Specialization of Lazy Functional Logic Programs*

M. Alpuente[‡] M. Falaschi[§] P. Julián[¶] G. Vidal[‡]

[‡] DSIC, UPV, Camino de Vera s/n, 46071 Valencia, Spain. E-mail: {alpuente,gvidal}@dsic.upv.es.

[§] Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. E-mail: falaschi@dimi.uniud.it.

[¶] Dep. de Informática, Ronda de Calatrava s/n, 13.071 Ciudad Real, Spain. E-mail: pjulian@inf-cr.uclm.es.

Abstract

Partial evaluation is a method for program specialization based on fold/unfold transformations [8, 25]. Partial evaluation of pure functional programs uses mainly static values of given data to specialize the program [15, 44]. In logic programming, the so-called static/dynamic distinction is hardly present, whereas considerations of determinacy and choice points are far more important for control [12]. We discuss these issues in the context of a (lazy) functional logic language. We formalize a two-phase specialization method for a non-strict, first order, integrated language which makes use of lazy narrowing to specialize the program w.r.t. a goal. The basic algorithm (first phase) is formalized as an instance of the framework for the partial evaluation of functional logic programs of [2, 3], using lazy narrowing. However, the results inherited by [2, 3] mainly regard the termination of the PE method, while the (strong) soundness and completeness results must be restated for the lazy strategy. A post-processing renaming scheme (second phase) is necessary which we describe and illustrate on the well-known matching example. This phase is essential also for other non-lazy narrowing strategies, like innermost narrowing, and our method can be easily extended to these strategies. We show that our method preserves the lazy narrowing semantics and that the inclusion of simplification steps in narrowing derivations can improve control during specialization.

Keywords: integration of functional and logic programming, term rewriting systems, lazy narrowing, partial evaluation, post-processing renaming.

1 Introduction

Many proposals for the integration of functional and logic programming are based on narrowing (see [18] for a recent survey). Narrowing is a natural extension of the evaluation mechanism of functional languages to incorporate unification. Narrowing solves equations by computing unifiers w.r.t. an equational theory usually described by means of a (conditional) term rewriting system. In order to avoid useless computations and to deal with nonterminating and nonstrict functions, lazy narrowing strategies have recently been proposed [4, 29, 35, 39]. One main advantage of an integrated functional logic language is the reduction of the search space by exploiting functional computations. Hence, an important improvement of (lazy) narrowing is the incorporation of deterministic simplification steps which can largely reduce both run time and search space in comparison to pure logic programs, since normalization can avoid the creation of useless choice points in sequential implementations [17, 18].

Program specialization refers to the technique of how to derive a specialized instance of a program to a restricted set of inputs. Particular cases include Partial Evaluation (PE) of functional [25] and logic [12, 34] programs. PE of functional programs, as in [25], is usually restricted to constant (static values) propagation, whereas PE techniques for logic languages exploit unification-based information propagation, which results in more powerful transformations. The basic technique for PE of logic programs was presented in [12]. Turchin's *driving* transformation for functional programs achieves the same effect as the PE of logic programs, by virtue of unification [15, 45, 46].

The PE of functional logic languages is a relatively new area of research. As far as we know, [2, 3] formalize the first PE scheme for functional logic languages which can improve the original program w.r.t. the ability of computing the set of answer substitutions. In contrast to the approach usually taken with pure functional languages, in [2] we use the unification-based computation mechanism of narrowing for the specialization of the program as well as for the execution. The basic PE algorithm is parametric w.r.t. the narrowing strategy which is used for the automatic construction of (finite) narrowing trees. We have defined the notions of *closedness* and *independence* that are essential to prove the computational equivalence of the original and the partially evaluated programs, for a restricted set of goals. We have proved that these conditions suffice for correctness in the case of unrestricted narrowing. An appropriate abstraction operator is also introduced which guarantees termination of the PE process. However, the independence condition which guarantees that the derived program does not produce additional answers is not obtained automatically and, for some particular narrowing strategies, the partially evaluated program might not satisfy the restrictions on the theories which are necessary for the completeness of the strategy.

In this paper, we formalize a call-by-name partial eval-

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

uator for functional logic languages with a lazy narrowing semantics like that of [39]. Then, we formalize a renaming transformation of the residual program, which removes any remaining lack of independence. This is a post-processing stage whereby new function symbols are introduced and a transformed program and goal are obtained. We prove that, for the renamed queries, the transformed program computes the same answers as the original program. We note that the post-processing phase is also crucial to guarantee the correctness of the whole process. In general, the partially evaluated program resulting from the PE phase might not satisfy one of the basic assumptions for the completeness of lazy narrowing (the so-called 'constructor discipline'), which may prevent the lazy strategy from being able to narrow a goal in the partially evaluated program. As a rough example, consider the following

Example 1 *Consider the program* $\mathcal{R} \equiv \{\mathbf{f}(\mathbf{x}) \to \mathbf{x}\}$ *which derives the residual program* $\mathcal{R}'$*:*

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

when specialized w.r.t. the function calls $\mathbf{f}(\mathbf{f}(\mathbf{x}))$ *and* $\mathbf{f}(0)$*. This program contains a nested function application* $\mathbf{f}(\mathbf{f}(\mathbf{x}))$*. Also, the considered set of specialized calls* $\{\mathbf{f}(0), \mathbf{f}(\mathbf{f}(\mathbf{x}))\}$ *is not independent, since it contains two different pairs of interfering calls, namely* $\mathbf{f}(0)$ *with* $\mathbf{f}(\mathbf{x})$ *and* $\mathbf{f}(\mathbf{x})$ *with* $\mathbf{f}(\mathbf{f}(\mathbf{x}))$*.*

The post-processing phase generates a constructor based program, i.e., no left-hand side argument contains a defined function symbol, and the renamed set of calls is independent. Our method is also useful for non lazy narrowing strategies, such as innermost narrowing. Also in this case the resulting program must be constructor based. The extension of our method is easy [1].

Our method passes the so-called Knuth-Morris-Pratt test [15, 24], i.e. the specialization of a naïve pattern matcher w.r.t. a fixed pattern essentially obtains the efficiency of the Knuth, Morris and Pratt matching algorithm [27].

We show that the inclusion of a normalization process between narrowing steps not only saves time and space but also yields a better optimization strategy which does not increase the size of the program since no choices are unfolded. We show that, by using normalization, it is possible to improve the ability to eliminate intermediate data structures with a narrowing-based partial evaluator. We also prove that the partial evaluator can achieve in a principled, non ad-hoc way, the same transformation effect as some (postunfolding) techniques that eliminate intermediate functions in the driving approach to program specialization [15, 42, 43]. The preference for deterministic computations is a good heuristic to avoid code explosion which is also comparable to the determinism-based criterium of [13] that explores maximal deterministic paths.

1.1 Plan of the Paper

The structure of the paper is as follows. Basic definitions are given in Section 2. Section 3 recalls the general scheme for the PE of functional logic languages of [2]. In Section 4, a two-phase specialization method is described and shown to be correct. Section 5 discusses the related work and Section 6 concludes. More details and missing proofs can be found in [1].

2 Preliminaries

We briefly recall some known results about rewrite systems and functional logic programming [10, 18, 21, 26]. Throughout this paper, $\mathbf{V}$ will denote a countably infinite set of variables and Σ denotes a set of function symbols $\mathbf{f/n}$, each with a fixed associated arity $\mathbf{n}$. $\tau(\Sigma \cup \mathbf{V})$ and $\tau(\Sigma)$ denote the sets of terms and ground terms built on Σ and $\mathbf{V}$, respectively. We assume that the alphabet Σ contains some primitive symbols, including at least the nullary constructor **true** and a binary equality function symbol, say $=$, written in infix notation, which allows us to interpret equations $\mathbf{s} = \mathbf{t}$ as terms, with $\mathbf{s}, \mathbf{t} \in \tau(\Sigma \cup \mathbf{V})$. The term **true** is also considered an equation. Terms are viewed as labeled trees in the usual way. 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. $\mathbf{O}(\mathbf{t})$ denotes the set of occurrences of a term $\mathbf{t}$. $\mathbf{O}_{\mathcal{V}}(\mathbf{t})$ and $\bar{\mathbf{O}}(\mathbf{t})$ denote the set of variable occurrences and nonvariable occurrences of the term $\mathbf{t}$, respectively. $\mathbf{t}_{|\mathbf{u}}$ is the subterm at the occurrence $\mathbf{u}$ of $\mathbf{t}$. $\mathbf{t}[\mathbf{r}]_{\mathbf{u}}$ is the term $\mathbf{t}$ with the subterm at the occurrence $\mathbf{u}$ replaced with $\mathbf{r}$. These notions extend to sequences of equations in a natural way. For instance, the nonvariable occurrence set of a sequence of equations $\mathbf{g} \equiv (\mathbf{e}_1, \ldots, \mathbf{e_n})$ can be defined as follows: $\bar{\mathbf{O}}(\mathbf{g}) = \{\mathbf{i.u} \mid \mathbf{u} \in \bar{\mathbf{O}}(\mathbf{e_i}), \mathbf{i} = 1, \ldots, \mathbf{n}\}$. Identity of syntactic objects is denoted by $\equiv$. $\mathbf{Var}(\mathbf{s})$ 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 **Sub**. 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 and $\mathbf{Cod}(\sigma) = \{\mathbf{x}\sigma \mid \mathbf{x} \in \mathbf{Dom}(\sigma)\}$ is called the codomain of σ. 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 $\mathbf{W}$. We write $\theta = \gamma\,[\mathbf{W}]$ if $\mathbf{x}\theta = \mathbf{x}\gamma\ \forall \mathbf{x} \in \mathbf{W}$. 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 $\theta = \{\mathbf{x}_1/\mathbf{t}_1, \ldots, \mathbf{x_n}/\mathbf{t_n}\}$ is the set of equations $\widehat{\theta} = \{\mathbf{x}_1 = \mathbf{t}_1, \ldots, \mathbf{x_n} = \mathbf{t_n}\}$. A substitution θ is a *unifier* of an equation set $\mathbf{E}$ iff $\widehat{\theta} \Rightarrow \mathbf{E}$. A set of equations $\mathbf{E}$ is unifiable, if there exists $\vartheta \in \mathbf{Sub}$ such that for all $\mathbf{s} = \mathbf{t}$ in $\mathbf{E}$. $\mathbf{s}\vartheta \equiv \mathbf{t}\vartheta$. ϑ is called a unifier of $\mathbf{E}$. We let $\mathbf{mgu}(\mathbf{E})$ denote the *most general unifier* of the equation set $\mathbf{E}$ (see, e.g., [30]). 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 $\mathbf{t}\rho \equiv \mathbf{t}'$. 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 $\langle \mathbf{t}, \Theta\rangle = \mathbf{msg}(\mathbf{S})$, if 1) $\langle \mathbf{t}, \Theta\rangle$ is a generalization of $\mathbf{S}$, and 2) for every other generalization $\langle \mathbf{t}', \Theta'\rangle$ of $\mathbf{S}$, $\mathbf{t}'$ is more general than $\mathbf{t}$ (i.e. $\mathbf{t}' \leq \mathbf{t}$). The **msg** of a set of terms is unique up to variable renaming.

An equational Horn theory $\mathcal{E}$ consists of a finite set of equational Horn clauses of the form $\mathbf{e} \Leftarrow \mathbf{C}$. The head $\mathbf{e}$ is an equation $(\lambda = \rho)$ and 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.

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 $\mathbf{s}$ and $\mathbf{t}$, we say that they are $\mathcal{E}$-**unifiable** iff there exists a substitution σ such that $\mathbf{s}\sigma =_{\mathcal{E}} \mathbf{t}\sigma$. The substitution σ is called an $\mathcal{E}$-**unifier** of $\mathbf{s}$ and $\mathbf{t}$. 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 $\sigma =_{\mathcal{E}} \theta\gamma\ [\mathbf{Var}(\mathbf{E})]$ 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 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 $\mathbf{r} \equiv (\lambda \to \rho \Leftarrow \mathbf{C})$, λ, $\rho \in \tau(\Sigma \cup \mathbf{V})$, $\lambda \notin \mathbf{V}$ and $\mathbf{Var}(\rho) \subseteq \mathbf{Var}(\lambda)$. The reduction rule $\mathbf{r}$ is left–linear if λ is a linear term, i.e. no variable occurs twice or more in λ. If a rewrite rule has no condition we write $\lambda \to \rho$. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$.

Operationally, equational Horn clauses will be used as conditional rewrite rules. A term $\mathbf{s}$ conditionally rewrites to a term $\mathbf{t}$, written $\mathbf{s} \to_{\mathcal{R}} \mathbf{t}$, if there exists $\mathbf{u} \in \mathbf{O}(\mathbf{s})$, $(\lambda \to \rho \Leftarrow \mathbf{s}_1 = \mathbf{t}_1, \ldots, \mathbf{s_n} = \mathbf{t_n}) \in \mathcal{R}$ 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 $\mathbf{s_i}\sigma \to^*_{\mathcal{R}} \mathbf{w_i}\ {}_{\mathcal{R}}^{*}\!\!\leftarrow \mathbf{t_i}\sigma$. The CTRS $\mathcal{R}$ is said to be confluent if, for all terms $\mathbf{s}$, $\mathbf{t}_1$, $\mathbf{t}_2$ with $\mathbf{t}_1\ {}_{\mathcal{R}}^{*}\!\!\leftarrow \mathbf{s} \to^*_{\mathcal{R}} \mathbf{t}_2$, there exists a term $\mathbf{t}_3$ such that $\mathbf{t}_1 \to^*_{\mathcal{R}} \mathbf{t}_3\ {}_{\mathcal{R}}^{*}\!\!\leftarrow \mathbf{t}_2$. $\mathcal{R}$ is weak orthogonal if each rule of $\mathcal{R}$ is left–linear and $\mathcal{R}$ contains only trivial critical pairs. $\mathcal{R}$ is normal if the terms $\mathbf{t}_1, \ldots, \mathbf{t_n}$ in the condition $\mathbf{s}_1 = \mathbf{t}_1, \ldots, \mathbf{s_n} = \mathbf{t_n}$ of each program rule are ground normal forms w.r.t. the unconditional part of the CTRS. *Weakly orthogonal normal* CTRSs are confluent [6, 26]. 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 $\mathbf{s}$. A substitution σ is *normalized*, if $\mathbf{x}\sigma$ is a normal form for all $\mathbf{x} \in \mathbf{Dom}(\sigma)$. 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 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 $\mathbf{u} \in \mathbf{O}(\mathbf{t}) - \{\Lambda\}$, 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 $\mathbf{s} = \mathbf{t}$ in $\mathbf{C}$ (roughly speaking, in each conditional equation used for rewriting, the rhs and the condition terms must be smaller than the lhs w.r.t. some termination ordering).

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 $\Sigma = \mathcal{C} \uplus \mathcal{F}$, where $\mathcal{C}$ is the set of irreducible (*constructor*) function symbols and $\mathcal{F}$ is the set of defined function symbols or operations. The terms in $\tau(\mathcal{C} \cup \mathbf{V})$ are called *constructor terms*. A substitution σ is *(ground) constructor*, if $\mathbf{x}\sigma$ is (ground) constructor for all $\mathbf{x} \in \mathbf{Dom}(\sigma)$.

Narrowing Semantics

The computation mechanism of functional logic languages is based on narrowing, an evaluation mechanism that uses unification for parameter passing [40]. 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 $\mathbf{g}$ 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* (*red*ucible *ex*pression) 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 redex $\mathbf{t}_{|\mathbf{u}}$ is an *outermost redex* if there is no redex $\mathbf{t}_{|\mathbf{u}'}$ of $\mathbf{t}$ with $\mathbf{u}' \leq \mathbf{u}$. 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 $\sigma = \mathbf{mgu}(\{\mathbf{s} = \mathbf{t}\})$. We use $\top$ as a 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 $\theta_{\lceil \mathbf{Var}(\mathbf{g})}$ is called a computed answer substitution (c.a.s.) for $\mathbf{g}$ in $\mathcal{R}$. The narrowing derivations can be represented by a (possibly infinite) finitely branching tree. A *failing leaf* is a goal which is not $\top$ and which cannot be further narrowed. Following [34], we adopt the convention in this paper that any derivation is potentially incomplete (a branch thus can be failed, incomplete, successful or 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 [37]. Conditional narrowing has been shown to be a complete $\mathcal{E}$-unification algorithm for theories satisfying different restrictions [18, 21, 37].

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* [22], *innermost* [11], *innermost basic* [21] or *lazy* narrowing [40]. The innermost and the lazy strategies mimic the strict and lazy evaluation known from functional programming languages. Formally, a narrowing strategy φ is a mapping that assigns to every goal $\mathbf{g}$ (different from $\top$) a set of triples [4]. If $\langle \mathbf{u}, \mathbf{r}, \sigma \rangle \in \varphi(\mathbf{g})$, then $\mathbf{u} \in \bar{\mathbf{O}}(\mathbf{g})$, $\mathbf{r} \ll \mathcal{R}$, and σ is a substitution such that $\mathbf{g} \overset{[\mathbf{u},\mathbf{r},\sigma]}{\leadsto} \mathbf{g}'$. 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 [18].

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 [23]. It is useful to apply rewriting between narrowing steps, whenever it is possible, since rewriting may reduce an infinite search space to a finite one and can save a lot of time and space [18]. A *normalizing conditional narrowing step* w.r.t. $\mathcal{R}$, is given by a normalization $\mathbf{g} \to^*_{\mathcal{R}} \mathbf{g}\!\downarrow$ followed by a narrowing step $\mathbf{g}\!\downarrow \stackrel{\sigma}{\leadsto} \mathbf{g}'$. The idea of exploiting deterministic computations by including normalization has been applied to almost all narrowing strategies, e.g. basic [21, 41], innermost [11], innermost basic [21], and lazy narrowing [17].

Lazy narrowing reduces expressions at outermost narrowable positions. Narrowing at inner positions is performed only if it is demanded (by the pattern in the lhs of some rule) and contributes to some later narrowing step at an outer position. Since the notion of "demanded position" is not unique, different lazy narrowing strategies have been proposed [4, 29, 35, 39, 40] (although some lazy strategies are "lazier" than others). In the following, we specify our lazy narrowing strategy, similar to [39].

Lazy Narrowing

The following definitions are necessary for our formalization of lazy narrowing. An *pattern* $\mathbf{t}$ is an operation applied to constructor terms, i.e. $\mathbf{t} = \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t_k})$, where $\mathbf{f} \in \mathcal{F}$ and, for all $\mathbf{i} = 1, \ldots, \mathbf{k}$, $\mathbf{t_i} \in \tau(\mathcal{C} \cup \mathbf{V})$. A CTRS is *constructor-based* (CB) if the left-hand side (lhs) of each rule is a pattern. It implies that there can be neither functional nestings on the lhs of the head of the clauses nor equations between constructors. This is a reasonable class from the functional programming point of view. Many functional logic languages follow this discipline, e.g. ALF [16], Babel [39], K-LEAF [14], LPG [7], and SLOG [11]. A *head normal form* is a variable or a constructor-rooted term. In CB normal programs, each condition $\mathbf{s} = \mathbf{t}$ in the body of a program rule has the property that $\mathbf{t}$ is ground constructor [19]. We assume that these conditions hold for all programs we consider in this paper. This requirement is also made in Babel and K-LEAF. In practice, this is not a real restriction, since we can provide an explicit definition of a boolean–valued equality function $\approx$ between constructor terms and consider that an equation $\mathbf{s} \approx \mathbf{t}$ in a condition of a CB normal CTRS denotes the equation $(\mathbf{s} \approx \mathbf{t}) = \mathbf{true}$ [19].

The unification of redexes of the goal with lhs's of program rules gives rise to the following particular kind of unification problems.

Definition 2.1 (linear unification problem)
A linear unification problem is a pair of terms:
$\langle \mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d_n}), \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t_n}) \rangle$,
where $\mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d_n})$ *and* $\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t_n})$ *do not share variables, and the term* $\mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d_n})$ *is linear and innermost.*

Linear unification problems can be solved by any version of the unification algorithm (see, e.g., [30]). Following [39], we distinguish the case when an attempted unification does not succeed because of a clash between a constructor $\mathbf{c}$ and an operation $\mathbf{f}$, since such a situation can be thought of as a demand of further evaluation of $\mathbf{f}$. This differs from the standard syntactic unification algorithm [30]. The following is a reformulation of the unification algorithm for linear unification problems of [39].

Definition 2.2 (LU configuration) *An* LU configuration *is a pair* $(\mathbf{U}, \sigma)$*, where* $\mathbf{U}$ *is a set and* σ *is a substitution.*

The following definition adapts the standard syntactic unification algorithm [30] to the linear unification case. Because of linearity, an occur-check is not needed.

Definition 2.3 (unification relation $\to_{\mathsf{LU}}$)
We define the unification relation $\to_{\mathsf{LU}}$ *between* LU *configurations as the smallest relation satisfying:*

1. $(\{\mathbf{c}(\mathbf{d}_1, \ldots, \mathbf{d_m}) \downarrow_{\mathbf{u}} \mathbf{c}(\mathbf{t}_1, \ldots, \mathbf{t_m})\} \cup \mathbf{U}, \sigma)$
   $\to_{\mathsf{LU}} (\{\mathbf{d}_1 \downarrow_{\mathbf{u}.1} \mathbf{t}_1, \ldots, \mathbf{d_m} \downarrow_{\mathbf{u}.\mathbf{m}} \mathbf{t_m}\} \cup \mathbf{U}, \sigma)$,
   where $(\mathbf{c}/\mathbf{m}) \in \mathcal{C}, \mathbf{m} \geq 0$.
2. $(\{\mathbf{x} \downarrow_{\mathbf{u}} \mathbf{t}\} \cup \mathbf{U}, \sigma) \to_{\mathsf{LU}} (\mathbf{U}\{\mathbf{x}/\mathbf{t}\}, \sigma\{\mathbf{x}/\mathbf{t}\})$,
   where $\mathbf{t} \notin \mathbf{V}$.
3. $(\{\mathbf{d} \downarrow_{\mathbf{u}} \mathbf{x}\} \cup \mathbf{U}, \sigma) \to_{\mathsf{LU}} (\mathbf{U}\{\mathbf{x}/\mathbf{d}\}, \sigma\{\mathbf{x}/\mathbf{d}\})$.
4. $(\{\mathbf{c}(\mathbf{d}_1, \ldots, \mathbf{d_m}) \downarrow_{\mathbf{u}} \mathbf{c}'(\mathbf{t}_1, \ldots, \mathbf{t_p})\} \cup \mathbf{U}, \sigma)$
   $\to_{\mathsf{LU}} (\{\mathtt{fail}\}, \sigma)$,
   where $(\mathbf{c}/\mathbf{m}), (\mathbf{c}'/\mathbf{p}) \in \mathcal{C}$, $\mathbf{c} \not\equiv \mathbf{c}'$, *and* $\mathbf{m}, \mathbf{p} \geq 0$.

Definition 2.4 (initial LU configuration)
Let $\langle \mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d_n}), \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t_n}) \rangle$ *be a linear unification problem. The* initial LU configuration *is:*
$(\mathbf{U}_0, \sigma_0) \equiv (\{\mathbf{d}_1 \downarrow_1 \mathbf{t}_1, \ldots, \mathbf{d_n} \downarrow_{\mathbf{n}} \mathbf{t_n}\}, \epsilon)$.

Linear unification (LU) can either succeed, fail or suspend. When it suspends, it returns the set of positions which "demand" further evaluation. Formally, given a LU configuration $(\mathbf{U}, \sigma)$, a position $\mathbf{u}$ is *demanded*, if $(\mathbf{c}(\mathbf{d}_1, \ldots, \mathbf{d_m}) \downarrow_{\mathbf{u}} \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t_p})) \in \mathbf{U}$.

Definition 2.5 (behaviour of $\to_{\mathsf{LU}}$)
Let $\Gamma \equiv \langle \mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d_n}), \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t_n}) \rangle$ *be a linear unification problem. Let* $(\mathbf{U}_0, \sigma_0) \equiv (\{\mathbf{d}_1 \downarrow_1 \mathbf{t}_1, \ldots, \mathbf{d_n} \downarrow_{\mathbf{n}} \mathbf{t_n}\}, \epsilon) \to^*_{\mathsf{LU}} (\mathbf{U}, \sigma) \not\to_{\mathsf{LU}}$. *We define the function* $\mathsf{LU}(\Gamma)$ *as follows:*

$$\mathsf{LU}(\Gamma) = \begin{cases} (\textsc{Succ}, \sigma) & \textit{if } \mathbf{U} = \emptyset \\ (\textsc{Fail}, \emptyset) & \textit{if } \mathbf{U} = \{\mathtt{fail}\} \\ (\textsc{Demand}, \mathbf{P}) & \textit{otherwise, where } \mathbf{P} \textit{ is the set} \\ & \textit{of demanded positions} \end{cases}$$

Let us define a lazy narrowing strategy ($\varphi_{\blacktriangleright}$) as follows. Roughly speaking, in the following definition, the set-valued function $\varphi_{\blacktriangleright}(\mathbf{g})$ returns the set of triples $(\mathbf{u}, \mathbf{k}, \sigma)$ such that $\mathbf{u} \in \bar{\mathbf{O}}(\mathbf{g})$ is a demanded position of $\mathbf{g}$ which can be narrowed by the rule $\mathbf{r_k}$ with narrowing substitution σ. We assume the rules of $\mathcal{R}$ to be numbered with $\mathbf{r}_1, \ldots, \mathbf{r_m}$.

Definition 2.6 (lazy narrowing strategy) *We define a* lazy narrowing strategy $\varphi_{\blacktriangleright} : \mathbf{Goal} \to \wp(\mathbf{N}^* \times \mathbf{N} \times \mathbf{Sub})$ *which, for a given goal* $\mathbf{g} \equiv (\mathbf{e}_1, \ldots, \mathbf{e_n})$*, computes the set of triples* $\langle \mathbf{u}, \mathbf{k}, \sigma \rangle$*, with* $\mathbf{u} \in \bar{\mathbf{O}}(\mathbf{g})$*,* $\mathbf{k}$ *a natural number (indicating the program rule* $\mathbf{r_k}$*) and* σ *a substitution, as follows:*

$$\begin{array}{lcl} \varphi_{\blacktriangleright}(\mathbf{g}) & = & \varphi_{-}(\mathbf{g}, \mathbf{i}), \\ & & \quad \textit{where } \mathbf{i} = \mathbf{select\_don't\_care}(\{1, \ldots, \mathbf{n}\}) \\ \varphi_{-}(\mathbf{g}, \mathbf{u}) & = & \bigcup_{\mathbf{k}=1}^{\mathbf{m}} \varphi_{--}(\mathbf{g}, \mathbf{u}, \mathbf{k}) \\ \varphi_{--}(\mathbf{g}, \mathbf{u}, \mathbf{k}) & = & \textit{if } \lambda_{\mathbf{k}}[\Lambda] \equiv \mathbf{g}[\mathbf{u}] \textit{ then} \\ & & \quad \textit{case } \mathsf{LU}(\langle \lambda_{\mathbf{k}}, \mathbf{g}_{|\mathbf{u}} \rangle) \textit{ of} \\ & & \quad \begin{cases} (\textsc{Succ}, \sigma) : & \{(\mathbf{u}, \mathbf{k}, \sigma)\} \\ (\textsc{Fail}, \emptyset) : & \emptyset \\ (\textsc{Demand}, \mathbf{P}) : & \bigcup_{\mathbf{u}' \in \mathbf{P}} \varphi_{-}(\mathbf{g}, \mathbf{u}.\mathbf{u}') \end{cases} \\ & & \quad \textit{else } \emptyset \end{array}$$

where $\mathbf{r_k} \equiv (\lambda_{\mathbf{k}} \to \rho_{\mathbf{k}} \Leftarrow \mathbf{C_k}) \ll \mathcal{R}$ *and the "choice" function* $\mathbf{select\_don't\_care}(\mathbf{S})$ *arbitrarily chooses one element of the set* $\mathbf{S}$ *accordingly to some fixed criterium.*

Our lazy strategy is essentially equivalent to the demand driven reduction mechanism formalized in [39], where sequences of equations are interpreted as terms by defining the boolean conjunction operation ',' as a binary predefined symbol (whose rules are implicitly added to the program). As opposed to [39], our lazy narrowing calculus manipulates sequences of equations rather than terms, as in other lazy narrowing calculi (e.g. [21]).

We would like to note that there are three sources of nondeterminism in our narrowing calculus: the choice of the equation, given by $\mathbf{i}$, the choice of the subterm, given by $\mathbf{u}$, and the choice of the program rule, given by $\mathbf{k}$. The last two choices are don't-know nondeterministic, meaning that in general all possible choices have to be considered to ensure completeness. Only the choice of the equation is don't-care nondeterministic. In general, for the selected equation, the set of lazy redexes is not a singleton, and all such redexes have to be narrowed (trying to apply all program rules to each one) to guarantee completeness.

Definition 2.7 (lazy conditional narrowing)
We define lazy conditional narrowing *as a labeled transition system* $(\mathbf{Goal}, \mathbf{Sub}, \leadsto_{\blacktriangleright})$ *whose transition relation* $\leadsto_{\blacktriangleright} \subseteq (\mathbf{Goal} \times \mathbf{Sub} \times \mathbf{Goal})$ *is the smallest relation which satisfies:*

$$\frac{(\mathbf{u}, \mathbf{k}, \sigma) \in \varphi_{\blacktriangleright}(\mathbf{g}) \wedge \mathbf{r_k} \equiv (\lambda \rightarrow \rho \Leftarrow \mathbf{C}) \ll \mathcal{R}}{\mathbf{g} \overset{\sigma}{\leadsto}_{\blacktriangleright} (\mathbf{C}, \mathbf{g}[\rho]_{\mathbf{u}})\sigma}$$

Similarly to the formalization of [39], our calculus allows some narrowing derivations that do not contribute to any later steps. It is not difficult to strengthen the demand driven nature of our strategy by somehow forcing the use of the rule that demands evaluation of a suspended argument [18, 29, 35, 39]. [4] presents an optimal lazy narrowing strategy (for a restricted class of programs) which avoids superfluous steps by dropping the restriction to mgu's. Since these types of optimizations are not relevant to the subject of this paper, we do not consider them here for the sake of simplicity.

Due to the presence of nonterminating functions, completeness results for lazy narrowing are stated w.r.t. a non-standard interpretation of equality. Functional logic languages with a lazy narrowing operational semantics define the validity of an equation as a strict equality $\approx$ on terms. Strict equality regards two terms as equal iff they have the same ground constructor normal form. The semantics of $\approx$ is defined by the following set StrEq of confluent CB rules:
$\textsc{StrEq} = \{\mathbf{c}(\mathbf{x}_1, \ldots, \mathbf{x_n}) \approx \mathbf{c}(\mathbf{y}_1, \ldots, \mathbf{y_n}) \rightarrow \mathbf{true} \Leftarrow \mathbf{x}_1 \approx \mathbf{y}_1, \ldots, \mathbf{x_n} \approx \mathbf{y_n} \mid (\mathbf{c}/\mathbf{n}) \in \mathcal{C},\ \mathbf{n} \geq 0\}$.
Note that strict equality does not have the reflexivity property $\mathbf{t} \approx \mathbf{t}$ for all terms $\mathbf{t}$. When we consider lazy narrowing, we assume that the equality symbol in the goal (and in the conditions of program rules) is $\approx$.

In weakly orthogonal, CB programs, lazy narrowing is complete w.r.t. StrEq for constructor substitutions:

Proposition 2.8 [18, 39] *Let $\mathcal{R}$ be a weakly orthogonal, CB program, $\mathbf{g}$ be a goal and σ be a (ground constructor) substitution such that, for all $\mathbf{s} = \mathbf{t}$ in $\mathbf{g}$, there exists a ground constructor term $\mathbf{u}$ s.t. $\mathbf{s}\sigma \rightarrow^*_{\mathcal{R}} \mathbf{u}$ and $\mathbf{t}\sigma \rightarrow^*_{\mathcal{R}} \mathbf{u}$. Then, there is a c.a.s. θ of $(\mathcal{R} \cup \textsc{StrEq}) \cup \{\mathbf{g}\}$ using $\leadsto_{\blacktriangleright}$, and a substitution γ such that $\theta\gamma = \sigma[\mathbf{Var}(\mathbf{g})]$.*

In [17], Hanus showed how deterministic (lazy) simplification steps can be performed between nondeterministic (lazy) narrowing steps without loosing completeness. Roughly speaking, in the presence of nonterminating functions, only a *terminating subset* of the program rules is used for simplification[1]. Example 3 in Section 4 shows how the integration of simplification into (lazy) narrowing derivations is not only a main source for speedups but can also strengthen the specialization, with the added benefit that the optimization is 'compiled-in' in the program.

3 Partial Evaluation of Functional Logic Programs

In this section, we recall a generic procedure for the PE of functional logic programs which appeared in [2]. 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) an *abstraction operator* used to guarantee the finiteness of the PE process. We let $\leadsto_{\varphi}$ denote a generic (possibly normalizing) narrowing relation which uses the narrowing strategy φ. The definitions of this section are quoted from [2].

Definition 3.1 (resultant)
Let $\mathcal{R}$ be a program and $\mathbf{s} = \mathbf{y}$ an equation, where the variable $\mathbf{y} \notin \mathbf{Var}(\mathbf{s})$. Let $[(\mathbf{s} = \mathbf{y}) \overset{\theta}{\leadsto}{}^*_{\varphi} \mathbf{g}, \mathbf{e}]$ be a derivation in $\mathcal{R}$. Let $\sigma = \mathbf{mgu}(\mathbf{e})$. Then the resultant of the derivation is: $((\mathbf{s} \rightarrow \mathbf{y})\theta \Leftarrow \mathbf{g})\sigma$.

Example 2 *Let the rule $(\mathbf{f}(\mathbf{x}) \rightarrow \mathbf{x} \Leftarrow \mathbf{a} \approx \mathbf{x}, \mathbf{b} \approx \mathbf{x})$ be in $\mathcal{R}$. The resultant of the lazy narrowing derivation:*

1. $[\mathbf{f}(\mathbf{z}) \approx \mathbf{y} \overset{\{\mathbf{z}/\mathbf{x}\}}{\leadsto}_{\blacktriangleright} \mathbf{a} \approx \mathbf{x}, \mathbf{b} \approx \mathbf{x}, \mathbf{x} \approx \mathbf{y}]$ *is the rule:* $\mathbf{f}(\mathbf{y}) \rightarrow \mathbf{y} \Leftarrow \mathbf{a} \approx \mathbf{y}, \mathbf{b} \approx \mathbf{y}$.
Note that, without applying the mgu $\sigma = \{\mathbf{x}/\mathbf{y}\}$ ($\sigma = \{\mathbf{y}/\mathbf{x}\}$) of the last equation $\mathbf{x} \approx \mathbf{y}$, we would have obtained the rule: $\mathbf{f}(\mathbf{x}) \rightarrow \mathbf{y} \Leftarrow \mathbf{a} \approx \mathbf{x}, \mathbf{b} \approx \mathbf{x}, \mathbf{x} \approx \mathbf{y}$, which contains an extra-variable $\mathbf{y}$ in the rhs of the head.

2. $[\mathbf{f}(\mathbf{z}) \approx \mathbf{y} \overset{\{\mathbf{z}/\mathbf{x}\}}{\leadsto}_{\blacktriangleright} \mathbf{a} \approx \mathbf{x}, \mathbf{b} \approx \mathbf{x}, \mathbf{x} \approx \mathbf{y} \overset{\{\mathbf{x}/\mathbf{a}\}}{\leadsto}_{\blacktriangleright} \mathbf{true}, \mathbf{b} \approx \mathbf{a}, \mathbf{a} \approx \mathbf{y}]$ *is:* $\mathbf{f}(\mathbf{a}) \rightarrow \mathbf{a} \Leftarrow \mathbf{true}, \mathbf{b} \approx \mathbf{a}$.

A PE is derived by extracting rules from non–failing, root–to–leaf paths in a narrowing tree.

Definition 3.2 (partial evaluation of a term)
Let τ be a finite (possibly incomplete) narrowing tree for the goal $\mathbf{s} = \mathbf{y}$ ($\mathbf{y} \notin \mathbf{Var}(\mathbf{s})$) in the 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 τ and $\mathcal{R}' = \{\mathbf{r_i} \mid \mathbf{i} = 1, \ldots, \mathbf{k}\}$ the resultants associated with the derivations $\{(\mathbf{s} = \mathbf{y}) \overset{\sigma_i}{\leadsto}{}^+_{\varphi} \mathbf{g_i} \mid \mathbf{i} = 1, \ldots, \mathbf{k}\}$. Then, $\mathcal{R}'$ is a PE *of* $\mathbf{s}$ *in* $\mathcal{R}$ *(using τ).*

The above definition lifts in the natural way to partial evaluation of a set of terms $\mathbf{S}$ (which are considered modulo variants). 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 $\{\mathbf{s}_1, \mathbf{t}_1, \ldots, \mathbf{s_n}, \mathbf{t_n}\}$.

Following [34], we introduce a closedness condition which guarantees that all calls which might occur during the execution of the resulting program are covered by some program

[1] In the conditional case, the additional requirement for decreasing rules is needed [18].

rule. The function $\mathbf{terms(O)}$ extracts the terms appearing in the syntactic object $\mathbf{O}$.

Definition 3.3 (closedness) *Let* $\mathbf{S}$ *and* $\mathbf{T}$ *be two finite sets of terms. We say that* $\mathbf{T}$ *is* $\mathbf{S}$*-closed if* $\mathbf{closed(S,T)}$, *where the predicate* closed *is defined inductively as follows:*

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

We say that a program $\mathcal{R}$ *is* $\mathbf{S}$*-closed if* $\mathbf{closed(S,terms(}\mathcal{R}\mathbf{))}$.

Now we introduce an independence condition which guarantees that the derived program $\mathcal{R}'$ does not produce additional answers. In the case of pure logic programming, only (pairwise) non-unifiability of the partially evaluated calls (atoms) in $\mathbf{S}$ is required for ensuring the independence of the resultant procedures thus guaranteeing the strong correctness of the transformation. We need a more involved condition. Namely, we have to consider overlaps among the specialized calls (see [3]).

Definition 3.4 (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. If* $\mathbf{s} \equiv \mathbf{t}$*, we require that* $\mathbf{t}$ *be unifiable with a proper nonvariable subterm of* $\mathbf{s}$.

Definition 3.5 (independence) *A set of terms* $\mathbf{S}$ *is* independent *if there are no terms* $\mathbf{s}$ *and* $\mathbf{t}$ *in* $\mathbf{S}$ *such that* $\mathbf{s}$ *overlaps* $\mathbf{t}$.

Given a goal $\mathbf{g}$ and a program $\mathcal{R}$, in general, there exists an infinite number of different partial evaluations of $\mathbf{g}$ in $\mathcal{R}$. A fixed rule for generating resultants called an unfolding rule is used, which ensures that infinite unfolding is not attempted.

Definition 3.6 (unfolding rule) *An* unfolding rule $\mathbf{U}_\varphi$ *is a function which, when given a program* $\mathcal{R}$*, a term* $\mathbf{s}$ *and a narrowing 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* $\leadsto_\varphi$.

This definition lifts naturally to a set of terms $\mathbf{S}$.

Starting with the set of calls (terms) which appear in the initial goal $\mathbf{g}$, we partially evaluate them by using a finite unfolding strategy, and recursively specialize the terms which are introduced dynamically during this process. Assuming that it terminates, the procedure computes a set of partially evaluated terms $\mathbf{S}'$ and a set of rules $\mathcal{R}'$ (the PE of $\mathbf{S}'$ in $\mathcal{R}$) such that the closedness condition for $\mathcal{R}' \cup \{\mathbf{g}\}$ is satisfied.

We let $\mathbf{c[S]} \in \mathbf{State}$ 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 partially evaluated terms $\mathbf{S}$. When $\mathbf{S}$ is clear from the context, $\mathbf{c[S]}$ will simply be denoted by $\mathbf{c}$.

Definition 3.7 (PE transition relation $\longmapsto_\mathcal{P}$) *We define the PE relation* $\longmapsto_\mathcal{P} \subseteq \mathbf{State} \times \mathbf{State}$ *as the smallest relation satisfying:*

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

where the function $\mathbf{abstract(c,T)}$ *extends the current configuration* $\mathbf{c}$ *with (an abstraction of) the set of terms* $\mathbf{T}$ *which are not closed w.r.t.* $\mathbf{S}$*, giving a new PE configuration.*

Similarly to [36], applying *abstract* in every iteration allows us to tune the control of polyvariance as much as needed. Also, it is within the *abstract* operator that the progress towards termination resides.

Definition 3.8 (behaviour of the $\longmapsto_\mathcal{P}$ calculus) *Let* $\mathbf{c}_0$ *be the "empty" PE state. We define the function:* $\mathcal{P}(\mathcal{R},\mathbf{g}) = \mathbf{S}$ *if* $\mathbf{abstract(c_0,terms(g))} \longmapsto^*_\mathcal{P} \mathbf{c[S]}$ *and* $\mathbf{c[S]} \longmapsto_\mathcal{P} \mathbf{c[S]}$.

The PE procedure in Definition 3.8 computes the set of partially evaluated terms $\mathbf{S}$ which unambiguously determines its associated PE $\mathcal{R}'$ in $\mathcal{R}$ (using $\mathbf{U}_\varphi$).

4 A Call-by-Name Partial Evaluator

In this section, we formulate an instance of the generic PE procedure introduced in Definition 3.8 and show how it works. We consider the *(normalizing) lazy conditional narrowing* $\leadsto_\blacktriangleright$ of Section 2 to construct search trees. We define a hnf-PE $\mathcal{R}'$ of $\mathbf{s}$ in $\mathcal{R}$ as a PE of $\mathbf{s}$ in $\mathcal{R}$ such that each derivation used to build $\mathcal{R}'$ has the form $[(\mathbf{s} \approx \mathbf{y}) \overset{\theta}{\leadsto}{}^*_\blacktriangleright \mathbf{g}, (\mathbf{s}' \approx \mathbf{y})]$, where the equality symbol of the original equation is not narrowed using the rules StrEq. This restriction avoids the term $\mathbf{s}$ to be evaluated beyond the form possibly demanded by a context.

In [2], we have developed a rather simple criterium for avoiding looping. Our strategy is based on the intuitive notion of orderings in which a term that is "syntactically simpler" than another is smaller than the other. The following definition extends the homeomorphic embedding ("syntactically simpler") relation [10] to nonground terms. In [31], an extension of this relation is defined which provides a finer handle of variables.

Definition 4.1 (embedding relation [43]**)** *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* $\mathbf{s} \equiv \mathbf{f(s_1,\ldots,s_m)} \trianglelefteq \mathbf{g(t_1,\ldots,t_n)} \equiv \mathbf{t}$*, 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* $\mathbf{j}$*,* $1 \leq \mathbf{j} \leq \mathbf{n}$.

The embedding relation $\trianglelefteq$ is a well-quasi ordering of the set $\tau(\Sigma \cup \mathbf{V})$ for finite Σ [2, 28], that is, any infinite sequence of terms $\mathbf{t}_1, \mathbf{t}_2, \ldots$ with a finite number of operators is self-embedding, i.e., there are numbers $\mathbf{j},\mathbf{k}$ with $\mathbf{j} < \mathbf{k}$ and $\mathbf{t_j} \trianglelefteq \mathbf{t_k}$. 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. When the compared calls are in the embedding relation, the derivation is stopped. We consider here this *nonembedding unfolding rule* (instantiated with the lazy calculus relation) $\mathbf{U}^\trianglelefteq_{\leadsto_\blacktriangleright}$ to control the expansion of the trees.

We define a PE$^\trianglelefteq$ *configuration* as a sequence of terms $(\mathbf{t}_1,\ldots,\mathbf{t_n}) \in \mathbf{State}^\trianglelefteq$. Upon each iteration of the algorithm in Definition 3.7, the current configuration $\mathbf{q} \equiv (\mathbf{t}_1, \ldots, \mathbf{t_n})$ is transformed in order to 'cover' the set $\mathbf{T}$ of terms which result from the PE of $\mathbf{q}$ in $\mathcal{R}$, that is:

$$\mathbf{T} = \mathbf{terms}(\mathbf{U}^{\trianglelefteq}_{\leadsto_\blacktriangleright}(\{\mathbf{t}_1, \ldots, \mathbf{t_n}\}, \mathcal{R})).$$

This transformation is done using a specific abstraction operation $\mathbf{abstract}^{\trianglelefteq}$ [2]. Informally, this operator does the following. Let $\mathbf{T}$ be the set of new terms introduced by the unfolding. They are compared to those already generated (recorded in $\mathbf{q}$) and, if the new term is not larger than any of the preceding ones, it is just appended to $\mathbf{q}$. Otherwise, if the new term $\mathbf{s}$ is an instance of some term $\mathbf{t} = \mathbf{s}\sigma$ in $\mathbf{q}$, then the terms in σ are recursively abstracted and introduced in $\mathbf{q}$; else the two terms are slightly generalized by taking their most specific generalization [30]. The abstraction operator $\mathbf{abstract}^{\trianglelefteq}$ guarantees the global termination of the algorithm and the closedness of the partially evaluated program.

Our first example, the 'double-append', is a standard test of elimination of intermediate data structures. It illustrates the fact that our method can eliminate intermediate data structures and turn multiple–pass programs into one–pass programs. This effect is also achieved by deforestation [47], tupling [9] and positive supercompilation [42], among others, while standard PE generally does not get this optimization [44]. This example also shows that normalization between narrowing steps can be used as a safe replacement for some ad-hoc optimizations used in other methodologies for program transformation. This normalization does not risk looping, because by using a terminating subset of the program rules for normalization, each single narrowing step terminates and thus narrowing trees are built in finite time. It does not lose completeness either, because alternative clauses are discarded only if no solutions are lost.

Example 3 (double-append) Consider the well-known, terminating, program *app/2:*

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

with initial query $\mathbf{app}(\mathbf{app}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}) \approx \mathbf{y}$. 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 lazy narrowing. Starting with the sequence: $\mathbf{q} = \mathbf{app}(\mathbf{app}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s})$, and by using the procedure described in Definition 3.8, we compute the trees depicted in Figure 3 for the sequence of terms: $\mathbf{q}' = \mathbf{app}(\mathbf{app}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}), \mathbf{app}(\mathbf{x_s}, \mathbf{y_s})$. Note that *"app"* has been abbreviated to *"a"* in the picture. Then we get the following residual program $\mathcal{R}'$*:*

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

which is able to append the three lists by traversing its input only once. Note that the key to success in this example is the use of normalization. Without the simplification step, the embedding ordering would have been satisfied too early in the rightmost branch of the top tree of Figure 3. The driving algorithm in [43] (the only version of positive supercompilation guaranteeing termination) achieves the same effect by means of an ad-hoc technique like the 'transient reductions' [43], which can incur the risk of nontermination, as opposed to our method.

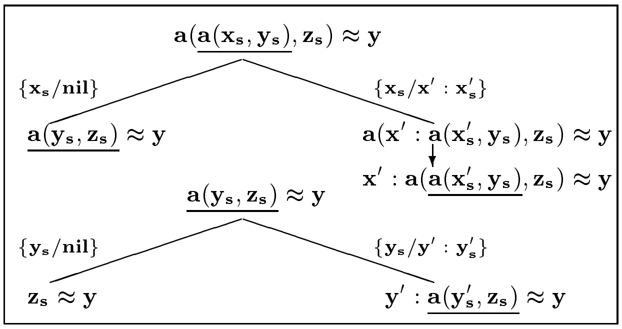

Figure 1: Normalizing lazy narrowing trees for $\mathbf{a}(\mathbf{a}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}) \approx \mathbf{y}$ and $\mathbf{a}(\mathbf{x_s}, \mathbf{y_s}) \approx \mathbf{y}$.

In [31], *characteristic trees* have been proposed as an alternative to transient reductions which ensure termination.

We note that the normalization of goals implements a strategy where we compute in a deterministic way as long as possible, and it is thus comparable to Gallagher's preference for *unfolding determinate goals* [12, 43] in that it avoids the creation of superfluous choice points for alternative rules but it requires a sort of "looking-ahead" in the search trees.

Example 3 illustrates the fact that the resulting partially evaluated program is not guaranteed to be CB, which may prevent the lazy strategy from being able to narrow a goal in the transformed, partially evaluated program. Example 3 also shows that the resulting set of (partially evaluated) terms is not guaranteed to be independent. In our example, PE uses the same function symbol for two different specializations of a definition (namely, for the procedures $\mathbf{app}(\mathbf{app}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s})$ and $\mathbf{app}(\mathbf{x_s}, \mathbf{y_s})$). Some type of transformation is required to guarantee that there is no interference between the corresponding sets of rules, as would be the case if the definition of 'double-append' were used to narrow a nested call $\mathbf{app}(\_, \_)$ in a goal $\mathbf{app}(\underline{\mathbf{app}(\_, \_)}, \_)$.

4.1 Post-Processing Renaming

In this section, we formalize a suitable *renaming* phase able to guarantee that lazy narrowing can execute the goal in the transformed program and that different specializations of a function are not confused, while the (lazy) computed answer semantics is preserved.

Definition 4.2 (independent renaming) *Let* $\mathbf{S}$ *be a set of terms. We define an independent renaming* $\mathbf{S}'$ *of* $\mathbf{S}$ *as follows:*

$$\mathbf{S}' = \{\langle \mathbf{s}, \mathbf{s}' \rangle \mid \mathbf{s} \in \mathbf{S} \wedge \mathbf{s}' = \mathbf{f}^{\mathbf{s}}(\mathbf{x}_1, \ldots, \mathbf{x_m})\}$$

where $\{\mathbf{x}_1, \ldots, \mathbf{x_m}\}$ *are the distinct variables in* $\mathbf{s}$ *in the order of their first occurrence, and the* $\mathbf{f}^{\mathbf{s}}$*'s are new function symbols, which are different from those in* $\mathcal{R}$ *and* $\mathbf{S}$*.*

The post-processing renaming can be formally defined as follows.

Definition 4.3 (post-processing renaming) *Let* $\mathcal{R}$ *be a program and* $\mathbf{S}$ *a set of terms. Let* $\mathcal{R}'$ *be a partial evaluation of* $\mathcal{R}$ *w.r.t.* $\mathbf{S}$*, and* $\mathbf{S}'$ *an independent renaming of* $\mathbf{S}$*. We define the post-processing renaming* $\mathcal{R}'' = \mathbf{ppren}_\blacktriangleright(\mathcal{R}', \mathbf{S}')$

of $\mathcal{R}'$ *w.r.t.* $\mathbf{S}'$ *as follows:* $\mathbf{ppren}_{\blacktriangleright}(\mathcal{R}', \mathbf{S}') =$

$$\bigcup_{\langle \mathbf{s},\mathbf{s}'\rangle \in \mathbf{S}'} \{\mathbf{r}' \mid \quad (\mathbf{s}\theta \to \rho \Leftarrow \mathbf{C}) \in \mathcal{R}', \text{ and } \mathbf{r}' \equiv (\mathbf{s}'\theta \to \mathbf{ren}(\rho, \mathbf{S}') \Leftarrow \mathbf{ren}(\mathbf{C}, \mathbf{S}'))\}$$

where the nondeterministic function $\mathbf{ren}(\mathbf{o}, \mathbf{S}')$ *is defined inductively as follows:* $\mathbf{ren}(\mathbf{o}, \mathbf{S}') =$

$$\begin{cases} \mathbf{ren}(\mathbf{e}_1, \mathbf{S}'), \ldots, \mathbf{ren}(\mathbf{e_n}, \mathbf{S}') & \text{if } \mathbf{o} \equiv (\mathbf{e}_1, \ldots, \mathbf{e_n}) \\ \mathbf{ren}(\mathbf{s}, \mathbf{S}') \approx \mathbf{ren}(\mathbf{t}, \mathbf{S}') & \text{if } \mathbf{o} \equiv (\mathbf{s} \approx \mathbf{t}) \\ \mathbf{x} & \text{if } \mathbf{o} \equiv \mathbf{x} \in \mathbf{V} \\ \mathbf{c}(\mathbf{ren}(\mathbf{t}_1, \mathbf{S}'), \ldots, \mathbf{ren}(\mathbf{t_n}, \mathbf{S}')) & \text{if } \mathbf{o} \equiv \mathbf{c}(\mathbf{t}_1, \ldots, \mathbf{t_n}),\ \mathbf{c} \in \mathcal{C},\ \mathbf{n} \geq 0 \\ \mathbf{s}'\theta' & \text{if } \mathbf{o} = \mathbf{s}\theta,\ \langle \mathbf{s}, \mathbf{s}'\rangle \in \mathbf{S}', \text{and } \theta' = \{(\mathbf{x}/\mathbf{ren}(\mathbf{x}\theta, \mathbf{S}')) \mid \mathbf{x} \in \mathbf{Dom}(\theta)\}. \end{cases}$$

Roughly speaking, in Definition 4.3 we derive specialized procedures for each term in $\mathbf{S}'$, and perform fold on every function call in $\mathcal{R}'$ (replacing the original term by a call to the newly defined function) using the corresponding renaming in $\mathbf{S}'$, to produce the new, renamed, filtered program $\mathcal{R}''$. The idea behind this transformation is that, for any $\mathbf{S}$-closed query $\mathbf{g}$, lazy narrowing computes the same answers for $\mathbf{g}$ in $\mathcal{R}$ as for the goal which results from the renaming of $\mathbf{g}$ (according to $\mathbf{S}'$) in $\mathcal{R}''$. Note that the postunfolding process terminates.

We now illustrate these definitions with an example.

Example 4 *Consider again the double-append goal and program app/2 of Example 3. An independent renaming* $\mathbf{S}'$ *of* $\mathbf{S}$ *is:*

$$\mathbf{S}' = \{ \quad \langle \mathbf{app}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{a}_1(\mathbf{x_s}, \mathbf{y_s})\rangle, \langle \mathbf{app}(\mathbf{app}(\mathbf{x_s}, \mathbf{y_s}), \mathbf{z_s}), \mathbf{a}_2(\mathbf{x_s}, \mathbf{y_s}, \mathbf{z_s})\rangle \ \}.$$

The post-processing renaming $\mathcal{R}''$ *of* $\mathcal{R}'$ *w.r.t.* $\mathbf{S}'$ *is:*

$$\begin{array}{rcl} \mathbf{a}_2(\mathbf{nil}, \mathbf{y_s}, \mathbf{z_s}) & \to & \mathbf{a}_1(\mathbf{y_s}, \mathbf{z_s}) \\ \mathbf{a}_2(\mathbf{x} : \mathbf{x_s}, \mathbf{y_s}, \mathbf{z_s}) & \to & \mathbf{x} : \mathbf{a}_2(\mathbf{x_s}, \mathbf{y_s}, \mathbf{z_s}) \\ \mathbf{a}_1(\mathbf{nil}, \mathbf{y_s}) & \to & \mathbf{y_s} \\ \mathbf{a}_1(\mathbf{x} : \mathbf{x_s}, \mathbf{y_s}) & \to & \mathbf{x} : \mathbf{a}_1(\mathbf{x_s}, \mathbf{y_s}) \end{array}$$

We note that, for a given set of terms $\mathbf{S}'$, the filtered form of a program may depend on the strategy which selects the term from $\mathbf{S}'$ which is used to rename a given term $\mathbf{o}$ in $\mathbf{S}$, since there may exist, in general, more than one $\mathbf{s}$ in $\mathbf{S}'$ that covers the call $\mathbf{o}$. Hence, the specialized form of a program is not unique. Some potential specialization might be lost due to an inconvenient choice. The problem of defining some plausible heuristics able to produce the better potential specialization is still pending research.

Renaming ensures independence of the specialized procedures, as stated in the following.

Proposition 4.4 (independence) *Let* $\mathcal{R}$ *be a weakly orthogonal, CB program,* $\mathbf{S}$ *be a finite set of terms, and* $\mathbf{g}$ *be an* $\mathbf{S}$*-closed goal. Let* $\mathcal{R}'$ *be a hnf-PE of* $\mathcal{R}$ *w.r.t.* $\mathbf{S}$ *such that* $\mathcal{R}'$ *is* $\mathbf{S}$*-closed. Let* $\mathbf{S}'$ *be an independent renaming of* $\mathbf{S}$*,* $\mathcal{R}'' = \mathbf{ppren}_{\blacktriangleright}(\mathcal{R}', \mathbf{S}')$ *the renaming of* $\mathcal{R}'$ *w.r.t.* $\mathbf{S}'$*, and* $\mathbf{g}'' = \mathbf{ren}(\mathbf{g}, \mathbf{S}')$ *the renaming of the goal* $\mathbf{g}$ *w.r.t.* $\mathbf{S}'$*. Then,*

1. $\mathcal{A} \equiv \{\mathbf{s}' \mid \langle \mathbf{s}, \mathbf{s}'\rangle \in \mathbf{S}'\}$ *is independent,*
2. $\mathcal{R}''$ *is a left linear CB program, and*
3. $\mathcal{R}'' \cup \{\mathbf{g}''\}$ *is* $\mathcal{A}$*-closed.*

We now state the correctness of the partial evaluator with the post-processing renaming.

Theorem 4.5 (strong soundness and completeness) *Let* $\mathcal{R}$ *be a weakly orthogonal, CB program,* $\mathbf{g}$ *a goal, and* $\mathbf{S}$ *be a finite set of terms. Let* $\mathcal{R}'$ *be a hnf-PE of* $\mathcal{R}$ *w.r.t.* $\mathbf{S}$ *such that* $\mathcal{R}' \cup \{\mathbf{g}\}$ *is* $\mathbf{S}$*-closed. Let* $\mathbf{S}'$ *be an independent renaming of* $\mathbf{S}$*,* $\mathcal{R}''$ *be a renaming of* $\mathcal{R}'$ *w.r.t.* $\mathbf{S}'$*, and* $\mathbf{g}'' = \mathbf{ren}(\mathbf{g}, \mathbf{S}')$*. Then* θ *is a computed answer substitution for* $\mathbf{g}$ *in* $\mathcal{R}$ *iff* θ *is a computed answer substitution for* $\mathbf{g}''$ *in* $\mathcal{R}''$*.*

We finally illustrate the power of the call-by-name PE procedure on the matching program **match** of [27]. This example is discussed by [24, 43, 44], among others.

4.2 Pattern matching in strings

A standard example in the literature on PE is the derivation of an efficient string matcher by PE of a (more or less) naïve pattern matcher w.r.t. a given pattern [15, 44]. The source program $\mathcal{R}$ listed below checks whether a string pattern $\mathbf{p}$ occurs within another string $\mathbf{s}$ by iteratively comparing $\mathbf{p}$ with a prefix of $\mathbf{s}$. In the case of a mismatch, the first element of the target string $\mathbf{s}$ is cut off and the process is restarted with the tail of $\mathbf{s}$. 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) PE nor deforestation can perform automatically [44]. Partial deduction of logic programs and positive supercompilation of functional programs can pass the test [44]. Our method also performs satisfactorily on the problem, as the following example illustrates. We assume that matching is on bit-strings, i.e. strings containing only zeroes and ones. Note that this choice of a binary (or finite) alphabet is sensible for an equational definition of the *false* value for equalities, but it is not essential for the specialization, since the algorithm does not rely on it and is not exploited during the specialization process.

Example 5 *Let* $\mathcal{R}$ *be the naïve pattern matching program* (a) *of Figure 2. 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-name evaluator to the term* $\mathbf{match}(001, \mathbf{s})$*, and subsequently evaluating new terms according to our method, gives the program* $\mathcal{R}'$[2] *(Figure 2,* (b)*). After the post-processing renaming transformation, we obtain the specialized program* $\mathcal{R}''$ *(Figure 2,* (c)*). The amount of specialization obtained in this program is essentially analogous to that of the rules produced by the algorithm in [43]. The specialized algorithm acts like a KMP-style pattern matcher. This gives the same advantages of the KMP algorithm over the naïve one in terms of complexity. Namely, for each fixed pattern* $\mathbf{p}$ *the specialized algorithm makes much less comparisons, furthermore as* $\mathbf{p}$ *gets larger the complexity of the naïve algorithm becomes larger, while for the specialized algorithm there exists an upper bound which does not depend on* $\mathbf{p}$*.*

[2]For simplicity, we have omitted the rules that reduce functions to false. We have used the simplification and eager variable elimination [38] rules in order to get better specialization.

(a) Naïve pattern matcher $\mathcal{R}$:

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

(b) CBN PE $\mathcal{R}'$ of $\mathbf{match}(001, \mathrm{s})$ in $\mathcal{R}$:

$$
\begin{array}{rcl}
\mathbf{match}(001, \mathrm{s}) & \rightarrow & \mathbf{loop}(001, \mathrm{s}, 001, \mathrm{s}) \\
\mathbf{loop}(001, 0 : \mathbf{ss}, 001, 0 : \mathbf{ss}) & \rightarrow & \mathbf{loop}(01, \mathbf{ss}, 001, 0 : \mathbf{ss}) \\
\mathbf{loop}(001, \mathbf{s} : \mathbf{ss}, 001, \mathbf{s} : \mathbf{ss}) & \rightarrow & \mathbf{loop}(001, \mathbf{ss}, 001, \mathbf{ss}) \Leftarrow (0 \approx \mathrm{s}) \approx \mathbf{false} \\
\mathbf{loop}(01, 0 : \mathbf{ss}', 001, 00 : \mathbf{ss}') & \rightarrow & \mathbf{loop}(1, \mathbf{ss}', 001, 00 : \mathbf{ss}') \\
\mathbf{loop}(01, \mathbf{s}' : \mathbf{ss}', 001, 0 : \mathbf{s}' : \mathbf{ss}') & \rightarrow & \mathbf{loop}(001, \mathbf{ss}', 001, \mathbf{s}' : \mathbf{ss}') \Leftarrow (0 \approx \mathrm{s}') \approx \mathbf{false} \\
\mathbf{loop}(1, 1 : \mathbf{ss}'', 001, 001 : \mathbf{ss}'') & \rightarrow & \mathbf{true} \\
\mathbf{loop}(1, \mathbf{s}'' : \mathbf{ss}'', 001, 00 : \mathbf{s}'' : \mathbf{ss}'') & \rightarrow & \mathbf{loop}(01, \mathbf{s}'' : \mathbf{ss}'', 001, 0 : \mathbf{s}'' : \mathbf{ss}'') \Leftarrow (1 \approx \mathrm{s}'') \approx \mathbf{false}
\end{array}
$$

(c) Post-processing renaming $\mathcal{R}''$ of $\mathcal{R}'$:

$$
\begin{array}{lcllcl}
\mathbf{match}'(\mathrm{s}) & \rightarrow & \mathbf{loop\_001}(\mathrm{s}) & & & \\
\mathbf{loop\_001}(0 : \mathbf{ss}) & \rightarrow & \mathbf{loop\_01}(\mathbf{ss}) & \mathbf{loop\_001}(\mathrm{s} : \mathbf{ss}) & \rightarrow & \mathbf{loop\_001}(\mathbf{ss}) \Leftarrow (0 \approx \mathrm{s}) \approx \mathbf{false} \\
\mathbf{loop\_01}(0 : \mathbf{ss}) & \rightarrow & \mathbf{loop\_1}(\mathbf{ss}) & \mathbf{loop\_01}(\mathrm{s} : \mathbf{ss}) & \rightarrow & \mathbf{loop\_001}(\mathbf{ss}) \Leftarrow (0 \approx \mathrm{s}) \approx \mathbf{false} \\
\mathbf{loop\_1}(1 : \mathbf{ss}) & \rightarrow & \mathbf{true} & \mathbf{loop\_1}(\mathrm{s} : \mathbf{ss}) & \rightarrow & \mathbf{loop\_01}(\mathrm{s} : \mathbf{ss}) \Leftarrow (1 \approx \mathrm{s}) \approx \mathbf{false}
\end{array}
$$

Figure 2: KMP example.

Let us conclude with an example that illustrates the fact that, in some cases, a call-by-name partial evaluator can derive more efficient residual programs than a call-by-value partial evaluator and hence it can be worthwhile constructing them, even for terminating programs. We note that most of the actual partial evaluators are based on call-by-value procedures.

Example 6 *Consider the following program* $\mathcal{R} \equiv \{\mathbf{g}(\mathbf{x}) \rightarrow 0,\ \mathbf{f}(0) \rightarrow 0,\ \mathbf{f}(\mathbf{c}(\mathbf{x})) \rightarrow \mathbf{f}(\mathbf{x})\}$. *Let us consider a call-by-value partial evaluation based on innermost narrowing [11]. If we evaluate the goal* $\mathbf{g}(\mathbf{f}(\mathbf{x})) = \mathbf{y}$ *in* $\mathcal{R}$ *using innermost narrowing, we obtain the following renamed specialized program* $\mathcal{R}_1'' \equiv \{\mathbf{g}'(0) \rightarrow 0,\ \mathbf{g}'(\mathbf{c}(\mathbf{x})) \rightarrow \mathbf{g}'(\mathbf{x})\}$. *The partial evaluator based on our lazy narrowing strategy produces the following residual program* $\mathcal{R}_2'' \equiv \{\mathbf{g}'(\mathbf{x}) \rightarrow 0\}$, *whose complexity is* $\mathbf{O}(1)$, *whereas the complexity of* $\mathcal{R}_1''$ *is* $\mathbf{O}(\mathbf{n})$ *(where* $\mathbf{n}$ *is the number of* 0*'s in the input argument).*

5 Related Work

Early work on automatic specialization of functional programs includes Turchin's supercompiler [45] and the positive supercompiler [44]. Supercompilation is based on *driving*, a unification-based (call-by-name) transformation technique for a strict (call-by-value) functional language tailored for supercompilation. The mechanism of driving covers the activities of specializing and unfolding in PE. The positive supercompiler is a reformulation of Turchin's supercompiler for a simpler call-by-name language with tree-structured patterns. In this language, all arguments of a function are input parameters. Furthermore, there is only pattern-matching for non-nested linear patterns. The formulation is generalizable to less restrictive languages at the cost of more complex driving algorithms. In our functional logic language, this is not the case, since we allow for more powerful matching via unification on all arguments, both in execution and during specialization.

In positive supercompilation [15], local and global control are not (explicitly) distinguished as only one-step unfolding is performed. A large evaluation structure is built which comprises something similar to both the local narrowing trees and the global configurations of [36]. Each call $\mathbf{t}$ in the driving process graph produces a residual function. The body of the definition of the new function is derived from the descendants of $\mathbf{t}$ in the graph. As in Partial Deduction, the body of each residual rule in our framework is built from the final node of the derivation, which results in a fewer number of rules [15].

The post-processing renaming we have introduced in this paper can be seen as an extension of the post-processing renaming defined in [5] to the case of nested functions in expressions. For simple patterns, our post-unfolding renaming essentially boils down to the simpler transformation of [5].

A more closely related approach can be found in [32], where conjunctions of calls (atoms) can appear in the heads of partially evaluated programs, which is somehow comparable to nested function applications. As in our framework, in [32] renaming is mandatory in order to derive an executable specialized program from the partially evaluated one. There exist two main differences between our post-processing renaming and the renaming function introduced in [32]. First, their definition describes renaming functions which can *erase* variables of the renamed calls. As noted in [32], this approach can produce incorrect transformations. A safe technique has been later introduced in [33]. The second difference is in the renaming of clause bodies. As we pointed out before, our renaming transformation is nondeterministic, and thus the function **ren** of Definition 4.3 can produce different outcomes. Similarly, the renaming transformation of [32] involves some nondeterminism concerning the selection of conjunctions from the clause bodies to be renamed. However, they introduce a *partitioning* function in order to deterministically choose one of the possible renamings. This extension is also possible in our framework, by simply adding a new parameter to the renaming function **ren** to indicate how the specialized calls have been proved **S**-closed, which might be used to guide the renaming process.

6 Conclusions

The interest in combining the most important declarative paradigms, namely functional and logic programming, has grown over the last decade (see [18] for a survey). However, integrated functional logic languages are currently not widely used. In order to develop useful and practical integrated languages, it is essential to succeed in shrinking the efficiency gap with respect to imperative languages, as is already being done for Prolog. To this goal, formally based, practical tools for the analysis and transformation of functional logic programs which are able to improve the current implementations are a pressing need. Since functional logic languages have been widely investigated from the point of view of the semantics, it becomes of natural concern to study formal manipulation techniques based on the semantics, which are able to improve program performances without changing the computational meaning.

PE is a semantics-preserving program transformation based on unfolding and specializing procedures. In this paper we have considered the case of (normalizing) lazy narrowing, which has been shown to be a reasonable improvement over pure logic SLD resolution strategy [17]. The main innovations in our work are: 1) our procedure applies to (lazy) functional logic languages such as Babel whose (lazy narrowing) operational semantics corresponds to SLD resolution but with the additional feature of exploiting determinism by the 'dynamic cut' [18], and 2) we present a renaming transformation for guaranteeing: (a) the independence of the set of partially evaluated terms, (b) that the partially evaluated program does not lose some of the basic requirements for the completeness of lazy narrowing and (c) the equivalence of the computed answer substitution semantics of the original and the partially evaluated programs for the intended queries.

We note that our post-processing transformation is necessary also for some non-lazy strategies, such as innermost narrowing, and the extension results straightforward [1].

As future work we mention the investigation of the application of our framework to optimal versions of lazy narrowing strategies, such as needed narrowing [4] (and its extension to a higher order framework) which has been proposed as the basic operational principle of Curry [20], a language which is intended to become a standard in this area.

Acknowledgements

We thank Morten Sørensen for providing us with very valuable feedback on a draft of the paper.

References

[1] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Call-by-Name Partial Evaluation of Functional Logic Programs. TR DSIC-II/34/96, U.P. Valencia, 1996.

[2] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. Riis Nielson, editor, *Proc. of the 6th European Symp. on Programming, ESOP'96*, pages 45–61. Springer LNCS 1058, 1996.

[3] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Specialization of Functional Logic Programs. TR DSIC-II/33/96, U.P. Valencia. Extended and revised version of [2]. Available from URL http://www.dsic.upv.es/users/elp/papers.html, 1996.

[4] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.

[5] 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.

[6] J.A. Bergstra and J.W. Klop. Conditional Rewrite Rules: confluence and termination. *Journal of Computer and System Sciences*, 32:323–362, 1986.

[7] 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.

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

[9] W. Chin. Towards an Automated Tupling Strategy. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 119–132. ACM, New York, 1993.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42, 1991.

[15] 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.

[16] 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.

[17] 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.

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

[19] M. Hanus. On Extra Variables in (Equational) Logic Programming. In *Proc. of 20th Int'l Conf. on Logic Programming*, pages 665–678. The MIT Press, Cambridge, MA, 1995.

[20] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

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

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

[23] H. Hussman. Unification in Conditional-Equational Theories. In *Proc. European Conf. on Computer Algebra EUROCAL'85*, pages 543–553. Springer LNCS 204, 1985.

[24] 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.

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

[26] 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.

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

[28] J. B. Kruskal. The Theory of Well-Quasi-Ordering: A Frequently Discovered Concept. *Journal of Combinatorial Theory*, A(13):297–305, 1972.

[29] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. of the Int'l Conf. on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317, 1990.

[30] 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.

[31] M. Leuschel and B. Martens. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. Technical Report CW-220, Department of Computer Science, K.U. Leuven, Belgium, December 1995.

[32] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.

[33] M. Leuschel and M.H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proc. of Int'l Workshop on Logic Programming Synthesis and Transformation, LOPSTR'96*. To appear in Springer LNCS, 1996.

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

[35] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93, Tallinn (Estonia)*, pages 184–200. Springer LNCS 714, 1993.

[36] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.

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

[38] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. In *Proc. of the Fuji Int'l Workshop on Functional and Logic Programming*, pages 104–118. World Scientific, 1995.

[39] J.J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[40] 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.

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

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

[43] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In J.W. Lloyd, editor, *Proc. of ILPS'95*, pages 465–479. The MIT Press, Cambridge, MA, 1995.

[44] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 1996. To appear.

[45] 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.

[46] 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.

[47] 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.