Specialization of Inductively Sequential Functional Logic Programs*

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

† DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain. {alpuente,slucas,gvidal}@dsic.upv.es
‡ Informatik II, RWTH Aachen, D-52056 Aachen, Germany. hanus@informatik.rwth-aachen.de

Abstract

Functional logic languages combine the operational principles of the most important declarative programming paradigms, namely functional and logic programming. Inductively sequential programs admit the definition of optimal computation strategies and are the basis of several recent (lazy) functional logic languages. In this paper, we define a partial evaluator for inductively sequential functional logic programs. We prove strong correctness of this partial evaluator and show that the nice properties of inductively sequential programs carry over to the specialization process and the specialized programs. In particular, the structure of the programs is preserved by the specialization process. This is in contrast to other partial evaluation methods for functional logic programs which can destroy the original program structure. 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 (see [25] for a survey). 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, which combines reduction (from the functional part) and variable instantiation (from the logic part) [48, 32, 47]. 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 program equation.

Example 1.1 *Consider the data type* `Nat` *defined as*

```
data Nat = 0 | S Nat
```

and the operator $\leqslant$ :: `Nat` $\to$ `Nat` $\to$ `Bool` *which is defined by the following equations:*

$$\begin{array}{rcl} \texttt{0} \leqslant \texttt{n} & = & \texttt{True} \\ (\texttt{S m}) \leqslant \texttt{0} & = & \texttt{False} \\ (\texttt{S m}) \leqslant (\texttt{S n}) & = & \texttt{m} \leqslant \texttt{n} \end{array}$$

The expression $(\texttt{S m}) \leqslant \texttt{y}$ *can be evaluated (i.e., reduced to a value) by instantiating* `y` *to* `(S n)` *to apply the third equation, followed by the instantiation of* `m` *to* `0` *to apply the first equation:*

$$(\texttt{S m}) \leqslant \texttt{y} \quad \leadsto_{\{\texttt{y} \mapsto (\texttt{S n})\}} \quad \texttt{m} \leqslant \texttt{n} \quad \leadsto_{\{\texttt{m} \mapsto \texttt{0}\}} \quad \texttt{True}$$

Narrowing provides completeness in the sense of logic programming (computation of all answers, i.e., substitutions leading to successful evaluations) 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. 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., [9, 22, 41, 43]). *Needed narrowing* [9] 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 equations in Example 1.1 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 term, here `0` or $(\texttt{S}\ \Box)$. Depending on this instantiation, either the first equation is applied or the second argument t_2 is evaluated. Needed narrowing is currently the best narrowing strategy for first-order (inductively sequential) functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions [9] and it can be efficiently implemented by pattern matching and unification (e.g., [26, 41]). Moreover, it has recently been extended to higher-order functions and λ-terms as data structures and proved optimal w.r.t. independency 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 [12, 13, 18, 36, 42], functional programming [15, 33], and logic programming [21, 40]. Although the

*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.

To appear in the 1999 International Conference on Functional Programming (ICFP'99), September 1999, Paris, France.

©1999 ACM. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission.

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, 44, 49].

Narrowing-driven PE [4, 5] is the first generic algorithm for the (on-line) 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 inspired by the theoretical framework established in [40] 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 [39] (a comparison can be found in [1, 5, 6]).

To perform reductions at specialization time, a partial evaluator normally includes an interpreter [15, 24]. This implies that the power of the transformation is highly influenced by the properties of the evaluation strategy from the underlying interpreter. The contribution of this paper is the definition of a partial evaluator for functional logic programs based on needed narrowing. We provide the following results:

- We prove strong correctness for such a partial evaluator, i.e., the answers and values computed by needed narrowing in the original and the partially evaluated programs coincide.
- We relate this partial evaluator to PE based on the lazy narrowing strategy of [43] and show its advantages.
- We prove that PE 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.
- We show that the specialized programs do not lose their abilities for deterministic reduction, which is important from an implementation point of view and is not obtained by PE 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 extension of Haskell with features for 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 briefly introduce in Section 3 the computation models for lazy functional logic programs which we consider in this paper. The definition of partial evaluation based on needed narrowing is provided in Section 4 together with results about the structure of specialized programs and the (strong) correctness of the transformation. Section 5 shows the practical importance of our specialization techniques by means of some benchmarks and Section 6 concludes. Proofs of all technical results can be found in [7].

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) [11, 34, 46]. Within this framework, the class of inductively sequential programs has been defined, studied, and used for the implementation of programming languages which provide for optimal computations both in functional and functional logic programming [8, 9, 27, 28, 41]. 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 [17] 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[1] $f(\overline{d_n})$ 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). Given a term t, we let $\mathcal{P}os(t)$ and $\mathcal{NVP}os(t)$ denote the set of positions and the set of nonvariable 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.

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

[1] Note the difference with the usual notion in functional programming: a constructor term.

$\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ 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 $\theta_{|V}$ the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta_{|V} = \sigma_{|V}$, 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 $\sigma(s) = \sigma(t)$. 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. In the remainder of this paper, a functional logic *program* is a 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 [30, 43].

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*. A (constructor) *head normal form* is either a variable or 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* [25]). 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 $\approx$ used in some examples is defined, as in functional languages, as the *strict equality* on terms, 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., [9]):

$$\begin{array}{rcll} \mathtt{C} \approx \mathtt{C} & \to & \mathtt{True} & \%\ \mathtt{C/0} \in \mathcal{C} \\ \mathtt{C}(\overline{\mathtt{x_n}}) \approx \mathtt{C}(\overline{\mathtt{y_n}}) & \to & (\mathtt{x_1} \approx \mathtt{y_1}) \wedge \ldots \wedge (\mathtt{x_n} \approx \mathtt{y_n}) & \%\ \mathtt{C/n} \in \mathcal{C} \\ \mathtt{True} \wedge \mathtt{x} & \to & \mathtt{x} & \end{array}$$

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 $e \leadsto^*_\sigma \mathtt{True}$.

3 Lazy Computation Models for Functional Logic Programs

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. In the following, we briefly outline the computation models which we consider in this paper: lazy narrowing and needed narrowing. A formal description of these strategies can be found in [7].

3.1 Lazy Narrowing

Lazy narrowing reduces expressions at outermost narrowable positions. Narrowing at inner positions is performed only if it is demanded (by the lhs of some rule). Following [43], we define the lazy narrowing strategy as a function $\lambda_{lazy}(t)$ which returns a set of triples (p, R, σ) such that p is a demanded position of t which can be narrowed by the rule R with substitution σ (where σ is a most general unifier of $t|_p$ and the lhs of R).

Example 3.1 *Consider the following rules:*

$$\begin{array}{rcll} \mathtt{0} \leqslant \mathtt{n} & \to & \mathtt{True} & \mathtt{0} + \mathtt{n} \to \mathtt{n} \\ \mathtt{S(m)} \leqslant \mathtt{0} & \to & \mathtt{False} & \mathtt{S(m)} + \mathtt{n} \to \mathtt{S(m + n)} \\ \mathtt{S(m)} \leqslant \mathtt{S(n)} & \to & \mathtt{m} \leqslant \mathtt{n} & \end{array}$$

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:*

$$\begin{array}{lll} \mathtt{x} \leqslant \mathtt{x} + \mathtt{x} & \leadsto_{\{\mathtt{x} \mapsto \mathtt{0}\}} & \mathtt{True} \\ \mathtt{x} \leqslant \mathtt{x} + \mathtt{x} & \leadsto_{\{\mathtt{x} \mapsto \mathtt{0}\}} & \mathtt{0} \leqslant \mathtt{0} \\ \mathtt{x} \leqslant \mathtt{x} + \mathtt{x} & \leadsto_{\{\mathtt{x} \mapsto \mathtt{S(m)}\}} & \mathtt{S(m)} \leqslant \mathtt{S(m + S(m))} \end{array}$$

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

3.2 Needed Narrowing

Needed narrowing extends the Huet and Lévy's notion of a needed reduction [31]. The definition of *needed narrowing* [9] 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*. A precise definition of this class of programs and the needed narrowing strategy is based on the notion of a definitional tree [8]. Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves contain all (and only) the rules used to define f and whose inner nodes contain information to guide the (optimal) pattern matching during the evaluation of expressions. Each inner node contains a

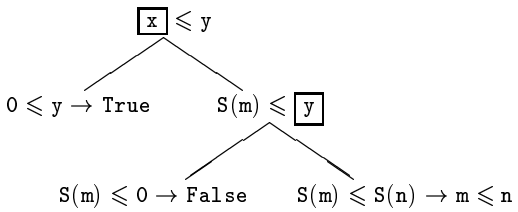

Figure 1: Definitional tree for the operator "⩽" of Example 3.1

pattern and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\overline{x_n})$, where $\overline{x_n}$ are different variables. A graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules illustrates this notion (see Figure 1).

Definitional trees are similar to standard matching trees of functional programming[2] [20]. However, differently from left-to-right matching trees used in either Hope, Miranda, or Haskell, definitional trees can deal with more complex dependencies between arguments of functional patterns (e.g., right-to-left evaluations of arguments). As a good point, optimality is achieved when definitional trees are used (in this sense, they are closer to *matching dags* or *index trees* for TRSs [31, 19, 28]). 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[3].

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 $\lambda_{needed}(t, \mathcal{P})$. Then, for all $(p, R, \sigma) \in \lambda_{needed}(t, \mathcal{P})$, $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*. We call this step *deterministic* if $\lambda_{needed}(t, \mathcal{P})$ contains exactly one element. Informally speaking, needed narrowing applies a rule, if possible, or checks the subterm corresponding to the inductive position of the branch: if it is a variable, it is instantiated to the constructor of a child; if it is already a constructor, we proceed with the corresponding child; if it is a function, we evaluate it by recursively applying needed narrowing.

Example 3.2 *Consider the rules for "⩽" and "+" in Example 3.1. Then the function λ_{needed} computes the following set for the initial term* $\mathtt{x} \leqslant \mathtt{x} + \mathtt{x}$*:*

$$\{(\ \wedge, \mathtt{0} \leqslant \mathtt{n} \to \mathtt{True}, \{\mathtt{x} \mapsto \mathtt{0}\}),\\ (2, \mathtt{S(m)} + \mathtt{n} \to \mathtt{S(m+n)}, \{\mathtt{x} \mapsto \mathtt{S(m)}\})\}$$

This corresponds to the narrowing steps

$$\begin{array}{lll} \mathtt{x} \leqslant \mathtt{x} + \mathtt{x} & \leadsto_{\{\mathtt{x} \mapsto \mathtt{0}\}} & \mathtt{True} \\ \mathtt{x} \leqslant \mathtt{x} + \mathtt{x} & \leadsto_{\{\mathtt{x} \mapsto \mathtt{S(m)}\}} & \mathtt{S(m)} \leqslant \mathtt{S(m + S(m))} \end{array}$$

The main properties of needed narrowing are formalized as follows:

[2] Definitional trees can also be encoded using *case expressions*, another well-known technique to implement pattern matching in functional languages [45, 29].

[3] For CB-TRSs, inductive sequentiality and Huet and Lévy's strong sequentiality coincide [28].

Theorem 3.3 [9] *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 $\mathcal{V}ar(e)$.*

4 Partial Evaluation of Lazy Functional Logic Programs

In narrowing-driven PE [5], specialized program rules are constructed from narrowing derivations using *resultants*.

Definition 4.1 (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 $\sigma(s) \to t$.*

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 in Definition 4.7 a post-processing renaming transformation which not only eliminates redundant structures but also obtains *independent* specializations (in the sense of [40]) 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.

Narrowing derivations can be represented by a (possibly infinite) finitely branching tree. Following [40], in this work we adopt the convention that any derivation is potentially partial, i.e., not fully expanded. Thus, a branch can be failed, partial, successful, or infinite. A *failing leaf* is an expression which is not a constructor term and which cannot be further narrowed. The *(pre–)partial evaluation* of a term s is obtained by constructing a (possibly partial) narrowing tree for s and then extracting the specialized definitions (the resultants) from the non–failing, root–to–leaf paths of the tree.

Definition 4.2 (pre–partial evaluation) *Let $\mathcal{R}$ be a TRS and s a term. Let T be a finite (possibly partial) narrowing tree for s in $\mathcal{R}$ such that no (constructor) head normal form 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 evaluate beyond head normal forms in pre–partial evaluations cannot be dropped. This is due to the fact that a pre–partial evaluation beyond the head normal form might propagate bindings which do not occur in the execution of the original program.

Example 4.3 *Consider the following program $\mathcal{R}$:*

$$\begin{array}{rcl} \mathtt{f(0)} & \to & \mathtt{0} \\ \mathtt{g(x)} & \to & \mathtt{S(f(x))} \\ \mathtt{h(S(x))} & \to & \mathtt{S(0)} \end{array}$$

with the set of calls $S = \{\mathtt{g(x)}, \mathtt{h(x)}\}$. Then, a pre–partial evaluation of S in $\mathcal{R}$ without the restriction to not evaluate beyond head normal forms is the program $\mathcal{R}'$:

$$\begin{array}{rcl} \mathtt{g(0)} & \to & \mathtt{S(0)} \\ \mathtt{h(S(x))} & \to & \mathtt{S(0)} \end{array}$$

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

$$\begin{array}{lll} \mathtt{h(\underline{g(S(0))})} \approx \mathtt{x} & \leadsto & \underline{\mathtt{h(S(f(S(0))))}} \approx \mathtt{x} \\ & \leadsto & \mathtt{S(0)} \approx \mathtt{x} \\ & \leadsto^*_{\{x \mapsto S(0)\}} & \mathtt{True} \end{array}$$

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

$closed(S,t) \Leftrightarrow$

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

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 (nondeterministic) 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 [1, 5] for the definition of concrete unfolding rules).

Example 4.5 *Consider the well-known concatenation operator:*

$$\begin{array}{rcl} \mathtt{[]\ ++\ ys} & \to & \mathtt{ys} \\ \mathtt{(x : xs)\ ++\ ys} & \to & \mathtt{x : (xs\ ++\ ys)} \end{array}$$

with the set of calls $S = \{\mathtt{(xs{+}{+}ys){+}{+}zs}, \mathtt{xs{+}{+}ys}\}$. A pre–partial evaluation of S in $\mathcal{R}$ using needed narrowing is the S-closed program:

$$\begin{array}{rcl} \mathtt{([]\ ++\ ys)\ ++\ zs} & \to & \mathtt{ys\ ++\ zs} \\ \mathtt{((x : xs)\ ++\ ys)\ ++\ zs} & \to & \mathtt{x : ((xs\ ++\ ys)\ ++\ zs)} \\ \mathtt{[]\ ++\ zs} & \to & \mathtt{zs} \\ \mathtt{(y : ys)\ ++\ zs} & \to & \mathtt{y : (ys\ ++\ zs)} \end{array}$$

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 and 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.

Definition 4.6 (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.

Definition 4.7 (partial evaluation) *Let $\mathcal{R}$ be a TRS, S a finite set of terms and $\mathcal{R}'$ a pre–partial evaluation of $\mathcal{R}$ w.r.t. S. Let ρ be an independent renaming of S. We define the partial evaluation $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. S (under ρ) as follows:*

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

where the nondeterministic renaming function ren_ρ is de-

fined as follows:

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

Similarly to the test for closedness, an equation $s \approx t$ can be (nondeterministically) 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).

We now illustrate these definitions with an example.

Example 4.8 *Consider again the definition of the operator* `++` *and the set S of Example 4.5. An independent renaming ρ for S is the mapping:*

$$\{\ \mathtt{xs\ ++\ ys} \mapsto \mathtt{app(xs, ys)},\ \mathtt{(xs\ ++\ ys)\ ++\ zs} \mapsto \mathtt{dapp(xs, ys, zs)}\ \}.$$

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

```
dapp([], ys, zs)   →  app(ys, zs)
dapp(x : xs, ys, zs) → x : dapp(xs, ys, zs)
app([], ys)        →  ys
app(x : xs, ys)    →  x : app(xs, ys)
```

The following theorem states an important property of PE w.r.t. needed narrowing: if the input program is inductively sequential, then the specialized program is also inductively sequential so that we can apply the optimal needed narrowing strategy to the specialized program.

Theorem 4.9 *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.*

4.1 Needed-PE vs. Lazy-PE

The correctness of LN-PE is stated in [1, 3] for orthogonal programs (i.e., programs without overlapping left-hand sides). In the following we show that the partial evaluation w.r.t. needed narrowing can also be obtained (but possibly with more steps) by partial evaluation of a 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.

In functional programming, the class of so-called uniform programs was introduced to ease the efficient implementation of the pattern matching [45]. For the implementation of needed narrowing, a similar class of programs has been studied. These are the *uniform* programs of [51], where 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. Uniform programs are inductively sequential. A different definition of uniform programs can be found in [35].

There is a simple mapping $\mathcal{U}$ from inductively sequential into uniform programs which can be found in [51] and is based on flattening nested patterns. For instance, if $\mathcal{R}$ is the set of rules defining "$\leqslant$" (see Example 3.1), then $\mathcal{U}(\mathcal{R})$ consists of the rules

```
0 ⩽ n    →  True          m ⩽' 0      →  False
S(m) ⩽ n →  m ⩽' n        m ⩽' S(n1)  →  m ⩽ n1
```

where $\leqslant'$ is a new function symbol. Lazy narrowing and needed narrowing coincide on this class of programs, since the transformed uniform program embeds the information carried on the definitional trees of the original one. Now we can precisely relate NN-PE with LN-PE.

Lemma 4.10 *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 $\mathcal{R}_u$.*

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

```
f(x, B) → g(x)
g(A)    → A
```

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

```
f2(A, B) → 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 4.12 will positively answer this question.

Corollary 4.12 *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.*

The uniformity condition in Corollary 4.12 cannot be weakened to inductive sequentiality when LN-PEs are considered, as demonstrated by the following counterexample.

Example 4.13 *Let $\mathcal{R}$ be the inductively sequential program:*

```
f(A, A, A) → B      h(A, B, x) → B
f(B, B, x) → B      h(E, x, K) → B
g(A, B, x) → B      i(x, C, D) → B
g(x, C, D) → B      i(E, x, K) → B
```

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:

```
f3(A, B, x) → ···
f3(E, x, K) → ···
f3(x, C, D) → ···
```

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 4.14 *Consider again the rules of Example 3.2 and the input term* $\mathtt{x} \leqslant \mathtt{x} + \mathtt{y}$. *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 5). 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.

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

Example 4.15 *Consider the following orthogonal program:*

$$\begin{array}{rcl} \mathtt{f(0,0)} & \to & \mathtt{S(f(0,0))} \\ \mathtt{f(S(n),x)} & \to & \mathtt{S(f(n,x))} \\ \mathtt{g(0)} & \to & \mathtt{g(0)} \\ \mathtt{h(S(x))} & \to & \mathtt{0} \end{array}$$

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* $\mathtt{h2(x,y)}$*):*

$$\begin{array}{rcl} \mathtt{h0} & \to & \mathtt{h0} \\ \mathtt{h2(0,0)} & \to & \mathtt{h0} \\ \mathtt{h2(S(x),y)} & \to & \mathtt{0} \end{array}$$

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

$$\begin{array}{rcl} \mathtt{h1(x)} & \to & \mathtt{h1(x)} \\ \mathtt{h1(S(x))} & \to & \mathtt{0} \\ \mathtt{h2(x,0)} & \to & \mathtt{h1(x)} \\ \mathtt{h2(S(x),y)} & \to & \mathtt{0} \\ \mathtt{h2(S(x),0)} & \to & \mathtt{0} \end{array}$$

Note that the program specialized by LN-PE in the example above 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 $\mathtt{h(f(S(0),g(0)))}$. The evaluation of this term has a finite derivation tree w.r.t. lazy narrowing as well as needed narrowing. However, the renamed term $\mathtt{h2(S(0),0)}$ 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 $\mathtt{h2(x,0)} \to \mathtt{h1(x)}$ and $\mathtt{h1(x)} \to \mathtt{h1(x)}$.

4.2 Correctness of NN-PE and Preservation of Deterministic Evaluations

The strong correctness of NN-PE is stated in the following theorem, 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).

Theorem 4.16 (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,* $e \leadsto^*_\sigma \mathtt{True}$ *is a needed narrowing derivation for* e *in* $\mathcal{R}$ *iff there exists a needed narrowing derivation* $e' \leadsto^*_{\sigma'} \mathtt{True}$ *in* $\mathcal{R}'$ *such that* $\sigma' = \sigma\,[V]$ *(up to renaming).*

Now, we show another practically interesting property of NN-PE. One can prove 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. 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 4.17 *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.*

This kind of determinism in computations is an important advantage of functional logic languages in comparison to pure logic languages as it can avoid the evaluation of potential non-deterministic expressions. For instance, consider again the rules in Example 3.1 and the term $\mathtt{0} \leqslant \mathtt{x} + \mathtt{x}$. Needed narrowing evaluates this term by one deterministic step to $\mathtt{True}$. 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 $+(\mathtt{x},\mathtt{x},\mathtt{z})$.[4] Another reason for the improved operational

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

behavior of functional logic languages is the ability of particular evaluation strategies (like needed narrowing or parallel narrowing [10]) to evaluate ground terms in a completely deterministic way, which is important to ensure an efficient implementation of purely functional evaluations.

Example 4.15 showed 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. The following proposition formalizes that deterministic normalizations w.r.t. the original program cannot cause a non-deterministic evaluation w.r.t. the specialized program using NN-PE.

Proposition 4.18 *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. $\mathcal{R}'$.*

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 4.15. 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.

In the next 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.

5 Experimental Results

A partial evaluator for functional logic programs based on needed narrowing as well as on lazy narrowing has been implemented in the INDY system[5] [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 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 4.15; `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 3.1 containing the rules for "+","−", and "⩽". Some of the examples are typical PD benchmarks (see [37, 38]) adapted to a functional logic syntax, while others come from the literature of functional program transformations, such as positive supercompilation [49], fold/unfold transformations [14, 16], and deforestation [50]. 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 Table 3.

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 "Improvement" shows the relative improvement achieved by NN-PE for each benchmark, obtained as the ratio (LN-St ÷ 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-Speedup" and "NN-Speedup" show the speedups for computing the first solution of each call in the programs specialized using LN-PE and NN-PE relative to the original program for the same goal. The column "Improvement" shows the relative improvement for each call, obtained as the ratio (NN-Speedup ÷ LN-Speedup). Our results show that the specialization achieved by using NN-PE in these experiments is better, with an average improvement factor of 2.66 in comparison to LN-PE. These results point to the superiority of the NN-PE strategy.

In general, partially evaluated programs cannot be guaranteed to be faster than the original ones, since there is a trade-off between the smaller number of computation steps and the larger number of rules after the specialization. Nevertheless, our experiments seem to indicate that the gain due to the smaller derivations makes up for the overhead of checking the applicability of the larger number of rules in the specialized programs.

6 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 and values computed by needed narrowing in the original and specialized programs 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

[5] 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.

	Original	LN-PE		NN-PE		
Benchmarks	Size	LN-Size	LN-St	NN-Size	NN-St	Improvement
`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.)

Specialized Expressions:	LN-Speedup	NN-Speedup	Improvement
$\texttt{ackermann(5)} \leq (5+5) \approx \texttt{True}$	1.20	1.49	1.24
$(20-\texttt{x})+((20-\texttt{x})+(20-\texttt{x})) \leq 40+40 \approx \texttt{True}$	2.33	6.67	2.87
$(20+\texttt{y})+(\texttt{y}+20) \leq 20+20 \approx \texttt{True}$	1.37	2.70	1.97
$10+\texttt{x} \leq (\texttt{x}+2)+\texttt{x} \approx \texttt{True}$	4.54	14.93	3.29
$(\texttt{x}-10)+((\texttt{x}-10)+(\texttt{x}-10)) \leq 20+20 \approx \texttt{True}$	1.15	4.55	3.95

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

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. The results in this paper are relevant for the optimization of Curry [27, 30], a language which is intended to become a standard in the functional logic programming community.

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.

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 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] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. Technical report 99-4, RWTH Aachen, 1999. Available from `ftp.informatik.rwth-aachen.de/pub/reports/`.

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

[9] 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.

[10] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the 14th Int'l Conf. on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] I. Durand. Bounded, Strongly Sequential and Forward-Branching Term Rewriting Systems. *Journal of Symbolic Computation*, 18(4):319–352, 1994.

[20] A.J. Field and P.G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, 1988.

[21] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, 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. of 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. 5th Int'l 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*, 9(1):33–75, 1999.

[30] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Version 0.5, Jan. 1999. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

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

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

[34] 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.

[35] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. of ALP'90*, pages 298–317. Springer LNCS 463, 1990.

[36] 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.

[37] 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.

[38] 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.

[39] 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 Int'l Conference and Symposium on Logic Programming, JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

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

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

[48] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

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

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

[51] 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.

applast

```
applast(xs,x) -> last(append(xs,[x])
last([x]) -> x
last(x:xs) -> last(xs)
append([],y) -> y
append(x:xs,y) -> x:append(xs,y)
```

call: `applast(xs,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(xs) -> allones(length(xs))
allones(0) -> []
allones(S(n)) -> 1:allones(n)
length([]) -> 0
length(x:xs) -> sum(S(0),length(xs))
sum(0,y) -> y
sum(S(x),y) -> S(sum(x,y))
```

call: `f(xs)`

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(xs) -> sum(sumlist(xs),prodlist(xs))
sumlist([]) -> 0
sumlist(x:xs) -> sum(x,sumlist(xs))
prodlist([]) -> S(0)
prodlist(x:xs) -> prod(x,prodlist(xs))
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(xs)`

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:ps,[],op,os) -> False
loop(p:ps,s:ss,op,os) ->
 if(eq(p,s),loop(ps,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(xs) -> eqlist(reverse(xs),xs)
reverse(xs) -> rev(xs,[])
rev([],xs) -> xs
rev(x:xs,ys) -> rev(xs,x:ys)

eqlist([],[]) -> True
eqlist(a:as,b:bs) -> if(eq(a,b),eqlist(as,bs),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):xs)`

Table 3: Benchmark programs and specialized calls.