Specialization of Functional Logic Programs Based on Needed Narrowing

María Alpuente *Michael Hanus*
Salvador Lucas *Germán Vidal*

Aachener Informatik-Bericht
99-4

Specialization of Functional Logic Programs Based on Needed Narrowing*

María Alpuente[†] Michael Hanus [‡] Salvador Lucas[†] Germán Vidal[†]

Abstract

Functional logic languages 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. Needed narrowing is an optimal narrowing strategy and the basis of several recent functional logic languages. In this paper, we define a partial evaluator for functional logic programs based on needed narrowing. We prove strong correctness of this partial evaluator and show that the nice properties of needed narrowing carry over to the specialization process and the specialized programs. In particular, the structure of the specialized programs provides for the application of optimal evaluation strategies. This is in contrast to other partial evaluation methods for functional logic programs which can change the original program structure in a negative way. Finally, we present some experiments which highlight the practical advantages of our approach.

1 Introduction

Functional logic languages combine the operational principles of the most important declarative programming paradigms, namely functional and logic programming. Efficient demand-driven functional computations are amalgamated with the flexible use of logical variables providing for function inversion and search for solutions. The operational semantics of such languages is usually based on narrowing, a generalization of term rewriting which combines reduction and variable instantiation. A *narrowing step* instantiates variables of an expression and applies a reduction step to a redex of the instantiated expression. The instantiation of variables is usually computed by unifying a subterm of the entire expression with the left-hand side of some rule.

Example 1 *Consider the following rules which define the less-or-equal predicate "$\leqslant$" on natural numbers which are represented by terms built from* `0` *and* `s` *(note that variable names*

*This work has been partially supported by CICYT TIC 98-0445-C03-01, by Acción Integrada hispano–alemana HA1997-0073, and by the German Research Council (DFG) under grant Ha 2457/1-1.

†DSIC, UPV, Camino de Vera s/n, 46020 Valencia, Spain. `{alpuente,slucas,gvidal}@dsic.upv.es`

‡Informatik II, RWTH Aachen, D-52056, Germany. `hanus@informatik.rwth-aachen.de`

always start with an uppercase letter):

$$
\begin{array}{rcll}
\texttt{0} & \leqslant & \texttt{N} & \rightarrow \ \texttt{true} \\
\texttt{s(M)} & \leqslant & \texttt{0} & \rightarrow \ \texttt{false} \\
\texttt{s(M)} & \leqslant & \texttt{s(N)} & \rightarrow \ \texttt{M} \leqslant \texttt{N}
\end{array}
$$

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

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

Narrowing provides completeness in the sense of logic programming (computation of all solutions) as well as functional programming (computation of values). Since simple narrowing can have a huge search space, great effort has been made to develop sophisticated narrowing strategies without losing completeness (see [25] for a survey). To avoid unnecessary computations and to provide computations with infinite data structures as well as a demand-driven generation of the search space, the most recent work has advocated *lazy narrowing strategies* (e.g., [10, 22, 40, 44]). *Needed narrowing* [10] is based on the idea of evaluating only subterms which are *needed* in order to compute a result. For instance, in a term like $t_1 \leqslant t_2$, it is always necessary to evaluate t_1 (to some *head normal form*) since all three rules in Example 1 have a non-variable first argument. On the other hand, the evaluation of t_2 is only needed if t_1 is of the form $\texttt{s}(\Box)$. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor, here `0` or $\texttt{s}(\Box)$. Depending on this instantiation, either the first rule is applied or the second argument t_2 is evaluated. Needed narrowing is currently the best narrowing strategy for first-order functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions [10] and it can be efficiently implemented by pattern matching and unification (e.g., [26, 40]). Moreover, it has recently been extended to higher-order functions and λ-terms as data structures and proved optimal w.r.t. independence of computed solutions [29].

Partial evaluation (PE) is a semantics-preserving performance optimization technique for computer programs which consists of the specialization of the program w.r.t. parts of its input. PE has been widely applied in the fields of term rewriting systems [13, 14, 15, 20, 35, 43], functional programming [17, 32], and logic programming [21, 39]. Although the objectives are similar, the general methods are often different due to the distinct underlying models and the different perspectives (see [5] for a detailed comparison). This separation has the negative consequence of duplicated work since developments are not shared and many similarities are overlooked. A unified (narrowing-based) treatment can bring the different methodologies closer and lays the ground for new insights in all three fields [5, 6, 23, 46, 48].

To perform reductions at specialization time, a partial evaluator normally includes an interpreter [17, 24]. This implies that the power of the transformation is highly influenced by the properties of the evaluation strategy from the underlying interpreter. Narrowing-driven PE [4, 5] is the first generic algorithm for the specialization of functional logic programs. The method is parametric w.r.t. the narrowing strategy which is used for the automatic construction of the search trees. The method is formalized within the theoretical framework established in [39] for the partial evaluation of logic programs (also known as *partial deduction*), although a number of concepts have been generalized to deal with the functional component of the language (e.g., nested function calls in expressions, different evaluation strategies, etc.). This approach has better opportunities for optimization thanks to the functional dimension (e.g., by the inclusion of deterministic evaluation steps). Also, since unification is embedded into narrowing, it is able to automatically propagate syntactic information on the partial input (term structure) and not only constant values, similar to partial deduction. Using the terminology of [24], the narrowing-driven PE method of [5] is able to produce both *polyvariant* and *polygenetic* specializations, i.e., it can produce different specializations for the same function definition and can also combine distinct original function definitions into a comprehensive specialized function. This means that narrowing-driven PE has the same potential for specialization as *positive supercompilation* of functional programs [23] and *conjunctive partial deduction* of logic programs [38] (a comparison can be found in [1, 5, 6]).

The contribution of this paper is the definition of a partial evaluator for functional logic programs based on needed narrowing. To be more precise, we provide the following results:

- We prove strong correctness for such a partial evaluator, i.e., the answers computed by needed narrowing in the original and the partially evaluated programs coincide.
- We relate this partial evaluator to partial evaluation based on lazy narrowing [3] and show its advantages.
- We prove that partial evaluation based on needed narrowing keeps desirable properties during the specialization process, namely the inductively sequential structure of programs which is a prerequisite for optimal evaluation strategies. This is in contrast to partial evaluation based on lazy narrowing which can destroy such properties. Nevertheless, we also show a positive result about the structure of specialized programs obtained by PE based on lazy narrowing.
- We show that the specialized programs do not lose their abilities for deterministic reduction. This is important from an implementation point of view and it is not obtained by partial evaluation based on other operational models, like lazy narrowing.
- Moreover, we provide experimental evidence of the advantages of partial evaluation based on needed narrowing.

The multi-paradigm language Curry [27, 30] is an attempt to combine the paradigms of functional, logic and concurrent programming. Since the kernel of Curry (i.e., without the concurrency features) is based on needed narrowing and inductively sequential programs, the results of this paper can be applied to optimize a large class of Curry programs.

The structure of the paper is as follows. After some basic definitions in the next section, we recall in Section 3 the formal definition of inductively sequential programs and needed narrowing. Section 4 recalls the lazy narrowing strategy and relates it to needed narrowing. The definition of partial evaluation based on needed narrowing is provided in Section 5 together with results about the structure of specialized programs and the (strong) correctness of the transformation. Section 7 shows the practical importance of our specialization techniques by means of some benchmarks and Section 8 concludes.

2 Preliminaries

Term rewriting systems (TRSs) provide an adequate computational model for functional languages which allow the definition of functions by means of patterns (e.g., Haskell, Hope or Miranda) [12, 33, 47]. Within this framework, the class of inductively sequential programs, which we consider in this paper, has been defined, studied, and used for the implementation of programming languages which provide for optimal computations both in functional and functional logic programming [7, 10, 27, 28, 40]. Inductively sequential programs can be thought of as constructor-based TRSs with discriminating left-hand sides, i.e., typical functional programs. Thus, in the remainder of the paper we follow the standard framework of term rewriting [19] for developing our results.

We consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. There is at least one sort *Bool* containing the 0-ary Boolean constructors *true* and *false*. The set of *terms* and *constructor terms* with *variables* (e.g., x, y, z) from $\mathcal{X}$ are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$. A term is *linear* if it does not contain multiple occurrences of one variable. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$.

A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A term is *operation-rooted* if it has an operation symbol at the root. $root(t)$ denotes the symbol at the root of the term t. A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). Positions are ordered by the *prefix* ordering: $u \leq v$, if there exists w such that $u.w = v$. Positions u, v are *disjoint*, denoted $u \perp v$, if neither $u \leq v$ nor $v \leq u$. Given a term t, we let $\mathcal{P}os(t)$ and $\mathcal{NVP}os(t)$ denote the set of positions and the set of non-variable positions of t, respectively. $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [19] for details). For a set of (pairwise distinct, ordered) positions $P = \{p_1, \ldots, p_n\}$, we let $t[s_1, \ldots, s_n]_P = (((t[s_1]_{p_1})[s_2]_{p_2}) \ldots [s_n]_{p_n})$.

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

A term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$. This implies a *subsumption ordering* on terms which is defined by $t \leq t'$ iff t' is an instance of t. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. The unifier σ is *most general* if $\sigma \leq \sigma'\ [\mathcal{X}]$ for each other unifier σ'. Two substitutions σ and σ' are *independent* (on a set of variables V) iff there exists some $x \in V$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable.

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A TRS $\mathcal{R}$ is *left-linear* if l is linear for all $l \to r \in \mathcal{R}$.

A TRS is constructor-based (CB) if each lhs l is a pattern. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position $p \in \mathcal{NVP}os(l)$ and a most general unifier σ such that $\sigma(l|_p) = \sigma(l')$. The pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ is called a *critical pair* and is also called an *overlay* if $p = \Lambda$. A critical pair $\langle t, s \rangle$ is trivial if $t = s$. A left-linear TRS without critical pairs is called *orthogonal*. It is called *almost orthogonal* if its critical pairs are trivial overlays. If it only has trivial critical pairs it is called *weakly orthogonal*. Note that, in CB-TRSs, almost orthogonality and weak orthogonality coincide. In the remainder of this paper, a *functional logic program* is a finite left-linear CB-TRS. Conditions in program rules are treated by using the predefined functions `and`, `if_then_else`, `case_of` which are reduced by standard defining rules [44].

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$ (p and R will often be omitted in the notation of a computation step). The instantiated lhs $\sigma(l)$ is called a *redex*. We let $\mathcal{P}os_{\mathcal{R}}(t)$ denote the set of *redex positions* of the term t in the TRS $\mathcal{R}$. The inner reduction relation is $\to_{>\Lambda} = \to \backslash \to_{\Lambda}$. A term t is root-stable (often called a *head-normal form*) if it cannot be rewritten to a redex. A term is root-normalizing if it has a root-stable reduct. A *constructor root-stable* term is either a variable or a *constructor-rooted* term, that is, a term rooted by a constructor symbol. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$. $\to^+$ denotes the transitive closure of $\to$ and $\to^*$ denotes the reflexive and transitive closure of $\to$.

To evaluate terms containing variables, narrowing non-deterministically instantiates the variables such that a rewrite step is possible (usually by computing most general unifiers between a subterm and some lhs [25], but this requirement is relaxed in needed narrowing steps to obtain an optimal evaluation strategy [10]). Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to_{p,R} t'$. We denote by $t_0 \leadsto^*_{\sigma} t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. Since we are interested in computing *values* (constructor terms) as well as *answers* (substitutions) in functional logic programming, we say that the narrowing derivation $t \leadsto^*_{\sigma} c$ *computes the result c with answer* σ if c is a constructor term. The evaluation to ground constructor terms (and not to arbitrary expressions) is the intended semantics of functional languages and also of most functional logic languages. In particular, the equality predicate $\approx$ used in some examples is defined, as in functional languages, as the *strict equality* on terms (note that we do not require terminating rewrite systems and thus reflexivity is not desired), i.e., the equation $t_1 \approx t_2$ is satisfied if t_1 and t_2 are reducible to the same ground constructor term. Furthermore, a substitution σ is a *solution* for an equation $t_1 \approx t_2$ if $\sigma(t_1) \approx \sigma(t_2)$ is satisfied. The strict equality can be defined as a binary Boolean function by the following set of orthogonal rewrite rules (see, e.g., [10]):

$$
\begin{array}{rcll}
\mathtt{c} \approx \mathtt{c} & \to & \mathtt{true} & \text{for all } \mathtt{c}/\mathtt{0} \in \mathcal{C} \\
\mathtt{c}(\mathtt{X_1}, \ldots, \mathtt{X_n}) \approx \mathtt{c}(\mathtt{Y_1}, \ldots, \mathtt{Y_n}) & \to & (\mathtt{X_1} \approx \mathtt{Y_1}) \wedge \ldots \wedge (\mathtt{X_n} \approx \mathtt{Y_n}) & \text{for all } \mathtt{c}/\mathtt{n} \in \mathcal{C}, \mathtt{n} > \mathtt{0} \\
\mathtt{true} \wedge \mathtt{X} & \to & \mathtt{X} &
\end{array}
$$

Thus, we do not treat the strict equality in any special way, and it is sufficient to consider it as a Boolean function which must be reduced to the constant `true`. We say that σ is a *computed answer substitution* for an equation e if there is a narrowing derivation $e \leadsto^*_{\sigma}$ `true`. More details about strict equality can be found in [10, 22, 44].

Narrowing derivations can be represented by a (possibly infinite) finitely branching tree. Following [39], in this work we adopt the convention that any derivation is potentially incomplete (a branch thus can be failed, incomplete, successful, or infinite). A *failing leaf* contains an expression which is not a constructor term and which cannot be further narrowed.

3 Needed Narrowing

A challenge in the design of functional logic languages is the definition of a "good" narrowing strategy, i.e., a restriction λ on the narrowing steps issuing from t, without losing completeness. *Needed narrowing* [10] is currently the best known narrowing strategy due to its optimality properties w.r.t. the length of successful derivations and the number of computed solutions. Needed narrowing is defined on *inductively sequential programs.* To provide a precise definition of this class of programs and the needed narrowing strategy, we introduce definitional trees. In contrast to the original definition [7], here we use the "declarative" definition [8] since it is more appropriate for proving the results about partial evaluation based on needed narrowing.

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

Root property: There is a minimum element $pattern(\mathcal{P})$, also called the *pattern* of the definitional tree.

Leaves property: The maximal elements, called the *leaves*, are the elements of S. Non-maximal elements are also called *branches.*

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

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

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

It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position

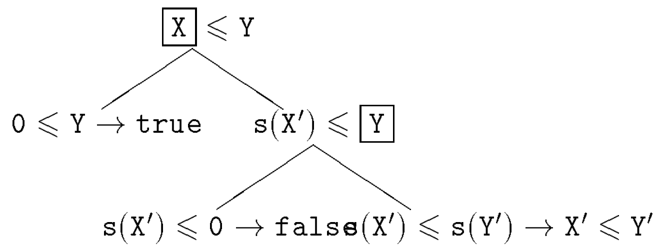

Figure 1: Definitional tree for the function "$\leqslant$"

in branches is surrounded by a box, and the leaves contain the corresponding rules. For instance, the definitional tree for the function "$\leqslant$" in Example 1 is illustrated in Figure 1.

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

Proposition 1 *If $f(\overline{t_n})$ is a linear pattern, then there exists a definitional tree for the set $\{f(\overline{t_n})\}$ with pattern $f(\overline{x_n})$.*

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

For the definition of needed narrowing, we assume that t is an operation-rooted term and $\mathcal{P}$ is a definitional tree with $pattern(\mathcal{P}) = \pi$ such that $\pi \leq t$. We define a function λ from terms and definitional trees to sets of tuples (position, rule, substitution) as the least set satisfying the following properties. We consider two cases for $\mathcal{P}$:[1]

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

2. If π is a branch, consider the inductive position o of π and a child $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. Let $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ be the definitional tree where all patterns are instances of π_i. Then we consider the following cases for the subterm $t|_o$:
$$\lambda(t, \mathcal{P}) \ni \begin{cases} (p, R, \sigma \circ \tau) & \text{if } t|_o = x \in \mathcal{X},\ \tau = \{x \mapsto c_i(\overline{x_n})\}, \\ & \quad \text{and } (p, R, \sigma) \in \lambda(\tau(t), \mathcal{P}_i); \\ (p, R, \sigma \circ id) & \text{if } t|_o = c_i(\overline{t_n}) \text{ and } (p, R, \sigma) \in \lambda(t, \mathcal{P}_i); \\ (o.p, R, \sigma \circ id) & \text{if } t|_o = f(\overline{t_n}) \text{ for } f \in \mathcal{F} \text{ and } (p, R, \sigma) \in \lambda(t|_o, \mathcal{P}') \\ & \quad \text{where } \mathcal{P}' \text{ is a definitional tree for } f. \end{cases}$$

Informally speaking, needed narrowing applies a rule, if possible (case 1), or checks the subterm corresponding to the inductive position of the branch (case 2): if it is a variable, it is instantiated to the constructor of a child; if it is already a constructor, we proceed with

[1]This description of a needed narrowing step is slightly different from [10], but it results in the same needed narrowing steps.

the corresponding child; if it is a function, we evaluate it by recursively applying needed narrowing. Thus, the strategy differs from lazy functional languages only in the instantiation of free variables.

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

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

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

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

Then the function λ computes the following set for the initial term $\mathtt{X} \leqslant \mathtt{X} + \mathtt{X}$:

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

This corresponds to the narrowing steps

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

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

Proposition 2 *If $(p, R, \varphi_k \circ \cdots \circ \varphi_1) \in \lambda(t, \mathcal{P})$ is a needed narrowing step, then, for $i = 1, \ldots, k$, $\varphi_i = id$ or $\varphi_i = \{x \mapsto c(\overline{x_n})\}$ (where $\overline{x_n}$ are pairwise different variables) with $x \in \mathcal{V}ar(\varphi_{i-1} \circ \cdots \circ \varphi_1(t))$.*

Proof. By induction on k. □

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

Lemma 3 *Let t be an operation-rooted term, $\mathcal{P}$ a definitional tree with $pattern(\mathcal{P}) \leq t$ and $(p, R, \varphi_k \circ \cdots \circ \varphi_1), (p', R', \varphi'_{k'} \circ \cdots \circ \varphi'_1) \in \lambda(t, \mathcal{P})$, $k \leq k'$. Then, for all $i \in \{1, \ldots, k\}$,*

- *either $\varphi_i \circ \cdots \circ \varphi_1 = \varphi'_i \circ \cdots \circ \varphi'_1$, or*

- *there exists some $j < i$ with*

  1. $\varphi_j \circ \cdots \circ \varphi_1 = \varphi'_j \circ \cdots \circ \varphi'_1$, *and*
  2. $\varphi_{j+1} = \{x \mapsto c(\cdots)\}$ *and* $\varphi'_{j+1} = \{x \mapsto c'(\cdots)\}$ *with* $c \neq c'$.

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

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

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

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

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

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

$\square$

For inductively sequential programs, needed narrowing is sound and complete w.r.t. strict equations and constructor substitutions as solutions (note that constructor substitutions are sufficient in practice since more general solutions would contain unevaluated or undefined expressions). Moreover, needed narrowing does not compute redundant solutions. These properties are formalized as follows:

Theorem 4 [10] *Let $\mathcal{R}$ be an inductively sequential program and e an equation.*

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

An important advantage of functional logic languages in comparison to pure logic languages is their improved operational behavior by avoiding non-deterministic computation steps. One reason for that is a demand-driven computation strategy which can avoid the evaluation of potential non-deterministic expressions. For instance, consider the rules in Examples 1 and 2 and the term $\mathtt{0} \leqslant \mathtt{X} + \mathtt{X}$. Needed narrowing evaluates this term by one deterministic step to `true`. In an equivalent logic program, this nested term must be flattened into a conjunction of two predicate calls, like $+(\mathtt{X},\mathtt{X},\mathtt{Z}) \wedge \leqslant(\mathtt{0},\mathtt{Z},\mathtt{B})$, which causes a non-deterministic computation due to the predicate call $+(\mathtt{X},\mathtt{X},\mathtt{Z})$.[2] Another reason for the improved operational behavior of functional logic languages is the ability of particular evaluation strategies (like needed narrowing or parallel narrowing [11]) to evaluate ground terms in a completely deterministic way, which is important to ensure an efficient implementation of purely functional evaluations. This property, which is obvious by the definition of needed narrowing, is formally stated in the following proposition. For this purpose, we call a term t *deterministically evaluable* (w.r.t. needed narrowing) if each step in a narrowing derivation issuing from t is deterministic. A term t *deterministically normalizes* to a constructor term c (w.r.t. needed narrowing) if t is deterministically evaluable and there is a needed narrowing derivation $t \leadsto^*_{id} c$ (i.e., c is the normal form of t).

Proposition 5 *Let $\mathcal{R}$ be an inductively sequential program and t be a term.*

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

4 Lazy Narrowing and Uniform Programs

In order to compare partial evaluators based on lazy narrowing and needed narrowing and to show the improvements obtained by using needed narrowing in partial evaluation, we provide a brief review of the lazy narrowing strategy in this section.

Lazy narrowing reduces expressions at outermost narrowable positions. Narrowing at inner positions is performed only if it is demanded (by the pattern in the lhs of some rule). In the following, we specify the lazy narrowing strategy similar to [44].

The following definitions are necessary for our formalization of lazy narrowing. A linear unification problem is a pair of terms: $\delta = \langle f(\overline{d_n}), f(\overline{t_n})\rangle$, where $f(\overline{d_n})$ and $f(\overline{t_n})$ do not share variables, and $f(\overline{d_n})$ is a linear pattern. Linear unification $\mathtt{LU}(\delta)$ can either succeed, fail or suspend, delivering $(\mathtt{Succ}, \sigma)$, $(\mathtt{Fail}, \varnothing)$ or $(\mathtt{Demand}, P)$, respectively, where P is the set of *demanded positions* which require further evaluation (details can be found in [3]).

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

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

Definition 6 (lazy narrowing strategy)

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

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

Example 3 *Consider the rules for "$\leqslant$" and "$+$" in Examples 1 and 2. Then lazy narrowing evaluates the term* $\mathtt{X} \leqslant \mathtt{X}+\mathtt{X}$ *by applying a narrowing step at the top (with the first rule for "$\leqslant$") or by applying a narrowing step to the second argument* $\mathtt{X}+\mathtt{X}$ *since this is demanded by the second and third rules for "$\leqslant$". Thus, there are three lazy narrowing steps:*

$$\mathtt{X} \leqslant \mathtt{X}+\mathtt{X} \quad \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} \quad \mathtt{true}$$

$$\mathtt{X} \leqslant \mathtt{X}+\mathtt{X} \quad \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} \quad \mathtt{0} \leqslant \mathtt{0}$$

$$\mathtt{X} \leqslant \mathtt{X}+\mathtt{X} \quad \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(M)}\}} \quad \mathtt{s(M)} \leqslant \mathtt{s(M+s(M))}$$

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

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

Proposition 7 [44] *Let $\mathcal{R}$ be an orthogonal program, e an equation, and σ a constructor substitution that is a solution for e. Then there is a lazy narrowing derivation $e \leadsto^*_{\sigma'}$ true such that $\sigma' \leq \sigma$ $[\mathcal{V}ar(e)]$.*

Thus, lazy narrowing is complete for a larger class of programs than needed narrowing (since inductively sequential programs are always orthogonal)[3], but it has in some cases a behavior which is worse than needed narrowing (see Example 3). There exists a class of programs where the superfluous steps of lazy narrowing are avoided, since lazy narrowing and needed narrowing coincide on this class. These are the *uniform* programs [50] which are inductively sequential programs where at most one constructor occurs in the left-hand side of each rule. A program is *uniform* if each function f is defined by one rule $f(\overline{x_n}) \to r$ or the lhs of every rule R_i defining f has the form $f(\overline{x_k}, c_i(\overline{y_{n_i}}), \overline{z_m})$, where $\overline{x_k}, \overline{y_{n_i}}, \overline{z_m}$ are pairwise different variables and the constructors c_i are distinct in different rules. In the latter case, an evaluation of a call to f demands its $(k+1)$-th argument. A different definition of uniform programs can be found in [34].

[3] The idea of needed narrowing can also be extended to almost orthogonal programs [11], but then the optimality properties are lost.

There is a simple mapping $\mathcal{U}$ from inductively sequential into uniform programs which can be found in [50] and is based on flattening nested patterns. For instance, if $\mathcal{R}$ is the program in Example 1, then $\mathcal{U}(\mathcal{R})$ consists of the rules

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

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

The following theorem states a correspondence between needed narrowing derivations using the original program and lazy narrowing derivations in the transformed uniform program.

Theorem 8 [50] *Let $\mathcal{R}$ be an inductively sequential program, $\mathcal{U}(\mathcal{R})$ the transformed uniform program, and t an operation-rooted term. Then there exists a needed narrowing derivation $t \leadsto^*_\sigma s$ w.r.t. $\mathcal{R}$ to a constructor root-stable form s iff there exists a lazy narrowing derivation $t \leadsto^*_\sigma s$ w.r.t. $\mathcal{U}(\mathcal{R})$.*

5 Partial Evaluation with Needed Narrowing

In this section, we introduce the notion of partial evaluation of functional logic programs in order to define a partial evaluator based on needed narrowing. Moreover, we show some important properties of the specialized programs.

Specialized program rules are constructed from narrowing derivations using the notion of *resultant*.

Definition 9 (resultant) *Let $\mathcal{R}$ be a TRS and s be a term. Given a narrowing derivation $s \leadsto^+_\sigma t$, its associated resultant is the rewrite rule $\sigma(s) \rightarrow t$.*

We note that, whenever the specialized call s is not a linear pattern, lhs's of resultants may not be linear patterns either and hence resultants may not be program rules. In order to produce program rules, we will introduce a post-processing renaming transformation which not only eliminates redundant structures but also obtains *independent* specializations (in the sense of [39]) and is necessary for the correctness of the PE transformation. Roughly speaking, independence ensures that the different specializations for the same function definition are correctly distinguished, which is crucial for polyvariant specialization.

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

Definition 10 (pre–partial evaluation) *Let $\mathcal{R}$ be a TRS and s a term. Let T be a finite (possibly incomplete) narrowing tree for s in $\mathcal{R}$ such that no constructor root-stable term in the tree has been narrowed. Let $\overline{t_n}$ be the terms in the non-failing leaves of T. Then, the set of resultants for the narrowing sequences $\{s \leadsto^+_{\sigma_i} t_i \mid i = 1, \ldots, n\}$ is called a pre–partial evaluation of s in $\mathcal{R}$.*

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

The following example illustrates that the restriction to not surpass head normal forms in pre–partial evaluations cannot be dropped.

Example 4 *Consider the following program $\mathcal{R}$:*

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

with the set of calls $S = \{\mathtt{g(X)}, \mathtt{h(X)}\}$. Then, a pre–partial evaluation of S in $\mathcal{R}$ is the program $\mathcal{R}'$:

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

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

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

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

A recursive *closedness* condition, which guarantees that each call which might occur during the execution of the resulting program is covered by some program rule, is formalized by inductively checking that the different calls in the rules are sufficiently covered by the specialized functions.

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

Definition 11 (closedness) *Let S be a finite set of terms. We say that a term t is S-closed if $closed(S,t)$ holds, where the predicate closed is defined inductively as follows:*

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

We say that a set of terms T is S-closed, written $closed(S,T)$, if $closed(S,t)$ holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is S-closed if $closed(S, \mathcal{R}_{calls})$ holds. Here we denote by $\mathcal{R}_{calls}$ the set of the rhs's of the rules in $\mathcal{R}$.

According to the (non-deterministic) definition above, an expression rooted by a "primitive" function symbol, such as a conjunction $t_1 \wedge t_2$ or an equation $t_1 \approx t_2$, can be proven closed w.r.t. S either by checking that t_1 and t_2 are S-closed or by testing whether the conjunction (equation) is an instance of a call in S (followed by an inductive test of the subterms). This is useful when we are not interested in specializing complex expressions (like conjunctions or strict equations) but we still want to run them after specialization. Note that this is safe, since we consider that the rules which define the primitive functions are automatically added to each program, hence calls to these symbols are steadily covered in the specialized program. A general technique for dealing with primitive symbols which deterministically splits terms before testing them for closedness and is able to improve the specialization can be found in [1].

In general, given a call s and a program $\mathcal{R}$, there exists an infinite number of different pre–partial evaluations of s in $\mathcal{R}$. A fixed rule for generating resultants called an *unfolding rule* is assumed, which determines the expressions to be narrowed (by using a fixed narrowing strategy) and which decides how to stop the construction of narrowing trees (see [5, 1] for the definition of concrete unfolding rules).

Example 5 *Consider the well-known function* `append` *to concatenate two lists:*[4]

$$\begin{array}{l} \mathtt{append}(\mathtt{nil}, \mathtt{Y_s}) \to \mathtt{Y_s} \\ \mathtt{append}(\mathtt{X}:\mathtt{X_s}, \mathtt{Y_s}) \to \mathtt{X}:\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}) \end{array}$$

with the set of calls $S = \{\mathtt{append}(\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{Z_s}),\ \mathtt{append}(\mathtt{X_s}, \mathtt{Y_s})\}$. *A pre–partial evaluation of* S *in* $\mathcal{R}$ *using needed narrowing is the* S*-closed program:*

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

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

Definition 12 (independent renaming) *An independent renaming* ρ *for a set of terms* S *is a mapping from terms to terms defined as follows: for* $s \in S$, $\rho(s) = f_s(\overline{x_n})$, *where* $\overline{x_n}$ *are the distinct variables in* s *in the order of their first occurrence and* f_s *is a new function symbol, which does not occur in* $\mathcal{R}$ *or* S *and is different from the root symbol of any other* $\rho(s')$, *with* $s' \in S$ *and* $s' \neq s$. *By abuse, we let* $\rho(S)$ *denote the set* $S' = \{\rho(s) \mid s \in S\}$.

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

[4] We use `nil` and `:` as constructors of lists.

Definition 13 (partial evaluation) *Let $\mathcal{R}$ be a TRS, S a finite set of terms and $\mathcal{R}'$ a pre–partial evaluation of $\mathcal{R}$ w.r.t. S. Let ρ be an independent renaming of S. We define the partial evaluation $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. S (under ρ) as follows:*

$$\mathcal{R}'' = \bigcup_{s \in S} \{\theta(\rho(s)) \to ren_\rho(r) \mid \theta(s) \to r \in \mathcal{R}' \textit{ is a resultant for } s \textit{ in } \mathcal{R}\}$$

where the non-deterministic renaming function ren_ρ is defined as follows:

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

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

We now illustrate these definitions with an example.

Example 6 *Consider again the program* `append` *and the set S of Example 5. An independent renaming ρ for S is the mapping:*

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

A partial evaluation $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. S (under ρ) is:

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

We note that, for a given renaming ρ, the filtered form of a program $\mathcal{R}$ may depend on the strategy which selects the term from $\rho(S)$ which is used to rename a given call t in $\mathcal{R}$ (e.g., $\mathtt{append}(\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{Z_s})$), since there may exist, in general, more than one s in S that covers the call t. 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.

The correctness of LN-PE is stated in [1, 3]. The following lemma shows that the partial evaluation w.r.t. needed narrowing can also be obtained (but possibly with more steps) by partial evaluation of the transformed uniform program w.r.t. lazy narrowing. This shows that in some sense the specializations computed by a partial evaluator based on needed narrowing cannot be worse than the specializations computed by a lazy narrowing partial evaluator. On the other hand, we will show that there are cases where a LN-PE is worse than a NN-PE for the same original program.

Lemma 14 *Let $\mathcal{R}$ be an inductively sequential program, $\mathcal{R}_u = \mathcal{U}(\mathcal{R})$ the corresponding uniform program, and S a finite set of operation-rooted terms. If $\mathcal{R}'$ is a NN-PE of S in $\mathcal{R}$, then $\mathcal{R}'$ is also a LN-PE of S in $\mathcal{R}_u$.*

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

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

where $t \in S$ and $t \leadsto^+_\sigma s$ is a needed narrowing derivation w.r.t. $\mathcal{R}$. By Theorem 8, there exists a lazy narrowing derivation $t \leadsto^+_\sigma s$ w.r.t. $\mathcal{R}_u$ which has the same answer and result (note that Theorem 8 states this property only for derivations into constructor-rooted terms, but it also holds in the direction used here for arbitrary needed narrowing derivations since each needed narrowing step corresponds to a sequence of lazy narrowing steps w.r.t. the transformed uniform programs, which can be seen by the proof of this theorem). Thus, $\sigma(t) \to s$ is a resultant of this lazy narrowing derivation w.r.t. $\mathcal{R}_u$. $\square$

The following theorem states an important property of partial evaluation w.r.t. needed narrowing: if the input program is inductively sequential, then the specialized program is also inductively sequential so that we can also apply the optimal needed narrowing strategy to the specialized program. Firstly, this is only proved for partial evaluation w.r.t. linear patterns but later we extend this result to arbitrary sets of terms.

Theorem 15 *Let $\mathcal{R}$ be an inductively sequential program and t be a linear pattern. If $\mathcal{R}'$ is a pre-NN-PE of t in $\mathcal{R}$, then $\mathcal{R}'$ is inductively sequential.*

Proof. Due to the definition of pre-NN-PE, $\mathcal{R}'$ has the form

$$\begin{array}{l} \sigma_1(t) \to t_1 \\ \vdots \\ \sigma_n(t) \to t_n \end{array}$$

where $t \leadsto^+_{\sigma_i} t_i$, $i = 1, \ldots, n$, are all the derivations in the needed narrowing tree for t ending in a non-failing leaf. To show the inductive sequentiality of $\mathcal{R}'$, it suffices to show that there exists a definitional tree for the set $S = \{\sigma_1(t), \ldots, \sigma_n(t)\}$ with pattern $f(\overline{x_p})$ if t has the p-ary function f at the root. We prove this property by induction on the number of inner nodes of the narrowing tree for t.

Base case: If the number of inner nodes is 1, we first construct a definitional tree for the set $S = \{t\}$ containing only the pattern at the root of the narrowing tree. This is always possible by Proposition 1. Now we construct a definitional tree for the sons of the root by extending this initial definitional tree. This construction is identical to the induction step.

Induction step: Assume that s is a leaf in the narrowing tree, σ is the accumulated substitution from the root to this leaf, and $\mathcal{P}$ is a definitional tree for the set

$$\begin{array}{l} S = \{\theta(t) \mid t \leadsto^+_\theta s' \text{ is a derivation in the needed narrowing tree} \\ \qquad\qquad\qquad \text{with a non-failing leaf } s'\}. \end{array}$$

Now we extend the narrowing tree by applying one needed narrowing step to s, i.e., let

$$s \leadsto_{\varphi_1} s_1$$
$$\vdots$$
$$s \leadsto_{\varphi_m} s_m$$

be all needed narrowing steps for s. For the induction step, it is sufficient to show that there exists a definitional tree for

$$S' = (S \backslash \{\sigma(t)\}) \cup \{\varphi_1(\sigma(t)), \ldots, \varphi_m(\sigma(t))\} .$$

Consider for each needed narrowing step $s \leadsto_{\varphi_i} s_i$ the associated canonical representation $(p, R, \varphi_{ik_i} \circ \cdots \circ \varphi_{i1}) \in \lambda(s, \mathcal{P}_s)$ (where $\mathcal{P}_s$ is a definitional tree for the root of s). Let

$$\mathcal{P}' = \mathcal{P} \cup \{\varphi_{ij} \circ \cdots \circ \varphi_{i1} \circ \sigma(t) \mid 1 \leq i \leq m, 1 \leq j \leq k_i\}$$

We prove that $\mathcal{P}'$ is a definitional tree for S' by showing that each of the four properties of a definitional tree holds for $\mathcal{P}'$.

Root property: The minimum elements are identical for both definitional trees, i.e., $pattern(\mathcal{P}) = pattern(\mathcal{P}')$, since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.

Leaves property: The maximal elements of $\mathcal{P}$ are S. Since all substitutions computed by needed narrowing along different derivations are independent (Lemma 3), σ is independent to all other substitutions occurring in S and the substitutions $\varphi_1, \ldots, \varphi_m$ are pairwise independent. Thus, the replacement of the element $\sigma(t)$ in S by the set $\{\varphi_1(\sigma(t)), \ldots, \varphi_m(\sigma(t))\}$ does not introduce any comparable (w.r.t. the subsumption ordering) terms. This implies that S' is the set of maximal elements of $\mathcal{P}'$.

Parent property: Let $\pi \in \mathcal{P}' \backslash \{pattern(\mathcal{P}')\}$. We consider two cases for π:

1. $\pi \in \mathcal{P}$: Then the parent property trivially holds since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.
2. $\pi \notin \mathcal{P}$: By definition of $\mathcal{P}'$, $\pi = \varphi_{ij} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$ for some $1 \leq i \leq m$ and $1 \leq j \leq k_i$. We show by induction on j that the parent property holds for π.
   Base case ($j = 1$): Then $\pi = \varphi_{i1}(\sigma(t))$. It is $\varphi_{i1} \neq id$ (otherwise $\pi = \sigma(t) \in \mathcal{P}$). Thus, by Proposition 2, $\varphi_{i1} = \{x \mapsto c(\overline{x_n})\}$ with $x \in \mathcal{V}ar(s) \subseteq \mathcal{V}ar(\sigma(t))$. Due to the linearity of the initial pattern and all substituted terms (cf. Proposition 2), $\sigma(t)$ has a single occurrence o of the variable x and, therefore, $\pi = \sigma(t)[c(\overline{x_n})]_o$, i.e., $\sigma(t)$ is the unique parent of π.
   Induction step ($j > 1$): We assume that the parent property holds for $\pi' = \varphi_{i,j-1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$. Let $\varphi_{ij} \neq id$ (otherwise the induction step is trivial). By Proposition 2, $\varphi_{ij} = \{x \mapsto c(\overline{x_n})\}$ with $x \in \mathcal{V}ar(\varphi_{i,j-1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t))$ (since $\mathcal{V}ar(s) \subseteq \mathcal{V}ar(\sigma(t))$). Now we proceed as in the base case to show that π' is the unique parent of π.

Induction property: Let $\pi \in \mathcal{P}' \backslash S'$. We consider two cases for π:

1. $\pi \in \mathcal{P}\backslash\{\sigma(t)\}$: Then the induction property holds for π since it already holds in $\mathcal{P}$ and only instances of $\sigma(t)$ are added in $\mathcal{P}'$.
2. $\pi = \varphi_{ij} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$ for some $1 \leq i \leq m$ and $0 \leq j < k_i$. Assume $\varphi_{i,j+1} \neq id$ (otherwise, do the identical proof with the representation $\pi = \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$). By Proposition 2, $\varphi_{i,j+1} = \{x \mapsto c(\overline{x_n})\}$ and π has a single occurrence of the variable x (due to the linearity of the initial pattern and all substituted terms). Therefore, $\pi' = \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$ is a child of π. Consider another child $\pi'' = \varphi_{i'j'} \circ \cdots \circ \varphi_{i'1} \circ \sigma(t)$ of π (other patterns in $\mathcal{P}'$ cannot be children of π due to the induction property for $\mathcal{P}$). Assume $\varphi_{i'j'} \circ \cdots \circ \varphi_{i'1} \neq \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1}$ (otherwise, both children are identical). By Lemma 3, there exists some l with $\varphi_{i'l} \circ \cdots \circ \varphi_{i'1} = \varphi_{il} \circ \cdots \circ \varphi_{i1}$, $\varphi_{i',l+1} = \{x' \mapsto c'(\cdots)\}$, and $\varphi_{i,l+1} = \{x' \mapsto c''(\cdots)\}$ with $c' \neq c''$. Since π'' and π' are children of π (i.e., immediate successors w.r.t. the subsumption ordering) it must be $x' = x$ (otherwise, π' differs from π at more than one position) and $\varphi_{i',j'} = \cdots = \varphi_{i',l+2} = id$ (otherwise, π'' differs from π at more than one position). Thus, π' and π'' differ only in the instantiation of the variable x which has exactly one occurrence in their common parent π, i.e., there is a position o of π with $\pi|_o = x$ and $\pi' = \pi[c'(\overline{x_{n'_i}})]_o$ and $\pi'' = \pi[c''(\overline{x_{n''_i}})]_o$. Since π'' was an arbitrary child of π, the induction property holds.

□

Since partial evaluation is usually initiated with more than one term, we extend the previous theorem to this more general case.

Corollary 16 *Let $\mathcal{R}$ be an inductively sequential program and S be a finite set of linear patterns with pairwise different root symbols. If $\mathcal{R}'$ is a pre-NN-PE of S in $\mathcal{R}$, then $\mathcal{R}'$ is inductively sequential.*

Proof. This is a consequence of Theorem 15 since we can construct a definitional tree for each pre-NN-PE of a pattern of S. Since all patterns have different root symbols, the roots of these definitional trees do not overlap. □

Now we are able to show that, using needed narrowing, partial evaluation of an arbitrary set of terms w.r.t. an inductively sequential program always produces an inductively sequential program.

Theorem 17 *Let $\mathcal{R}$ be an inductively sequential program and S a finite set of operation-rooted terms. Then each NN-PE of $\mathcal{R}$ w.r.t. S is inductively sequential.*

Proof. Let $\mathcal{R}'$ be a pre-NN-PE of $\mathcal{R}$ w.r.t. S and let ρ be an independent renaming of S. Then each rule of a NN-PE $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. S (under ρ) has the form $\theta(\rho(s)) \to ren_\rho(r)$ for some rule $\theta(s) \to r \in \mathcal{R}'$. Consider the extended rewrite system

$$\mathcal{R}_\rho = \mathcal{R} \cup \{\rho(s) \to s \mid s \in S\}$$

where the renaming ρ is encoded by a set of rewrite rules. Note that $\mathcal{R}_\rho$ is inductively sequential since the new left-hand sides $\rho(s)$ are of the form $f_s(\overline{x_n})$ with new function symbols f_s.

Let $\mathcal{R}'_\rho$ be an arbitrary pre-NN-PE of $\mathcal{R}_\rho$ w.r.t. $\rho(S)$. Since $\rho(S)$ is a set of linear patterns with pairwise different root symbols, $\mathcal{R}'_\rho$ is inductively sequential by Corollary 16. It is obvious that each subset of an inductively sequential program is also inductively sequential (since only the left-hand sides of the rules are relevant for this property). Therefore, to complete the proof it is sufficient to show that all left-hand sides of rules from $\mathcal{R}''$ can also occur as left-hand sides in some $\mathcal{R}'_\rho$.

Each rule of $\mathcal{R}''$ has the form $\theta(\rho(s)) \to ren_\rho(r)$ for some rule $\theta(s) \to r \in \mathcal{R}'$. By definition of $\mathcal{R}'$, there exists a needed narrowing derivation $s \leadsto^+_\theta r$ w.r.t. $\mathcal{R}$. Hence,

$$\rho(s) \ \leadsto_{id} \ s \ \leadsto^+_\theta \ r$$

is a needed narrowing derivation w.r.t. $\mathcal{R}_\rho$. Thus, $\theta(\rho(s)) \to r$ is a resultant which can occur in some $\mathcal{R}'_\rho$. □

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

Example 7 *Let $\mathcal{R}$ be the uniform program:*

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

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

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

which is not uniform.

Note that the residual program $\mathcal{R}'$ in the example above is inductively sequential. This raises the question as to whether the LN-PE of a uniform program is always inductively sequential. Corollary 18 will positively answer this question.

Corollary 18 *Let $\mathcal{R}$ be a uniform program and S a finite set of operation-rooted terms. If $\mathcal{R}'$ is a LN-PE of S in $\mathcal{R}$, then $\mathcal{R}'$ is inductively sequential.*

Proof. Since a uniform program is inductively sequential and lazy narrowing steps w.r.t. uniform programs are also needed narrowing steps (cf. proof of Theorem 8), the proposition is a direct consequence of Theorem 17. □

The uniformity condition in Corollary 18 cannot be weakened to inductive sequentiality when LN-PEs are considered, as demonstrated by the following counterexample.

Example 8 *Let $\mathcal{R}$ be the inductively sequential program:*

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

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

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

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

Two main factors affecting the quality of a PE are determinacy and choice points [21]. The following examples illustrate the different way in which NN-PE and LN-PE "compile-in" choice points during unfolding, which is crucial to performance since a poor control choice during the construction of the computation trees can inadvertently introduce extra computation into a program.

Example 9 *Consider again the rules of Example 2 and the input term* $\mathtt{X} \leqslant \mathtt{X + Y}$. *The computed LN-PE is*

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

where the renamed initial term is $\mathtt{leq2(X,Y)}$. *The redundancy of lazy narrowing has the effect that the first two rules of the specialized program are identical (up to renaming). A good specialization without generating redundant rules is obtained with partial evaluation based on needed narrowing, since the NN-PE consists of the rules*

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

which are computed in half of the time needed for LN-PE (see Section 7). A call-by-value partial evaluator based on innermost narrowing (without normalization) [5] has an even worse behavior in this example since it does not specialize the program at all.

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

Example 10 *Lazy evaluation strategies are necessary if one wants to deal with infinite data structures and possibly non-terminating function calls. The following orthogonal program makes extensive use of these features:*

$$\begin{array}{rcl} \mathtt{f(0,0)} & \to & \mathtt{s(f(0,0))} \\ \mathtt{f(s(N),X)} & \to & \mathtt{s(f(N,X))} \end{array} \qquad \begin{array}{rcl} \mathtt{g(0)} & \to & \mathtt{g(0)} \\ \mathtt{h(s(X))} & \to & \mathtt{0} \end{array}$$

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

```
h0  →  h0                h2(0,0)  →  h0
                       h2(s(X),Y)  →  0
```

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

```
   h1(X)  →  h1(X)          h2(X,0)  →  h1(X)
h1(s(X))  →  0           h2(s(X),Y)  →  0
                         h2(s(X),0)  →  0
```

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

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

Proposition 19 *Let $\mathcal{R}$ be an inductively sequential program, S a finite set of operation-rooted terms, ρ an independent renaming of S, and e an equation. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If e deterministically normalizes to true w.r.t. $\mathcal{R}$, then e' deterministically normalizes to true w.r.t. $\mathcal{R}'$.*

Proof. Since e deterministically normalizes to $true$ w.r.t. $\mathcal{R}$, there is a needed narrowing derivation $e \leadsto^*_{id} true$ in $\mathcal{R}$. By Theorem 20 (see below), there is a needed narrowing derivation $e' \leadsto^*_\sigma true$ in $\mathcal{R}'$ with $\sigma = id\ [\mathcal{V}ar(e)]$. This implies $\sigma = id$ by definition of needed narrowing. Therefore, e' deterministically normalizes to $true$ w.r.t. $\mathcal{R}'$ by Proposition 5. □

This property of the specialized programs is desirable and important from an implementation point of view, since the implementation of non-deterministic steps is an expensive operation in logic-oriented languages. Moreover, additional non-determinism in the specialized programs can result in additional infinite derivations, as shown in Example 10. This might have the effect that solutions are no longer computable in a sequential implementation based on backtracking. Therefore, this property is also desirable in partial deduction of logic programs, but as far as we know, no similar results are known for partial deduction of logic programs.

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

Theorem 20 (strong correctness) *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. Then, $e \leadsto^*_\sigma true$ is a needed narrowing derivation for e in $\mathcal{R}$ iff there exists a needed narrowing derivation $e' \leadsto^*_{\sigma'} true$ in $\mathcal{R}'$ such that $\sigma' = \sigma\ [V]$ (up to renaming).*

6 Strong Correctness of NN-PE

In this section, we prove Theorem 20, i.e., the strong correctness of NN-PE, and introduce some necessary auxiliary notions and results for this proof. The proof proceeds essentially as follows. Firstly, we prove the soundness (resp. completeness) of the transformation, i.e., we prove that for each answer computed by needed narrowing in the original (resp. specialized) program there exists a more general answer in the specialized (resp. original) program for the considered queries. Then, by using the minimality of needed narrowing, we conclude the strong correctness of NN-PE, i.e., the answers computed in the original and the partially evaluated programs coincide (up to renaming).

In order to simplify the proofs, we assume (without loss of generality) that the rules of the strict equality are automatically added to the original as well as the partially evaluated program. We also assume that the set of specialized terms always contains the calls $x \approx y$ and $x \wedge y$, and by abuse we take $\rho(\approx) = \approx$ and $\rho(\wedge) = \wedge$. This allows us to handle the strict equality rules in $\mathcal{R}'$ as ordinary resultants derived from the one-step needed narrowing derivations for the calls $x \approx y$ and $x \wedge y$ in $\mathcal{R}$.

6.1 Soundness

The following lemmata are auxiliary to prove that reduction sequences in the specialized program can also be performed in the original program (up to renaming of terms and programs).

Lemma 21 *Let $\mathcal{R}$ be an inductively sequential program and s be an operation-rooted term. Let $s \leadsto^+_\sigma r$ be a needed narrowing derivation w.r.t. $\mathcal{R}$ whose associated resultant is $R = (\sigma(s) \to r)$. If $t \to_{p,R} t'$ for some position $p \in \mathcal{P}os(t)$, then $t \to^+ t'$ w.r.t. $\mathcal{R}$.*

Proof. Given the derivation $s \leadsto^+_\sigma r$, by the soundness of needed narrowing (claim 1 of Theorem 4), we have $\sigma(s) \to^+ r$. Since $t \to_{p,R} t'$, there exists a substitution θ such that $\theta(\sigma(s)) = t|_p$ and $t' = t[\theta(r)]_p$. Since $\sigma(s) \to^+ r$, by the stability of rewriting, we have $\theta(\sigma(s)) \to^+ \theta(r)$. Therefore $t = t[\theta(\sigma(s))]_p \to^+ t[\theta(r)]_p = t'$ w.r.t. $\mathcal{R}$, which concludes the proof. $\square$

Lemma 22 *Let S be a finite set of terms and ρ an independent renaming for S. Let $R = (\theta(s) \to r)$ be a rewrite rule such that θ is constructor and $s \in S$, and let $R' = (l' \to r')$ be a renaming of R where $l' = \theta(\rho(s))$ and $r' = ren_\rho(r)$. Given a term t_1 and one of its renamings $t'_1 = ren_\rho(t_1)$, if $t'_1 \to_{p',R'} t'_2$ then $t_1 \to_{p,R} t_2$ where p is the corresponding position of p' in t'_1 and $t'_2 = ren_\rho(t_2)$.*

Proof. Immediate by definition of ren_ρ. □

The following proposition is the key to prove the soundness of NN-PE.

Proposition 23 *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, S a finite set of operation-rooted terms, ρ an independent renaming of S, and $\mathcal{R}'$ a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e' \to^* true$ in $\mathcal{R}'$ then $e \to^* true$ in $\mathcal{R}$.*

Proof. We prove the claim by induction on the number n of rewrite steps in $e' \to^* true$ (considering e' an arbitrary S'-closed expression).

Base case. If $n = 0$, we have $e' = true$ and the claim follows trivially since $ren_\rho(true) = true$ by definition.

Inductive case. Let us consider a rewrite sequence of the form $e' \to_{p',R'} h' \to^* true$, with $R' = (l' \to r')$. By definition of NN-PE, R' has been obtained by applying the post-processing renaming to a rule $R = (\theta(s) \to r)$ in the pre-NN-PE, where θ is constructor, $l' = \theta(\rho(s))$, and $r' = ren_\rho(r)$. By Lemma 22, we have $e \to_{p,R} h$ where p is the corresponding position of p' in e' and $h' = ren_\rho(h)$. By definition of pre-NN-PE, there exists a needed narrowing derivation $s \leadsto^+_\theta r$ which produced the resultant R. Since $e \to_{p,R} h$, we have $e \to^+ h$ in $\mathcal{R}$ by Lemma 21.

Since the terms in S' are linear and $\mathcal{R}'$ is S'-closed, h' is trivially S'-closed. By applying the inductive hypothesis to the subderivation $h' \to^* true$ in $\mathcal{R}'$, there exists a sequence $h \to^* true$ in $\mathcal{R}$. Together with the initial sequence $e \to^* h$ we get the desired derivation in $\mathcal{R}$.

□

Now we state and prove the soundness of NN-PE.

Theorem 24 *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e' \leadsto^*_{\sigma'} true$ is a needed narrowing derivation for e' in $\mathcal{R}'$, then there exists a needed narrowing derivation $e \leadsto^*_\sigma true$ in $\mathcal{R}$ such that $\sigma \leq \sigma'$ $[V]$.*

Proof. Since $e' \leadsto^*_{\sigma'} true$ in $\mathcal{R}'$ and $\mathcal{R}'$ is inductively sequential (Theorem 17), by the soundness of needed narrowing (claim 1 of Theorem 4), we have $\sigma'(e') \to^* true$. Since e' is S'-closed and σ' is constructor, by definition of closedness, $\sigma'(e')$ is also S'-closed and $\sigma'(e') = ren_\rho(\sigma'(e))$. By Proposition 23, there exists a rewrite sequence $\sigma'(e) \to^* true$ in $\mathcal{R}$. Therefore, by the completeness of needed narrowing (claim 2 of Theorem 4), there exists a needed narrowing derivation $e \leadsto^*_\sigma true$ in $\mathcal{R}$ such that $\sigma \leq \sigma'$ $[V]$, which completes the proof. □

6.2 Completeness

Firstly, we consider the notions of descendants and traces. Let $A = (t \to_{u,l\to r} t')$ be a reduction step of some term t into t' at position u with rule $l \to r$. The set of *descendants* [31] of a position v of t by A, denoted $v\backslash A$, is

$$v\backslash A = \begin{cases} \varnothing & \text{if } u = v, \\ \{v\} & \text{if } u \not\leq v, \\ \{u.p'.q \mid r|_{p'} = x\} & \text{if } v = u.p.q \text{ and } l|_p = x, \text{ where } x \in \mathcal{X}. \end{cases}$$

The set of *traces* of a position v of t by A, denoted $v\backslash\!\backslash A$ is

$$v\backslash\!\backslash A = \begin{cases} \{v\} & \text{if } u = v, \\ \{v\} & \text{if } u \not\leq v, \\ \{u.p'.q \mid r|_{p'} = x\} & \text{if } v = u.p.q \text{ and } l|_p = x, \text{ where } x \in \mathcal{X}. \end{cases}$$

The set of descendants of a position v by a reduction sequence B is defined inductively as follows

$$v\backslash B = \begin{cases} \{v\} & \text{if } B \text{ is the null derivation,} \\ \displaystyle\bigcup_{w \in v\backslash B'} w\backslash B'' & \text{if } B = B'B'', \text{ where } B' \text{ is the initial step of } B. \end{cases}$$

Given a set of positions P, we let $P\backslash B = \bigcup_{p \in P} p\backslash B$. The definition of the set of traces of a position by a reduction sequence is perfectly analogous.

A redex s in a term t is root-needed, if s (itself or one of its descendants) is contracted in every rewrite sequence from t to a root-stable term [42].

In the remainder of this section, we consider *outermost-needed* reduction sequences as defined[5] in [9].

Definition 25 ([7]) *Let $\mathcal{R}$ be an inductively sequential program. The (partial) function φ takes arguments $t = f(\overline{t})$ for a given $f \in \mathcal{F}$, and a definitional tree[6] $\mathcal{P}$ such that $pattern(\mathcal{P}) \leq t$, and yields a redex occurrence $p \in \mathcal{P}os_{\mathcal{R}}(t)$ called an* outermost-needed *redex:*

$$\varphi(t, \mathcal{P}) = \begin{cases} \Lambda & \text{if } \mathcal{P} = \{\pi\} \\ \varphi(t, \mathcal{P}_i) & \text{if } \mathcal{P} = branch(\pi, p, \mathcal{P}_1, \ldots, \mathcal{P}_n) \\ & \quad \text{and } pattern(\mathcal{P}_i) \leq t \text{ for some } i,\ 1 \leq i \leq n \\ p.\varphi(t|_p, \mathcal{P}_g) & \text{if } \mathcal{P} = branch(\pi, p, \mathcal{P}_1, \ldots, \mathcal{P}_n), \\ & \quad root(t|_p) = g \in \mathcal{F}, \text{ and} \\ & \quad \mathcal{P}_g \text{ is a definitional tree for } g. \end{cases}$$

The following technical results are auxiliary.

Lemma 26 [42] *Let $\mathcal{R}$ be an almost orthogonal TRS. If t is root-stable and $s \to^*_{>\Lambda} t$, then s is root-stable.*

[5] This is a slightly different though equivalent definition, since we do not allow for *exempt* nodes, as in [7].

[6] In this definition, we write $branch(\pi, p, \mathcal{P}_1, \ldots, \mathcal{P}_n)$ for a definitional tree $\mathcal{P}$ with pattern π if π is a branch with inductive position p and children $\pi_1, \ldots, \pi_n$ where $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$, $i = 1, \ldots, n$.

Theorem 27 *Let $\mathcal{R}$ be an inductively sequential program and t be a non-root-stable term. Every outermost-needed redex is root-needed.*

Proof. By Theorem 18 in [28], outermost-needed redexes are addressed by strong indices. By Theorem 5.6 in [41], nv-indices (hence strong indices, see [45]) in non-root-stable terms address root-needed redexes. □

Theorem 28 *Let $\mathcal{R}$ be a weakly orthogonal CB-TRS and t be a term. Let $P = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}os(t)$ be a set of disjoint positions of t such that each $t|_{p_i}$ for $1 \leq i \leq n$ is operation-rooted. If t admits a root-normalizing derivation which does not root-normalize any $t|_{p_i}$, then t admits a root-normalizing derivation which does not reduce any $t|_{p_i}$.*

Proof. If t is root-stable, the result is immediate. If t is not root-stable, then there exists a root-stable reduct $\sigma(r)$ and a derivation $A : t \to^* \sigma(l) \to_\Lambda \sigma(r)$ for some rule $l \to r$ in $\mathcal{R}$ which, by hypothesis, root-normalizes t without root-normalizing any $t|_{p_i}$. Let $\overline{y_n} = y_1, \ldots, y_n$ be new, distinct variables each of which is used to name a subterm $t|_{p_i}$. The substitution θ_t defined by $\theta_t(y_i) = t|_{p_i}$ associates a subterm to each variable. Note that $\theta_t(t[\overline{y_n}]_P) = t$. As an intermediate step of the demonstration, first we prove, by induction on the length $N+1$ of the derivation A, that there exists a substitution σ' such that $t[\overline{y_n}]_P \to^* \sigma'(l)$ and $\theta_t(\sigma'(x)) \to^* \sigma(x)$ for all $x \in \mathcal{V}ar(l)$.

First we note that, since the derivation A does not root-normalize any $t|_{p_i}$, we have that $p_i > \Lambda$ for every $1 \leq i \leq n$ (otherwise $P = \{\Lambda\}$ and we obtain a contradiction with the initial hypothesis).

1. If $N = 0$, then $t = \sigma(l)$. Since $p_i > \Lambda$, $t|_{p_i}$ is operation-rooted for $1 \leq i \leq n$, and $\mathcal{R}$ is constructor based, then for each p_i there exists a variable position $v_i \in \mathcal{P}os(l)$ such that $p_i = v_i.w_i$ and $l|_{v_i}$ is a variable. Then, for each $x \in \mathcal{V}ar(l)$, we let $\sigma'(x) = t[\overline{y_n}]_P|_{v_x}$ where v_x is the position of x in l. Hence, $t[\overline{y_n}]_P = \sigma'(l)$ and $\theta_t(\sigma'(x)) = \theta_t(t[\overline{y_n}]_P|_{v_x}) = \sigma(x)$ for each $x \in \mathcal{V}ar(l)$. Thus, $\theta_t(\sigma'(x)) \to^* \sigma(x)$.

2. If $N > 0$, then we consider the derivation $t \to_q t' \to^* \sigma(l)$. Let $P' = q \backslash\backslash P = \{p'_1, \ldots, p'_{n'}\}$ be the traces of P w.r.t. the rewriting step $t \to_q t'$ (note that the traces are well defined since every $t|_{p_i}$ is operation-rooted). By hypothesis, the derivation $t \to^* \sigma(l) \to \sigma(r)$ does not root-normalize any $t|_p$ for $p \in P$. In particular, the step $t \to_q t'$ does not root-normalize any $t|_p$ for $p \in P$. Therefore, each $t'|_{p'}$ for every $p' \in P'$ is operation-rooted and the derivation $t' \to^* \sigma(l) \to \sigma(r)$ does not root-normalize any $t'|_{p'}$ for $p' \in P'$. Thus, by the induction hypothesis, $t'[\overline{z_{n'}}]_{P'} \to^* \sigma'(l)$ and $\theta_{t'}(\sigma'(x)) \to^* \sigma(x)$ for all $x \in \mathcal{V}ar(l)$ where $\overline{z_{n'}} = z_1, \ldots, z_{n'}$ are new, distinct variables which identify the subterms in t' addressed by P', i.e., $\theta_{t'}(z_i) = t'|_{p'_i}$ for $1 \leq i \leq n'$. We connect variables in $\overline{z_{n'}}$ and variables in $\overline{y_n}$ by means of a substitution $\tau : \overline{z_{n'}} \to \overline{y_n}$ as follows: $\tau(z_i) = y_j$ iff p'_i is a trace of p_j (w.r.t. the step $t \to t'$) for $1 \leq i \leq n'$ and $1 \leq j \leq n$. Now we consider two cases:

   (a) If there is no $p \in P$ such that $p \leq q$, then, since each $t|_{p_i}$ is operation-rooted and $\mathcal{R}$ is constructor-based, we have that $t[\overline{y_n}]_P \to_q \tau(t'[\overline{z_{n'}}]_{P'})$. Moreover, since no $t|_{p_i}$ changes in this rewriting step, we have $\theta_t(\tau(z)) = \theta_{t'}(z)$ for all $z \in \overline{z_{n'}}$, i.e., $\theta_{t'} = \theta_t \circ \tau$. Since $t'[\overline{z_{n'}}]_{P'} \to^* \sigma'(l)$, by stability, $\tau(t'[\overline{z_{n'}}]_{P'}) \to^* \tau(\sigma'(l))$. Thus, $t[\overline{y_n}]_P \to^* \tau(\sigma'(l))$. Since $\theta_{t'}(\sigma'(x)) = \theta_t(\tau(\sigma'(x))) \to^* \sigma(x)$, the conclusion follows.

(b) If there is $p \in P$ such that $p \leq q$, then $P' = P$, $n = n'$ and we can take $\overline{y_n} = \overline{z_{n'}}$. Hence, $t[\overline{y_n}]_P = t'[\overline{y_n}]_P = t'[\overline{z_{n'}}]_{P'}$. Now we have that $\theta_t(y_i) \to \theta_{t'}(y_i)$ if $p = p_i$, for some $1 \leq i \leq n$ whereas $\theta_t(y_j) = \theta_{t'}(y_j)$ for all $j \neq i$, and the conclusion also follows.

Since $\theta_t(\sigma'(x)) \to^* \sigma(x)$ for all $x \in \mathcal{V}ar(l)$, we consider two possibilities:

1. If $r \notin \mathcal{X}$, then, since $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, we have that $\theta_t(\sigma'(r)) \to^*_{>\Lambda} \sigma(r)$.

2. If $r = x \in \mathcal{X}$, then, we prove that this implies that $\theta_t(\sigma'(r)) = \theta_t(\sigma'(x)) \to^*_{>\Lambda} \sigma(x) = \sigma(r)$. Otherwise, it is necessary that $\sigma'(x)$ be a variable. In this case, it must be $\sigma'(x) = y_i$ for some $1 \leq i \leq n$ (otherwise, $\theta_t(\sigma'(x)) = \sigma'(x)$ is a variable and it cannot be rewritten to $\sigma(x)$ in zero or more steps unless $\sigma'(x) = \sigma(x)$ in which case, we trivially have that $\theta_t(\sigma'(x)) \to^*_{>\Lambda} \sigma(x)$). Since $\sigma(r) = \sigma(x)$ is root-stable, the existence of the derivation $\theta_t(\sigma'(x)) = \theta_t(y_i) \to^* \sigma(x)$ implies (since each reduction step in the derivation $\theta_t(\sigma'(x)) \to^* \sigma(x)$ has been taken from the derivation A) that the derivation A root-normalizes the subterm $\theta_t(y_i) = t|_{p_i}$. This contradicts our initial hypothesis.

Thus, in all cases, we have that $\theta_t(\sigma'(r)) \to^*_{>\Lambda} \sigma(r)$ and, since $\sigma(r)$ is root-stable, by Lemma 26 (remember that weak orthogonality and almost orthogonality coincide for CB-TRSs), $\theta_t(\sigma'(r))$ is root-stable. Note that we have also proved that $t[\overline{y_n}]_P \to^* \sigma'(l) \to \sigma'(r)$ and therefore, by stability, $t = \theta_t(t[\overline{y_n}]_P) \to^* \theta_t(\sigma'(r))$ is a root-normalizing derivation for t which does not reduce any $t|_{p_i}$ for $1 \leq i \leq n$. □

Theorem 29 *Let $\mathcal{R}$ be a weakly orthogonal CB-TRS and t be a term. Let $P = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}os(t)$ be a set of disjoint positions of t such that each $t|_{p_i}$ for $1 \leq i \leq n$ is a root-stable, operation-rooted term. If t is root-normalizing, then t admits a root-normalizing derivation which does not reduce any $t|_{p_i}$.*

Proof. The proof is perfectly analogous to the proof of Theorem 28. Assume the same notations for the proof. Only one difference arises in the last part of the proof: we need not distinguish the cases $r \in \mathcal{X}$ and $r \notin \mathcal{X}$. This is because the fact that each $t|_{p_i}$ is root-stable and the fact that $\theta_t(\sigma'(x)) \to^* \sigma(x)$ for all $x \in \mathcal{V}ar(l)$ allows us to immediately derive that $\theta_t(\sigma'(x)) \to^*_{>\Lambda} \sigma(x)$. Now, it suffices to consider that every $t|_{p_i}$ and their possible reducts are operation-rooted, which easily follows from the fact that each $t|_{p_i}$ is root-stable and operation-rooted. □

Theorem 30 *Let $\mathcal{R}$ be a weakly orthogonal CB-TRS, t be a term, and $p \in \mathcal{P}os(t)$. Let s be a root-stable, operation-rooted subterm of t. If t is root-normalizing, then s does not have redexes which are root-needed in t.*

Proof. Immediate, by using Theorem 29. □

Theorem 31 *Let $\mathcal{R}$ be a weakly orthogonal CB-TRS and t be a term. If s is an operation-rooted subterm of t that contains a redex which is root-needed in t, then every root-needed redex in s is root-needed in t.*

Proof. Since t contains at least a root-needed redex, then t is not root-stable. If t has no root-stable form, it is trivial, since every redex is root-needed in t. Hence, we assume that t has a root-stable reduct. Let $s|_q$ be a root-needed redex in s. Then, s is not root-stable. Let $s|_{q'}$ be a root-needed redex in t. If $s|_q$ is not root-needed in t, then it is possible to root-normalize t without reducing the redex $s|_q$. However, without reducing the redex $s|_q$ it is not possible to root-normalize s. Therefore, it is possible to root-normalize t without root-normalize s. By Theorem 28, it is possible to root-normalize t without reducing s, hence without reducing $s|_{q'}$, which yields a contradiction. $\square$

The following auxiliary definition is useful to deal with closed terms (it is a slight refinement of the same notion in [5]).

Definition 32 (covering set, closure set) *Let S be a finite set of terms and t be an S-closed term. We define the covering set of t w.r.t. S as follows:*

$$CSet(S,t) = \{O \mid O \in c\_set(S,t),\ (u.0, fail) \notin O,\ u \in I\!N^*\}$$

where the auxiliary function c_set, used to compute each closure set O, is defined inductively as follows:

$$c\_set(S,t) \ni \begin{cases} \varnothing & \text{if } t \in \mathcal{X} \cup \mathcal{C}, \\ \bigcup_{i=1}^{n} \{(i.p, s) \mid (p,s) \in c\_set(S, t_i)\} & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C} \cup \{\approx, \wedge\}, \\ \{(\Lambda, s)\} \cup \{(q.p, s') \mid s|_q \in \mathcal{X},\ (p, s') \in c\_set(S, \theta(s|_q))\} & \text{if } \exists \theta, \exists s \in S.\ \theta(s) = t, \\ \{(0, fail)\} & \text{otherwise.} \end{cases}$$

Note that positions ending with the mark '0' identify the situation in which some subexpression of t is not an instance of any of the terms in S. Thus, a set containing a pair of the form $(u.0, fail)$ is not considered a closure set.

Roughly speaking, given a set of terms S and a term t which is S-closed, each set in $CSet(S,t)$ identifies a concrete way in which t can be proven S-closed, thus avoiding the non-determinism which is implicit in the definition of closedness.

The following *lifting* lemma is a slight variant of the completeness result for needed narrowing.

Lemma 33 *Let $\mathcal{R}$ be an inductively sequential program. Let σ be a constructor substitution, V a finite set of variables, and s an operation-rooted term with $\mathcal{V}ar(s) \subseteq V$. If $\sigma(s) \to_{p_1,R_1} \cdots \to_{p_n,R_n} t$ is an outermost-needed reduction sequence, then there exists a needed narrowing derivation $s \leadsto_{p_1,R_1,\sigma_1} \cdots \leadsto_{p_n,R_n,\sigma_n} t'$ and a constructor substitution σ' such that $\sigma'(t') = t$ and $\sigma' \circ \sigma_n \circ \cdots \circ \sigma_1 = \sigma\ [V]$.*

Proof. It is perfectly analogous to the proof of Theorem 4 (completeness) in [9]. □

Now we prove two technical results which are necessary for a useful generalization of the lifting lemma. We need to make the lemma applicable even when the considered substitution is not constructor, as long as it still does not introduce a needed redex. In order to do this extension, we need to ensure that it is possible to get rid of some operation-rooted subterms which are introduced by instantiation whenever they are not contracted in the considered derivation. We prove this in the following lemmata.

Lemma 34 *Let $\mathcal{R}$ be program. Let t and s be operation-rooted terms and $P_0 \subseteq \mathcal{P}os(t)$ be a nonempty set of disjoint positions such that $t|_p = s$ for all $p \in P_0$. Let*

$$t[s,\ldots,s]_{P_0} = t_0 \to_{p_1,R_1} \cdots \to_{p_n,R_n} t_n = t'[s,\ldots,s]_{P_n}$$

be a reduction sequence where $A_i = (t_{i-1} \to_{p_i,R_i} t_i)$ and $P_i = P_{i-1}\backslash A_i$ for all $i = 1,\ldots,n$, $n \geq 0$. If $p \not\leq p_i$ for all $p \in P_{i-1}$, $i = 1,\ldots,n$, then there exists a reduction sequence

$$t[x,\ldots,x]_{P_0} \to_{p_1,R_1} \cdots \to_{p_n,R_n} t'[x,\ldots,x]_{P_n}$$

Proof. By induction on the number n of steps in the former reduction:

$n = 0$. Trivial.

$n > 0$. Consider $A_1 = (t[s,\ldots,s]_{P_0} \to_{p_1,R_1} t''[s,\ldots,s]_{P_1})$, where $R_1 = (l_1 \to r_1)$, $\sigma_1(l_1) = t|_{p_1}$, and $P_1 = P_0\backslash A_1$. We distinguish two cases depending on the relative position of p_1 (the case $p \leq p_1$, for some $p \in P_0$, is not considered since the subterms in s are not contracted, i.e., $p \not\leq p_1$ for all $p \in P_0$):

$\forall p \in P_0.\ p_1 \perp p$. In this case, we have that $\sigma_1(l_1) = (t[x,\ldots,x]_{P_0})|_{p_1}$ and, by definition of descendant, $P_0 = P_1$. Therefore $t[x,\ldots,x]_{P_0} \to_{p_1,R_1} t''[x,\ldots,x]_{P_0}$, and the claim follows by applying the inductive hypothesis to the sequence

$$B = (t''[s,\ldots,s]_{P_0} \to_{p_2,R_2} \cdots \to_{p_n,R_n} t'[s,\ldots,s]_{P_n}),$$

where $P_n = P_0\backslash B$.

$\exists p \in P_0.\ p_1 < p$. Since s is operation-rooted and l_1 is a linear pattern, then there exists a substitution σ_1' such that $\sigma_1'(l_1) = (t[x,\ldots,x]_{P_0})|_{p_1}$ (i.e., $\{x \mapsto s\} \circ \sigma_1' = \sigma_1$). Therefore, the following reduction step can be proven:

$$t[x,\ldots,x]_{P_0} \to_{p_1,R_1} t''[x,\ldots,x]_{P_1}$$

and the claim follows by applying the inductive hypothesis to

$$B = (t''[s,\ldots,s]_{P_1} \to_{p_2,R_2} \cdots \to_{p_n,R_n} t'[s,\ldots,s]_{P_n}),$$

where $P_n = P_1\backslash B$.

□

Lemma 35 *Let $\mathcal{R}$ be a program. Let $\theta = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$ be an idempotent substitution such that s_i is an operation-rooted term for all $i = 1, \ldots, m$. Let s be an operation-rooted term and $\theta(s) = t_0 \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} t_n$ be a reduction sequence where $A_i = (t_{i-1} \rightarrow_{p_i,R_i} t_i)$ and $P_i = P_{i-1} \backslash A_i$, for $i = 1, \ldots, n$, $n \geq 0$. If $p \not\leq p_i$ for all $p \in P_{i-1}$, $i = 1, \ldots, n$, then there exists a reduction sequence $s \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} s'$ such that $\theta(s') = t_n$.*

Proof. By induction on the number m of bindings in θ:

Base case. Consider $\theta = \{x_1 \mapsto s_1\}$. Then $\theta(s) = t_0[s_1, \ldots, s_1]_P$ and $s = t_0[x_1, \ldots, x_1]_P$, where $P = \{p \in \mathcal{P}os(s) \mid s|_p = x_1\}$. Then, the claim follows directly by Lemma 34.

Induction step. Consider $\theta = \theta_1 \cup \theta'$, where $\theta_1 = \{x_1 \mapsto s_1\}$ and $\theta' = \{x_2 \mapsto s_2, \ldots, x_m \mapsto s_m\}$. Then, $\theta(s) = t_0[s_1, \ldots, s_1]_P$ and $\theta'(s) = t_0[x_1, \ldots, x_1]_P$, where $P = \{p \in \mathcal{P}os(s) \mid s|_p = x_1\}$. Applying Lemma 34, we have that $t_0[x_1, \ldots, x_1]_P \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} s''$ is a reduction sequence such that $\theta_1(s'') = t_n$. By applying the inductive hypothesis to this derivation, we have that $s \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} s'$ is a reduction sequence such that $\theta'(s') = s''$. Therefore, since $\mathcal{D}om(\theta_1) \cap \mathcal{D}om(\theta') = \varnothing$, we get $\theta(s') = (\theta_1 \circ \theta')(s') = \theta_1(s'') = t_n$, which proves the claim.

□

Now we are ready to extend the lifting lemma for needed narrowing (Lemma 33) to non-constructor substitutions which do not introduce needed redexes.

Theorem 36 *Let $\mathcal{R}$ be an inductively sequential program. Let σ be a substitution and V a finite set of variables. Let s be an operation-rooted term and $\mathcal{V}ar(s) \subseteq V$. Let $\sigma(s) \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} t$ be an outermost-needed rewrite sequence such that, for all root-needed redex $\sigma(s)|_p$ of $\sigma(s)$, $p \in \mathcal{NVP}os(s)$. Then, there exists a needed narrowing derivation $s \leadsto_{p_1,R_1,\sigma_1} \cdots \leadsto_{p_n,R_n,\sigma_n} t'$ and a substitution σ' such that $\sigma'(t') = t$ and $\sigma' \circ \sigma_n \circ \cdots \circ \sigma_1 = \sigma$ $[V]$.*

Proof. We consider two cases:

Let σ be a constructor substitution. In this case, the claim follows directly by Lemma 33, and σ' is a constructor substitution too.

Let σ be a non-constructor substitution. Then, there exist substitutions θ_1 and θ_2 such that $\sigma = \theta_2 \circ \theta_1$, the substitution θ_1 is constructor, and for all $x \mapsto s' \in \theta_2$, s' is operation-rooted. Then $\sigma(s) = \theta_2(\theta_1(s))$. By applying Lemma 35, we have $\theta_1(s) \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} s''$ such that $\theta_2(s'') = t$. On the other hand, since σ does not introduce root-needed redexes (i.e., if $\sigma(s)|_p$ is a root-needed redex then $p \in \mathcal{NVP}os(s)$), then the sequence is an outermost-needed derivation. Now, applying Lemma 33 to this reduction sequence, there exists a needed narrowing derivation $s \leadsto_{p_1,R_1,\sigma_1} \cdots \leadsto_{p_n,R_n,\sigma_n} t'$ and a constructor substitution σ'' such that $\sigma''(t') = s''$ and $\sigma'' \circ \sigma_n \circ \cdots \circ \sigma_1 = \theta_1$ $[V]$. By taking $\sigma' = \theta_2 \circ \sigma''$, we have $\sigma'(t') = (\theta_2 \circ \sigma'')(t') = \theta_2(\sigma''(t')) = \theta_2(s'') = t$. Finally, since $\sigma'' \circ \sigma_n \circ \cdots \circ \sigma_1 = \theta_1$ $[V]$, we have $\theta_2 \circ \sigma'' \circ \sigma_n \circ \cdots \circ \sigma_1 = \theta_2 \circ \theta_1$ $[V]$, and hence $\sigma' \circ \sigma_n \circ \cdots \circ \sigma_1 = \sigma$ $[V]$, which completes the proof.

□

The next lemma establishes a strong correspondence between the closedness of an expression t and that of one renaming of t.

Lemma 37 *Let S be a finite set of terms, ρ an independent renaming of S, and $S' = \rho(S)$. Given a term t, $ren_\rho(t)$ is S'-closed iff t is S-closed.*

Proof. By induction on the structure of the terms. □

The following lemma states that, if some term t has an operation-rooted subterm s that contains a redex which is root-needed in t, then the outermost-needed redex in s is also root-needed in t.

Lemma 38 *Let $\mathcal{R}$ be an inductively sequential program and t be a term. If s is an operation-rooted subterm of t which contains a root-needed redex in t, then every outermost-needed redex in s is root-needed in t.*

Proof. Since t contains at least a root-needed redex, t is not root-stable. If t has no root-stable form, then every redex in t is root-needed. Therefore, we assume that t has a root-stable reduct. If s contains an outermost-needed redex, then, by hypothesis and by Theorem 30, s is not root-stable. Hence, by Theorem 27, such a redex is root-needed in s. By Theorem 31, the conclusion follows. □

The following lemma is helpful.

Lemma 39 *Let $\mathcal{R}$ be an inductively sequential program and t be a term. If s is an operation-rooted subterm of t which contains a root-needed redex in t and there is a subterm s' of s which does not contain any root-needed redex in t, then there is no outermost-needed derivation from s to a root-stable form which contracts any redex (or residual) in s'.*

Proof. If s is root-stable, the claim is trivially true. If s is not root-stable and there is an outermost-needed derivation starting from s which contracts a (residual of a) redex s'' in s', then, by Theorem 27 such a redex is root-needed in s. Therefore, by Theorem 31, s'' is root-needed in t, thus leading to a contradiction. □

Proposition 40 *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, S a finite set of operation-rooted terms, ρ an independent renaming of S, and $\mathcal{R}'$ a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e \to^* true$ in $\mathcal{R}$ then $e' \to^* true$ in $\mathcal{R}'$.*

Proof. Since e' is S'-closed, by Lemma 37, e is S-closed. Now we prove that, for any reduction sequence $e \to^* true$ in $\mathcal{R}$ for an S-closed term e (not necessarily an equation), there exists a reduction sequence $e' \to^* true$ in $\mathcal{R}'$, with $e' = ren_\rho(e)$. Let $B_1, \ldots, B_j$ be all possible needed reduction sequences from e to $true$, and let k_i be the number of contracted redexes in B_i, $i = 1, \ldots, j$. We prove the claim by induction on the maximum number $n = max(k_1, \ldots, k_j)$ of contracted needed redexes which are necessary to reduce e to $true$.

$n = 0$. This case is trivial since $e' = ren_\rho(true) = true$.

$n > 0$. Since e is S-closed, there exists a closure set $\{(p_1, s_1), \ldots, (p_m, s_m)\} \in CSet(S, e)$, $m > 0$, where $p_i \in \mathcal{P}os(e)$ and $s_i \in S$, $i = 1, \ldots, m$. Since e contains at least one needed redex, there exists some $i \in \{1, \ldots, m\}$ such that $e|_{p_i} = \theta(s_i)$ and the following facts hold:

- there exists at least one position $q \in \mathcal{P}os(e|_{p_i})$ such that $e|_{p_i.q}$ is a needed redex in e, and
- for all needed redex $e|_{p_i.q'}$ in e, we have $q' \in \mathcal{NVP}os(s_i)$.

Informally, p_i addresses an "innermost" subterm of e (according to the partition imposed by the closure set) in the sense that $e|_{p_i}$ contains at least one needed redex and there is no inner subterm $e|_{p_j}$, $p_i < p_j$, which contains needed redexes. Since both e and $e|_{p_i}$ are operation-rooted terms, by Lemma 38 we know that each outermost-needed redex in $e|_{p_i}$ is also a needed redex in e (note that, for derivations $e \to^* true$ in confluent TRSs, the notions of neededness and root-neededness coincide, since $true$ is the only root-stable form of e). Let us assume that q_1 is the position of such an outermost-needed redex. Since there is no inner subterm $e|_{p_j}$, $p_i < p_j$, which contains needed redexes in e, then by Lemma 39, we have that there is no inner subterm $e|_{p_j}$, $p_i < p_j$, which contains root-needed redexes in $e|_{p_i}$. Hence, we can consider a reduction sequence

$$e[\theta(s_i)]_{p_i} \to_{p_i.q_1,R_1} \ldots \to_{p_i.q_k,R_k} e[s_i']_{p_i} \to^* true$$

such that the corresponding sequence for $\theta(s_i)$

$$\theta(s_i) \to_{q_1,R_1} \ldots \to_{q_{k-1},R_{k-1}} s_i'' \to_{q_k,R_k} s_i'$$

is outermost-needed, s_i' is root-stable, and s_i'' is not root-stable, $k > 0$.

Now, we prove that s_i' is constructor-rooted. Assume that s_i' is operation-rooted. Then, since $t' = e[s_i']_{p_i}$ is root-normalizing, by Theorem 29, there exists a reduction sequence $t' \to^* true$ which does not reduce s_i'. Since s_i' is operation-rooted and $\mathcal{R}$ is constructor-based, then there exists a reduction sequence $e[x]_{p_i} \to^* true$, with $x \notin \mathcal{V}ar(e)$. Therefore, $e \to^* true$ without reducing $e|_{p_i}$, which contradicts the initial hypothesis that $e|_{p_i}$ contains a root-needed redex in e. Hence, s_i' is constructor-rooted.

Let V be a finite set of variables containing $\mathcal{V}ar(s_i)$. By Theorem 36, we know that there exists a needed narrowing derivation $s_i \leadsto_{q_1,R_1,\sigma_1} \ldots \leadsto_{q_k,R_k,\sigma_k} s_i''$ which contracts the same positions using the same rules and in the same order. By definition of NN-PE, some resultant of $\mathcal{R}'$ derives from a prefix of this needed narrowing derivation. Assume that the following subderivation

$$s_i \leadsto_{q_1,R_1,\sigma_1} \ldots \leadsto_{q_j,R_j,\sigma_j} t', \quad 0 < j \leq k$$

is the one which has been used to construct such a resultant. Let $\sigma'' = \sigma_j \circ \cdots \circ \sigma_1$. Since $\theta(s_i) \rightarrow_{q_1,R_1} \ldots \rightarrow_{q_j,R_j} t$, again by Theorem 36, there exists a substitution σ' such that $\sigma'(t') = t$ and $\sigma' \circ \sigma'' = \theta\ [V]$. Thus, the considered resultant has the form

$$R' = (\sigma''(\rho(s_i)) \rightarrow ren_\rho(t'))$$

and the considered reduction sequence in $\mathcal{R}$ has the form

$$e = e[\theta(s_i)]_{p_i} \rightarrow_{p_i.q_1,R_1} \ldots \rightarrow_{p_i.q_j,R_j} e[t]_{p_i} \rightarrow^* true$$

Now, we prove that e' can be reduced at position p'_i using R', where p'_i is the corresponding position of p_i in e after renaming. By construction, $\theta(x)$ is S-closed for all $x \in \mathcal{D}om(\theta)$. Moreover, since σ'' is constructor and $\sigma' \circ \sigma'' = \theta\ [V]$, we have that $\sigma'(x)$ is also S-closed for all $x \in \mathcal{D}om(\sigma')$. Then, there exists a substitution $\theta' = \{x \mapsto ren_\rho(\sigma'(x)) \mid x \in \mathcal{D}om(\sigma')\}$ such that $\theta'(x)$ is S'-closed for all $x \in \mathcal{D}om(\theta')$. By definition of post-processing renaming, $e'|_{p'_i} = ren_\rho(e|_{p_i}) = ren_\rho(\theta(s_i))$. Since $\mathcal{V}ar(s_i) = \mathcal{V}ar(\rho(s_i))$ and σ'' is constructor, we have $ren_\rho(\theta(s_i)) = ren_\rho(\sigma' \circ \sigma''(s_i)) = \theta'(\sigma''(ren_\rho(s_i))) = \theta'(\sigma''(\rho(s_i)))$. Therefore, the following rewrite step can be proven

$$e'|_{p'_i} = \theta'(\sigma''(\rho(s_i))) \rightarrow_{\Lambda,R'} \theta'(ren_\rho(t')) = ren_\rho(\sigma'(t')) = ren_\rho(t)$$

and thus $e' \rightarrow_{p'_i,R'} e'[ren_\rho(t)]_{p'_i}$. Then, it is immediate to see that $e'[ren_\rho(t)]_{p'_i} = ren_\rho(e[t]_{p_i})$.

Let us now consider the S-closedness of $e[t]_{p_i}$. Since $\mathcal{R}'$ is S'-closed, $ren_\rho(t')$ is also S'-closed. By Lemma 37, t' is S-closed. Since $\sigma'(x)$ is S-closed for all $x \in \mathcal{D}om(\sigma')$, by definition of closedness, $\sigma'(t') = t$ is also S-closed. Now we distinguish two cases:

$p_i = \Lambda$. Then $e[t]_{p_i}$ is trivially S-closed since t is S-closed.

$p_i \neq \Lambda$. Let $j \in \{1, \ldots, m\}$ such that $p_j < p_i$ and there is no $k \in \{1, \ldots, m\}$ with $p_j < p_k < p_i$. Let $e|_{p_j} = \gamma(s_j)$ where $y \mapsto s_i \in \gamma$, and consider the set $P_y = \{p_j.q \in \{p_1, \ldots, p_m\} \mid s_j|_q = y\}$. Now we have two possibilities:

P_y is a singleton. Then $e[t]_{p_i}$ is trivially S-closed, since $(p_i, s_i) \in CSet(S, e)$ and t is S-closed.

P_y is not a singleton. In this case, we have $e = e[\theta(s_i), \ldots, \theta(s_i)]_{P_y}$. By considering again the reduction sequences $\theta(s_i) \rightarrow^* t$ for each s_i, we get

$$e[\theta(s_i), \ldots, \theta(s_i)]_{P_y} \rightarrow \cdots \rightarrow e[t, \ldots, t]_{P_y}$$

and, by definition of closedness, it is immediate to see that $e[t, \ldots, t]_{P_y}$ is S-closed. Moreover, we can construct the following reduction sequence:

$$e'[\theta'(\sigma''(\rho(s_i))), \ldots, \theta'(\sigma''(\rho(s_i)))]_{P'_y} \rightarrow \ldots \rightarrow e'[ren_\rho(t), \ldots, ren_\rho(t)]_{P'_y}$$

where P'_y corresponds to the positions of P_y in e after renaming. Then, we have

$$e'[ren_\rho(t), \ldots, ren_\rho(t)]_{P'_y} = ren_\rho(e[t, \ldots, t]_{P_y}).$$

Putting all pieces together, we conclude that there exists a reduction sequence

$$e \to^{+} e[t, \ldots, t]_P \to^{*} true$$

in $\mathcal{R}$, where $P = \{p_i\}$ or $P = P_y$, such that there exists a reduction sequence

$$e' \to^{+} e'[ren_\rho(t), \ldots, ren_\rho(t)]_{P'}$$

in $\mathcal{R}'$, where $P' = \{p'_i\}$ or $P' = P'_y$, respectively. Since $e \to^{+} e[t, \ldots, t]_P$ has reduced at least one needed redex in e and $e'[ren_\rho(t), \ldots, ren_\rho(t)]_{P'} = ren_\rho(e[t, \ldots, t]_P)$, by applying the induction hypothesis to $e[t, \ldots, t]_P \to^{*} true$ in $\mathcal{R}$, we get

$$e'[ren_\rho(t), \ldots, ren_\rho(t)]_{P'} \to^{*} true$$

in $\mathcal{R}'$. By composing this sequence with the previous sequence

$$e' \to^{+} e'[ren_\rho(t), \ldots, ren_\rho(t)]_{P'}$$

we get the desired result.

$\square$

The completeness of NN-PE is a direct consequence of the previous proposition and the soundness and completeness of needed narrowing.

Theorem 41 (completeness) *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e \leadsto^{*}_{\sigma} true$ is a needed narrowing derivation for e in $\mathcal{R}$, then there exists a needed narrowing derivation $e' \leadsto^{*}_{\sigma'} true$ in $\mathcal{R}'$ such that $\sigma' \leq \sigma$ $[V]$.*

Proof. Since $e \leadsto^{*}_{\sigma} true$, by the soundness of needed narrowing (claim 1 of Theorem 4), we have $\sigma(e) \to^{*} true$. Since e' is S'-closed and σ is constructor, by definition of closedness, $\sigma(e')$ is also S'-closed and $\sigma(e') = ren_\rho(\sigma(e))$. By Proposition 40, there exists a rewrite sequence $\sigma(e') \to^{*} true$ in $\mathcal{R}'$. Therefore, since σ is a solution of e' in $\mathcal{R}'$ and $\mathcal{R}'$ is inductively sequential (Theorem 17), by the completeness of needed narrowing (claim 2 of Theorem 4), there exists a needed narrowing derivation $e' \leadsto^{*}_{\sigma'} true$ such that $\sigma' \leq \sigma$ $[V]$. $\square$

6.3 Strong Correctness

Finally, the strong correctness of the transformation can be easily proved as a direct consequence of Theorems 24 and 41, together with the independence of solutions computed by needed narrowing.

Theorem 20 (strong correctness) *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. Then, $e \leadsto^*_\sigma true$ is a needed narrowing derivation for e in $\mathcal{R}$ iff there exists a needed narrowing derivation $e' \leadsto^*_{\sigma'} true$ in $\mathcal{R}'$ such that $\sigma' = \sigma\ [V]$ (up to renaming).*

Proof. We consider the two directions separately:

Strong soundness. We prove the claim by contradiction. Assume that there exists some substitution σ' computed by needed narrowing for e' in $\mathcal{R}'$ such that there is no substitution θ computed by needed narrowing for e in $\mathcal{R}$ with $\theta = \sigma'\ [V]$ (up to renaming).

By Theorem 24 (soundness of NN-PE) and the assumption above, we conclude that there must be some substitution σ computed by needed narrowing for e in $\mathcal{R}$ such that $\sigma < \sigma'\ [V]$. Then, by Theorem 41, there exists a substitution θ' computed by needed narrowing for e' in $\mathcal{R}'$ such that $\theta' \leq \sigma\ [V]$. Since $\theta' \leq \sigma\ [V]$ and $\sigma < \sigma'\ [V]$, we have $\theta' < \sigma'\ [V]$ which contradicts the independence of solutions computed by needed narrowing (claim 3 of Theorem 4).

Strong completeness. The proof is perfectly analogous, by considering the completeness of NN-PE (Theorem 41) in the place of the soundness of NN-PE (Theorem 24).

□

7 Experimental Results

In this section, we report on some experiments which highlight the practical advantages of our approach and demonstrate that NN-PE can not only produce better specialized programs in comparison with lazy narrowing, but it also leads to better specialization times.

A partial evaluator for functional logic programs based on needed narrowing as well as on lazy narrowing has been implemented in the INDY system[7] [2] in order to compare the run time of the partial evaluator and the effects of both narrowing strategies on the specialized programs.

We have measured the improvements by some experiments which we summarize in Tables 1 and 2. Here we have benchmarked the speed and specialization achieved by our

[7]The INDY system gives the user the choice of the narrowing strategy as well as the unfolding rule which controls the construction of the computation trees and which ensures the finiteness of the unfolding process. In the experiments, we use the homeomorphic embedding ordering on comparable ancestors of selected redexes. This expands derivations while new redexes are less than previous *comparable* redexes (i.e., with the same root function symbol) appeared in the same branch (using the homeomorphic embedding ordering).

	Original	LN-PE		NN-PE		
Benchmarks	Size	LN-Size	LN-St	NN-Size	NN-St	Speedup
`ackermann`	4	20	2690	17	1370	1.96
`allones`	6	4	140	4	80	1.75
`applast`	5	4	340	4	190	1.78
`exam`	5	5	180	3	80	2.25
`fibonacci`	5	15	960	15	730	1.31
`kmp`	12	14	1290	14	1100	1.17
`palindrome`	12	19	1810	19	1400	1.29
`sumprod`	8	18	1110	18	880	1.26
`matmult`	10	24	1610	24	1190	1.35
`sumleq`	7	6	92	6	50	1.85

Table 1: NN-PE *vs.* LN-PE: size of specialized code and specialization times (in ms.)

Goal:	LN-Rt	NN-Rt	Speedup
$\mathtt{ackermann(5)} \leq (\mathtt{5}+\mathtt{5}) \approx \mathtt{true}$	83	67	1.24
$(\mathtt{20}-\mathtt{X})+((\mathtt{20}-\mathtt{X})+(\mathtt{20}-\mathtt{X})) \leq \mathtt{40}+\mathtt{40} \approx \mathtt{true}$	43	15	2.87
$(\mathtt{20}+\mathtt{Y})+(\mathtt{Y}+\mathtt{20}) \leq \mathtt{20}+\mathtt{20} \approx \mathtt{true}$	73	37	1.97
$\mathtt{10}+\mathtt{X} \leq (\mathtt{X}+\mathtt{2})+\mathtt{X} \approx \mathtt{true}$	22	6.7	3.28
$(\mathtt{X}-\mathtt{10})+((\mathtt{X}-\mathtt{10})+(\mathtt{X}-\mathtt{10})) \leq \mathtt{20}+\mathtt{20} \approx \mathtt{true}$	87	22	3.95

Table 2: NN-PE *vs.* LN-PE: relative runtimes

implementation (including size and execution time of specialized code). Times were measured on a HP 712/60 workstation, running under HP Unix v10.01. They are expressed in milliseconds and are the average of 10 executions. The benchmarks used for the analysis were: `ackermann`, the classical ackermann function; `allones`, which transforms all elements of a list into `1`; `applast`, which appends an element at the end of a given list and returns the last element of the resulting list; `exam`, the program of Example 10; `fibonacci`, fibonacci's function; `kmp`, the specialization of a semi-naïve string pattern matcher; `palindrome`, a program to check whether a given list is a palindrome; `sumprod`, which obtains the sum and the product of the elements of a list; `matmul`, a program for matrix multiplication, and `sumleq`, the program of Example 2 containing the rules for "+","−", and "⩽". Some of the examples are typical PD benchmarks (see [36, 37]) adapted to a functional logic syntax, while others come from the literature of functional program transformations, such as positive supercompilation [48], fold/unfold transformations [16, 18], and deforestation [49]. Runtime input goals were chosen to give a reasonably long overall time. The complete code for benchmarks and the specialized goals can be found in Appendix A.

Table 1 compares the performances of NN-PE w.r.t. LN-PE. The columns "Size", "LN-Size" and "NN-Size" are the number of rewrite rules in the original program, the specialized program using LN-PE and the program specialized by NN-PE, respectively. The columns "LN-St" and "NN-St" are the corresponding specialization times. The column "Speedup" shows the relative improvement achieved by NN-PE for each benchmark, obtained as the ratio (LN-St $\div$ NN-St). In all benchmarks, the NN-PE specialization times were considerably better, with an average speedup of 1.6 in comparison with LN-PE.

Table 2 summarizes our findings w.r.t. the quality of the specialization achieved. The experiments reported in this table correspond to a combination of the benchmarks `ackerman` and `sumleq`, which were executed using different running calls (including nested calls to these functions). Remember that natural numbers are implemented by $0/s$-terms. The columns "LN-Rt" and "NN-Rt" show the runtimes for computing the first solution of each call in the programs specialized using LN-PE and NN-PE, respectively. Runtimes are expressed as percent of the time taken by the original program for the same goal. The column "Speedup" shows the relative improvement for each call, obtained as the ratio (NN-Rt $\div$ LN-Rt). Our results show that the specialization achieved by using NN-PE in these experiments is better, with an average speedup of 2.66 in comparison to LN-PE. These results point to the superiority of the NN-PE strategy.

8 Conclusions

Few attempts have been made to investigate powerful and effective PE techniques which can be applied to term rewriting systems, logic programs and functional programs. In this paper, we have presented a partial evaluator for functional logic programs based on needed narrowing and we have shown its strong correctness, i.e., the answers computed by needed narrowing in the original and specialized programs for the considered queries are identical (up to renaming). Furthermore, we have shown that the partial evaluation process keeps the inductively sequential structure of programs so that the optimal needed narrowing strategy can also be applied to the specialized programs. As a consequence, the partial evaluation process preserves the desirable determinism property of functional logic programs: deterministic evaluations w.r.t. the original program are still deterministic in the specialized program. This property is nontrivial as witnessed by counterexamples for the case of lazy narrowing. We have also empirically verified that the use of needed narrowing in a partial evaluator speeds up the specialization time in comparison to lazy narrowing and it does not remove indexing information from the program, which is needed to obtain fast unification. Thus, we conclude that needed narrowing is the best known framework for specialising functional logic programs. We are currently working on the development of some abstract interpretation techniques for the detection and removal of redundant arguments and useless clauses from the partially evaluated program in order to further enhance the specialization.

We conclude by mentioning some further research. Needed narrowing is only defined for functional logic programs with inductively sequential function definitions. Although this class covers typical functional programs and many logic programs, it might be interesting to consider the more general class of almost orthogonal programs where rules with overlapping left-hand sides are allowed. In principle, needed narrowing can be extended to this class (weakly needed or parallel narrowing [11]), but then some of the optimality properties are lost. However, it seems that weakly needed narrowing is superior to lazy narrowing even in this case, and, thus, it is interesting for future research to investigate the properties of a partial evaluator based on such extensions of needed narrowing. Another interesting topic is the inclusion of concurrency features as in the extension of needed narrowing proposed in [27].

Acknowledgements

We wish to thank Elvira Albert and Santiago Escobar for many helpful remarks and for their valuable contribution to the implementation and testing work.

References

[1] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In G. Levi, editor, *Proc. of Static Analysis Symposium, SAS'98*, pages 262–277. Springer LNCS 1503, 1998.

[2] E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical Report DSIC-II/12/98, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

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

[4] 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.

[5] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[6] M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.

[7] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin, 1992.

[8] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, volume 1298 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, Berlin, 1997.

[9] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. Technical report MPI-I-93-243, Max-Planck-Institut für Informatik, Saarbrücken, 1993.

[10] 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.

[11] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.

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

[13] F. Bellegarde. ASTRE: Towards a fully automated program transformation system. In Jieh Hsiang, editor, *Proceedings of RTA'95*, pages 403–407. Springer LNCS 914, 1995.

[14] M. P. Bonacina. Partial evaluation by completion. In G. Italiani et al., editor, *Conferenza dell'Associazione italiana per il calcolo automatico*, 1988.

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

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

[17] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

[18] J. Darlington. Program transformation. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.

[19] 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.

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

[21] 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.

[22] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.

[23] 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.

[24] R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110, February 1996.

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

[26] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.

[27] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93. ACM, New York, 1997.

[28] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

[29] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming (to appear)*, 1999.

[30] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.

[31] G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.

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

[33] 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.

[34] 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.

[35] L. Lafave and J.P. Gallagher. Constraint-based Partial Evaluation of Rewriting-based Functional Logic Programs. In *Proc. of LOPSTR'97*, pages 168–188. Springer LNCS 1463, 1997.

[36] J. Lam and A. Kusalik. A Comparative Analysis of Partial Deductors for Pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1991. Revised April 1991.

[37] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Technical report, Accessible via `http://www.cs.kuleuven.ac.be/~lpai`, 1998.

[38] 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.

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

[40] 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.

[41] S. Lucas. Root-neededness and approximations of neededness. *Information Processing Letters*, 67(5):245–254, 1998.

[42] A. Middeldorp. Call by Need Computations to Root-Stable Form. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 94–105. ACM, New York, 1997.

[43] A. Miniussi and D. J. Sherman. Squeezing Intermediate Construction in Equational Programs. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 284–302. Springer LNCS 1110, February 1996.

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

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

[46] A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 355–385. Springer LNCS 1110, 1996.

[47] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

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

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

[50] F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In P. Van Hentenryck, editor, *Proc. of the 4th Int'l Static Analysis Symposium, SAS'97*, pages 141–159. Springer LNCS 1302, 1997.

A Benchmark Code and Specialized Goals

applast

```
applast(L,X) -> last(append(L,[X])
last([X]) -> X
last([X|R]) -> last(R)
append([],Y) -> Y
append([X|R],Y) -> [X|append(R,Y)]
```

call: `applast(L,X)`

ackermann

```
ackermann(N) -> ack(s(s(0)),N)
ack(0,N) -> s(N)
ack(s(M),0) -> ack(M,s(0))
ack(s(M),s(N)) -> ack(M,ack(s(M),N))
```

call: `ackermann(N)`

allones

```
f(L) -> allones(length(L))
allones(0) -> []
allones(s(N)) -> [1|allones(N)]
length([]) -> 0
length([H|T]) -> sum(s(0),length(T))
sum(0,Y) -> Y
sum(s(X),Y) -> s(sum(X,Y))
```

call: `f(L)`

sumleq

```
sum(0,X) -> X
sum(s(X),Y) -> s(sum(X,Y))
sub(X,0)  -> X
sub(s(X),s(Y)) -> sub(X,Y)
leq(0,X) -> true
leq(s(X),0) -> true
leq(s(X),s(Y)) -> leq(X,Y)
```

call: `leq(X,sum(X,Y))`

fibonacci

```
fib(0) -> s(0)
fib(s(0)) -> s(0)
fib(s(s(N))) -> sum(fib(s(N)),fib(N))
sum(0,Y) -> Y
sum(s(X),Y) -> s(sum(X,Y))
```

call: `fib(N)`

sumprod

```
sumprod(L) -> sum(sumlist(L),prodlist(L))
sumlist([]) -> 0
sumlist([H|T]) -> sum(H,sumlist(T))
prodlist([]) -> s(0)
prodlist([H|T]) -> prod(H,prodlist(T))
sum(0,Y) -> Y
sum(s(X),Y) -> s(sum(X,Y))
prod(0,Y) -> 0
prod(s(X),Y) -> sum(prod(X,Y),Y)
```

call: `sumprod(L)`

exam

```
f(0,0) -> s(f(0,0))
f(s(N),X) -> s(f(N,X))
g(0) -> g(0)
h(s(X)) -> 0
```

call: `h(f(X,g(Y))`

matmult

```
matmult([X|Xs],Y) -> [rowmult(X,Y)|matmult(Xs,Y)]
matmult([],Y) -> []
rowmult(X,[Y|Ys]) -> [dotmult(X,Y)|rowmult(X,Ys)]
rowmult(X,[]) -> []
dotmult([X|Xs],[Y|Ys]) ->
       plus(mult(X,Y),dotmult(Xs,Ys))
dotmult([],[]) -> 0
sum(0,X) -> X
sum(s(X),Y) -> s(sum(X,Y))
```

call: `matmult([X,Y,Z],W)`

kmp

```
match(P,S) -> loop(P,S,P,S)
loop([],SS,OP,OS) -> true
loop([P|PP],[],OP,OS) -> false
loop([P|PP],[S|SS],OP,OS) ->
 if(eq(P,S),loop(PP,SS,OP,OS),next(OP,OS))
next(OP,[]) -> false
next(OP,[S|SS]) -> loop(OP,SS,OP,SS)
if(true,A,B) -> A
if(false,A,B) -> B
eq(a,a) -> true
eq(b,b) -> true
eq(a,b) -> false
eq(b,a) -> false
```

call: `match([a,a,b],S)`

palindrome

```
palindrome(L) -> eqlist(reverse(L),L)
reverse(L) -> rev(L,[])
rev([],L) -> L
rev([X|L],Y) -> rev(L,[X|Y])
eqlist([],[]) -> true
eqlist([A|RA],[B|RB]) ->
        if(eq(A,B),eqlist(RA,RB),false)
if(true,A,B) -> A
if(false,A,B) -> B
eq(0,0) -> true
eq(0,s(M)) -> false
eq(s(N),0) -> false
eq(s(N),s(M)) -> eq(N,M)
```

call: `palindrome([s(0)|L])`