mathbf{Var}(\rho) \cup \mathbf{Var(C)} \subseteq \mathbf{Var}(\lambda)$. If a rewrite rule has no condition we write $\lambda \to \rho$. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$.

A Horn equational theory $\mathcal{E}$ which satisfies the above assumptions can be viewed as a CTRS $\mathcal{R}$, where the rules are the heads (implicitely oriented from left to right) and the conditions are the respective bodies. We assume that these assumptions hold for all theories we consider in this paper. The equational theory $\mathcal{E}$ is said to be canonical, if the binary one-step rewriting relation $\to_\mathcal{R}$ defined by $\mathcal{R}$ is noetherian and confluent. A term $\mathbf{s}$ conditionally rewrites to a term $\mathbf{t}$, written $\mathbf{s} \to_\mathcal{R} \mathbf{t}$, if there exists $\mathbf{u} \in \mathbf{O(s)}$, $(\lambda \to \rho \Leftarrow \mathbf{s_1 = t_1}, \ldots, \mathbf{s_n = t_n}) \in \mathcal{R}$ and substitution σ such that $\mathbf{s}_{|\mathbf{u}} = \lambda\sigma$, $\mathbf{t} = \mathbf{s}[\rho\sigma]_\mathbf{u}$ and $\forall \mathbf{i}.\ 1 \leq \mathbf{i} \leq \mathbf{n}.\ \exists \mathbf{w_i}$ such that $\mathbf{s_i}\sigma \to^*_\mathcal{R} \mathbf{w_i}$ and $\mathbf{t_i}\sigma \to^*_\mathcal{R} \mathbf{w_i}$. The instantiated left-hand side $\lambda\sigma$ of a reduction rule $(\lambda \to \rho \Leftarrow \mathbf{C})$ is called a *redex* (*red*ucible *ex*pression) with *contractum* $\rho\sigma$. When no confusion can arise, we omit the subscript $\mathcal{R}$. A term $\mathbf{s}$ is a *normal form*, if there is no term $\mathbf{t}$ with $\mathbf{s} \to_\mathcal{R} \mathbf{t}$. We let $\mathbf{s}\!\downarrow$ denote the normal form of $\mathbf{s}$. For CTRS $\mathcal{R}$, $\mathbf{r} \ll \mathcal{R}$ denotes that $\mathbf{r}$ is a new variant of a rule in $\mathcal{R}$ such that $\mathbf{r}$ contains no variable previously met during computation (standardised apart). A substitution σ is *normalized*, if $\mathbf{x}\sigma$ is a normal form for all $\mathbf{x} \in \mathbf{Dom}(\sigma)$. A CTRS is *decreasing* if there exists a well-founded extension $\succ$ of the rewrite relation $\to_\mathcal{R}$ with the following properties: 1) $\succ$ has the *subterm property*, i.e. $\mathbf{t} \succ \mathbf{t}_{|\mathbf{u}}$ for all $\mathbf{u} \in \mathbf{O(t)} - \{\Lambda\}$, and 2) if $(\lambda \to \rho \Leftarrow \mathbf{C}) \in \mathcal{R}$ and σ is a substitution, then $\lambda\sigma \succ \rho\sigma$ and $\lambda\sigma \succ \mathbf{s}\sigma, \lambda\sigma \succ \mathbf{t}\sigma$ for all $\mathbf{s = t}$ in $\mathbf{C}$.

Narrowing is the operational principle for languages that integrate functional and logic programming. Narrowing solves equations by computing unifiers with respect to the given program.

Given a CTRS $\mathcal{R}$, an equational goal $\mathbf{g}$ conditionally narrows into a goal clause $\mathbf{g}'$ (in symbols $\mathbf{g} \overset{[\mathbf{u},\mathbf{r},\theta]}{\leadsto} \mathbf{g}'$, $\mathbf{g} \overset{[\mathbf{u},\theta]}{\leadsto} \mathbf{g}'$ or simply $\mathbf{g} \overset{\theta}{\leadsto} \mathbf{g}'$), if there exists an occurrence $\mathbf{u} \in \bar{\mathbf{O}}(\mathbf{g})$, a standardised apart variant $\mathbf{r} \equiv (\lambda \to \rho \ \Leftarrow \mathbf{C}) \ll \mathcal{R}$ and a substitution θ such that $\theta = \mathbf{mgu}(\{\mathbf{g}_{|\mathbf{u}} = \lambda\})$ and $\mathbf{g}' = (\mathbf{C}, \{\mathbf{g}[\rho]_{\mathbf{u}}\})\theta$. $\mathbf{s}$ is called a (narrowing) *redex* iff there exists a new variant $(\lambda \to \rho \ \Leftarrow \mathbf{C})$ of a reduction rule in $\mathcal{R}$ and a substitution σ such that $\mathbf{s}\sigma \equiv \lambda\sigma$. A *narrowing derivation* for $\mathbf{g}$ in $\mathcal{R}$ is defined by $\mathbf{g} \overset{\theta}{\leadsto}{}^* \mathbf{g}'$ iff $\exists\theta_1, \ldots, \theta_{\mathbf{n}}.\ \mathbf{g} \overset{\theta_1}{\leadsto} \ldots \overset{\theta_{\mathbf{n}}}{\leadsto} \mathbf{g}'$ and $\theta = \theta_1 \ldots \theta_{\mathbf{n}}$. We say that the derivation has length $\mathbf{n}$. If $\mathbf{n} = 0$, then $\theta = \epsilon$. In order to treat syntactical unification as a narrowing step, we add the rule $(\mathbf{x} = \mathbf{x} \to \mathbf{true})$, $\mathbf{x} \in \mathbf{V}$, to the CTRS $\mathcal{R}$. Then $\mathbf{s} = \mathbf{t} \overset{\sigma}{\leadsto} \mathbf{true}$ holds iff $\sigma = \mathbf{mgu}(\{\mathbf{s} = \mathbf{t}\})$. The extension of a CTRS $\mathcal{R}$ with the rewrite rule $(\mathbf{x} = \mathbf{x} \to \mathbf{true})$ is denoted by $\mathcal{R}_+$. We use $\top$ as a generic notation for sequences of the form $\mathbf{true}, \ldots, \mathbf{true}$. A successful derivation for $\mathbf{g}$ in $\mathcal{R}_+$ is a narrowing derivation $\mathbf{g} \overset{\theta}{\leadsto}{}^* \top$, and $\theta_{\lceil \mathbf{Var(g)}}$ is called a computed answer substitution for $\mathbf{g}$ in $\mathcal{R}$. We define the *success set* operational semantics of an equational goal $\mathbf{g}$ in the program $\mathcal{R}$ as the set of the computed answer substitutions corresponding to all successful narrowing derivations for $\mathbf{g}$ in $\mathcal{R}_+$, in symbols $\mathcal{O}_{\mathcal{R}}(\mathbf{g}) = \{\theta_{\lceil \mathbf{Var(g)}} \mid \mathbf{g} \overset{\theta}{\leadsto}{}^* \top\}$. The calculus of narrowing defines an algorithm, since all derivations can be easily enumerated. These derivations can be represented by a (possibly infinite) finitely branching tree. Following [46], we adopt the convention in this paper that any derivation is potentially incomplete (a branch thus can be failed, incomplete, successful or infinite). A *failing leaf* is a goal which is not $\top$ and which cannot be further narrowed.

Each equational Horn theory $\mathcal{E}$ generates a smallest congruence relation $=_{\mathcal{E}}$ called **$\mathcal{E}$-equality** on the set of terms $\tau(\Sigma \cup \mathbf{V})$ (the least equational theory which contains all logic consequences of $\mathcal{E}$ under the entailment relation $\models$ obeying the axioms of equality for $\mathcal{E}$). $\mathcal{E}$ is a presentation or axiomatization of $=_{\mathcal{E}}$. In abuse of notation, we sometimes speak of the equational theory $\mathcal{E}$ to denote the theory axiomatized by $\mathcal{E}$. Given two terms s and t, we say that they are **$\mathcal{E}$-unifiable** iff there exists a substitution σ such that $\mathbf{s}\sigma =_{\mathcal{E}} \mathbf{t}\sigma$, i.e. such that $\mathcal{E} \models \mathbf{s}\sigma = \mathbf{t}\sigma$. The substitution σ is called an **$\mathcal{E}$-unifier** of s and t. By abuse of notation, it is often called *solution*. **$\mathcal{E}$-unification** is semidecidable.

Given a set of variables $\mathbf{W} \subseteq \mathbf{V}$, **$\mathcal{E}$-equality** is extended to substitutions in the standard way, by $\sigma =_{\mathcal{E}} \theta[\mathbf{W}]$ iff $\mathbf{x}\sigma =_{\mathcal{E}} \mathbf{x}\theta\ \forall \mathbf{x} \in \mathbf{W}$. $\mathbf{W}$ will be omitted if equal to $\mathbf{V}$. We say σ is an $\mathcal{E}$-instance of σ' and σ' is more general than σ on $\mathbf{W}$, in symbols $\sigma' \leq_{\mathcal{E}} \sigma[\mathbf{W}]$ iff $(\exists\rho)\ \sigma =_{\mathcal{E}} \sigma'\rho[\mathbf{W}]$.

A set $\mathbf{S}$ of $\mathcal{E}$-unifiers of the equation set $\mathbf{E}$ is complete iff every $\mathcal{E}$-unifier σ of $\mathbf{E}$ factors into $\sigma =_{\mathcal{E}} \theta\gamma$ for some substitutions $\theta \in \mathbf{S}$ and γ. A complete set of $\mathcal{E}$-unifiers of a system of equations may be infinite. A narrowing algorithm is *complete* if it generates a complete set of **$\mathcal{E}$-unifiers** for all input equation systems. Formally, (a kind of) narrowing is complete for (a class of) CTRS's if the following condition holds: If $\mathbf{s}\sigma =_{\mathcal{E}} \mathbf{t}\sigma$, then there exists a narrowing derivation $\mathbf{s} = \mathbf{t} \overset{\theta}{\leadsto}{}^* \top$ such that $\theta \leq_{\mathcal{E}} \sigma[\mathbf{Var}(\mathbf{s}) \cup \mathbf{Var}(\mathbf{t})]$. It is well-known that the subscript $\mathcal{E}$ in $\theta \leq_{\mathcal{E}} \sigma$ can be dropped if we only consider completeness w.r.t. normalized substitutions [49]. Conditional narrowing has been shown to be a complete $\mathcal{E}$-unification algorithm for canonical theories satisfying different restrictions [31, 32, 49].

Since unrestricted narrowing has quite a large search space, several strategies to control the selection of redexes have been devised to improve the efficiency of narrowing by getting rid of some useless derivations. A *narrowing strategy* (or *position constraint*) is any well-defined criterion which obtains a smaller search space by permitting narrowing to reduce only some chosen positions, e.g. *basic* [33], *innermost* [20], *innermost basic* [32] or *lazy* narrowing [54]. Formally, a narrowing strategy φ is a mapping that assigns to every goal $\mathbf{g}$ (different from $\top$) a subset $\varphi(\mathbf{g})$ of $\bar{\mathbf{O}}(\mathbf{g})$ such that for all $\mathbf{u} \in \varphi(\mathbf{g})$ the goal $\mathbf{g}$ is narrowable at occurrence $\mathbf{u}$. An important property of a narrowing strategy φ is completeness, meaning that the narrowing constrained by φ is still complete. A survey of results about the completeness of narrowing strategies can be found in [12, 18, 51].

In the case of a confluent and decreasing CTRS $\mathcal{R}$, we can further improve narrowing without losing completeness by normalizing the goal before a narrowing step is applied [34]. A *normalizing* *conditional narrowing step* w.r.t. $\mathcal{R}$, $\mathbf{g} \overset{\sigma}{\leadsto} \mathbf{g'}\!\downarrow$, is given by a narrowing step $\mathbf{g} \overset{\sigma}{\leadsto} \mathbf{g'}$ followed by a normalization $\mathbf{g'} \to^*_{\mathcal{R}} \mathbf{g'}\!\downarrow$. It is useful to apply rewriting between narrowing steps, whenever it is possible, since rewriting may cut an infinite search space to a finite one and can save a lot of time and space [28]. The idea of exploiting deterministic computations by including normalization has been applied to almost all narrowing strategies, e.g. basic [50, 55], innermost [20], innermost basic [32], LSE [7] and lazy narrowing [30, 38].

3 PE of Functional Logic Programs

In this section, we present a generic procedure for the partial evaluation of functional logic programs and give its correctness. We use a terminology consistent with that of [46, 48].

In logic programming, the idea of PE is basically the following [46]. Let us consider a program $\mathbf{P}$ and an atomic goal $\mathbf{G}$. Then construct a (finite) SLD-tree for $\mathbf{P} \cup \{\mathbf{G}\}$ containing at least one nonroot node. From this tree the set of clauses $\{\mathbf{G}\theta_\mathbf{i} \leftarrow \mathbf{G_i}\}$, called resultants, is obtained by collecting the goal $\mathbf{G_i}$ and the corresponding $\theta_\mathbf{i}$, from each nonfailed leaf. Intuitively, resultants are conditional answers for the initial goal.

When considering functional programs with narrowing semantics, it is not immediate what a *resultant* should be. We formalize the resultant of a derivation for a term $\mathbf{s}$ in a canonical program $\mathcal{R}$ as follows.

Definition 3.1 *Let* $\mathbf{s}$ *be a term and* $\mathcal{R}$ *a canonical program. Consider the equation* $\mathbf{s} = \mathbf{y}$*, with the variable* $\mathbf{y} \notin \mathbf{Var}(\mathbf{s})$*. Let* $\mathbf{D} \equiv [(\mathbf{s} = \mathbf{y}) \overset{\theta}{\leadsto}{}^* \mathbf{g}, \mathbf{e}]$ *be a derivation for the goal* $\mathbf{s} = \mathbf{y}$ *in the extended program* $\mathcal{R}_+$*. Let* $\sigma = \mathbf{mgu}(\mathbf{e})$*. Then the resultant of the derivation is:* $\mathbf{resultant}(\mathbf{D}) = ((\mathbf{s} \to \mathbf{y})\theta \Leftarrow \mathbf{g})\sigma$*.*

The definition above looks a bit more involved than the one for logic programming. Let us try to give the intuition behind the computation of σ through a few simple examples.

Example 1 *Let the rule* $(\mathbf{f}(\mathbf{x}) \to \mathbf{x} \Leftarrow \mathbf{a} = \mathbf{x}, \mathbf{b} = \mathbf{x})$ *be in* $\mathcal{R}$*. The resultant of the derivation:*

1) $\mathbf{f}(\mathbf{f}(\mathbf{z})) = \mathbf{y} \overset{\{\mathbf{x}/\mathbf{f}(\mathbf{z})\}}{\leadsto} \mathbf{a} = \mathbf{f}(\mathbf{z}), \mathbf{b} = \mathbf{f}(\mathbf{z}), \mathbf{f}(\mathbf{z}) = \mathbf{y} \overset{\{\mathbf{y}/\mathbf{f}(\mathbf{z})\}}{\leadsto} \mathbf{a} = \mathbf{f}(\mathbf{z}), \mathbf{b} = \mathbf{f}(\mathbf{z}), \mathbf{true}$
is: $\mathbf{f}(\mathbf{f}(\mathbf{z})) \to \mathbf{f}(\mathbf{z}) \Leftarrow \mathbf{a} = \mathbf{f}(\mathbf{z}), \mathbf{b} = \mathbf{f}(\mathbf{z})$.

2) $\mathbf{f}(\mathbf{z}) = \mathbf{y} \overset{\{\mathbf{z}/\mathbf{x}\}}{\leadsto} \mathbf{a} = \mathbf{x}, \mathbf{b} = \mathbf{x}, \mathbf{x} = \mathbf{y}$
is: $\mathbf{f}(\mathbf{y}) \to \mathbf{y} \Leftarrow \mathbf{a} = \mathbf{y}, \mathbf{b} = \mathbf{y}$.
Note that, without applying the mgu $\sigma = \{\mathbf{x}/\mathbf{y}\}$ $(\sigma = \{\mathbf{y}/\mathbf{x}\})$ *of the last equation* $\mathbf{x} = \mathbf{y}$*, we would have obtained the rule:* $\mathbf{f}(\mathbf{x}) \to \mathbf{y} \Leftarrow \mathbf{a} = \mathbf{x}, \mathbf{b} = \mathbf{x}, \mathbf{x} = \mathbf{y}$*, which contains an extra-variable* $\mathbf{y}$*.*

3) $\mathbf{f}(\mathbf{z}) = \mathbf{y} \overset{\{\mathbf{z}/\mathbf{x}\}}{\leadsto} \mathbf{a} = \mathbf{x}, \mathbf{b} = \mathbf{x}, \mathbf{x} = \mathbf{y} \overset{\{\mathbf{x}/\mathbf{a}\}}{\leadsto} \mathbf{true}, \mathbf{b} = \mathbf{a}, \mathbf{a} = \mathbf{y}$
is: $\mathbf{f}(\mathbf{a}) \to \mathbf{a} \Leftarrow \mathbf{true}, \mathbf{b} = \mathbf{a}$.

The PE of a goal is defined by constructing incomplete search trees for the goal and extracting the specialized definition – the resultants – associated with the leaves of the trees. A resultant is trivial if it has the form $\mathbf{s} \to \mathbf{s}$.

Definition 3.2 *Let* $\mathcal{R}$ *be a program,* $\mathbf{s}$ *a term and* $\mathbf{y} \notin \mathbf{Var}(\mathbf{s})$ *a variable. Let* τ *be a finite (possibly incomplete) narrowing tree for the goal* $\mathbf{s} = \mathbf{y}$ *in the extended program* $\mathcal{R}_+$ *containing at least one nonroot node. Let* $\{\mathbf{g_i} \mid \mathbf{i} = 1, \ldots, \mathbf{k}\}$ *be the nonfailing leaves of the branches of* τ *and* $\{\mathbf{r_i} \mid \mathbf{i} = 1, \ldots, \mathbf{k}-1\}$ *the nontrivial resultants associated with the derivations* $\{(\mathbf{s} = \mathbf{y}) \overset{\sigma_\mathbf{i}}{\leadsto}{}^+ \mathbf{g_i} \mid \mathbf{i} = 1, \ldots, \mathbf{k}\}$*. Then, the set* $\{\mathbf{r_i} \mid \mathbf{i} = 1, \ldots, \mathbf{k}-1\}$ *is called a* partial evaluation of $\mathbf{s}$ in $\mathcal{R}$*. We also say that this set is* the *partial evaluation of* $\mathbf{s}$ *in* $\mathcal{R}$ using τ*.*

If $\mathbf{S}$ *is a finite set of terms (modulo variants), then a* partial evaluation *of* $\mathbf{S}$ *in* $\mathcal{R}$ *(or partial evaluation of* $\mathcal{R}$ *w.r.t.* $\mathbf{S}$*) is the union of the partial evaluations in* $\mathcal{R}$ *of the elements of* $\mathbf{S}$*. A partial evaluation of an equational goal* $\mathbf{s_1} = \mathbf{t_1}, \ldots, \mathbf{s_n} = \mathbf{t_n}$ *in* $\mathcal{R}$ *is the partial evaluation in* $\mathcal{R}$ *of the set* $\{\mathbf{s_1}, \ldots, \mathbf{s_n}, \mathbf{t_1}, \ldots, \mathbf{t_n}\}$*.*

From the above definition, it may appear that some potential for specialization might be lost due to the fact that the terms $\mathbf{s_i}$ and $\mathbf{t_i}$ of each equation $\mathbf{s_i} = \mathbf{t_i}$ of the goal are specialized independently. However, there is no real loss of generality in restricting our discussion to partial evaluation of terms, since the problem of solving an equation $\mathbf{s} = \mathbf{t}$ in the extended program $\mathcal{R}_+ = \mathcal{R} \cup \{\mathbf{x} = \mathbf{x} \to \mathbf{true}\}$ is equivalent to narrowing the 'term' $\mathbf{s} = \mathbf{t}$ to $\mathbf{true}$. Moreover, if there is at least one binary free function symbol $\otimes$ in the signature, the problem of solving a conjunction like $\mathbf{s_1} = \mathbf{t_1}, \ldots, \mathbf{s_n} = \mathbf{t_n}$ is equivalent to the problem of solving the equation $\otimes(\mathbf{s_1}, \ldots \otimes(\mathbf{s_{n-1}}, \mathbf{s_n})) = \otimes(\mathbf{t_1}, \ldots \otimes(\mathbf{t_{n-1}}, \mathbf{t_n}))$ [39]. The assumption that the initial goal is atomic simplifies the formal development of our framework and, as pointed out in [46], this requirement guarantees that, at each step of a derivation, the associated resultant is indeed a program rule. Also, due to the form of the initial query $\mathbf{s} = \mathbf{y}$, we need not care about how the new rule should be oriented one way or another in the case when $\mathcal{R}$ is terminating (by using a suitable ordering), since the rules $(\mathbf{s} \to \mathbf{y})\theta\sigma$ which are the heads of the produced resultants can be proven terminating, as we state in the following proposition.

Proposition 3.3 *The program obtained as the partial evaluation of a term in a noetherian program is noetherian.*

Following [46], we introduce a closedness condition under which our transformation is sound and complete w.r.t. the operational semantics of functional logic programs.

The following definitions are necessary for our notion of closedness. A function symbol $\mathbf{f} \in \Sigma$ is irreducible iff there is no rule $(\lambda \to \rho \Leftarrow \mathbf{C}) \in \mathcal{R}$ such that $\mathbf{f}$ occurs as the outermost function symbol in λ, otherwise it is a defined function symbol. In theories where the above distinction is made, the signature Σ is partitioned as $\Sigma = \mathcal{C} \biguplus \mathcal{F}$, where $\mathcal{C}$ is the set of irreducible function symbols and $\mathcal{F}$ is the set of defined function symbols. The members of $\mathcal{C}$ are also called *constructors*. A substitution σ is *(ground) constructor*, if $\mathbf{x}\sigma$ is (ground) constructor for all $\mathbf{x} \in \mathbf{Dom}(\sigma)$.

The function $\mathbf{terms}(\mathbf{O})$ extracts the terms appearing in the (set of) rule(s) or equation(s) $\mathbf{O}$.

Definition 3.4 *Let* $\mathbf{O}$ *be an expression. We define* $\mathbf{terms}(\mathbf{O})$ *as follows:*

$$\mathbf{terms}(\mathbf{O}) = \begin{cases} \bigcup_{\mathbf{i}=1}^{\mathbf{n}} \mathbf{terms}(\mathbf{o_i}) & \textit{if } \mathbf{O} \equiv \{\mathbf{o_1}, \ldots, \mathbf{o_n}\} \\ \mathbf{terms}(\{\mathbf{e_1}, \ldots, \mathbf{e_n}\}) \cup \{\rho\} & \textit{if } \mathbf{O} \equiv (\lambda \to \rho \Leftarrow \mathbf{e_1}, \ldots, \mathbf{e_n}) \\ \{\mathbf{s}, \mathbf{t}\} & \textit{if } \mathbf{O} \equiv (\mathbf{s} = \mathbf{t}) \end{cases}$$

Definition 3.5 *Let* $\mathbf{S}$ *and* $\mathbf{T}$ *be two finite set of terms. We say that* $\mathbf{T}$ *is* $\mathbf{S}$*-closed iff* $\mathbf{closed}(\mathbf{S}, \mathbf{T})$*, where the predicate* closed *is defined inductively as follows:*

$$\mathbf{closed}(\mathbf{S}, \mathbf{O}) \Leftrightarrow \begin{cases} \mathbf{true} & \textit{if } \mathbf{O} \equiv \emptyset \textit{ or } \mathbf{O} \equiv \mathbf{x} \in \mathbf{V} \\ \mathbf{closed}(\mathbf{S}, \mathbf{t_1}) \wedge \ldots \wedge \mathbf{closed}(\mathbf{S}, \mathbf{t_n}) & \textit{if } \mathbf{O} \equiv \{\mathbf{t_1}, \ldots, \mathbf{t_n}\} \\ \mathbf{closed}(\mathbf{S}, \{\mathbf{t_1}, \ldots, \mathbf{t_n}\}) & \textit{if } \mathbf{O} \equiv \mathbf{c}(\mathbf{t_1}, \ldots, \mathbf{t_n}),\ \mathbf{c} \in \mathcal{C} \\ (\exists \mathbf{s} \in \mathbf{S}.\ \mathbf{s}\theta = \mathbf{O}) \wedge \mathbf{closed}(\mathbf{S}, \mathbf{terms}(\widehat{\theta})) & \textit{if } \mathbf{O} \equiv \mathbf{f}(\mathbf{t_1}, \ldots, \mathbf{t_n}),\ \mathbf{f} \in \mathcal{F} \end{cases}$$

We say that a term $\mathbf{t}$ *is* $\mathbf{S}$*-closed if* $\mathbf{closed}(\mathbf{S}, \mathbf{t})$*.*

Definition 3.6 *Let* $\mathcal{R}$ *be a program and* $\mathbf{S}$ *be a finite set of terms. We say that* $\mathcal{R}$ *is* $\mathbf{S}$*-closed iff* $\mathbf{closed}(\mathbf{S}, \mathbf{terms}(\mathcal{R}))$*.*

The following example illustrates the need for the recursive inspection of subterms in the definition of closedness.

Example 2 *Consider the following program $\mathcal{R}$:*

$$\begin{array}{rcl}\mathbf{h}(\mathbf{x}) & \rightarrow & \mathbf{x}\\ \mathbf{f}(0) & \rightarrow & 0\\ \mathbf{f}(\mathbf{c}(\mathbf{x})) & \rightarrow & \mathbf{h}(\mathbf{f}(\mathbf{x}))\end{array}$$

and the initial goal $\mathbf{f}(\mathbf{c}(\mathbf{x})) = \mathbf{y}$. We consider that narrowing trees are expanded up to the frontier at depth one (i.e. trees are constructed by one-step trivial unfolding of goals). Then, the partial evaluation of $\mathcal{R}$ w.r.t. $\mathbf{S} = \{\mathbf{f}(\mathbf{c}(\mathbf{x})),\ \mathbf{h}(\mathbf{x})\}$ is the specialized program $\mathcal{R}'$:

$$\begin{array}{rcl}\mathbf{h}(\mathbf{x}) & \rightarrow & \mathbf{x}\\ \mathbf{f}(\mathbf{c}(\mathbf{x})) & \rightarrow & \mathbf{h}(\mathbf{f}(\mathbf{x}))\end{array}$$

Although each term appearing in $\mathcal{R}'$ is an instance of some term in $\mathbf{S}$, the program $\mathcal{R}'$ should not be considered closed w.r.t. $\mathbf{S}$ since the call $\mathbf{f}(\mathbf{x})$ occurring in the term $\mathbf{h}(\mathbf{f}(\mathbf{x}))$ (which appears in the rhs of the second rule of $\mathcal{R}'$) is not covered sufficiently by the rules of $\mathcal{R}'$. Actually, the goal $\mathbf{f}(\mathbf{c}(0)) = 0$, which is closed w.r.t. $\mathbf{S}$, succeeds in the program $\mathcal{R}$ with c.a.s. ϵ whereas it fails in $\mathcal{R}'$.

The partial evaluation theorem is formulated using the closedness condition.

Theorem 3.7 *Let $\mathcal{R}$ be a canonical program, $\mathbf{g}$ a goal, $\mathbf{S}$ a finite set of terms, and $\mathcal{R}'$ a partial evaluation of $\mathcal{R}$ w.r.t. $\mathbf{S}$. Then,*

1. (Soundness) $\theta \in \mathcal{O}_{\mathcal{R}'}(\mathbf{g}) \implies \exists\gamma \in \mathcal{O}_{\mathcal{R}}(\mathbf{g})$ *s.t.* $\gamma \leq_{\mathcal{E}} \theta\ [\mathbf{Var}(\mathbf{g})]$[2].
2. (Completeness) $\mathcal{O}_{\mathcal{R}}(\mathbf{g}) \subseteq \mathcal{O}_{\mathcal{R}'}(\mathbf{g})$, *if $\mathcal{R}' \cup \{\mathbf{g}\}$ is $\mathbf{S}$-closed.*

Now we introduce an independence condition that allows us to obtain a stronger version of the theorem as follows.

Definition 3.8 (overlap) *A term $\mathbf{s}$ overlaps a term $\mathbf{t}$ if there is a nonvariable subterm $\mathbf{s}_{|\mathbf{u}}$ of $\mathbf{s}$ such that $\mathbf{s}_{|\mathbf{u}}$ and $\mathbf{t}$ unify.*

Definition 3.9 (independence) *A set of terms $\mathbf{S}$ is independent if there are no different terms $\mathbf{s}$ and $\mathbf{t}$ in $\mathbf{S}$ such that $\mathbf{s}$ overlaps $\mathbf{t}$.*

Theorem 3.10 *Let $\mathcal{R}$ be a canonical program, $\mathbf{g}$ a goal, and $\mathbf{S}$ a finite set of terms such that for all $\mathbf{t} \in \mathbf{terms}(\mathbf{g})$, there exist $\mathbf{s} \in \mathbf{S}$ and a constructor substitution θ s.t. $\mathbf{s}\theta = \mathbf{t}$. Let $\mathcal{R}'$ be a partial evaluation of $\mathcal{R}$ w.r.t. $\mathbf{S}$ such that $\mathcal{R}'$ is $\mathbf{S}$-closed. Then,*

1. (Strong Soundness) $\mathcal{O}_{\mathcal{R}'}(\mathbf{g}) \subseteq \mathcal{O}_{\mathcal{R}}(\mathbf{g})$, *if $\mathbf{S}$ is independent.*
2. (Completeness) $\mathcal{O}_{\mathcal{R}'}(\mathbf{g}) \supseteq \mathcal{O}_{\mathcal{R}}(\mathbf{g})$.

The following example illustrates that the independence condition cannot be dropped.

Example 3 *Consider the following program $\mathcal{R}$:*

$$\begin{array}{rcl}\mathbf{g}(\mathbf{x}) & \rightarrow & \mathbf{x}\\ \mathbf{f}(0) & \rightarrow & 0\end{array}$$

and the set $\mathbf{S} = \{\mathbf{f}(\mathbf{x}), \mathbf{f}(\mathbf{g}(\mathbf{x}))\}$. A PE of $\mathcal{R}$ w.r.t. $\mathbf{S}$ is $\mathcal{R}' = \{\mathbf{f}(0) \rightarrow 0,\ \mathbf{f}(\mathbf{g}(\mathbf{x})) \rightarrow \mathbf{f}(\mathbf{x}),\ \mathbf{f}(\mathbf{g}(0)) \rightarrow 0\}$. Then $\mathcal{R}' \cup \{\mathbf{f}(\mathbf{x}) = \mathbf{y}\}$ is $\mathbf{S}$-closed and also has a refutation with computed answer $\theta = \{\mathbf{x}/\mathbf{g}(0), \mathbf{y}/0\}$. However, while $\mathcal{R} \cup \{\mathbf{f}(\mathbf{x}) = \mathbf{y}\}$ has a refutation, it does not have one with computed answer θ.

[2]In this result we consider that $\mathcal{E}$ is the theory axiomatized by $\mathcal{R}$.

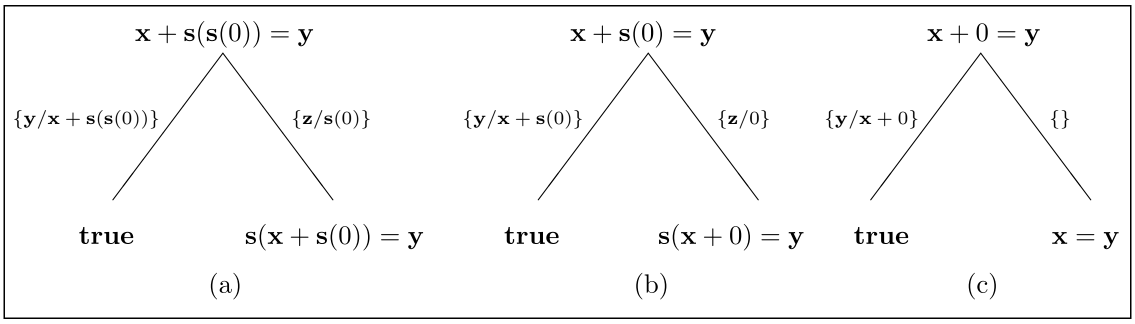

Figure 1: Narrowing trees for the goals $\mathbf{x} + \mathbf{s}(\mathbf{s}(0)) = \mathbf{y}$, $\mathbf{x} + \mathbf{s}(0) = \mathbf{y}$ and $\mathbf{x} + 0 = \mathbf{y}$.

Theorem 3.7 and Theorem 3.10 do not address the question of how the set **S** of terms should be computed to satisfy the required closedness (and independence) condition(s), or how PE should actually be performed. For the sake of simplicity, we do not consider in this paper the problem of the independence of the set of (to be) partially evaluated terms **S**, which should be obtained through some proper post-processing renaming transformation similar to that in [3, 4, 23].

Let us think of a simple PE method for functional logic programs which proceeds as follows. For a given goal **g** and program $\mathcal{R}$, a PE for **S** in $\mathcal{R}$ is computed, with **S** initialized to the set of terms appearing in **g**. Then this process is repeated for any term occurring in the right-hand side and in the body of the resulting rules which is not closed w.r.t. the set of terms already evaluated. Assuming that it terminates, the procedure computes a set of partially evaluated terms $\mathbf{S}'$ and a set of rules $\mathcal{R}'$ (the partial evaluation of $\mathbf{S}'$ in $\mathcal{R}$) such that each term in **S** is closed w.r.t. the set $\mathbf{S}'$ and the closedness condition for $\mathcal{R}' \cup \{\mathbf{g}\}$ is satisfied. The following example illustrates the method.

Example 4 *Consider the following program $\mathcal{R}$ that defines the addition on natural numbers which are constructed by* 0 *and* **s***:*

$$\begin{array}{rcl} \mathbf{x} + 0 & \rightarrow & \mathbf{x} \\ \mathbf{x} + \mathbf{s}(\mathbf{z}) & \rightarrow & \mathbf{s}(\mathbf{x} + \mathbf{z}) \end{array}$$

and the initial goal $\mathbf{x} + \mathbf{s}(\mathbf{s}(0)) = \mathbf{y}$. We consider that narrowing trees are expanded up to the frontier at depth one (i.e. trees are constructed by one-step trivial unfolding of goals). Starting with $\mathbf{S} = \{\mathbf{x} + \mathbf{s}(\mathbf{s}(0))\}$, and by using the procedure described above, we compute the trees depicted in Figure 1 for the set of terms $\mathbf{S}' = \{\mathbf{x} + \mathbf{s}(\mathbf{s}(0)), \mathbf{x} + \mathbf{s}(0), \mathbf{x} + 0\}$. The partial evaluation of $\mathbf{S}'$ in $\mathcal{R}$ is the following residual program $\mathcal{R}'$:

$$\begin{array}{rcl} \mathbf{x} + \mathbf{s}(\mathbf{s}(0)) & \rightarrow & \mathbf{s}(\mathbf{x} + \mathbf{s}(0)) \\ \mathbf{x} + \mathbf{s}(0) & \rightarrow & \mathbf{s}(\mathbf{x} + 0) \\ \mathbf{x} + 0 & \rightarrow & \mathbf{x} \end{array}$$

Note that $\mathcal{R}' \cup \{\mathbf{g}\}$ is $\mathbf{S}'$-closed.

It is worthwhile noting that, since the fact that **t** is an instance of a term in **S** does not ensure that **t** is **S**-closed, completeness is not guaranteed for all instances of the initial goal, as opposed to the case of pure logic programming.

Example 5 *Consider again the program of Example 4 and the goal $\mathbf{g}' \equiv (0 + \mathbf{w}) + \mathbf{s}(\mathbf{s}(0)) = \mathbf{y}$, which is not $\mathbf{S}'$-closed* (**with** $\mathbf{S}' = \{\mathbf{x} + \mathbf{s}(\mathbf{s}(0)), \mathbf{x} + \mathbf{s}(0), \mathbf{x} + 0\}$) *even if it can be obtained from the initial goal $\mathbf{x} + \mathbf{s}(\mathbf{s}(0)) = \mathbf{y}$ by instantiation. The c.a.s. $\theta = \{\mathbf{w}/\mathbf{s}(\mathbf{z}), \mathbf{y}/\mathbf{s}(\mathbf{s}(\mathbf{s}(0 + \mathbf{z})))\}$ for $\mathcal{R}$ with* **g** *cannot be obtained for* **g** *using the rules of $\mathcal{R}'$.*

There are two issues of correctness for a PE procedure: termination – given any input goal, execution should always reach a stage for which there is no way to continue, and (partial) correctness – (if execution terminates, then) the operational semantics of the goal w.r.t. the residual program and w.r.t. the original program would coincide.

As for termination, the PE procedure outlined above involves two termination problems, both concerned with unfolding [48, 44]. Given a term $\mathbf{s}$ and a program $\mathcal{R}$, there exist, in general, an infinite number of different partial evaluations of $\mathbf{s}$ in $\mathcal{R}$, since the construction of an unfolding tree for the initial goal in the program $\mathcal{R}_+$ is nondeterministic. The first problem when performing PE – the so-called "local termination" problem – is the termination of unfolding, or how to control and keep finite the expansion of the narrowing trees which provide partial evaluations for individual terms.

The global level of control concerns the termination of recursive unfolding, or how to stop recursively constructing narrowing trees while still guaranteeing that the desired amount of specialization is retained and that the closedness condition is reached. As we mentioned before, the set of terms $\mathbf{S}$ appearing in the goal with which the specialization is performed usually needs to be augmented in order to fulfill the closedness condition. This brings up the problem of how to keep this set finite throughout the PE process by means of some appropriate abstraction operator which guarantees termination. In the following, we establish a clear distinction between local and global control. This contrasts with the method proposed in [26, 59, 63], where both concerns are addressed simultaneously.

As for local termination, depth-bounds, loop-checks and well-founded orderings are some commonly used tools for controlling the unfolding during the construction of the search tree [10]. We will not address the details of local control in this section. We abstract from this problem by assuming that some suitable unfolding strategy is at our disposal which decides which terms to unfold and how to stop unfolding. We postpone to Section 4.1 the problem of guaranteeing that all sensible unfolding, and therefore specialization, is obtained.

As for global control, one possibility which does not just simply impose an ad-hoc bound to the number of terms in the set $\mathbf{S}$ is to use a proper (well-founded) generalization operator based on the notion of *most specific generalization*. As is well-known, using the msg can induce a loss of precision. Dynamic renaming, as defined in [3], takes advantage of the implicit static analysis performed by the partial evaluation to reduce limitations to be placed on the choice of the set and provides additional opportunities for polyvariant specialization (the possibility of producing a number of independent specializations for a given call using different data). The set of terms used for the transformation can also be found by means of abstract interpretation, as in [23].

The approach we follow originates from the framework for ensuring global termination of logic programs given in [48]. The extension of this method to a functional framework is a nontrivial one. In the following, we formalize a general algorithm for partial evaluation of functional logic programs based on narrowing which is proven to terminate (for proper instances) while ensuring that the closedness condition is satisfied, and still provides the right amount of polyvariance which allows us not to lose too much precision. Our algorithm is generic w.r.t. 1) the *narrowing relation* that constructs search trees, 2) the *unfolding rule* which determines when and how to terminate the construction of the trees, and 3) the *abstract operator* used to guarantee that the set of terms obtained during PE is finite.

We let $\leadsto_\varphi$ denote a generic (possibly normalizing) narrowing relation which uses the narrowing strategy φ. All notions concerning narrowing introduced so far can be extended to a narrower with strategy φ by replacing $\leadsto$ with $\leadsto_\varphi$ in the corresponding definition. In the following definition, we formalize the notion of a generic unfolding strategy $\mathbf{U}_{\leadsto_\varphi}$ (that we simply denote by $\mathbf{U}_\varphi$ when no confusion can arise) which constructs a (possibly incomplete) finite $\leadsto_\varphi$-narrowing tree and then extracts the resultants of the derivations of the tree. We consider each call, i.e. each operation $\mathbf{f} \in \Sigma$ applied to arbitrary terms, to be selectable in the unfolding process.

Definition 3.11 *An* unfolding rule $\mathbf{U}_\varphi$ *is a function which, when given a program* $\mathcal{R}$*, a term* $\mathbf{s}$

and a narrowing transition relation $\leadsto_\varphi$, returns a finite set of resultants $\mathbf{U}_\varphi(\mathbf{s},\mathcal{R})$ that is a partial evaluation of $\mathbf{s}$ *in $\mathcal{R}$ using $\leadsto_\varphi$.*

If $\mathbf{S}$ is a finite set of terms and $\mathcal{R}$ is a program, then the set of resultants obtained by applying $\mathbf{U}_\varphi$ to the term $\mathbf{s}$, for each $\mathbf{s} \in \mathbf{S}$, is called a partial evaluation of $\mathbf{S}$ in $\mathcal{R}$ using $\mathbf{U}_\varphi$ *(in symbols, $\mathbf{U}_\varphi(\mathbf{S},\mathcal{R})$).*

We formulate our method to compute a PE of a program $\mathcal{R}$ w.r.t. a finite set of terms $\mathbf{S}$ using $\mathbf{U}_\varphi$, by means of a transition system $(\mathbf{State}, \longmapsto_\mathcal{P})$ whose transition relation $\longmapsto_\mathcal{P} \subseteq \mathbf{State}\times\mathbf{State}$ formalizes the computation steps. The set $\mathbf{State}$ of PE configurations (states) is a parameter of the definition. The notion of state has to be instantiated in the specialization process. We let $\mathbf{c}[\mathbf{S}] \in \mathbf{State}$ denote a generic configuration whose structure is left unspecified as it depends on the specific PE algorithm, but which includes at least the set of (to be) partially evaluated terms $\mathbf{S}$. When $\mathbf{S}$ is clear from the context, $\mathbf{c}[\mathbf{S}]$ will simply be denoted by $\mathbf{c}$.

Definition 3.12 (PE transition relation $\longmapsto_\mathcal{P}$) *We define the PE relation $\longmapsto_\mathcal{P}$ as the smallest relation satisfying*

$$\frac{\mathcal{R}' = \mathbf{U}_\varphi(\mathbf{S},\mathcal{R})}{\mathbf{c}[\mathbf{S}] \longmapsto_\mathcal{P} \mathbf{abstract}(\mathbf{c}[\mathbf{S}], \mathbf{terms}(\mathcal{R}'))}$$

where the function $\mathbf{abstract}(\mathbf{c},\mathbf{T})$ extends the current configuration $\mathbf{c}$ with the set of terms $\mathbf{T}$ giving a new PE configuration.

Roughly speaking, at each computation step, the set of partially evaluated terms $\mathbf{S}$ (recorded in $\mathbf{c}$) is evaluated (using $\mathbf{U}_\varphi$). Then the terms appearing in the residual program $\mathcal{R}'$ which are not closed w.r.t. $\mathbf{S}$ are (properly) added to $\mathbf{c}$, as they are to be partially evaluated in the next iteration of the algorithm. To ensure termination, this combination is performed by applying an abstraction operator, which guarantees the finiteness of the set of terms for which partial evaluations are produced. Following [48], applying *abstract* in every iteration allows us to tune the control of polyvariance as much as needed. Also, it is within the *abstract* operation that the progress towards termination resides.

Definition 3.13 (initial PE configuration) *Let $\mathbf{g}$ be a goal and $\mathbf{c}_0$ be the "empty" PE state. The initial PE configuration is: $\mathbf{abstract}(\mathbf{c}_0, \mathbf{terms}(\mathbf{g}))$.*

Definition 3.14 (behaviour of the $\longmapsto_\mathcal{P}$ calculus) *Let us define the function:*
$\mathcal{P}(\mathcal{R},\mathbf{g}) = \mathbf{S}$ *if* $\mathbf{abstract}(\mathbf{c}_0, \mathbf{terms}(\mathbf{g})) \longmapsto^*_\mathcal{P} \mathbf{c}[\mathbf{S}]$ *and* $\mathbf{c}[\mathbf{S}] \longmapsto_\mathcal{P} \mathbf{c}[\mathbf{S}]$.

The procedure in Definition 3.14 computes the set of partially evaluated terms $\mathbf{S}$ which unambiguosly determines its associated partial evaluation $\mathcal{R}'$ in $\mathcal{R}$ (using $\mathbf{U}_\varphi$). The following theorem establishes the correctness of the PE method.

Theorem 3.15 (partial correctness of $\mathcal{P}$) *Let* abstract *be any abstraction operator satisfying that, if $\mathbf{abstract}(\mathbf{c}_1[\mathbf{S}_1], \mathbf{S}') = \mathbf{c}_2[\mathbf{S}_2]$, $\mathbf{then}(\mathbf{S}_1 \cup \mathbf{S}')$ is $\mathbf{S}_2$-closed. If $\mathcal{P}(\mathcal{R},\mathbf{g})$ terminates computing the set of terms $\mathbf{S}$, then $\mathcal{R}' \cup \{\mathbf{g}\}$ is $\mathbf{S}$-closed, where the specialized program $\mathcal{R}' = \mathbf{U}_\varphi(\mathbf{S},\mathcal{R})$.*

Definition 3.12 incorporates only the scheme of a complete method for partial evaluation. The resulting partial evaluations can be further optimized by eliminating redundant functors and unnecessary repetition of variables, trying to adapt standard techniques presented in [3, 4, 22, 23]. This is an interesting open problem in our setting, where functions appearing as arguments of calls are by no way "dead" structures, but can also generate new calls to function definitions. We consider this issue as a task for further research. The resulting mechanism should serve, among other purposes, to remove any remaining lack of independence.

In the following section we present our solution to the termination problem.

4 Ensuring Termination

4.1 Local Termination

In Section 3, the problem of obtaining (sensibly expanded) finite narrowing trees was shifted to that of defining sensible unfolding strategies that somehow ensure that infinite unfolding is not attempted. In this section, we introduce an unfolding rule which tries to maximize unfolding while retaining termination. Our strategy is simple but less crude than imposing an ad-hoc depth-bound, and still guarantees finite unfolding in all cases. The inspiration for our method comes from [59].

A commonly used tool for proving termination properties is based on the intuitive notion of orderings in which a term that is "syntactically simpler" than another is smaller than the other. The next definition extends the homeomorphic embedding ("syntactically simpler") relation $\trianglelefteq$ [15] to nonground terms.

Definition 4.1 (homeomorphic embedding relation) [59] *The homeomorphic embedding relation $\trianglelefteq$ on terms in $\tau(\Sigma \cup \mathbf{V})$ is defined as the smallest relation satisfying $\mathbf{x} \trianglelefteq \mathbf{y}$ for all $\mathbf{x}, \mathbf{y} \in \mathbf{V}$ and:*

$$\mathbf{s} \equiv \mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_\mathbf{m}) \trianglelefteq \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t}_\mathbf{n}) \equiv \mathbf{t}$$

if and only if
1) $\mathbf{f} \equiv \mathbf{g}$ (and $\mathbf{m} \equiv \mathbf{n}$) and $\mathbf{s_i} \trianglelefteq \mathbf{t_i}$ for all $\mathbf{i} = 1, \ldots, \mathbf{n}$, or
2) $\mathbf{s} \trianglelefteq \mathbf{t_j}$, for some $\mathbf{j}$, $1 \le \mathbf{j} \le \mathbf{n}$.

Roughly speaking, $\mathbf{s} \trianglelefteq \mathbf{t}$ if $\mathbf{s}$ may be obtained from $\mathbf{t}$ by deletion of operators. For example, $\surd\surd(\mathbf{u} \times (\mathbf{u} + \mathbf{v})) \trianglelefteq (\mathbf{w} \times \surd\underline{\surd\surd}((\surd\underline{\mathbf{u}} + \surd\mathbf{u})\underline{\times}(\surd\underline{\mathbf{u}+}\surd\underline{\mathbf{v}})))$.

A derivation $\mathbf{t}_1 \Rightarrow \mathbf{t}_2 \Rightarrow \ldots$ is self-embedding if $\mathbf{t_j} \trianglelefteq \mathbf{t_k}$ for some $\mathbf{j} < \mathbf{k}$. The following result is a consequence of Kruskal's Tree Theorem.

Theorem 4.2 [59] *Any infinite sequence of terms $\mathbf{t}_1, \mathbf{t}_2, \ldots$ with a finite number of operators is self-embedding, that is, there are numbers $\mathbf{j}, \mathbf{k}$ with $\mathbf{j} < \mathbf{k}$ and $\mathbf{t_j} \trianglelefteq \mathbf{t_k}$.*

The embedding relation $\trianglelefteq$ will be used in Section 4.2 to define an abstraction operator that guarantees global termination of the selected instance of the PE method. Now we use $\trianglelefteq$ to give a sufficient condition for local termination, that is, a condition which guarantees that narrowing trees are not expanded infinitely in depth.

The following criteria makes use of the embedding relation in a constructive way to produce finite narrowing trees. In order to avoid an infinite sequence of "diverging" calls, we compare each narrowing redex of the current goal with the selected redexes in the ancestor goals of the same derivation, and expand the narrowing tree under the constraints imposed by the comparison. When the compared calls are in the embedding relation, we stop the derivation. We need the following notations.

Definition 4.3 (comparable terms) *Let $\mathbf{s}$ and $\mathbf{t}$ be terms. We say that $\mathbf{s}$ and $\mathbf{t}$ are comparable iff the outermost function symbol of $\mathbf{s}$ and $\mathbf{t}$ coincide. Formally,*
$\mathbf{comparable}(\mathbf{s}, \mathbf{t}) \Leftrightarrow \mathbf{s} \equiv \mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s_n}) \ \wedge\ \mathbf{t} \equiv \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t_m}) \ \wedge\ \mathbf{f} \equiv \mathbf{g}$.

Definition 4.4 (admissible derivation) *Let $\mathbf{D}$ be a narrowing derivation for $\mathbf{g}_0$ in $\mathcal{R}$. We say that $\mathbf{D}$ is admissible iff it does not contain a pair of comparable redexes included in the embedding relation $\trianglelefteq$. Formally,*

$$\mathbf{admissible}(\mathbf{g}_0 \overset{[\mathbf{u}_0, \theta_0]}{\leadsto_\varphi} \ldots \overset{[\mathbf{u}_{\mathbf{n}-1}, \theta_{\mathbf{n}-1}]}{\leadsto_\varphi} \mathbf{g_n}) \Leftrightarrow \ \forall \mathbf{i} = 1, \ldots, \mathbf{n},\ \forall \mathbf{u} \in \varphi(\mathbf{g_i}),\ \forall \mathbf{j} = 0, \ldots, \mathbf{i} - 1. \ (\mathbf{comparable}(\mathbf{g}_{\mathbf{j}|\mathbf{u_j}}, \mathbf{g}_{\mathbf{i}|\mathbf{u}}) \Rightarrow \mathbf{g}_{\mathbf{j}|\mathbf{u_j}} \not\trianglelefteq \mathbf{g}_{\mathbf{i}|\mathbf{u}}).$$

To formulate the unfolding strategy, we also introduce the following preparatory definition.

Definition 4.5 (nonembedding narrowing tree $\tau_\varphi^{\trianglelefteq}$)

$$\tau_\varphi^{\trianglelefteq}(\mathbf{g_0},\mathcal{R}) = \{ \quad \mathbf{g_0} \overset{[\mathbf{u_0},\theta_0]}{\rightsquigarrow_\varphi} \ldots \overset{[\mathbf{u_{n-1}},\theta_{\mathbf{n-1}}]}{\rightsquigarrow_\varphi} \mathbf{g_n} \overset{[\mathbf{u_n},\theta_\mathbf{n}]}{\rightsquigarrow_\varphi} \mathbf{g_{n+1}} \mid$$
$$\mathbf{admissible}(\mathbf{g_0} \overset{[\mathbf{u_0},\theta_0]}{\rightsquigarrow_\varphi} \ldots \overset{[\mathbf{u_{n-1}},\theta_{\mathbf{n-1}}]}{\rightsquigarrow_\varphi} \mathbf{g_n}) \textit{ and}$$
$$(\mathbf{g_{n+1}} = \top, \textit{ or } \mathbf{g_{n+1}} \textit{ is a failing leaf, or}$$
$$(\exists \mathbf{u} \in \varphi(\mathbf{g_{n+1}}),\ \exists \mathbf{i} \in \{1,\ldots,\mathbf{n}\}.\ \mathbf{comparable}(\mathbf{g_{i|u_i}},\mathbf{g_{n+1|u}}) \wedge \mathbf{g_{i|u_i}} \trianglelefteq \mathbf{g_{n+1|u}}))\}.$$

Roughly speaking, derivations are stopped when they either fail, succeed or the considered redexes satisfy the embedding ordering. Before illustrating Definition 4.5 by means of one simple example, we state the following theorem.

Theorem 4.6 (local termination) *For a program $\mathcal{R}$ and goal* $\mathbf{g}$*, $\tau_\varphi^{\trianglelefteq}(\mathbf{g},\mathcal{R})$ is a finite (possibly incomplete) narrowing tree for $\mathcal{R} \cup \{\mathbf{g}\}$ using $\rightsquigarrow_\varphi$.*

Example 6 *Consider the well-known program append/2 with initial query* $\mathbf{append}(1:2:\mathbf{x_s},\mathbf{y_s}) = \mathbf{y}$. *We use '***nil***' and ':' as constructors of lists*[3].

$$\begin{array}{rcl} \mathbf{append}(\mathbf{nil},\mathbf{y_s}) & \to & \mathbf{y_s} \\ \mathbf{append}(\mathbf{x}:\mathbf{x_s},\mathbf{y_s}) & \to & \mathbf{x}:\mathbf{append}(\mathbf{x_s},\mathbf{y_s}) \end{array}$$

There exists the following infinite branch in the (unrestricted) narrowing tree (at each step we show, by means of underlining, the redex selected for narrowing):

$$\begin{array}{ll} \underline{\mathbf{append}(1:2:\mathbf{x_s},\mathbf{y_s})} = \mathbf{y} & \overset{\{\}}{\rightsquigarrow} 1:\underline{\mathbf{append}(2:\mathbf{x_s},\mathbf{y_s})} = \mathbf{y} \\ & \overset{\{\}}{\rightsquigarrow} 1:2:\underline{\mathbf{append}(\mathbf{x_s},\mathbf{y_s})} = \mathbf{y} \\ & \overset{\{\mathbf{x_s}/\mathbf{x}':\mathbf{x}'_\mathbf{s}\}}{\rightsquigarrow} 1:2:\mathbf{x}':\underline{\mathbf{append}(\mathbf{x}'_\mathbf{s},\mathbf{y_s})} = \mathbf{y} \\ & \overset{\{\mathbf{x}'_\mathbf{s}/\mathbf{x}'':\mathbf{x}''_\mathbf{s}\}}{\rightsquigarrow} \ldots \end{array}$$

According to Definition 4.5, the development of this branch is stopped at the fourth goal, since the derivation $\mathbf{append}(1:2:\mathbf{x_s},\mathbf{y_s}) \overset{\{\}}{\rightsquigarrow} 1:\mathbf{append}(2:\mathbf{x_s},\mathbf{y_s}) = \mathbf{y} \overset{\{\}}{\rightsquigarrow} 1:2:\mathbf{append}(\mathbf{x_s},\mathbf{y_s}) = \mathbf{y}$ *is admissible, and the step* $1:2:\underline{\mathbf{append}(\mathbf{x_s},\mathbf{y_s})} = \mathbf{y} \overset{\{\mathbf{x_s}/\mathbf{x}':\mathbf{x}'_\mathbf{s}\}}{\rightsquigarrow} 1:2:\mathbf{x}':\underline{\mathbf{append}(\mathbf{x}'_\mathbf{s},\mathbf{y_s})} = \mathbf{y}$ *fulfils the ordering, because* $\mathbf{append}(\mathbf{x_s},\mathbf{y_s}) \trianglelefteq \mathbf{append}(\mathbf{x}'_\mathbf{s},\mathbf{y_s})$.

Now we introduce the unfolding strategy induced by our notion of nonembedding tree.

Definition 4.7 (nonembedding unfolding rule $\mathbf{U}_\varphi^{\trianglelefteq}$)
We define $\mathbf{U}_\varphi^{\trianglelefteq}(\mathbf{s},\mathcal{R})$ as the partial evaluation of $\mathbf{s}$ *in $\mathcal{R}$ using $\tau_\varphi^{\trianglelefteq}(\mathbf{s}=\mathbf{y},\mathcal{R})$, $\mathbf{y} \notin \mathbf{Var}(\mathbf{s})$.*

Nontermination of the PE procedure can be caused not only by the creation of an infinite narrowing tree but also by never reaching the closedness condition. In the following section we are faced up to the problem of ensuring global termination.

4.2 Global Termination

Below we show how the abstract operator which is a parameter of the generic algorithm in Definition 3.12 can be defined using a simple kind of structure consisting in sequences of terms that we manipulate in such a way that termination of the specialized algorithm is guaranteed. For a more sophisticated and more expensive kind of tree-like structure which could improve the amount of specialization in some cases, see [48].

[3]We sometimes use the notation xyz as a shorthand for the list x:y:z.

Definition 4.8 (PE* configuration) *Let* $\mathbf{State}^* = \tau(\Sigma \cup \mathbf{V})^*$ *be the standard free monoid over the set of terms, with the empty sequence of terms denoted by* **nil** *and the concatenation operation denoted by ",". We define a* PE* configuration *as a sequence of terms* $(\mathbf{t_1}, \ldots, \mathbf{t_n}) \in \mathbf{State}^*$*. The* empty PE* configuration *is* **nil**.

Upon each iteration, the current configuration $\mathbf{q} \equiv (\mathbf{t_1}, \ldots, \mathbf{t_n})$ is transformed in order to 'cover' the terms which result from the partial evaluation of $\mathbf{q}$ in $\mathcal{R}$, that is, $\mathbf{terms}(\mathbf{U}_\varphi(\{\mathbf{t_1}, \ldots, \mathbf{t_n}\}, \mathcal{R}))$. This transformation is done using the following abstraction operation $\mathbf{abstract}^*(\mathbf{q}, \mathbf{T})$.

Definition 4.9 *Let* $\mathbf{q}$ *be a* $\mathbf{PE}^*$ *configuration and* $\mathbf{T}$ *be an expression. We define* $\mathbf{abstract}^*(\mathbf{q}, \mathbf{T})$ *inductively as follows:*

$$\mathbf{abstract}^*(\mathbf{q}, \mathbf{T}) = \begin{cases} \mathbf{q} & \text{if } \mathbf{T} \equiv \emptyset \text{ or } \mathbf{T} \equiv \mathbf{x} \in \mathbf{V} \\ \mathbf{abstract}^*(\ldots \mathbf{abstract}^*(\mathbf{q}, \mathbf{t_1}), \ldots, \mathbf{t_n}) & \text{if } \mathbf{T} \equiv \{\mathbf{t_1}, \ldots, \mathbf{t_n}\},\ \mathbf{n} \geq 1 \\ \mathbf{abstract}^*(\mathbf{q}, \{\mathbf{t_1}, \ldots, \mathbf{t_n}\}) & \text{if } \mathbf{T} \equiv \mathbf{c}(\mathbf{t_1}, \ldots, \mathbf{t_n}),\ \mathbf{c} \in \mathcal{C},\ \mathbf{n} \geq 0 \\ \mathbf{abs\_call}(\mathbf{q}, \mathbf{T}) & \text{if } \mathbf{T} \equiv \mathbf{f}(\mathbf{t_1}, \ldots, \mathbf{t_n}),\ \mathbf{f} \in \mathcal{F},\ \mathbf{n} \geq 0 \end{cases}$$

where the function $\mathbf{abs\_call}(\mathbf{q}, \mathbf{T})$ *is defined as follows:*

$$\mathbf{abs\_call}(\mathbf{nil}, \mathbf{T}) = \mathbf{T}$$

$$\mathbf{abs\_call}((\mathbf{q_1}, \ldots, \mathbf{q_n}), \mathbf{T}) = \begin{cases} (\mathbf{q_1}, \ldots, \mathbf{q_n}, \mathbf{T}) & \text{if } \nexists \mathbf{i} \in \{1, \ldots, \mathbf{n}\}.\ (\mathbf{comparable}(\mathbf{q_i}, \mathbf{T}) \\ & \quad \text{and } \mathbf{q_i} \trianglelefteq \mathbf{T}) \\ \mathbf{abstract}^*((\mathbf{q_1}, \ldots, \mathbf{q_n}), \mathbf{T}') & \text{if } \mathbf{i} = \max\limits_{\mathbf{j}=1,\ldots,\mathbf{n}} (\mathbf{comparable}(\mathbf{q_j}, \mathbf{T})), \\ & \quad \mathbf{q_i} \trianglelefteq \mathbf{T},\ \exists\theta.\ \mathbf{q_i}\theta = \mathbf{T}, \text{ and} \\ & \quad \mathbf{T}' = \mathbf{terms}(\widehat{\theta}) \\ \mathbf{abstract}^*(\mathbf{q}', \mathbf{T}') & \text{if } \mathbf{i} = \max\limits_{\mathbf{j}=1,\ldots,\mathbf{n}} (\mathbf{comparable}(\mathbf{q_j}, \mathbf{T})), \\ & \quad \mathbf{T} \text{ is not an instance of } \mathbf{q_i}, \\ & \quad \mathbf{msg}(\{\mathbf{q_i}, \mathbf{T}\}) = \langle \mathbf{w}, \{\theta_1, \theta_2\} \rangle, \\ & \quad \mathbf{q}' \equiv (\mathbf{q_1}, \ldots, \mathbf{q_{i-1}}, \mathbf{q_{i+1}}, \ldots, \mathbf{q_n})), \text{ and} \\ & \quad \mathbf{T}' = \{\mathbf{w}\} \cup \mathbf{terms}(\widehat{\theta_1} \cup \widehat{\theta_2}) \end{cases}$$

The following example illustrates how our method achieves both, termination and specialization. The positive supercompiler of [26, 58, 60] does not terminate on this example, due to the infinite generation of "fresh" calls which, because of the growing accumulating parameter, are not an instance of any call that was obtained before. The partial deduction procedure of [3, 4] results in the same nontermination pattern for a logic programming version of this program.

Example 7 *Consider the following program, which checks whether a sequence is a palindrome by using a reversing function with accumulating parameter:*

$$\begin{array}{rcll} \mathbf{palindrome}(\mathbf{x}) & \rightarrow & \mathbf{true} & \Leftarrow \mathbf{reverse}(\mathbf{x}) = \mathbf{x} \\ \mathbf{reverse}(\mathbf{x}) & \rightarrow & \mathbf{rev}(\mathbf{x}, \mathbf{nil}) & \\ \mathbf{rev}(\mathbf{nil}, \mathbf{y_s}) & \rightarrow & \mathbf{y_s} & \\ \mathbf{rev}(\mathbf{x} : \mathbf{x_s}, \mathbf{y_s}) & \rightarrow & \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : \mathbf{y_s}) & \end{array}$$

and consider the goal $\mathbf{palindrome}(1 : 2 : \mathbf{x}) = \mathbf{y}$*. We follow the algorithm of Definition 3.12, with the nonembedding unfolding rule* $\mathbf{U}^{\trianglelefteq}_{\rightsquigarrow}$ *of Definition 4.7 to stop the (normalizing conditional) narrowing derivations, and the* $\mathbf{abstract}^*$ *operator of Definition 4.9 to ensure total correctness.*

The initial $\mathbf{PE}^*$ *configuration is:*

$\mathbf{q_1} = \mathbf{abstract}^*(\mathbf{nil}, \mathbf{palindrome}(1 : 2 : \mathbf{x_s})) = \mathbf{palindrome}(1 : 2 : \mathbf{x_s})$.

The partial evaluation of $\mathbf{palindrome}(1 : 2 : \mathbf{x_s})$ *in* $\mathcal{R}$ *is:*

$\mathcal{R}_1 = \{\ \ \mathbf{palindrome}(1 : 2 : \mathbf{x} : \mathbf{x_s}) \ \rightarrow\ \mathbf{true} \ \Leftarrow\ \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : 2 : 1 : \mathbf{nil}) = 1 : 2 : \mathbf{x} : \mathbf{x_s}\ \ \}$

with $\mathbf{terms}(\mathcal{R}_1) = \{\mathbf{true}, \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : 2 : 1 : \mathbf{nil}), 1 : 2 : \mathbf{x} : \mathbf{x_s}\}$*. Then we obtain:*

$$\begin{array}{rcl} \mathbf{q_2} & = & \mathbf{abstract}^*(\mathbf{palindrome}(1 : 2 : \mathbf{x_s}), \{\mathbf{true}, \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : 2 : 1 : \mathbf{nil}), 1 : 2 : \mathbf{x} : \mathbf{x_s}\}) \\ & = & (\mathbf{palindrome}(1 : 2 : \mathbf{x_s}), \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : 2 : 1 : \mathbf{nil})). \end{array}$$

The partial evaluation of $\mathbf{rev}(\mathbf{x_s}, \mathbf{x} : 2 : 1 : \mathbf{nil})$ *in* $\mathcal{R}$ *is:*

$$\begin{array}{lrcl} \mathcal{R}_2 = \{ & \mathbf{rev}(\mathbf{nil}, \mathbf{x} : 2 : 1 : \mathbf{nil}) & \rightarrow & \mathbf{x} : 2 : 1 : \mathbf{nil} \\ & \mathbf{rev}(\mathbf{x}' : \mathbf{x}'_s, \mathbf{x} : 2 : 1 : \mathbf{nil}) & \rightarrow & \mathbf{rev}(\mathbf{x}'_s, \mathbf{x}' : \mathbf{x} : 2 : 1 : \mathbf{nil}) \ \} \end{array}$$

with $\mathbf{terms}(\mathcal{R}_2) = \{\mathbf{x} : 2 : 1 : \mathbf{nil}, \mathbf{rev}(\mathbf{x}'_s, \mathbf{x}' : \mathbf{x} : 2 : 1 : \mathbf{nil})\}$*. Then we obtain:*

$$\begin{array}{lcl} \mathbf{q}_3 & = & \mathbf{abstract}^*((\mathbf{palindrome}(1 : 2 : \mathbf{x_s}), \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : 2 : 1 : \mathbf{nil})), \{\mathbf{x} : 2 : 1 : \mathbf{nil}, \mathbf{rev}(\mathbf{x}'_s, \mathbf{x}' : \mathbf{x} : 2 : 1 : \mathbf{nil})\}) \\ & = & (\mathbf{palindrome}(1 : 2 : \mathbf{x_s}), \mathbf{rev}(\mathbf{x_s}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s})). \end{array}$$

The partial evaluation of $\mathbf{rev}(\mathbf{x_s}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s})$ *in* $\mathcal{R}$ *is:*

$$\begin{array}{lrcl} \mathcal{R}_3 = \{ & \mathbf{rev}(\mathbf{nil}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}) & \rightarrow & \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s} \\ & \mathbf{rev}(\mathbf{x} : \mathbf{x_s}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}) & \rightarrow & \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}) \ \} \end{array}$$

with $\mathbf{terms}(\mathcal{R}_3) = \{\mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}, \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s})\}$*. Then we obtain:*

$$\begin{array}{lcl} \mathbf{q}_4 & = & \mathbf{abstract}^*((\mathbf{palindrome}(1 : 2 : \mathbf{x_s}), \mathbf{rev}(\mathbf{x_s}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s})), \{\mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}, \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s})\}) \\ & = & (\mathbf{palindrome}(1 : 2 : \mathbf{x_s}), \mathbf{rev}(\mathbf{x_s}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s})) \ \equiv \ \mathbf{q}_3. \end{array}$$

Thus, the specialized program $\mathcal{R}'$ *resulting from the partial evaluation of* $\mathcal{R}$ *w.r.t. the set of terms* $\mathbf{S}' = \{\mathbf{palindrome}(1 : 2 : \mathbf{x_s}), \mathbf{rev}(\mathbf{x_s}, \mathbf{y_s})\}$ *is:*

$$\begin{array}{lrcl} \mathcal{R}' = \{ & \mathbf{palindrome}(1 : 2 : \mathbf{x} : \mathbf{x_s}) & \rightarrow & \mathbf{true} \ \Leftarrow \ \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : 2 : 1 : \mathbf{nil}) = 1 : 2 : \mathbf{x} : \mathbf{x_s} \\ & \mathbf{rev}(\mathbf{nil}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}) & \rightarrow & \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s} \\ & \mathbf{rev}(\mathbf{x} : \mathbf{x_s}, \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}) & \rightarrow & \mathbf{rev}(\mathbf{x_s}, \mathbf{x} : \mathbf{x}_1 : \mathbf{x}_2 : \mathbf{x}_3 : \mathbf{y_s}) \ \} \end{array}$$

where we have saved some infeasible branches which end with **fail** *at specialization time. Note that all computations on the partially static structure have been performed. In the new partially evaluated program, the known elements of the list in the argument of* palindrome *are "passed on" to the list in the second argument of* rev*. The logical inferences needed to perform this, whose number* $\mathbf{n}$ *is linear in the number of the known elements of the list, is done at PE time, which is thus more efficient than performing the* $\mathbf{n}$ *logical inferences at run-time. Note that the resulting set of terms* $\mathbf{S}'$ *is independent.*

The following theorems establish the correctness of the resulting algorithm.

Lemma 4.10 (partial correctness) *If* $\mathbf{abstract}^*(\mathbf{q}, \mathbf{S}) = \mathbf{q}'$*, then* $\mathbf{terms}(\mathbf{q}) \cup \mathbf{S}$ *is closed w.r.t.* $\mathbf{terms}(\mathbf{q}')$*.*

Theorem 4.11 (termination) *The algorithm in Definition 3.12 terminates for the domain* $\mathbf{State}^*$ *of PE* configurations and the abstraction operator* $\mathbf{abstract}^*$*.*

The last example in this section illustrates the fact that our method can also eliminate intermediate data structures and turn multiple–pass programs into one–pass programs, as the deforestation method of [65] and the positive supercompiler of [58] do.

Example 8 *Consider again the program append/2 of Example 6 with initial query* $\mathbf{append}(\mathbf{append}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}) = \mathbf{y}$*. This goal appends three lists by appending the two first, yielding an intermediate list, and then appending the last one to that. We evaluate the goal by using normalizing conditional narrowing. Starting with the sequence* $\mathbf{q} = \mathbf{append}(\mathbf{append}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s})$*, and by using the procedure described in Definition 3.12, we compute the trees depicted in Figure 2 for the sequence of terms* $\mathbf{q}' = \mathbf{append}(\mathbf{append}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}), \mathbf{append}(\mathbf{x_s}, \mathbf{y_s})$*. Note that "append" has been abbreviated to "a" in the picture. Then we get the following residual program* $\mathcal{R}'$*:*

$$\begin{array}{rcl} \mathbf{append}(\mathbf{append}(\mathbf{nil}, \mathbf{y_s}), \mathbf{z_s}) & \rightarrow & \mathbf{append}(\mathbf{y_s}, \mathbf{z_s}) \\ \mathbf{append}(\mathbf{append}(\mathbf{x} : \mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}) & \rightarrow & \mathbf{x} : \mathbf{append}(\mathbf{append}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}) \\ \mathbf{append}(\mathbf{nil}, \mathbf{z_s}) & \rightarrow & \mathbf{z_s} \\ \mathbf{append}(\mathbf{y} : \mathbf{y_s}, \mathbf{z_s}) & \rightarrow & \mathbf{y} : \mathbf{append}(\mathbf{y_s}, \mathbf{z_s}) \end{array}$$

which is able to append the three lists by passing over its input only once. This effect has been obtained in our method by virtue of normalization. Without the normalization step, the ordering would have been satisfied too early in the rightmost branch of the top tree of Figure 2. Note that we did not adopt any specific strategy (like the call-by-name or the call-by-value ones) for executing the

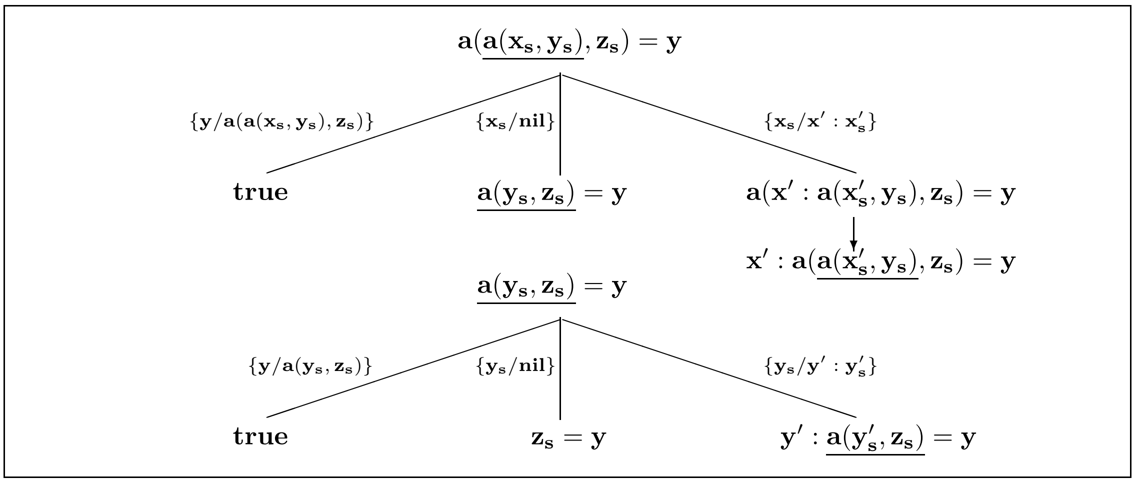
$\mathbf{a}(\underline{\mathbf{a}(\mathbf{x_s},\mathbf{y_s})},\mathbf{z_s}) = \mathbf{y}$
$\{\mathbf{y}/\mathbf{a}(\mathbf{a}(\mathbf{x_s},\mathbf{y_s}),\mathbf{z_s})\}$
$\{\mathbf{x_s}/\mathbf{nil}\}$
$\{\mathbf{x_s}/\mathbf{x}' : \mathbf{x}'_\mathbf{s}\}$
true
$\underline{\mathbf{a}(\mathbf{y_s},\mathbf{z_s})} = \mathbf{y}$
$\mathbf{a}(\mathbf{x}' : \mathbf{a}(\mathbf{x}'_\mathbf{s},\mathbf{y_s}),\mathbf{z_s}) = \mathbf{y}$
$\mathbf{x}' : \mathbf{a}(\underline{\mathbf{a}(\mathbf{x}'_\mathbf{s},\mathbf{y_s})},\mathbf{z_s}) = \mathbf{y}$
$\underline{\mathbf{a}(\mathbf{y_s},\mathbf{z_s})} = \mathbf{y}$
$\{\mathbf{y}/\mathbf{a}(\mathbf{y_s},\mathbf{z_s})\}$
$\{\mathbf{y_s}/\mathbf{nil}\}$
$\{\mathbf{y_s}/\mathbf{y}' : \mathbf{y}'_\mathbf{s}\}$
true
$\mathbf{z_s} = \mathbf{y}$
$\mathbf{y}' : \underline{\mathbf{a}(\mathbf{y}'_\mathbf{s},\mathbf{z_s})} = \mathbf{y}$

Figure 2: Narrowing trees for the goals $\mathbf{a}(\mathbf{a}(\mathbf{x_s},\mathbf{y_s}),\mathbf{z_s}) = \mathbf{y}$ and $\mathbf{a}(\mathbf{x_s},\mathbf{y_s}) = \mathbf{y}$.

goal. This seems to answer negatively an interesting question left open in [58, 60]. The resulting set of terms $\{\mathbf{append}(\mathbf{append}(\mathbf{x_s},\mathbf{y_s}),\mathbf{z_s}), \mathbf{append}(\mathbf{x_s},\mathbf{y_s})\}$ *in* $\mathbf{q}'$ *is not independent. This example illustrates the need for an extra* renaming *phase able to produce an independent set of terms such as* $\{\mathbf{append}(\mathbf{append}(\mathbf{x_s},\mathbf{y_s}),\mathbf{z_s}), \mathbf{append}'(\mathbf{x_s},\mathbf{y_s})\}$ *and associated specialized program:*

$$\begin{array}{rcl}
\mathbf{append}(\mathbf{append}(\mathbf{nil},\mathbf{y_s}),\mathbf{z_s}) & \to & \mathbf{append}'(\mathbf{y_s},\mathbf{z_s}) \\
\mathbf{append}(\mathbf{append}(\mathbf{x} : \mathbf{x_s},\mathbf{y_s}),\mathbf{z_s}) & \to & \mathbf{x} : \mathbf{append}(\mathbf{append}(\mathbf{x_s},\mathbf{y_s}),\mathbf{z_s}) \\
\mathbf{append}'(\mathbf{nil},\mathbf{z_s}) & \to & \mathbf{z_s} \\
\mathbf{append}'(\mathbf{y} : \mathbf{y_s},\mathbf{z_s}) & \to & \mathbf{y} : \mathbf{append}'(\mathbf{y_s},\mathbf{z_s})
\end{array}$$

which does have the same computed answers as the original program $\mathbf{append}/2$ *for the query* $\mathbf{append}(\mathbf{append}(\mathbf{x_s},\mathbf{y_s}),\mathbf{z_s})$ *(modulo the renaming transformation).*

The use of efficient forms of narrowing can significantly improve the accuracy of the specialization method as well as to increase the efficiency of the resulting program, because some run-time optimizations (e.g. normalization steps) can be performed at compile time. In the following section, we formalize a highly efficient instance of the generic PE procedure of Definition 3.12 which make use of the simple mechanisms introduced so far to achieve (both local and global) termination. For simplicity, we consider the *leftmost innermost* narrowing strategy. This corresponds to the evaluation strategy of Prolog. However, all the following results should be generalized to a suitable class of narrowing strategies, and we plan to develop this as future work.

5 A call-by-value Partial Evaluator

In this section, we are interested in an innermost narrowing strategy where a narrowing step is performed at the (leftmost) innermost occurrence. This corresponds to eager evaluation in functional languages and allows us to formalize a call-by-value partial evaluator for functional logic programs. We briefly recall the definition of *innermost conditional narrowing* [20].

5.1 Innermost Conditional Narrowing

An *innermost* term $\mathbf{t}$ is an operation applied to constructor terms, i.e. $\mathbf{t} = \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t_k})$, where $\mathbf{f} \in \mathcal{F}$ and, for all $\mathbf{i} = 1, \dots, \mathbf{k}$, $\mathbf{t_i} \in \mathcal{T}(\mathcal{C} \cup \mathbf{V})$. A CTRS is *constructor-based* (CB) if the left-hand side (lhs) of each rule is an innermost term. It implies that there can be neither functional nestings on the lhs of the head of the clauses or axioms between constructors. This is a reasonable class from the functional programming point of view. Many equational theories which occur in practice follow this discipline, e.g. in the specification of abstract data types.

A function symbol is *completely-defined* (everywhere defined [8]) if it does not occur in any ground term in normal form, that is to say functions are reducible on all ground terms of an appropriate sort. $\mathcal{R}$ is said to be completely-defined (CD) if each defined function symbol is completely-defined. In a completely-defined CTRS the set of ground normal terms is the set of ground constructor terms $\tau(\mathcal{C})$ over $\mathcal{C}$. In one-sorted theories, completely-defined theories occur only rarely. But they are usual when using types and each function is defined for all constructors of their argument types. Examples of functional logic languages following the CB-CD discipline are: SLOG [20] and LPG [5, 6].

Let $\varphi_{\blacktriangleleft}(\mathbf{g})$ be the narrowing strategy which assigns the occurrence $\mathbf{u}$ of the leftmost innermost subterm of $\mathbf{g}$ to the goal $\mathbf{g}$. We formulate innermost conditional narrowing by means of a labelled transition system [52] $(\mathbf{Goal}, \mathbf{Sub}, \rightsquigarrow_{\varphi_{\blacktriangleleft}})$ whose transition relation $\rightsquigarrow_{\varphi_{\blacktriangleleft}} \subseteq \mathbf{Goal} \times \mathbf{Sub} \times \mathbf{Goal}$ formalizes the computation steps. In the following, we will abbreviate $\rightsquigarrow_{\varphi_{\blacktriangleleft}}$ to $\rightsquigarrow_{\blacktriangleleft}$.

Definition 5.1 *Let $\mathcal{R}$ be a CTRS. We define the innermost conditional narrowing relation $\rightsquigarrow_{\blacktriangleleft}$ as the smallest relation satisfying*

$$\frac{\mathbf{u} = \varphi_{\blacktriangleleft}(\mathbf{g}) \ \wedge \ (\lambda \to \rho \Leftarrow \mathbf{C}) \lll \mathcal{R} \ \wedge \ \sigma = \mathbf{mgu}(\{\mathbf{g}_{|\mathbf{u}} = \lambda\})}{\mathbf{g} \overset{\sigma}{\rightsquigarrow}_{\blacktriangleleft} (\mathbf{C}, \{\mathbf{g}[\rho]_{\mathbf{u}}\})\sigma}$$

The following proposition establishes the completeness of innermost conditional narrowing for constructor-based completely-defined canonical CTRS's.

Proposition 5.2 [20] *Let $\mathcal{R}$ be a CB-CD canonical program, $\mathbf{g}$ be a goal and σ be a ground constructor solution of $\mathbf{g}$ such that $\mathbf{Var}(\mathbf{g}) \subseteq \mathbf{Dom}(\sigma)$. Then, there is a computed answer substitution θ of $\mathcal{R} \cup \{\mathbf{g}\}$ using $\rightsquigarrow_{\blacktriangleleft}$, and a substitution γ such that $\theta\gamma = \sigma$.*

The condition $\mathbf{Var}(\mathbf{g}) \subseteq \mathbf{Dom}(\sigma)$ in the premise of Proposition 5.2 guarantees that $\mathbf{g}\sigma$ is ground. The following example reveals that this condition cannot be dropped, contrary to what is generally believed [20, 31, 32].

Example 9 *Consider the following CB-CD canonical program $\mathcal{R} \equiv \{\mathbf{f}(0, \mathbf{y}) \to \mathbf{y},\ \mathbf{g}(0) \to 0\}$. The ground constructor substitution $\sigma = \{\mathbf{x}/0\}$ is a solution of the equation $\mathbf{f}(\mathbf{x}, \mathbf{g}(\mathbf{y})) = \mathbf{g}(\mathbf{y})$. However, innermost conditional narrowing is not able to compute a more general answer. Note that the substitution σ does not satisfy the condition $\mathbf{Var}(\mathbf{g}) \subseteq \mathbf{Dom}(\sigma)$.*

It is easy to extend this strategy to incompletely-defined functions, by just adding a so-called innermost reflection rule which skips an innermost function call that cannot be reduced [32]. For the sake of simplicity we assume in the following that all functions are completely defined, i.e. innermost narrowing is sufficient to compute all answers.

In the following section we consider the normalizing innermost conditional narrowing $\rightsquigarrow_{\blacktriangleleft}$, where the computation of the normal form between narrowing steps is performed by innermost conditional rewriting (i.e., a rewrite rule is applied to a term only if each of its subterms is in normal form). This simplifies our proofs and can also be efficiently implemented [27, 28]. In order to ensure that the normal form of a goal uniquely exists and can be computed by any strategy of rewriting, we require programs to be decreasing as well. Normalizing innermost narrowing is the foundation of several functional logic programming languages like SLOG [20], LPG [5, 6] and (a subset of) ALF [27] and RAP [24]. Thus, the method that we propose can be used for the optimization of programs written in these languages. It has been shown that, since functions allow more deterministic evaluation than predicates, normalizing innermost narrowing has the effect that functional logic programs are more efficiently executable than equivalent logic programs [20, 28]. Following a similar argument, we are able to show that functional logic programs are also more efficiently specializable than pure logic programs.

5.2 The call-by-value Partial Evaluator

In this section we consider the specialization of the generic PE procedure introduced in Definition 3.12 which results from using:

1. the *normalizing innermost conditional narrowing* $\rightsquigarrow_{\blacktriangleleft}$ of Section 5.1 to construct search trees,
2. the *nonembedding unfolding rule* $\mathbf{U}^{\trianglelefteq}_{\blacktriangleleft}$ (i.e. $\mathbf{U}^{\trianglelefteq}_{\rightsquigarrow_{\blacktriangleleft}}$) of Section 4.1 to control the expansion of the trees,
3. the domain **State*** for encoding PE* configurations, and
4. the abstraction operator **abstract*** of Section 4.2 which guarantees correctness and global termination.

The specialized algorithm inherits all results which were proved for the general method. Also, we are able to strengthen some of these results, when we restrict the method to the evaluation of innermost (not necessarily ground) terms, as we formalize in the following theorems.

The restriction to innermost terms is necessary for the residual program to adhere the CB-CD discipline, thus avoiding to produce non-data answers (which, for the particular case of innermost narrowing, amounts to computing only the solutions produced by $\mathcal{R}$).

Proposition 5.3 *The PE of a decreasing CB-CD program w.r.t. an innermost term is CB-CD decreasing.*

Theorem 5.4 *Let $\mathcal{R}$ be a canonical program,* $\mathbf{S}$ *a finite set of innermost terms, and* $\mathbf{g}$ *a goal such that for all* $\mathbf{t} \in \mathbf{terms}(\mathbf{g})$, $\mathbf{t}$ *is innermost. Let $\mathcal{R}'$ be a partial evaluation of $\mathcal{R}$ w.r.t.* $\mathbf{S}$ *such that* $\mathcal{R}' \cup \{\mathbf{g}\}$ *is* $\mathbf{S}$*-closed. Then,*

1. (Strong Soundness) $\mathcal{O}_{\mathcal{R}'}(\mathbf{g}) \subseteq \mathcal{O}_{\mathcal{R}}(\mathbf{g})$*, if* $\mathbf{S}$ *is independent.*

2. (Completeness) $\mathcal{O}_{\mathcal{R}'}(\mathbf{g}) \supseteq \mathcal{O}_{\mathcal{R}}(\mathbf{g})$.

A way to test a transformation method's strength is to check whether it can derive efficient programs from equivalent naïve and inefficient programs [36]. We now illustrate the power of the call-by-value PE procedure on the pattern matching program **match** of [41].

5.3 Pattern matching in strings

A standard example in the literature on partial evaluation is the derivation of an efficient string matcher by partial evaluation of a (more or less) naïve pattern matcher w.r.t. a given pattern [25, 36, 60]. The source program $\mathcal{R}$ listed below checks whether a string pattern **p** occurs within another string **s** by iteratively comparing **p** with a prefix of **s**. In case of a mismatch, the first element of the target string **s** is cut off and the process is restarted with the tail of **s**. The strategy is not optimal because the same elements in the string may be tested several times.

The power of a transformation can be made evident by checking whether it automatically performs the optimization central to the Knuth-Morris-Pratt (KMP) string matching algorithm which constructs a deterministic finite automaton. The 'KMP test' is often used to compare the strength of specializers. This example is particularly interesting because it is a kind of transformation that neither (conventional) partial evaluation or deforestation can perform automatically [58]. Partial deduction of logic programs and positive supercompilation of functional programs can pass the test (see [58] for references). In this section we show that our method also performs satisfactorily on the problem. We assume that matching is on bit-strings, i.e. strings containing only zeroes and ones.

Example 10 *Let $\mathcal{R}$ be the naïve pattern matching program:*

$$\begin{array}{rcll}
\mathbf{match}(\mathbf{p},\mathbf{s}) & \to & \mathbf{loop}(\mathbf{p},\mathbf{s},\mathbf{p},\mathbf{s}) & \\
\\
\mathbf{loop}(\mathbf{nil},\mathbf{ss},\mathbf{op},\mathbf{os}) & \to & \mathbf{true} & \\
\mathbf{loop}(\mathbf{p}:\mathbf{pp},\mathbf{nil},\mathbf{op},\mathbf{os}) & \to & \mathbf{false} & \\
\mathbf{loop}(\mathbf{p}:\mathbf{pp},\mathbf{p}:\mathbf{ss},\mathbf{op},\mathbf{os}) & \to & \mathbf{loop}(\mathbf{pp},\mathbf{ss},\mathbf{op},\mathbf{os}) & \%\ \texttt{continue} \\
\mathbf{loop}(\mathbf{p}:\mathbf{pp},\mathbf{s}:\mathbf{ss},\mathbf{op},\mathbf{os}) & \to & \mathbf{next}(\mathbf{op},\mathbf{os}) \Leftarrow (\mathbf{p}=\mathbf{s})=\mathbf{false} & \%\ \texttt{shift string} \\
\\
\mathbf{next}(\mathbf{op},\mathbf{nil}) & \to & \mathbf{false} & \\
\mathbf{next}(\mathbf{op},\mathbf{s}:\mathbf{ss}) & \to & \mathbf{loop}(\mathbf{op},\mathbf{ss},\mathbf{op},\mathbf{ss}) & \%\ \texttt{restart loop} \\
(0=1) & \to & \mathbf{false} & \\
(1=0) & \to & \mathbf{false} &
\end{array}$$

Suppose that the fixed pattern 001 *is given and we want to solve the pattern matching problem for the subject string* $\mathbf{s}$. *Applying the call-by-value evaluator to the term* $\mathbf{match}(001,\mathbf{s})$, *and subsequently evaluating new terms according to our method, gives the specialized program*[4]*:*

$$\begin{array}{rcl}
\mathbf{match}(001,001:\mathbf{ss}) & \to & \mathbf{true} \\
\mathbf{match}(001,000:\mathbf{ss}) & \to & \mathbf{loop}(1,\mathbf{ss},001,00:\mathbf{ss}) \\
\mathbf{match}(001,01:\mathbf{ss}) & \to & \mathbf{loop}(001,\mathbf{ss},001,\mathbf{ss}) \\
\mathbf{match}(001,1:\mathbf{ss}) & \to & \mathbf{loop}(001,\mathbf{ss},001,\mathbf{ss}) \\
\\
\mathbf{loop}(1,1:\mathbf{ss},001,001:\mathbf{ss}) & \to & \mathbf{true} \\
\mathbf{loop}(1,0:\mathbf{ss},001,000:\mathbf{ss}) & \to & \mathbf{loop}(1,\mathbf{ss},001,00:\mathbf{ss}) \\
\\
\mathbf{loop}(001,001:\mathbf{ss},001,001:\mathbf{ss}) & \to & \mathbf{true} \\
\mathbf{loop}(001,000:\mathbf{ss},001,000:\mathbf{ss}) & \to & \mathbf{loop}(1,\mathbf{ss},001,00:\mathbf{ss}) \\
\mathbf{loop}(001,01:\mathbf{ss},001,01:\mathbf{ss}) & \to & \mathbf{loop}(001,\mathbf{ss},001,\mathbf{ss}) \\
\mathbf{loop}(001,1:\mathbf{ss},001,1:\mathbf{ss}) & \to & \mathbf{loop}(001,\mathbf{ss},001,\mathbf{ss})
\end{array}$$

The amount of specialization obtained in this program is essentially analogous to the one of the rules produced by the algorithm in [59]. Redundant structure in the program can be easily be removed by some proper post-processing renaming such as [22]. The complexity of the specialized algorithm is $\mathbf{O}(\mathbf{n})$, *where* $\mathbf{n}$ *is the length of the string. The naïve pattern matcher has complexity* $\mathbf{O}(\mathbf{m}\times\mathbf{n})$, *where* $\mathbf{m}$ *is the length of the pattern. This is in essence a KMP pattern matcher.*

6 Conclusions and Further Research

Partial evaluation is a semantics-preserving program transformation based on unfolding and specializing procedures. Techniques in conventional partial evaluation of functional programs usually rely on the reduction of expressions and constant propagation, while transformation techniques for logic languages exploit unification-based parameter propagation [26]. The driving approach essentially achieves the same transformational effect for functional programs. Few attempts have been made to study the relationship of techniques used in logic and functional languages [26]. We think that the unified treatment of the problem lays the ground for comparisons and possibly generates new insights for further developments in both fields. Since we can use all known results about narrowing, our proofs are much simpler and our results are stronger, particularly the notion of correctness, which amounts to preserving the computed answer semantics of the goal, and not just the ground success set semantics of [26]. We have shown how can be defined a core PE procedure whose behaviour does not depend on the eager or lazy nature of the narrower. Then we have considered of normalizing innermost narrowing since it has been shown that this strategy is a reasonable improvement over pure logic left-to-right SLD resolution strategy [20, 29]. It is

[4]For simplicity, we have omitted the rules that reduce functions to false.

worthwhile to investigate the instantiation of our framework for other narrowing strategies, such as LSE narrowing [7] or lazy narrowing with normalization [30].

Turchin's supercompiler does not just propagate positive information (by applying unifiers) but also propagates negative information which can restrict the values that the variables can take by using enviroments of positive and negative bindings (bindings which do not hold) [63, 25]. We think that we can strengthen this effect in the setting of (equational) constraint logic programming [1, 35] by using some kind of narrowing procedure with disunification, such as the ones defined in [2, 19, 53], in order to propagate (negative) bindings which can be gathered during transformation as (disequality) constraints. Automatic generation of such generalized specializations is the subject of further work.

Our results are also relevant for the definition of a compositional semantics for functional logic programming which is useful for modular program analysis and transformation [9]. To achieve such a goal, we intend to study how to define an unfolding semantics using partial evaluation with an unfolding strategy which stops derivations that reach *open* function symbols (i.e. symbols to be defined in other program modules).

References

[1] M. Alpuente, M. Falaschi, and G. Levi. Incremental Constraint Satisfaction for Equational Logic Programming. *Theoretical Computer Science*, 142(1):27–57, 1995.

[2] P. Arenas, A. Gil, and F. López. Combining Lazy Narrowing with Disequality Constraints. In *Proc. of PLILP'94*, volume 844 of *Lecture Notes in Computer Science*, pages 385–399. Springer-Verlag, Berlin, 1994.

[3] K. Benkerimi and P.M. Hill. Supporting Transformations for the Partial Evaluation of Logic Programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.

[4] K. Benkerimi and J.W. Lloyd. A Partial Evaluation Procedure for Logic Programs. In S. Debray and M. Hermenegildo, editors, *Proc. of the 1990 North American Conf. on Logic Programming*, pages 343–358. The MIT Press, Cambridge, MA, 1990.

[5] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. of First European Symp. on Programming, ESOP'86*, pages 119–132. Springer LNCS 213, 1986.

[6] D. Bert and R. Echahed. Integrating Disequations in the Algebraic and Logic Programming Language LPG. In *Proc. of the ICLP'94 Post-Conference Workshop on Integration of Declarative Paradigms.* MPI-I-94-224, Saarbrucken, 1994.

[7] A. Bockmayr and A. Werner. LSE Narrowing for Decreasing Conditional Term Rewrite Systems. In *Conditional Term Rewriting Systems, CTRS'94, Jerusalem.* Springer-Verlag, Berlin, 1995.

[8] P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[9] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19-20:149–197, 1994.

[10] M. Bruynooghe, D. de Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.

[11] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[12] P.H. Cheong and L. Fribourg. A survey of the implementations of narrowing. In J. Darlington and R. Dietrich, editors, *Declarative Programming. Workshops in Computing*, pages 177–187. Springer-Verlag and BCS, 1992.

[13] C. Consel and O. Danvy. Partial Evaluation in Parallel. *LISP and Symbolic Computation*, 5(4):315–330, 1993.

[14] J. Darlington and H. Pull. A Program Development Methodology Based on a Unified Approach to Execution and Transformation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 117–131. North-Holland, Amsterdam, 1988.

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

[16] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[17] N. Dershowitz and U. Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494, 1993.

[18] R. Echahed. On completeness of narrowing strategies. In *Proc. of CAAP'88*, pages 89–101. Springer LNCS 299, 1988.

[19] M. Fernández. Narrowing Based Procedures for Equational Disunification. *Applicable Algebra in Engineering, Communication and Computing*, 3:1–26, 1992.

[20] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, New York, 1985.

[21] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[22] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.

[23] J. Gallagher and M. Bruynooghe. Some Low-Level Source Transformations for Logic Programs. In M. Bruynooghe, editor, *Proc. of 2nd Workshop on Meta-Programming in Logic*, pages 229–246. Department of Computer Science, KU Leuven, Belgium, 1990.

[24] A. Geser and H. Hussmann. Experiences with the RAP system – a specification interpreter combining term rewriting and resolution. In *Proc. of ESOP'86*, pages 339–350. Springer LNCS 213, 1986.

[25] R. Glück and A.V. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proc. of 3rd Int'l Workshop on Static Analysis, WSA'93*, pages 112–123. Springer LNCS 724, 1993.

[26] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *Proc. Int'l Symp. on Programming Language Implementation and Logic Programming, PLILP'94*, pages 165–181. Springer LNCS 844, 1994.

[27] M. Hanus. Compiling Logic Programs with Equality. In *Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[28] M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int'l Workshop on Processing Declarative Knowledge*, pages 344–365. Springer LNAI 567, 1991.

[29] M. Hanus. An Abstract Interpretation Algorithm for Residuating Logic Programs. In *Proc. of Second Int'l Workshop on Static Analysis, WSA'92, Bordeaux, France*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.

[30] M. Hanus. Combining Lazy Narrowing with Simplification. In *Proc. of 6th Int'l Symp. on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

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

[32] S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNAI 353, 1989.

[33] J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[34] H. Hussmann. Unification in conditional-equational theories. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1986.

[35] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of 14th Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[36] N.D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. In N.D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language and Computation*, pages 206–224. Springer LNCS 792, 1994.

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

[38] A. Josephson and N. Dershowitz. An Implementation of Narrowing. *Journal of Logic Programming*, 6:57–77, 1989.

[39] S. Kaplan. Fair conditional term rewriting systems: unification, termination and confluence. In H.-J. Kreowski, editor, *Recent Trends in Data Type Specification*, volume 116 of *Informatik-Fachberichte*, pages 136–155. Springer-Verlag, Berlin, 1986.

[40] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[41] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal of Computation*, 6(2):323–350, 1977.

[42] H.J. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proc. of 9th ACM Symp. on Principles of Programming Languages*, pages 255–267, 1982.

[43] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[44] M. Leuschel and D. De Schreye. An Almost Perfect Abstraction Operator for Partial Deduction. Technical Report CW-199, Department of Computer Science, K.U. Leuven, Belgium, December 1994.

[45] G. Levi and F. Sirovich. Proving Program Properties, Symbolic Evaluation and Logical Procedural Semantics. In *Proc. of MFCS'75*, pages 294–301. Springer LNCS 32, 1975.

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

[47] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. Technical Report CSTR-94-16, Computer Science Department, University of Bristol, December 1994.

[48] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In K. Furukawa and K. Ueda, editors, *Proc. of ICLP'95*, 1995.

[49] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[50] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7:295–317, 1989.

[51] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1988.

[52] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[53] M.J. Ramírez and M. Falaschi. Conditional Narrowing with Constructive Negation. In E. Lamma and P. Mello, editors, *Proc. of Third Int'l Workshop on Extensions of Logic Programing ELP'92*, volume 660 of *Lecture Notes in Computer Science*, pages 59–79. Springer-Verlag, Berlin, 1993.

[54] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, New York, 1985.

[55] P. Réty. Improving basic narrowing techniques. In *Proc. of the Conf. on Rewriting Techniques and Applications*, pages 228–241. Springer LNCS 256, 1987.

[56] A. Romanenko. Inversion and metacomputation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–22. Sigplan Notices, 26(9), ACM, New York, September 1991.

[57] P. Sestoft and H. Søndergaard. A bibliography on partial evaluation. *Sigplan Notices*, 23(2):19–27, Feb 1988.

[58] M.H. Sørensen. Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation. Technical Report 94/7, Master's Thesis, Department of Computer Science, University of Copenhagen, Denmark, 1994.

[59] M.H. Sørensen and R. Glück. Generalization in Positive Supercompilation. In J.W. Lloyd, editor, *Proc. of ILPS'95*, 1995.

[60] M.H. Sørensen, R. Glück, and N.D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems, ESOP'94. 5th European Symp. on Programming*, pages 485–500. Springer LNCS 788, 1994.

[61] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

[62] V.F. Turchin. Semantic definitions in Refal and automatic production of compilers. In N.D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 441–474. Springer LNCS 94, 1980.

[63] V.F. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, 1985*, pages 257–281. Springer LNCS 217, 1986.

[64] V.F. Turchin. The Algorithm of Generalization in the Supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, Amsterdam, 1988.

[65] P.L. Wadler. Deforestation: transforming programs to eliminate trees. In *Proc of the European Symp. on Programming, ESOP'88*, pages 344–358. Springer LNCS 300, 1988.