Narrowing-driven Partial Evaluation of Functional Logic Programs *

M. Alpuente[1] and M. Falaschi[2] and G. Vidal[1]

[1] DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain. e.mail:{alpuente,gvidal}@dsic.upv.es
[2] Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze 206, 33100 Udine, Italy. e.mail:falaschi@dimi.uniud.it

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. We illustrate our method with several examples and discuss the relation with Supercompilation and Partial Evaluation. 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 [27]. 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 its input [9]. The main issues with automatic PE (specialization) concern the choice of the basic transformation techniques, termination of the process, preserving the semantics of the original program, and effectiveness of the transformation, i.e. execution speedup for a large class of programs. Two basic transformation techniques used in PE are the folding and unfolding transformations [5]. Unfolding is essentially the replacement of a call by its body, with appropriate substitutions. Folding is the

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

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. Because of the unification, the mechanism of PE for logic programs is in general more powerful than for functional programs, as it is also able to propagate syntactic information on the partial input, such as term structure, and not only constant values. PE has been extensively studied both in functional [19, 28] and in logic programming [10, 24].

In this paper, we show that, in the context of languages that integrate functional and logic programming [14], specialization can be based on the unification-based 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.

Due to its basic strategy, a PE can loop in two ways: either by unfolding infinitely a function call, or by creating infinitely many specialized definitions [22, 25]. Our PE procedure follows a structure similar to the framework developed for Logic Programming in [25]. Starting with the set of calls (terms) which appear in the initial goal, we partially evaluate them by using a finite unfolding strategy, and recursively specialize the terms which are introduced dynamically during this process. We introduce an appropriate abstract operator which ensures that this set is kept finite throughout the PE process (hence guaranteeing termination) and which also allows us to tune the specialization of the method.

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 [23], 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 [6], 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 work on supercompilation [31] 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* [12]. Driving can be understood as a unification-based function transformation mechanism, which uses some kind of evaluation machinery similar to (lazy) narrowing 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. Supercompilation subsumes PE and other standard transformations of functional programming languages [30]. For example, it is able to support certain forms of theorem proving, program synthesis and program inversion.

The driving process does not always terminate, and it does not preserve the semantics, as it can extend the domain of functions [19, 30]. Techniques to ensure termination of driving are studied in [29, 32]. The idea of [32] 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 [29], termination is guaranteed following a method which is comparable to the Martens-Gallagher general approach for ensuring global termination of PE for logic programs [25].

In [12], 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 construction 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 [12, 30] in order not to encumber the formulation of driving algorithms. Let us emphasize that our PE procedure guarantees the completeness of the transformed program w.r.t. a strong observable such as the set of computed answer substitutions of the original program. Our framework defines the first semantics-based PE scheme for functional logic programs.

Plan of the paper

This paper is organized as follows. In Section 2, basic definitions are given. Section 3 presents a general scheme for the PE of functional logic programs 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. Section 5 concludes the paper and discusses directions of future research. More details and missing proofs can be found in [2].

2 Preliminaries

We briefly recall some known results about rewrite systems and functional logic programming [7, 14, 15]. Throughout this paper, V will denote a countably infinite set of variables and Σ denotes a set of function symbols, each with a fixed arity. $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the sets of terms and ground terms built on Σ and V, respectively. A Σ-equation $s = t$ is a pair of terms $s, t \in \tau(\Sigma \cup V)$. Terms are viewed as labeled trees in the usual way. $\bar{O}(t)$ denotes the set of nonvariable occurrences of a term t. $t_{|u}$ is the subterm at the occurrence u of t. $t[r]_u$ is the term t with the subterm at the occurrence u replaced with r. These notions extend to equations and sequences of equations in a natural way. Identity of syntactic objects is denoted by $\equiv$. We restrict our interest to the set of idempotent substitutions over $\tau(\Sigma \cup V)$, which is denoted by Sub. The identity function on V is called the empty substitution and denoted ϵ. $t\theta$ denotes the application of θ to the syntactic object t. $\theta\gamma$ denotes the composition of θ and γ. $\theta_{\upharpoonright W}$ is the substitution obtained from θ by restricting its domain, $Dom(\theta)$, to W. The equational representation of a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ is the set of equations $\widehat{\theta} = \{x_1 = t_1, \ldots, x_n = t_n\}$. We let $mgu(E)$ denote the (syntactic) *most general unifier* of the equation set E [21]. A generalization of the nonempty set of terms $\{t_1, \ldots, t_n\}$ is a pair $\langle t, \{\theta_1, \ldots, \theta_n\}\rangle$ such that, for all $i = 1, \ldots, n$, $t\theta_i = t_i$. The pair $\langle t, \Theta\rangle$ is the *most specific generalization (msg)* of a set of terms S, written $\langle t, \Theta\rangle = msg(S)$, if 1) $\langle t, \Theta\rangle$ is a generalization of S, and 2) for every other generalization $\langle t', \Theta'\rangle$ of S, t' is more general than t.

An equational Horn theory $\mathcal{E}$ consists of a finite set of equational Horn clauses of the form $(\lambda = \rho) \Leftarrow C$. The condition C is a sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations. Variables in 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 $(\lambda \rightarrow \rho \ \Leftarrow C)$, $\lambda, \rho \in \tau(\Sigma \cup V)$, $\lambda \notin V$ and $Var(\rho) \cup Var(C) \subseteq Var(\lambda)$. If a rewrite rule has no condition we write $\lambda \rightarrow \rho$. We will often omit Σ. 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 (implicitly oriented from left to right) and the conditions are the respective bodies. The equational theory $\mathcal{E}$ is said to be canonical, if the binary one-step rewriting relation $\rightarrow_{\mathcal{R}}$ defined by $\mathcal{R}$ is noetherian and confluent. For CTRS $\mathcal{R}$, $r \ll \mathcal{R}$ denotes that r is a new variant of a rule in $\mathcal{R}$ such that r contains no variable previously met during computation (standardised apart).

Functional logic languages are extensions of functional languages with principles derived from logic programming. The computation mechanism of functional logic languages is based on narrowing, an evaluation mechanism that uses unification for parameter passing [27]. Narrowing solves equations by computing unifiers with respect to a given CTRS (which is called the 'program'). Given a CTRS $\mathcal{R}$, an equational goal g conditionally narrows into a goal clause g' (in symbols $g \overset{[u,r,\theta]}{\leadsto} g'$, $g \overset{[u,\theta]}{\leadsto} g'$ or simply $g \overset{\theta}{\leadsto} g'$), if there exists an occurrence $u \in \bar{O}(g)$, a standardised apart variant $r \equiv (\lambda \rightarrow \rho \ \Leftarrow C) \ll \mathcal{R}$ and a substitution θ such that $\theta = mgu(\{g_{|u} = \lambda\})$ and $g' = (C, \{g[\rho]_u\})\theta$. s is called a (narrowing) *redex* iff there exists a new variant $(\lambda \rightarrow \rho \ \Leftarrow C) \ll \mathcal{R}$ and a substitution σ such that $s\sigma \equiv \lambda\sigma$. A *narrowing derivation* for g in $\mathcal{R}$ is defined by $g \overset{\theta}{\leadsto}{}^* g'$ iff $\exists \theta_1, \ldots, \theta_n.\ g \overset{\theta_1}{\leadsto} \ldots \overset{\theta_n}{\leadsto} g'$ and $\theta = \theta_1 \ldots \theta_n$. If $n = 0$, then $\theta = \epsilon$. In order to treat syntactical unification as a narrowing step, we add the rule $(x = x \rightarrow true)$, $x \in V$, to the CTRS $\mathcal{R}$. Then $s = t \overset{\sigma}{\leadsto} true$ holds iff $\sigma = mgu(\{s = t\})$. The extension of a CTRS $\mathcal{R}$ with the rewrite rule $(x = x \rightarrow true)$ is denoted by $\mathcal{R}_+$. We use $\top$ as a generic notation for sequences of the form $true, \ldots, true$. A successful derivation for g in $\mathcal{R}_+$ is a narrowing derivation $g \overset{\theta}{\leadsto}^* \top$, and $\theta_{\lceil Var(g)}$ is called a computed answer substitution for g in $\mathcal{R}$. The *success set* operational semantics of an equational goal g in the program $\mathcal{R}$ is $\mathcal{O}_{\mathcal{R}}(g) = \{\theta_{\lceil Var(g)} \mid g \overset{\theta}{\leadsto}^* \top\}$. The set of narrowing derivations can be represented by a (possibly infinite) finitely branching tree. Following [24], in this paper we adopt the convention 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 $\tau(\Sigma \cup V)$ (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 $s\sigma =_{\mathcal{E}} t\sigma$, i.e. such that $\mathcal{E} \models s\sigma = t\sigma$. 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 $W \subseteq V$, *$\mathcal{E}$-equality* is extended to substitutions in the standard way, by $\sigma =_{\mathcal{E}} \theta[W]$ iff $x\sigma =_{\mathcal{E}} x\theta \ \forall x \in W$. We say σ is an $\mathcal{E}$-instance of σ' and σ' is more general than σ on W, in symbols $\sigma' \leq_{\mathcal{E}} \sigma[W]$ iff $(\exists\rho)\ \sigma =_{\mathcal{E}} \sigma'\rho[W]$.

A set S of $\mathcal{E}$-unifiers of the equation set E is complete iff every $\mathcal{E}$-unifier σ of E factors into $\sigma =_{\mathcal{E}} \theta\gamma$ for some substitutions $\theta \in S$ 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. Conditional narrowing has been shown to be a complete $\mathcal{E}$-unification algorithm for canonical theories satisfying different restrictions [14, 15, 26].

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* is any well-defined criterion which obtains a smaller search space by permitting narrowing to reduce only some chosen positions, e.g. *basic* [16], *innermost* [8], *innermost basic* [15] or *lazy* narrowing [27]. Formally, a narrowing strategy φ is a mapping that assigns to every goal g (different from $\top$) a subset $\varphi(g)$ of $\bar{O}(g)$ such that for all $u \in \varphi(g)$ the goal g is narrowable at occurrence u. A survey of results about the completeness of narrowing strategies can be found in [14]. In the case of a confluent and decreasing CTRS $\mathcal{R}$, we can further improve narrowing without losing completeness by normalizing the goal between narrowing steps [15, 17]. A *normalizing conditional narrowing step* w.r.t. $\mathcal{R}$, $g \overset{\sigma}{\leadsto} g'{\downarrow}$ is given by a narrowing step $g \overset{\sigma}{\leadsto} g'$ followed by a normalization $g' \rightarrow^*_{\mathcal{R}} g'{\downarrow}$. The idea of exploiting deterministic computations by including normalization has been applied to almost all narrowing strategies, e.g. basic [15], innermost [8], innermost basic [15], and lazy narrowing [13].

3 Partial Evaluation of Functional Logic Programs

In this section, we present a generic procedure for the PE of functional logic programs and show its correctness. Our construction is formalized within the theoretical framework established in [24, 25] for the partial deduction of logic programs.

In logic programming, the idea of PE is basically the following [24]. Let us consider a program P and an atomic goal G. Then construct a (finite) SLD-tree for $P \cup \{G\}$ containing at least one nonroot node. From this tree the set of clauses $\{G\theta_i \leftarrow G_i\}$, called resultants, is obtained by collecting the goal G_i and the corresponding θ_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 from a term s in a canonical program $\mathcal{R}$ as follows.

Definition 1. Let s be a term and $\mathcal{R}$ a canonical program. Consider the equation $s = y$, with the variable $y \notin Var(s)$. Let $D \equiv [(s = y) \overset{\theta}{\leadsto}{}^{*} g, e]$ be a conditional narrowing derivation for the goal $s = y$ in the program $\mathcal{R}_+ = \mathcal{R} \cup \{x = x \rightarrow true\}$. Let $\sigma = mgu(e)$. Then the resultant of the derivation is: $((s \rightarrow y)\theta \Leftarrow g)\sigma$.

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

Example 1. Let the rule $(f(x) \rightarrow x \Leftarrow a = x, b = x)$ be in $\mathcal{R}$. The resultant of:
1) the derivation $f(f(z)) = y \overset{\{x/f(z)\}}{\leadsto} a = f(z), b = f(z), f(z) = y \overset{\{y/f(z)\}}{\leadsto}$ $a = f(z), b = f(z), true$ is: $f(f(z)) \rightarrow f(z) \Leftarrow a = f(z), b = f(z)$.
2) the derivation $f(z) = y \overset{\{z/x\}}{\leadsto} a = x, b = x, x = y$ is: $f(y) \rightarrow y \Leftarrow a = y, b = y$.
Note that, without applying the mgu $\sigma = \{x/y\}$ ($\sigma = \{y/x\}$) of the last equation $x = y$, we would have obtained the rule: $f(x) \rightarrow y \Leftarrow a = x, b = x, x = y$, which contains an extra-variable y.
3) the derivation $f(z) = y \overset{\{z/x\}}{\leadsto} a = x, b = x, x = y \overset{\{x/a\}}{\leadsto} true, b = a, a = y$ is: $f(a) \rightarrow a \Leftarrow true, b = a$.

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 $s \rightarrow s$.

Definition 2. Let $\mathcal{R}$ be a program, s a term and $y \notin Var(s)$ a variable. Let τ be a finite (possibly incomplete) narrowing tree for the goal $s = y$ in the extended program $\mathcal{R}_+$ containing at least one nonroot node. Let $\{g_i \mid i = 1, \ldots, k\}$ be the nonfailing leaves of τ and $\{r_i \mid i = 1, \ldots, k-1\}$ the nontrivial resultants associated with the derivations $\{(s = y) \overset{\sigma_i}{\leadsto}{}^{+} g_i \mid i = 1, \ldots, k\}$. Then, the set $\{r_i \mid i = 1, \ldots, k-1\}$ is called a *partial evaluation of* s *in* $\mathcal{R}$ *(using* τ*)*.

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

The assumption that the initial goal is atomic simplifies the formal development of our framework and this requirement guarantees that, at each step of a derivation, the associated resultant is indeed a program rule, that is, the produced resultants do not contain extra-variables. Also, due to the form of the initial query $s = y$, 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 $(s \rightarrow y)\theta\sigma$ which are the heads of the produced resultants can be proven terminating, as we state in the following proposition.

Proposition 3. *The program obtained as the PE of a term in a noetherian program is noetherian.*

Following [24], we introduce a closedness condition under which our transformation is sound and complete w.r.t. the operational semantics of functional logic programs. Roughly speaking, the notion of closedness guarantees that all calls which might occur during the execution of the resulting program are covered by some program rule.

The following definitions are necessary for our notion of closedness. A function symbol $f \in \Sigma$ is irreducible iff there is no rule $(\lambda \rightarrow \rho \Leftarrow C) \in \mathcal{R}$ such that 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 (*constructors*) and $\mathcal{F}$ is the set of defined function symbols. A substitution σ is *(ground) constructor*, if $x\sigma$ is (ground) constructor for all $x \in Dom(\sigma)$. An expression can be a single rule/equation or a set of rules/equations. We let $terms(O)$ denote the function which extracts the terms appearing in the expression O.

Definition 4. Let S and T be two finite set of terms. We say that T is S-closed if $closed(S, T)$, where the predicate $closed$ is defined inductively as follows:

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

We say that a term t is S-closed if $closed(S, t)$, and we say that a program $\mathcal{R}$ is S-closed if $closed(S, terms(\mathcal{R}))$.

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

Example 2. Consider the following program: $\mathcal{R} = \{h(x) \rightarrow x,\ f(0) \rightarrow 0,\ f(c(x)) \rightarrow h(f(x))\}$, and the initial goal $f(c(x)) = y$. A PE of $\mathcal{R}$ w.r.t. $S = \{f(c(x)),\ h(x)\}$ is the specialized program $\mathcal{R}' = \{h(x) \rightarrow x,\ f(c(x)) \rightarrow h(f(x))\}$.

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

The PE theorem is formulated using the closedness condition.

Theorem 5. *Let $\mathcal{R}$ be a canonical program, g a goal, S a finite set of terms, and $\mathcal{R}'$ a partial evaluation of $\mathcal{R}$ w.r.t. S. Then,*

1. (SOUNDNESS) $\theta \in \mathcal{O}_{\mathcal{R}'}(g) \implies \exists \gamma \in \mathcal{O}_{\mathcal{R}}(g)$ *s.t.* $\gamma \leq_{\mathcal{E}} \theta\, [Var(g)]$[3].
2. (COMPLETENESS) $\mathcal{O}_{\mathcal{R}}(g) \subseteq \mathcal{O}_{\mathcal{R}'}(g)$, *if* $\mathcal{R}' \cup \{g\}$ *is* S*-closed.*

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

Definition 6 Overlap. A term s overlaps a term t if there is a nonvariable subterm $s_{|u}$ of s such that $s_{|u}$ and t unify. If $s \equiv t$ we require that t is unifiable with a proper nonvariable subterm of s.

Definition 7 Independence. A set of terms S is independent if there are no terms s and t in S such that s overlaps t.

Theorem 8. *Let $\mathcal{R}$ be a canonical program, g a goal, and S a finite set of terms. Let $\mathcal{R}'$ be a partial evaluation of $\mathcal{R}$ w.r.t. S such that $\mathcal{R}' \cup \{g\}$ is S-closed. Then,*

1. (STRONG SOUNDNESS) $\mathcal{O}_{\mathcal{R}'}(g) \subseteq \mathcal{O}_{\mathcal{R}}(g)$, *if S is independent.*
2. (COMPLETENESS) $\mathcal{O}_{\mathcal{R}'}(g) \supseteq \mathcal{O}_{\mathcal{R}}(g)$.

Roughly speaking, the condition of independence guarantees that the derived program $\mathcal{R}'$ does not produce additional answers. The following example illustrates that the independence condition cannot be dropped.

Example 3. Consider the following program: $\mathcal{R} = \{f(0) \rightarrow 0\}$, and the set $S = \{f(f(x)), f(0)\}$. A PE of $\mathcal{R}$ w.r.t. S is $\mathcal{R}' = \{f(f(0)) \rightarrow f(0),\ f(f(0)) \rightarrow 0,\ f(0) \rightarrow 0\}$. Then $\mathcal{R}' \cup \{f(f(x)) = y\}$ is S-closed and has a refutation with computed answer $\theta = \{x/f(0), y/f(0)\}$. $\mathcal{R} \cup \{f(f(x)) = y\}$ does not have a refutation with computed answer θ. Note that the specialized program $\mathcal{R}'$ is confluent.

Theorem 5 and Theorem 8 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 simplicity, in this paper we do not consider 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 [4, 11].

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

[3] In this result we consider $\mathcal{E}$ as the theory axiomatized by $\mathcal{R}$.

computed, with S initialized to the set of terms appearing in g. Then this process is repeated for any term occurring in the rhs 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 S' and a set of rules $\mathcal{R}'$ (the PE of S' in $\mathcal{R}$) such that each term in S is closed w.r.t. S' and the closedness condition for $\mathcal{R}' \cup \{g\}$ is satisfied.

As for termination, the PE procedure outlined above involves the two classical termination problems mentioned in Section 1. The first problem – 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 calls. 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 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 [12, 29, 31], where these two issues are not (explicitly) distinguished, as only one-step unfolding is performed, and a large evaluation structure is built which comprises something similar to both our local narrowing trees and the global configurations of [25].

The approach we follow originates from the framework for ensuring global termination of partial deduction given in [25]. The extension of this method to a functional framework is nontrivial. In the following, we formalize a general algorithm for PE of functional logic programs based on narrowing which is proven to terminate (for appropriate instances) while ensuring that the closedness condition is satisfied, and still provides the right amount of polyvariance (the possibility of producing a number of independent specializations for a given call using different data [19]) 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, we formalize the notion of a generic unfolding strategy $U_{\leadsto_\varphi}$ (that we simply denote by U_φ 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.

Definition 9. An *unfolding rule* U_φ is a function which, when given a program $\mathcal{R}$, a term s and a narrowing transition relation $\leadsto_\varphi$, returns a finite set of resultants $U_\varphi(s, \mathcal{R})$ that is a partial evaluation of s in $\mathcal{R}$ using $\leadsto_\varphi$.

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

We formulate our method to compute a PE of a program $\mathcal{R}$ w.r.t. a finite set of terms S using U_φ, by means of a transition system $(\mathit{State}, \longmapsto_{\mathcal{P}})$ whose transition relation $\longmapsto_{\mathcal{P}} \subseteq \mathit{State} \times \mathit{State}$ formalizes the computation steps. The set *State* of PE configurations is a parameter of the definition. The notion of state has to be instantiated in the specialization process. We let $c[S] \in \mathit{State}$ 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 partially evaluated terms S. When S is clear from the context, $c[S]$ will simply be denoted by c.

Definition 10 PE transition relation $\longmapsto_{\mathcal{P}}$. We define the PE relation $\longmapsto_{\mathcal{P}}$ as the smallest relation satisfying

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

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

Roughly speaking, at each computation step, the set of partially evaluated terms S (recorded in c) is evaluated (using U_φ). Then the terms appearing in the residual program $\mathcal{R}'$ which are not closed w.r.t. S are (properly) added to 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. Similarly to [25], applying *abstract* in every iteration allows us to tune the control of polyvariance as much as needed.

Definition 11 Initial PE configuration. Let g be a goal and c_0 the "empty" PE state. The initial PE configuration is: $\mathit{abstract}(c_0, \mathit{terms}(g))$.

Definition 12 Behavior of the $\longmapsto_{\mathcal{P}}$ calculus. Let us define the function: $\mathcal{P}(\mathcal{R}, g) = S$ if $\mathit{abstract}(c_0, \mathit{terms}(g)) \longmapsto^*_{\mathcal{P}} c[S]$ and $c[S] \longmapsto_{\mathcal{P}} c[S]$.

The procedure in Definition 12 computes the set of partially evaluated terms S which unambiguously determines its associated partial evaluation $\mathcal{R}'$ in $\mathcal{R}$ (using U_φ). The following theorem establishes the correctness of the PE method.

Theorem 13 Partial correctness of $\mathcal{P}$. *Let* abstract *be any abstraction operator satisfying that, if $\mathit{abstract}(c_1[S_1], S') = c_2[S_2]$, then $(S_1 \cup S')$ is S_2-closed. If $\mathcal{P}(\mathcal{R}, g)$ terminates computing the set of terms S, then $\mathcal{R}' \cup \{g\}$ is S-closed, where the specialized program $\mathcal{R}' = U_\varphi(S, \mathcal{R})$.*

Definition 12 incorporates only the scheme of a complete method for PE. The resulting partial evaluations might be further optimized by eliminating redundant functors and unnecessary repetition of variables, trying to adapt standard techniques presented in [4, 10, 11]. 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. The resulting mechanism should serve, among other purposes, to remove any remaining lack of independence. We consider this issue as a task for further research.

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 performed. 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 [29]. The next definition extends the homeomorphic embedding ("syntactically simpler") relation [7] to nonground terms.

Definition 14 Embedding relation. [29] The homeomorphic embedding relation $\trianglelefteq$ on terms in $\tau(\Sigma \cup V)$ is defined as the smallest relation satisfying: $x \trianglelefteq y$ for all $x, y \in V$, and $s \equiv f(s_1, \ldots, s_m) \trianglelefteq g(t_1, \ldots, t_n) \equiv t$, if and only if: 1) $f \equiv g$ (and $m \equiv n$) and $s_i \trianglelefteq t_i$ for all $i = 1, \ldots, n$, or 2) $s \trianglelefteq t_j$, for some j, $1 \leq j \leq n$.

Roughly speaking, $s \trianglelefteq t$ if s may be obtained from t by deletion of operators. For example, $\surd\surd(u \times (u + v)) \trianglelefteq (w \times \surd\underline{\surd\surd}((\surd\underline{u} + \surd u)\underline{\times}(\surd\underline{u+}\surd\underline{v})))$. The following result is a consequence of Kruskal's Tree Theorem.

Theorem 15. *Any infinite sequence of terms $t_1, t_2, \ldots$ with a finite number of operators is self-embedding, i.e., there are numbers j, k with $j < k$ and $t_j \trianglelefteq t_k$.*

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. 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 say that the terms s and t are comparable, in symbols $comparable(s, t)$, iff the outermost function symbol of s and t coincide. We also need the following notation.

Definition 16 Admissible derivation. Let D be a narrowing derivation for g_0 in $\mathcal{R}$. We say that D is admissible iff it does not contain a pair of comparable redexes included in the embedding relation $\trianglelefteq$. Formally,

$$
\begin{aligned}
admissible(g_0 \overset{[u_0,\theta_0]}{\leadsto_\varphi} \ldots \overset{[u_{n-1},\theta_{n-1}]}{\leadsto_\varphi} g_n) \Leftrightarrow \\
\forall i = 1, \ldots, n,\ \forall u \in \varphi(g_i),\ \forall j = 0, \ldots, i-1. \\
(comparable(g_{j|u_j}, g_{i|u}) \Rightarrow g_{j|u_j} \not\trianglelefteq g_{i|u}).
\end{aligned}
$$

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

Definition 17 Nonembedding narrowing tree $\tau_\varphi^{\trianglelefteq}$.
$\tau_\varphi^{\trianglelefteq}(g_0, \mathcal{R}) = \{g_0 \overset{[u_0,\theta_0]}{\leadsto_\varphi} \dots \overset{[u_{n-1},\theta_{n-1}]}{\leadsto_\varphi} g_n \overset{[u_n,\theta_n]}{\leadsto_\varphi} g_{n+1} \mid$
$\quad admissible(g_0 \overset{[u_0,\theta_0]}{\leadsto_\varphi} \dots \overset{[u_{n-1},\theta_{n-1}]}{\leadsto_\varphi} g_n) \wedge$
$\quad (g_{n+1} = \top \vee g_{n+1}$ *is a failing leaf* $\vee$
$\quad (\exists u \in \varphi(g_{n+1}),\ \exists i \in \{1, \dots, n\}.\ comparable(g_{i|u_i}, g_{n+1|u}) \wedge g_{i|u_i} \trianglelefteq g_{n+1|u}))\}$.

Hence, derivations are stopped when they either fail, succeed or the considered redexes satisfy the embedding ordering. Before illustrating Definition 17 by means of a simple example, we state the following.

Theorem 18 Local termination. *For a program $\mathcal{R}$ and goal g, $\tau_\varphi^{\trianglelefteq}(g, \mathcal{R})$ is a finite (possibly incomplete) narrowing tree for $\mathcal{R} \cup \{g\}$ using $\leadsto_\varphi$.*

Example 4. Consider the well-known program append/2

$$append(nil, y_s) \to y_s$$
$$append(x : x_s, y_s) \to x : append(x_s, y_s)$$

with initial query $append(1 : 2 : x_s, y_s) = y$. There exists the following infinite branch in the (unrestricted) narrowing tree (at each step, we underline the redex selected for narrowing):

$$\underline{append(1:2:x_s,y_s)} = y \overset{\{\}}{\leadsto} 1 : \underline{append(2:x_s,y_s)} = y \overset{\{\}}{\leadsto} 1:2:\underline{append(x_s,y_s)} = y$$
$$\overset{\{x_s/x':x_s'\}}{\leadsto} 1:2:x':\underline{append(x_s',y_s)} = y \overset{\{x_s'/x'':x_s''\}}{\leadsto} \dots$$

According to Definition 17, the development of this branch is stopped at the fourth goal, since the derivation $append(1 : 2 : x_s, y_s) = y \overset{\{\}}{\leadsto} 1 : append(2 : x_s, y_s) = y \overset{\{\}}{\leadsto} 1 : 2 : append(x_s, y_s) = y$ is admissible, and the step $1 : 2 : \underline{append(x_s, y_s)} = y \overset{\{x_s/x':x_s'\}}{\leadsto} 1 : 2 : x' : \underline{append(x_s', y_s)} = y$ fulfills the ordering, because $append(x_s, y_s) \trianglelefteq append(x_s', y_s)$.

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

Definition 19 Nonembedding unfolding rule $U_\varphi^{\trianglelefteq}$.
We define $U_\varphi^{\trianglelefteq}(s, \mathcal{R})$ as the PE of s in $\mathcal{R}$ using $\tau_\varphi^{\trianglelefteq}(s = y, \mathcal{R})$, $y \notin Var(s)$.

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.

4.2 Global Termination

In this section, we show how the abstract operator which is a parameter of the generic algorithm in Definition 12 can be defined using a simple kind of structure consisting of 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 [25].

Definition 20 PE* configuration. Let $State^* = \tau(\Sigma \cup V)^*$ 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 ",". A *PE* configuration* is a sequence of terms $(t_1, \ldots, t_n) \in State^*$. The *empty PE* configuration* is nil.

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

Definition 21. Let q be a PE^* configuration and T be an expression. We define $abstract^*$ inductively as follows: $abstract^*(q, T) =$

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

where, given a term T, the function $abs\_call(nil, T)$ is T, and $abs\_call(q, T)$ ($q \not\equiv nil$) is defined as follows: $abs\_call((q_1, \ldots, q_n), T) =$

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

The loss of precision caused by the use of the generalization operator msg is quite reasonable and is compensated by the simplicity of the resulting method. The following example illustrates how our method achieves both, termination and specialization. The positive supercompiler of [12, 30] 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 [4] results in the same nontermination pattern for a logic programming version of this program. The methods in [25, 29] would instead terminate on this example.

Example 5. Consider the following program, which checks whether a sequence is a palindrome by using a reversing function with accumulating parameter:

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

and consider the goal $palindrome(1 : 2 : x) = y$. Using the nonembedding unfolding rule $U_{\rightsquigarrow}^{\trianglelefteq}$ of Definition 19 to stop the (normalizing conditional) narrowing derivations, and the $abstract^*$ operator of Definition 21 to ensure total correctness, the specialized program $\mathcal{R}'$ resulting from the PE of $\mathcal{R}$ w.r.t. the set of terms

$S' = \{palindrome(1:2:x_s), rev(x_s, y_s)\}$ is:

$$\begin{aligned}
\mathcal{R}' = \{ \quad palindrome(1:2:x:x_s) &\rightarrow true \quad \Leftarrow \quad rev(x_s, x:2:1:nil) = 1:2:x:x_s \\
rev(nil, x_1:x_2:x_3:y_s) &\rightarrow x_1:x_2:x_3:y_s \\
rev(x:x_s, x_1:x_2:x_3:y_s) &\rightarrow rev(x_s, x:x_1:x_2:x_3:y_s) \quad \}
\end{aligned}$$

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*. Note that the resulting set of terms S' is independent.

The following theorems establish the correctness of the resulting algorithm.

Lemma 22 Partial correctness. *If $abstract^*(q, S) = q'$, then $terms(q) \cup S$ is closed w.r.t. $terms(q')$.*

Theorem 23 Termination. *The algorithm in Definition 12 terminates for the domain $State^*$ of* PE* *configurations and the abstraction operator $abstract^*$.*

The last example illustrates that our method can also eliminate intermediate data structures and turn multiple–pass programs into one–pass programs, as the deforestation method and the positive supercompiler of [30] do.

Example 6. Consider again the program append/2 of Example 4 with initial query $append(append(x_s, y_s), z_s) = y$. 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 $q = append(append(x_s, y_s), z_s)$, and by using the procedure described in Definition 12, we compute the trees depicted in Figure 1 for the sequence of terms $q' = append(append(x_s, y_s), z_s), append(x_s, y_s)$. Note that "append" has been abbreviated to "a" in the picture. Then we get the following residual program $\mathcal{R}'$:

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

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 1. Note that we did not adopt any specific strategy (like the call-by-name or the call-by-value ones) for executing the goal. Thus a lazy evaluation strategy does not seem essential in this example, contradicting a conjecture posed in [30]. The resulting set of terms $\{append(append(x_s, y_s), z_s), append(x_s, y_s)\}$ in q' is not independent. This example illustrates the need for an extra *renaming* phase able to produce an independent set of terms such as $\{append(append'(x_s, y_s), z_s), append''(x_s, y_s)\}$ and associated specialized program:

$$\begin{aligned}
append(append'(nil, y_s), z_s) &\rightarrow append''(y_s, z_s) \\
append(append'(x:x_s, y_s), z_s) &\rightarrow x : append(append'(x_s, y_s), z_s) \\
append''(nil, z_s) &\rightarrow z_s \\
append''(y:y_s, z_s) &\rightarrow y : append''(y_s, z_s)
\end{aligned}$$

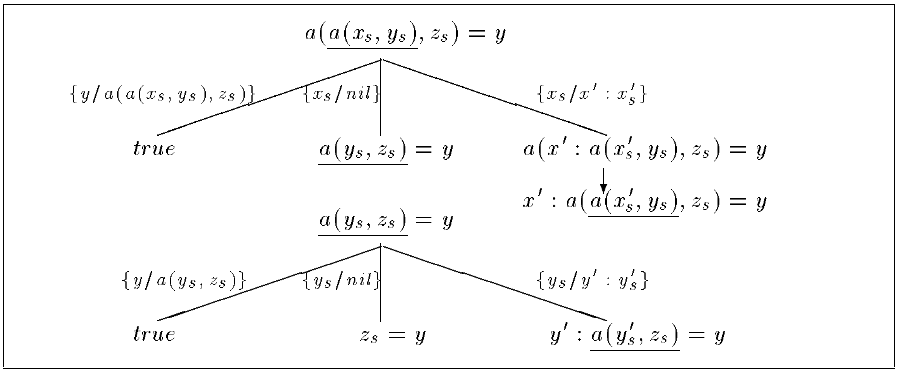

Fig. 1. Narrowing trees for the goals $a(a(x_s, y_s), z_s) = y$ and $a(x_s, y_s) = y$.

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

The use of efficient forms of narrowing can significantly improve the accuracy of the specialization method and increase the efficiency of the resulting program, because some run-time optimizations (e.g. normalization steps) can be performed at compile time. Different (highly efficient) instances of the framework can be considered, e.g. for innermost and lazy narrowing, which resemble the call-by-value and call-by-name cases in functional programming. The choice of an innermost narrowing strategy allows us to formalize in [2] a call-by-value partial evaluator for functional logic programs which makes use of the simple mechanisms introduced so far to achieve (both local and global) termination. Our method passes the so-called Knuth-Morris-Pratt test [12, 30], i.e. specializing a naïve pattern matcher w.r.t. a fixed pattern obtains the efficiency of the Knuth, Morris and Pratt matching algorithm [20].

5 Conclusions and Further Research

PE is a semantics-preserving program transformation based on unfolding and specializing procedures. Techniques in conventional PE of functional programs usually rely on the reduction of expressions and constant propagation, while transformation techniques for logic languages exploit unification-based parameter propagation [12]. The driving approach essentially achieves the same transformational effect for functional programs. Few attempts have been made to study the relationship between techniques used in logic and functional languages [12]. 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 simpler and some of 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 as in [12]. We have shown how a core

PE procedure whose behaviour does not depend on the eager or lazy nature of the narrower can be defined. In [2] we considered the case of normalizing innermost narrowing which is known to be a reasonable improvement over pure logic SLD resolution strategy [8, 14]. It is worthwhile to investigate the instantiation of our framework for other strategies, such as the definition of a call-by-name partial evaluator based on lazy narrowing [13, 27].

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 environments of positive and negative bindings (bindings which do not hold) [30, 31]. We think that we can strengthen this effect in the setting of (equational) constraint logic programming [1, 18] by using some kind of narrowing procedure with disunification, such as the one defined in [3], in order to propagate (negative) bindings which can be gathered during the transformation as (disequality) constraints. Automatic generation of such generalized specializations is the subject of further work.

References

1. M. Alpuente, M. Falaschi, and G. Levi. Incremental Constraint Satisfaction for Equational Logic Programming. *Theoretical Computer Science*, 142:27–57, 1995.
2. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven specialization of Functional Logic Programs. Technical Report DSIC-II/27/95, UPV, 1995.
3. P. Arenas, A. Gil, and F. López. Combining Lazy Narrowing with Disequality Constraints. In *Proc. of PLILP'94*, pages 385–399. Springer LNCS 844, 1994.
4. 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.
5. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
6. 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.
7. 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, 1990.
8. 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.
9. Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
10. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
11. 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.
12. R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *Proc. of PLILP'94*, pages 165–181. Springer LNCS 844, 1994.

13. M. Hanus. Combining Lazy Narrowing with Simplification. In *Proc. of PLILP'94*, pages 370–384. Springer LNCS 844, 1994.
14. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
15. S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNAI 353, 1989.
16. J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.
17. H. Hussmann. Unification in Conditional-Equational Theories. In *Proc. of EUROCAL'85*, pages 543–553. Springer LNCS 204, 1985.
18. 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.
19. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
20. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal of Computation*, 6(2):323–350, 1977.
21. 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.
22. 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.
23. 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.
24. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
25. 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*, pages 597–611, 1995.
26. A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.
27. 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.
28. P. Sestoft and H. Søndergaard. A bibliography on partial evaluation. *Sigplan Notices*, 23(2):19–27, Feb 1988.
29. M.H. Sørensen and R. Glück. Generalization in Positive Supercompilation. In J.W. Lloyd, editor, *Proc. of ILPS'95*, 1995.
30. M.H. Sørensen, R. Glück, and N.D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In D. Sannella, editor, *Proc. of ESOP'94*, pages 485–500. Springer LNCS 788, 1994.
31. V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
32. 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.