Preliminary version. Final version in Theoretical Computer Science (Elsevier), Vol. 311, No. 1-3, pp. 479-525, 2004

Rules + Strategies for Transforming Lazy Functional Logic Programs

María Alpuente [a] Moreno Falaschi [b] Ginés Moreno [c]
Germán Vidal [a]

[a] *DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.*
[b] *Dip. Mat. e Informatica, U. Udine, 33100 Udine, Italy.*
[c] *Dep. Informática, UCLM, 02071 Albacete, Spain.*

Abstract

This work introduces a transformation methodology for functional logic programs based on needed narrowing, the optimal and complete operational principle for modern declarative languages which integrate the best features of functional and logic programming. We provide correctness results for the transformation system w.r.t. the set of *computed values* and *answer substitutions* and show that the prominent properties of needed narrowing—namely, the optimality w.r.t. the length of derivations and the number of computed solutions—carry over to the transformation process and the transformed programs. We illustrate the power of the system by taking on in our setting two well-known transformation strategies (*composition* and *tupling*). We also provide an implementation of the transformation system which, by means of some experimental results, highlights the potentiality of our approach.

Key words: program transformation, functional logic programming, strategies

1 Introduction

Functional logic programming languages combine the operational principles of the most important declarative programming paradigms, namely functional

⋆ This work has been partially supported by CICYT under grant TIC 98-0445-C03.
⋆⋆A preliminary version of this article appeared in the *Proceedings of the Fourth International Symposium on Functional and Logic Programming*, FLOPS'99 [4].
Email addresses: alpuente@dsic.upv.es (María Alpuente), falaschi@dimi.uniud.it (Moreno Falaschi), gmoreno@info-ab.uclm.es (Ginés Moreno), gvidal@dsic.upv.es (Germán Vidal).

and logic programming (see [24] 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 integrated languages is usually based on narrowing, a combination of variable instantiation and reduction. The instantiation of variables is often computed by unifying a subterm of the goal expression with the left-hand side of some program rule; then narrowing reduces the instantiated goal using that rule. Needed narrowing is currently the best narrowing strategy for first-order, lazy functional logic programs due to its optimality properties [9]. Needed narrowing provides completeness in the sense of logic programming (computation of all solutions) as well as functional programming (computation of values), and it can be efficiently implemented by pattern matching and unification.

The fold/unfold transformation approach (also known as "rules+strategies" approach [41]) was first introduced in [14] to optimize functional programs and then used for logic programs [46]. This approach is based on the construction, by means of a *strategy*, of a sequence of equivalent programs—called *transformation sequence* and usually denoted by $\mathcal{R}_0, \ldots, \mathcal{R}_n$—where each program $\mathcal{R}_i$ is obtained from the preceding ones $\mathcal{R}_0, \ldots, \mathcal{R}_{i-1}$ by using an *elementary* transformation rule. The essential rules are *folding* and *unfolding*, i.e., contraction and expansion of subexpressions of a program using the definitions of this program (or of a preceding one). Other rules which have been considered are, e.g., instantiation, definition introduction/elimination, and abstraction.

There exists a large class of program optimizations which can be achieved by fold/unfold transformations and are not always possible by using a fully automatic method (such as *partial evaluation* [30]). Typical instances of this class are the strategies that perform *tupling* (also known as *pairing*) [14,20], which merges separate (nonnested) function calls with some common arguments into a single call to a (possibly new) recursive function which returns a tuple of the results of the separate calls. Tupling is able to avoid either multiple accesses to the same data structures or common subcomputations, similarly to the idea of *sharing* which is used in graph rewriting to improve the efficiency of computations in time and space [10]. A closely related strategy is *composition* [48] (also known as *fusion*, *deforestation*, or *vertical jamming* [22]), which essentially consists of the merging of nested function calls, where the inner function call builds up a composite object which is used by the outer call; composing these two calls into one has the effect to avoid the generation of the intermediate data structure. Composition can be made automatically [48], whereas tupling has only been automated to some extent (i.e., for particular classes of programs [15,16]) in the "rule+strategies" framework.

Although a lot of literature has been devoted to proving the correctness of fold/unfold systems w.r.t. the various semantics proposed for logic programs [12,23,31,36,40,46], in functional programming the problem of correctness has

received surprisingly little attention [42,43]. Of the very few studies of correctness of fold/unfold transformations in functional programming, the most general and recent work is [42], which contains correctness proofs for several well-known transformation methods (like Wadler's deforestation [48]).

In [3], we investigated fold/unfold rules in the context of a strict (*call-by-value*) functional logic language based on unrestricted (i.e., not optimized) narrowing. The use of narrowing empowers the fold/unfold system by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework [14] which introduces an instance of an existing equation) into unfolding by means of unification. However, [3] does not consider a general transformation system (only two rules: fold and unfold), and hence the composition or tupling transformations cannot be achieved. Also, [3] refers to a notion of "reversible" folding, whose optimization power within a transformation system is very poor. On the other hand, the use of unrestricted narrowing to perform unfolding may produce an important increase in the number of program rules.

In this work, we define a transformation methodology for lazy (*call-by-name*) functional logic programs. On the theoretical side, we extend the Tamaki and Sato transformation rules [46] for logic programs to cope with lazy functional logic programs based on needed narrowing. The transformation process consists of applying an arbitrary number of times the basic transformation rules, which are: definition introduction, definition elimination, unfolding, folding, and abstraction. Needed narrowing is complete for *inductively sequential* programs [7]. Thus, we demonstrate that such a program structure is preserved through a transformation sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, which is a key point for proving the correctness of the transformation system as well as for its effective applicability. For instance, by using other variants of narrowing (e.g., lazy narrowing [39]), the structure of the original program is not preserved, thus seriously restricting the applicability of the transformation system: the transformed program could not be further transformed and, what is even worse, it might not even be safely executed, as it might not satisfy the conditions for the completeness of the considered operational mechanism. The major technical result consists of proving strong correctness for the transformation system, namely that the values and answers computed by needed narrowing in the initial and the final program coincide (for goals constructed using the symbols of the initial program). The efficiency improvement of $\mathcal{R}_n$ with regard to $\mathcal{R}_0$ is not ensured by an arbitrary use of the elementary transformation rules but it rather depends on the heuristic which is employed. Hence, on the practical side, we investigate how the classical and powerful transformation methodologies of *tupling* and *composition* [41] transfer to our framework. We show the advantages of using needed narrowing in this framework, and illustrate the power of our transformation system by (automatically) optimizing several significant examples using a prototype implementation [2].

The structure of the paper is as follows. After recalling some basic definitions in the next section, we introduce the basic transformation rules and illustrate them by means of several examples in Section 3. We state the correctness of the transformation system and show some results about the structure of transformed programs in Section 4. Section 5 shows how to achieve the (automatic) composition and tupling strategies in our framework, whereas Section 6 presents an experimental evaluation of the method on a set of benchmarks. Finally, Section 7 concludes. We include an appendix containing the proofs of some technical results.

2 Preliminaries

We assume familiarity with basic notions of term rewriting [10] and functional logic programming [24]. We consider a *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 is *linear* if it does not contain multiple occurrences of any variable. We write $\overline{o_n}$ for the *list* of syntactic 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 $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. Note the difference with the usual notion of pattern in functional programming: a constructor term. A term is *operation-rooted* (*constructor-rooted*) if it has an operation (constructor) symbol at the root. 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: $p \leq q$, if $\exists w$ such that $p.w = q$. Positions p, q are *disjoint* if neither $p \leq q$ nor $q \leq p$. 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 (i.e., positions where a variable does not occur), 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. A substitution σ is *(ground) constructor*, if $\sigma(x)$ is (ground) constructor for all x such that $\sigma(x) \neq x$. The identity substitution is denoted by *id*. Given a substitution θ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta_{\restriction V}$ the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta_{\restriction V} = \sigma_{\restriction V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$. We write $t \leq t'$ (*subsumption*) if $t' = \sigma(t)$ for some substitution σ; moreover,

we write $t < t'$ (*strict subsumption*) if $\sigma \neq id$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ of s and t is *most general* (mgu) if $\sigma \leq \theta$ for each unifier θ of s and t (such a *mgu* is unique up to renaming). Two substitutions σ and σ' are *disjoint* (on a set of variables V) if 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). 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 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 [27,39]. 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 substitution σ such that $\sigma(l|_p) = \sigma(l')$. A left-linear TRS with no overlapping rules is called *orthogonal*. 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 denotation of a rewrite step). The instantiated lhs $\sigma(l)$ is called a *redex*. A term t is called a *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 $\to$.

In orthogonal TRSs, every term which is not a normal form has a *needed* redex that must "eventually" be reduced to normalize the term [28,32]. Therefore, for orthogonal TRSs, the best normalizing strategy which avoids unnecessary reductions is *needed rewriting* [28], a reduction strategy which only considers *needed* redexes, namely, redexes which must be contracted (themselves or some descendant) in every derivation leading to a normal form. Unfortunately, it is undecidable whether a redex is needed and can only be approximated. For CB-TRSs, Antoy's outermost-needed redexes [7] provide a decidable subclass of needed redexes which can be used to normalize terms in a particular class of programs, called inductively sequential systems, by applying outermost-needed redution. The formal definition of outermost-needed rewriting can be found in [8,9], and will be recalled in Appendix B.

In order to evaluate terms containing variables, the narrowing mechanism non-deterministically instantiates the variables so that a rewrite step is possible. 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 $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (where $\sigma = id$ for $n = 0$). 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, like 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. A *solution* of an equation $s \approx t$ is a substitution σ such that $\sigma(s \approx t)$ can be reduced to *true*. On the other hand, we say that σ is a *computed answer substitution* for an equation e if there is a narrowing derivation $e \leadsto^*_\sigma true$.

Needed Narrowing

A challenge in the design of functional logic languages is the definition of a "good" narrowing strategy, i.e., a restriction on the narrowing steps issuing from a term without losing completeness. *Needed narrowing* [9] is currently the best known narrowing strategy due to its optimality properties w.r.t. the length of the derivations and the number of computed solutions. It extends Huet and Lévy's notion of a needed reduction [28]. The definition of needed narrowing [9] uses the notion of a *definitional tree* [7]. Definitional trees are similar to standard matching trees of functional programming. However, differently from left-to-right matching trees used in either Hope, Miranda, or Haskell, definitional trees deal with *dependencies* between arguments of functional patterns. 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 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.

Formally, a *definitional tree* of a finite set of linear patterns S is a finite non-empty set $\mathcal{P}$ of linear patterns partially ordered by subsumption having the following properties (up to variable renaming):

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—the *inductive position*—of $pattern(\mathcal{P})$ with $pattern(\mathcal{P})|_o \in \mathcal{X}$ and constructors $c_1, \ldots, c_n \in$

$\mathcal{C}$ with $c_i \neq c_j$ for $i \neq j$, such that, for all $\pi_1, \ldots, \pi_n$ which have the parent π, $\pi_i = \pi[c_i(\overline{x_{n_i}})]_o$ (where $\overline{x_{n_i}}$ are new distinct variables) for all $1 \leq i \leq n$.

Given a defined function f/n in a program $\mathcal{R}$, we say that $\mathcal{P}$ is 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. 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. Observe that every inductively sequential TRS is orthogonal, but the inverse does not generally hold. For instance, the following orthogonal TRS:

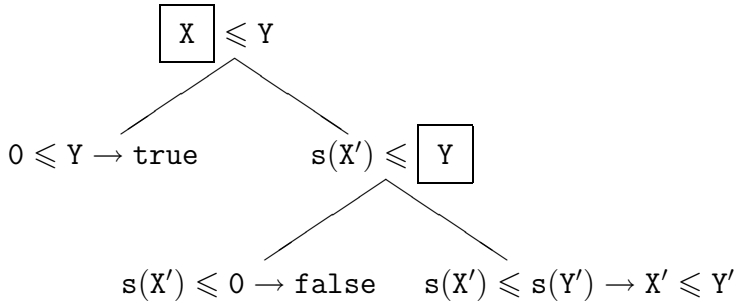
$$\mathtt{f(X,a,b)} \to \mathtt{a} \qquad \mathtt{f(a,X,b)} \to \mathtt{b} \qquad \mathtt{f(a,b,X)} \to \mathtt{c}$$

where $\mathtt{X}$ is a variable and $\mathtt{a}, \mathtt{b}, \mathtt{c}$ are constructor symbols, is not inductively sequential since there is no definitional tree for function f. An inductively sequential TRS can be viewed as a set of definitional trees, each one 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 the understanding to provide a graphic representation of definitional trees, where each node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules. For instance, the definitional tree for the function "$\leqslant$":

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

can be depicted as follows:



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) 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 some 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 $t|_o$:

$$\lambda(t, \mathcal{P}) \ni \begin{cases} (p, R, \sigma \circ \tau) & \text{if } t|_o = x \in \mathcal{X},\ \tau = \{x \mapsto c_i(\overline{x_n})\}, \text{ and} \\ & \quad (p, R, \sigma) \in \lambda(\tau(t), \mathcal{P}_i); \\ (p, R, \sigma \circ id) & \text{if } t|_o = c_i(\overline{t_n}) \text{ and } (p, R, \sigma) \in \lambda(t, \mathcal{P}_i); \\ (o.p, R, \sigma \circ id) & \text{if } t|_o = f(\overline{t_n}) \text{ for } f \in \mathcal{F} \text{ and} \\ & \quad (p, R, \sigma) \in \lambda(t|_o, \mathcal{P}') \text{ where } \mathcal{P}' \\ & \quad \text{is a definitional tree for } f. \end{cases}$$

Informally speaking, needed narrowing directly applies a rule if the term is an instance of some left-hand side (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 some 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. Thus, the strategy differs from lazy functional languages only in the instantiation of free variables. In contrast to more traditional narrowing strategies, needed narrowing does not compute *most general* unifiers. 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 binding φ_j is either the identity or the replacement of a single variable. This is called the *canonical representation* of a needed narrowing step.

As in proof procedures for logic programming, we assume that 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.*

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

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

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

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

This corresponds to the following narrowing steps (the subterm evaluated in the next step is underlined):

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

Needed narrowing is sound and complete for inductively sequential programs. Moreover, it is *minimal*, i.e., given two distinct needed narrowing derivations $e \leadsto^*_\sigma true$ and $e \leadsto^*_{\sigma'} true$, we have that σ and σ' are disjoint on $\mathcal{V}ar(e)$. These properties are formalized as follows:

Theorem 2 [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 disjoint on $\mathcal{V}ar(e)$.

3 The Transformation Rules

In this section, our aim is to define a set of program transformations which is strongly correct, i.e., sound and complete w.r.t. the semantics of values and answer substitutions computed by needed narrowing. Let us first give the rule for the introduction and elimination of function definitions in a similar style to [46], in which the set of definitions is partitioned into "old" and

"new" functions. In the following, we consider a fixed transformation sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq 0$.

Definition 3 (Definition introduction) *We may get program $\mathcal{R}_{k+1}$ by adding to $\mathcal{R}_k$ a new rule (called a "definition rule" or "eureka") of the form $f(\overline{t_n}) \to r$, such that:*

(1) $f(\overline{t_n})$ is a linear pattern and $\mathcal{V}ar(f(\overline{t_n})) = \mathcal{V}ar(r)$—i.e., the rule is non-erasing—,
(2) f does not occur in the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_k$ (f is new*), and*
(3) every defined function symbol occurring in r belongs to $\mathcal{R}_0$.

We say that f is a new *function symbol, and every function symbol belonging to $\mathcal{R}_0$ is called an* old *function symbol.*

The introduction of a new eureka definition is virtually always the first step of a transformation sequence. Determining which definitions should be introduced is a clever task (which justifies the name "eureka" for the new rules) which falls into the realm of *strategies* (see [40] for a survey), as we discuss in Section 5. Let us note that only one program rule can be introduced with our definition introduction rule. This means that function definitions consisting of several rules with different patterns in the left-hand side cannot be introduced. Nevertheless, this is not a limitation in practice since the rules introduced by the transformation strategies which we consider in Section 5 have the form $f(\overline{x_n}) \to exp$, where $\overline{x_n}$ are the different variables occurring in exp. This is common in almost all transformation strategies that can be found in the literature. Now, we would like to advance two important aspects which are strongly related with the folding operation that we will see later:

(1) A *eureka rule* maintains its status only as long as no transformation step is applied on it. Once we transform this rule—even if the resulting rule is syntactically equal to the original one—it is not considered a *eureka rule* anymore. As we will see in Definition 8, this is important for the folding operation, since we can only fold *non eureka rules* using *eureka rules*.
(2) The *non-erasing* condition is required for avoiding the creation of rules with extra-variables when performing folding steps. Consider, for instance, the folding of rule $\mathtt{f(X)} \to \mathtt{g(X)}$ using the (incorrect) eureka $\mathtt{new(X,Y)} \to \mathtt{g(X)}$ which would produce a new rule $\mathtt{f(X)} \to \mathtt{new(X,Y)}$ containing an extra-variable in its rhs (thus an illegal rule).

Definition 4 (Definition elimination) *We may get program $\mathcal{R}_{k+1}$ by deleting from program $\mathcal{R}_k$ all rules defining the functions $f_0, \ldots, f_n$ ($n \geq 0$), say R^f, such that $f_0, \ldots, f_n$ do not occur in $\mathcal{R}_0$ nor in $(\mathcal{R}_k - R^f)$.*

This rule has been initially proposed with the name of *deletion* (for logic programs) in [36] and also in [12], where it was called *restriction*. Note that

the deletion of the rules defining a function f implies that no function calls to f are allowed afterwards. However, subsequent transformation steps (in particular, folding steps) might introduce those deleted functions in the rhs's of the rules, thus producing inconsistencies in the resulting programs. We avoid this encumbrance by the usual requirement [40]: do not allow folding steps if a definition elimination has been previously performed.

Now we introduce our unfolding rule, which systematizes a fit combination of instantiation and classical (functional) unfolding into a single transformation. Essentially, it exploits the capability of narrowing to deal with logical variables.

Definition 5 (Unfolding) *Let $R = (l \to r) \in \mathcal{R}_k$ be a rule (the "unfolded rule") whose rhs is an operation-rooted term. We may get program $\mathcal{R}_{k+1}$ from $\mathcal{R}_k$ by replacing R with the set of rules:*

$$\{\theta(l) \to r' \mid r \leadsto_\theta r' \text{ is a needed narrowing step in } \mathcal{R}_k\} \ .$$

Here, it is worth noting that the requirement not to unfold a rule whose rhs is constructor-rooted can be left aside when functions are *totally defined* (which is quite usual in typed languages). The following example shows that the above requirement cannot be dropped in general.

Example 6 *Consider the following programs:*

$$\mathcal{R} = \left\{ \begin{array}{l} \mathtt{f(0)} \to \mathtt{0} \\ \mathtt{g(X)} \to \mathtt{s(f(X))} \\ \mathtt{h(s(X))} \to \mathtt{s(0)} \end{array} \right\} \qquad \mathcal{R}' = \left\{ \begin{array}{l} \mathtt{f(0)} \to \mathtt{0} \\ \mathtt{g(0)} \to \mathtt{s(0)} \\ \mathtt{h(s(X))} \to \mathtt{s(0)} \end{array} \right\}$$

We get $\mathcal{R}'$ from $\mathcal{R}$ by applying an unfolding step which considers the needed narrowing derivation $\mathtt{s(f(X))} \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} \mathtt{s(0)}$ for the rhs of the second rule in $\mathcal{R}$. Now, the term $\mathtt{h(g(s(0)))}$ has the following needed narrowing derivation in $\mathcal{R}$:

$$\mathtt{h}(\underline{\mathtt{g(s(0))}}) \ \leadsto \ \underline{\mathtt{h(s(f(s(0))))}} \ \leadsto \ \mathtt{s(0)}$$

whereas it is no longer possible in the transformed program. Essentially, completeness is lost because the unfolding rule, $\mathtt{f(0)} \to \mathtt{0}$, defines a function $\mathtt{f}$ which is not totally *defined. Hence, by unfolding the call $\mathtt{f(X)}$ we improperly "compile in" an unnecessary restriction in the domain of the function $\mathtt{g}$.*

The following example illustrates the fact that the choice of the operational principle of needed narrowing to drive the unfolding steps is not only convenient but also crucial to guarantee that the structure of the original program,

i.e., inductive sequentiality, is preserved through the transformation.

Example 7 *Consider the following inductively sequential (hence also orthogonal) program:*

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

Note that the goal $\mathtt{test(X,Y)}$ *reduces to* $\mathtt{0}$ *if* $\mathtt{X}$ *is bound to* $\mathtt{s(N)}$*, and it enters a loop if* $\mathtt{X}$ *is bound to* $\mathtt{0}$ *due to the non terminating evaluation of* $\mathtt{g(0)}$*.*

Now, by unfolding the first rule using needed narrowing and lazy narrowing,[1] *respectively, we obtain the following rules in the respective programs:*

$$\mathcal{R}_{needed} = \left\{ \begin{array}{c} \mathtt{test(0,0)} \to \mathtt{h(f(0,g(0)))} \\ \mathtt{test(s(X),Y)} \to \mathtt{h(s(f(X,g(Y))))} \end{array} \right\}$$

$$\mathcal{R}_{lazy} = \left\{ \begin{array}{c} \mathtt{test(X,0)} \to \mathtt{h(f(X,g(0)))} \\ \mathtt{test(s(X),Y)} \to \mathtt{h(s(f(X,g(Y))))} \end{array} \right\}$$

Note that $\mathcal{R}_{lazy}$ *is neither inductively sequential nor orthogonal, in contrast to the original program* $\mathcal{R}$*. This does not only mean that the unfolded program cannot be executed by using needed narrowing (nor lazy narrowing), but also that* $\mathcal{R}_{lazy}$ *has a worse termination behaviour than* $\mathcal{R}_{needed}$ *(and also worse than* $\mathcal{R}$*). For instance, the term* $\mathtt{test(s(0),0)}$ *has a finite derivation tree w.r.t.* $\mathcal{R}_{needed}$ *(using either needed or lazy narrowing) whereas the (lazy narrowing) derivation tree w.r.t.* $\mathcal{R}_{lazy}$ *is infinite in depth (caused by the application of the rules* $\mathtt{test(X,0)} \to \mathtt{h(f(X,g(0)))}$ *and* $\mathtt{g(0)} \to \mathtt{g(0)}$*).*

Now, let us introduce the folding rule, which is a counterpart of the previous transformation, i.e., the compression of a piece of code into an equivalent call.

Definition 8 (Folding) *Let* $R = (l \to r) \in \mathcal{R}_k$ *be a rule (the "folded rule") and let* $R' = (l' \to r') \in \mathcal{R}_j$*,* $0 \leq j \leq k$*, be a rule (the "folding rule") such that* $r|_p = \theta(r')$ *for some position* $p \in \mathcal{NVP}os(r)$ *and substitution* θ*, fulfilling*

[1] We consider the (demand–driven) lazy narrowing strategy described in [39], which is also driven by the patterns in the lhs's of the rules (though it is "less lazy" than needed narrowing).

the following conditions:

(1) the folded rule R is not a eureka rule and
(2) the folding rule R' is a eureka rule.

Then, we may get program $\mathcal{R}_{k+1}$ from program $\mathcal{R}_k$ by replacing the rule R with the new rule $l \to r[\theta(l')]_p$.

Roughly speaking, the folding operation proceeds in a contrary direction to the usual reduction steps, i.e., reductions are performed against the reversed folding rules. There are several points regarding our definition of the folding rule which are worth noticing:

- The condition which says that the folded rule R is not a eureka rule means that either l (the lhs of R) is rooted by an *old* function symbol or R is the result of unfolding a eureka rule at least once within the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_k$.
- In contrast to the unfolding rule, the subterm which is selected for the folding step needs not be a (needed) narrowing redex. This generality is not only safe but also helpful as it will become apparent in Section 5.
- The substitution θ of Definition 8 is not a unifier (as occurs in the more involved version of [3]) but just a matcher, similarly to many other folding rules for logic programs, which have been defined in a similar "functional style" (see, e.g., [12,31,41,46]). Moreover, it has the extra advantage that it is easier to check and can still produce—at a very low cost—many powerful optimizations.

Many proposals have also been made to define a folding transformation in a (pure) functional context [14,18,33,43]. A *marked* folding for a lazy (higher-order) functional language has been presented in [42], which preserves the semantics of (ground constructor) values under applicability conditions which are similar to ours. However, our correctness results are slightly stronger, since we preserve the (non-ground) semantics of computed values and *answers*.

As in our definition of folding, a large number of proposals also allow the folded and the folding rule to belong to different programs (see, e.g., [12,31,40,41,46]), which is crucial to achieve an effective optimization in many cases. Some other works in the literature have advocated a different style of folding which is *reversible* [23], i.e., a kind of folding which can always be undone by an unfolding step. This greatly simplifies the correctness proofs—correctness of folding follows immediately from the correctness of unfolding—, but usually require too strong applicability conditions, such as requiring that both the folded and the folding rules belong to the same program. This drastically reduces the power of the transformation. The folding rule proposed in [3] for a strict functional logic language is reversible and thus its transformational power is very limited. Our folding rule is more powerful and practical since the applicability conditions are less restrictive. Therefore, its use within a

transformation system—when guided by appropriate strategies—is able to produce more effective optimizations for (lazy) functional logic programs.

The set of rules presented so far constitutes the kernel of our transformation system. These rules suffice to automate the *composition* strategy (see Section 5). Other (functional) approaches include a new rule called abstraction [14,42] (often known as *where–abstraction* rule [41]) that can be simulated in our setting by applying appropriate definition introduction and folding steps. This rule is usually required to implement tupling and it essentially consists of replacing the occurrences of some expression e in the rhs of a rule R by a fresh variable z, adding the "local declaration" $z = e$ within a *where* expression in R. For instance, the rule

$$\texttt{double\_sum(X,Y)} \;\rightarrow\; \texttt{sum(sum(X,Y),sum(X,Y))}$$

can be transformed into the new rule

$$\texttt{double\_sum(X,Y)} \;\rightarrow\; \texttt{sum(Z,Z)} \;\; \texttt{where Z} = \texttt{sum(X,Y)} \; .$$

As noted by [41], the use of the where-abstraction rule has the advantage that, in the call-by-value mode of execution, the evaluation of the expression e is performed only once. This is also true in a lazy context under an implementation based on *sharing*, which allows us to keep track of variables which occur several times in the expression to be evaluated.

The new rules introduced by the where-abstraction rule often contain extra variables in the right-hand sides. However, as noted in [42], this can easily be amended by using standard "lambda lifting" techniques (which can be thought of as an appropriate application of a definition introduction step followed by a folding step). For instance, if we consider again the rule

$$\texttt{double\_sum(X,Y)} \;\rightarrow\; \texttt{sum(Z,Z)} \;\; \texttt{where Z} = \texttt{sum(X,Y)},$$

we can transform it (by lambda lifting [29]) into the new pair of rules:

$$\begin{aligned} \texttt{double\_sum(X,Y)} &\rightarrow \texttt{ds\_aux(sum(X,Y))} \\ \texttt{ds\_aux(Z)} &\rightarrow \texttt{sum(Z,Z)} \end{aligned}$$

Note that these rules can be directly generated from the initial definition by a definition introduction ($\texttt{ds\_aux(Z)} \rightarrow \texttt{sum(Z,Z)}$) and then by folding the original rule at the expression $\texttt{sum(sum(X,Y),sum(X,Y))}$ using as folding rule the newly generated definition for `ds_aux`. The inclusion of an abstraction rule is

traditional in functional fold/unfold frameworks [14,41–43]. In the case of logic programs, abstraction is only possible by means of the so called *generalization* strategy [41], which generalizes some calls to eliminate the mismatches that prevent a folding step.

Now, we are ready to formalize our abstraction rule, which is inspired by the standard lambda lifting transformation of functional programs. By means of the tuple constructor $\langle\ \rangle$, our definition allows the abstraction of different expressions in one go. For a sequence of (pairwise disjoint) positions $P = \overline{p_n}$, we let $t[\overline{s_n}]_P = (((t[s_1]_{p_1})[s_2]_{p_2}) \ldots [s_n]_{p_n})$. By abuse, we denote $t[\overline{s_n}]_P$ by $t[s]_P$ when $s_1 = \ldots = s_n = s$, as well as $((t[s_1]_{P_1}) \ldots [s_n]_{P_n})$ by $t[\overline{s_n}]_{\overline{P_n}}$.

Definition 9 (Abstraction) *Let $R = (f(\overline{t_n}) \to r) \in \mathcal{R}_k$ be a non eureka rule and let $\overline{P_j}$ be sequences of disjoint positions in $\mathcal{NVP}os(r)$ such that $r|_p = e_i$ for all p in P_i, $i = 1, \ldots, j$, i.e., $r = r[\overline{e_j}]_{\overline{P_j}}$. We may get program $\mathcal{R}_{k+1}$ from program $\mathcal{R}_k$ by replacing R with the rules:*

$$R_{abs} = (f(\overline{t_n}) \to f\_aux(\overline{y_m}, \langle e_1, \ldots, e_j \rangle))$$

$$R_{new} = (f\_aux(\overline{y_m}, \langle z_1, \ldots, z_j \rangle) \to r[\overline{z_j}]_{\overline{P_j}})$$

where $\overline{z_j}$ are fresh variables not occurring in $\overline{t_n}$, $\mathcal{V}ar(r[\overline{z_j}]_{\overline{P_j}}) = \{\overline{y_m}, \overline{z_j}\}$, f_aux is a new function symbol that does not occur in $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, and, on the contrary, $r[\overline{z_j}]_{\overline{P_j}}$ does not contain new function symbols.

As we mentioned before, the transformation which is achieved by our abstraction rule can also be obtained, in our framework, by a definition introduction step (that generates the eureka rule R_{new}) followed by a folding step (that folds the rule to be abstracted R using R_{new} and delivers the desired rule R_{abs}). Informally speaking, the two rules generated by the abstraction transformation can be seen as a syntactic variant of the following rule:

$$f(\overline{t_n}) \ \to\ r[\overline{z_j}]_{\overline{P_j}} \ \ \textit{where}\ \langle z_1, \ldots, z_j \rangle = \langle e_1, \ldots, e_j \rangle$$

using the conventional notation of `where` clauses. In the examples shown in sections 5 and 6, we will use this sugared syntax for readability reasons.

4 Correctness Properties of the Transformation System

In this section, we state and prove the main theoretical results for the transformation system based on the elementary rules introduced so far: definition introduction, definition elimination, unfolding, folding, and abstraction. We

state the correctness of the transformation system, as well as the preservation of the structure of programs, for transformation sequences constructed from an inductively sequential program. In proving this, we assume that no folding step is applied after a definition elimination, which guarantees that no function call to a previously deleted function is introduced [40]. The following example illustrates the need for this requirement.

Example 10 *Consider the following inductively sequential program $\mathcal{R}_i$:*

$$\mathtt{f(X)} \to \mathtt{0}\ (R_1)$$
$$\mathtt{g(X)} \to \mathtt{0}\ (R_2)$$

and assume that R_2 is a eureka rule. Firstly, we perform a definition elimination step on the new function `g`*. In this way, we obtain program R_{i+1}:*

$$\mathtt{f(X)} \to \mathtt{0}\ (R_1)$$

Now, we fold $R_1 \in \mathcal{R}_{i+1}$ using the folding rule $R_2 \in \mathcal{R}_i$, obtaining $\mathcal{R}_{i+2}$:

$$\mathtt{f(X)} \to \mathtt{g(X)}\ (R_3)$$

Observe that the term `f(X)` *can be reduced to* `0` *in $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ but not in $\mathcal{R}_{i+2}$. The problem arises because the folding rule (unlike any other transformation rule) is able to introduce new symbols in the rhs of a rule, even when these function symbols are undefined (since the rules defining them have been previously removed).*

Therefore, we only consider transformation sequences fulfilling the following conditions:

- the initial program of the transformation sequence is inductively sequential, and
- no folding step is applied after a definition elimination step.

Moreover, since the abstraction rule can easily be recast in terms of definition introduction and folding, w.l.o.g., in the following we restrict ourselves to transformation sequences which do not use this rule.

The following theorem states that transformation sequences preserve the inductively sequential structure of programs:

Theorem 11 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, be a transformation sequence such*

that $\mathcal{R}_0$ is an inductively sequential program. Then, $\mathcal{R}_i$ is inductively sequential, for $i = 1, \ldots, n$.

The formal proof can be found in Appendix A. Essentially, this result is a direct consequence of the following facts:

- By applying a definition introduction/elimination transformation to a given program, we simple add/remove some definitional trees for new/old rules, and hence the resulting program is inductively sequential.
- The folding operation does not change the lhs of any program rule, which also implies that the set of definitional trees is not modified.
- As for the unfolding operation, the result follows easily from [6], where a similar result is proven for the partial evaluation based on needed narrowing.

In the following, we state and prove the strong correctness of the transformation system, i.e., that the set of values and computed answer substitutions for a given goal are exactly the same (up to variable renaming) for each program of the transformation sequence.

In [42], Sands formalizes a syntactic *improvement* theory which restricts general fold/unfold transformations and can be applied to give correctness proofs for some existing transformation methods (such as, e.g., deforestation [48]). However, we find it more convenient to stick to the logic programming methods for proving the correctness because the narrowing mechanism can be properly seen as a generalization of the SLD-resolution method. In other words, instantiation is computed in a systematic way by the needed narrowing mechanism (as in the unfolding of logic programs), whereas it is not restricted in the Burstall and Darlington's fold/unfold framework considered in [42]. Unrestricted instantiation is problematic since it does not even preserve local equivalence (i.e., each local transformation step cannot always be proved correct); for this reason, the instantiation rule is not considered explicitly in [42]. As a consequence, the improvement theorem of [42] does not directly apply to our context.

Let us introduce the key ideas for the proof. Our proof technique is inspired by the original proof scheme of Tamaki and Sato [46] concerning the least Herbrand model semantics of logic programs (and the subsequent extension of Kawamura and Kanamori [31] for the semantics of computed answers). We have retained some of their notation, in particular, the notion of a *virtual* transformation sequence, and have introduced a new notion of "rank-consistency". The main idea behind the notion of a virtual sequence is to consider that the last program in a transformation sequence can always be obtained (in an ordered way) by anticipating all the definition introduction steps at the beginning of the sequence and by delaying all the definition elimination steps at the end of the same sequence. Note that this is always possible

since no folding is allowed after a definition elimination. Thus, we assume that no transformation step changes the signature of the program, i.e., the same set of (new and old) function symbols is fixed throughout.

Definition 12 (Virtual sequence) *Given a transformation sequence of the form* $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$*, we define a* virtual *transformation sequence* $(\mathcal{R}'_0, \ldots, \mathcal{R}'_n)$ *as a transformation sequence satisfying the following:*

(1) $\mathcal{R}'_0 = \mathcal{R}_0 \cup \mathcal{R}_{new}$*, where* $\mathcal{R}_{new}$ *contains all the eureka rules introduced in* $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$*.*
(2) The sequence $(\mathcal{R}'_0, \ldots, \mathcal{R}'_n)$ *is constructed by applying only the rules: unfolding, folding, and abstraction.*[2]
(3) If some definitions have been eliminated in $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$*, then by simply eliminating the same definitions in* $\mathcal{R}'_n$ *we obtain exactly* $\mathcal{R}_k$*.*

Intuitively speaking, a fold/unfold transformation system is correct if there are "at least as many unfolds as there are folds" or, equivalently, if "going backward in the computation (as folding does) does not prevail over going forward in the computation (as unfolding does)" [40,42]. This essentially means that there must be a kind of "computational cost" measure which is not increased either by folding or by unfolding steps. Several definitions for this measure can be found in the literature: the *rank of a goal* in [46], the *weight of a proof tree* in [31], or the notion of *improvement* in [42]. In our context, we have introduced the notion of *rank of a term* in order to measure the computational cost of a given term. Our definition is slightly more involved than previous ones in the literature since we have to take into account the particularities of the needed narrowing strategy, which we formalize as follows.

In the following, we denote by $\mathcal{D} = [s \rightarrow^* s']$ a rewrite sequence from term s to term s'; the number of rewrite steps in the reduction sequence $\mathcal{D}$ is expressed by $length(\mathcal{D})$.

Definition 13 (Rank of a term) *Let* $\mathcal{R}_0$ *be the initial program in a virtual transformation sequence. Let* s *be a term such that* $s \rightarrow^* t$ *in* $\mathcal{R}_0$*, where* t *is a constructor term. Let* $\mathcal{D} = [s \rightarrow^* s']$ *be a reduction sequence in* $\mathcal{R}_0$ *that only uses eureka rules, and such that no more eureka rules can be applied to* s'*. Let* $(\mathcal{D}_0, \ldots, \mathcal{D}_m)$*,* $m \geq 0$*, be all the possible needed reduction sequences from* s' *to* t *in* $\mathcal{R}_0$ *with* $n = max(length(\mathcal{D}_0), \ldots, length(\mathcal{D}_m))$*. Then,* $rank(s) = n$*.*

In the previous definition we require $(\mathcal{D}_0, \ldots, \mathcal{D}_m)$, $m \geq 0$, to be needed reduction sequences, i.e., derivations in which only needed redexes are contracted. This restriction is necessary in order to ensure that the rank of a term is never infinite (hence it is well defined), since it suffices to reduce a finite number of

[2] In practice, only folding and unfolding rules are considered, since abstraction is recast in terms of definition introduction and folding.

needed redexes in order to obtain the constructor normal form (if it exists) of a term [28].

We have considered the *longest* needed reduction sequence since unfolding based on needed narrowing may increase the length of (arbitrary) needed reductions and, thus, the notion of rank in terms of the shortest sequence would not be preserved by unfolding. Folding steps may also introduce some additional reduction steps in the needed reductions for a given term. In this case, we solve the problem by not taking into account the (finite number of) steps performed with eureka rules.

Following a proof scheme similar to [46], we prove the correctness of our transformation rules by showing that the following invariants hold for each program in a (virtual) transformation sequence.

Definition 14 (Invariants) *The invariants of a program $\mathcal{R}_i$ in a virtual transformation sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq i \geq 0$, are defined as follows:*

(I1) Given an equation e, $e \to^* true$ in $\mathcal{R}_i$ iff $e \to^* true$ in $\mathcal{R}_0$.
(I2) Let $\mathcal{D} = [e_0 \to^* e_n]$, $n \geq 0$, be a needed reduction sequence in $\mathcal{R}_i$ such that e_0 is an equation and $e_n = true$. For every reduction step $e_j \to_R e_{j+1}$ in $\mathcal{D}$, $0 \leq j \leq n-1$, we have that $rank(e_j) = rank(e_{j+1})$ if R is a eureka rule, and $rank(e_j) > rank(e_{j+1})$ otherwise.

In the following, we state and prove the strong correctness of the transformation system. As discussed before, we can restrict our discourse to *virtual* transformation sequences. Hence, we only need to prove that the unfolding and folding transformations preserve the invariants of Definition 14, since the final result is a direct consequence of this fact (as we will see afterwards). An advantage of our proof scheme is that, in order to prove the above result, it suffices to prove the soundness and completeness of folding/unfolding at the level of rewrite sequences. Then, correctness can easily be lifted to narrowing derivations by using the properties of Theorem 2. For the case of unfolding, the following result establishes this kind of soundness and completeness for the considered transformation. Analogously to Theorem 11, the following lemma is an immediate consequence of similar results for the partial evaluation technique presented in [6].

Lemma 15 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$, be a transformation sequence such that $\mathcal{R}_{i+1}$ has been obtained by unfolding and let e be an equation. Then,*

(Soundness) If $e \to^* true$ in $\mathcal{R}_{i+1}$ then $e \to^* true$ in $\mathcal{R}_i$.
(Completeness) If $e \to^* true$ in $\mathcal{R}_i$ then $e \to^* true$ in $\mathcal{R}_{i+1}$.

The following proposition states a useful property of folding.

Proposition 16 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_k, \mathcal{R}_{k+1})$, $k \geq 0$, be a transformation sequence. If $\mathcal{R}_{k+1}$ has been obtained by folding using the folded rule $R \in \mathcal{R}_k$, then R is not a eureka rule.*

PROOF. By Definition 8, we have that the folded rule $R = (l \to r) \in \mathcal{R}_k$ verifies that either l is rooted by an old function symbol or R is the result of at least one unfolding within the sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$. By Definition 3, we know that a eureka rule is rooted with a new function symbol and, moreover, it loses its status once it suffers a transformation (like unfolding). Hence, the sets of eureka and folded rules are disjoint.

Now we prove the preservation of invariants (see Definition 14) within a (virtual) transformation sequence. The following lemma states that the invariants hold in the initial program of the sequence.

Lemma 17 *Let $\mathcal{R}_0$ be the initial program of a virtual transformation sequence. Then, invariants (I1) and (I2) hold in $\mathcal{R}_0$.*

PROOF. Invariant (I1) trivially holds in $\mathcal{R}_0$, hence we proceed to prove that (I2) also holds in $\mathcal{R}_0$. Let $\mathcal{D} = [e_0 \to^* e_n]$, $n \geq 0$, be a needed reduction sequence in $\mathcal{R}_0$ such that e_0 is an equation and $e_n = \mathit{true}$. For every reduction step $e_j \to_R e_{j+1}$ in $\mathcal{D}$, $0 \leq j \leq n-1$, we must prove that $\mathit{rank}(e_j) = \mathit{rank}(e_{j+1})$ if R is a eureka rule, and $\mathit{rank}(e_j) > \mathit{rank}(e_{j+1})$ otherwise. We prove the claim by induction on $n = \mathit{length}(\mathcal{D})$:

$n = 0$**.** This case is immediate since $e_0 = \mathit{true}$.

$n > 0$**.** Then we have $\mathcal{D} = [e_0 \to_{q,R} e_1 \to^* \mathit{true}]$. If R is a eureka rule, by definition of rank, we have that $\mathit{rank}(e_0) = \mathit{rank}(e_1)$. Hence, the claim follows by the inductive hypothesis.

If R is not a eureka rule, we have the following pair of reduction sequences in $\mathcal{R}_0$:

$$\mathcal{D}_0 = [e_0 \to^* e'_0] \quad \text{and} \quad \mathcal{D}_1 = [e_1 \to^* e'_1]$$

where only eureka rules have been used in $\mathcal{D}_0$ and $\mathcal{D}_1$, and no more eureka rules are applicable to e'_0 and e'_1. Now, observe that $e_0|_q$ is a needed redex rooted with an old function symbol, since it can be reduced by the non eureka rule R. Since only eureka rules are used in $\mathcal{D}_0$ and they are *non-erasing* (see Definition 3), we have that there exists at least one descendant of $e_0|_q$ in e'_0. Moreover, since $e_0|_q$ is a needed redex in e_0, then we have that its descendants are also needed redexes in e'_0. Assume that such needed redexes are $e'_0|_{p_1}, \ldots, e'_0|_{p_k}$, $\{p_1, \ldots, p_k\} \subseteq \mathcal{P}os(e'_0)$, $k > 0$. Hence, the following needed reduction sequence $e'_0 \to_{p_1,R} \ldots \to_{p_k,R} e'_1$ can be proven in $\mathcal{R}_0$ by

repeatedly contracting all the needed redexes using the (non eureka) rule R.

By Definition 13, we have that $rank(e_0) = rank(e'_0)$ and $rank(e_1) = rank(e'_1)$. Assume that the longest needed reduction sequence for e'_1 to *true* is $\mathcal{D}'_1 = [e'_1 \rightarrow^m true]$. Then $rank(e_1) = rank(e'_1) = m$. Thus, since e'_0 admits a needed reduction sequence to *true* of the form $e'_0 \rightarrow_{p,R} e'_1 \rightarrow^m true$, then the rank of e'_0 is at least $m+1$. This implies that $rank(e_0) = rank(e'_0) \geq m + 1 > m = rank(e'_1) = rank(e_1)$. Finally, since $rank(e_0) > rank(e_1)$, the claim follows by the inductive hypothesis.

The following lemmata prove that invariants are preserved both by folding and unfolding steps.

Lemma 18 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$, be a virtual transformation sequence and e an equation. If the invariant (I1) holds in $\mathcal{R}_i$ then, if $e \rightarrow^* true$ in $\mathcal{R}_{i+1}$ then $e \rightarrow^* true$ in $\mathcal{R}_i$.*

PROOF. Let e be an equation such that $\mathcal{D} = [e \rightarrow^* true]$ in $\mathcal{R}_{i+1}$. We prove the claim by induction on $length(\mathcal{D})$.

$n = 0$**.** This case is immediate since $e = true$.

$n > 0$**.** Let $\mathcal{D} = [e \rightarrow_{q,R^*} e' \rightarrow^* true]$. If $R^* \in \mathcal{R}_i$, then the claim follows by the inductive hypothesis. Otherwise, R^* has been obtained from $\mathcal{R}_i$ by an unfolding or folding step. In the first case, the claim holds by Lemma 15 (claim 1). In the second case, there exist rules $R = (l \rightarrow r) \in \mathcal{R}_i$ and $R' = (l' \rightarrow r') \in \mathcal{R}_0$ such that R^* is the result of folding R using R'. Then $r|_p = \theta(r')$ for some position $p \in Pos(r)$ and substitution θ, with $R^* = (l \rightarrow r[\theta(l')]_p)$. Hence, we have $\mathcal{D} = [e \rightarrow_{q,R^*} e[\sigma(r[\theta(l')]_p)]_q = e' \rightarrow^* true]$ where $e|_q = \sigma(l)$. By the inductive hypothesis, we have $e[\sigma(r[\theta(l')]_p)]_q \rightarrow^* true$ in $\mathcal{R}_i$. Since the invariant (I1) holds in $\mathcal{R}_i$, we have that $e[\sigma(r[\theta(l')]_p)]_q \rightarrow^* true$ in $\mathcal{R}_0$ too. By definition of folding, the folding rule $R' = (l' \rightarrow r')$ must be a eureka rule, and hence $R' \in \mathcal{R}_0$. Therefore, the following rewrite step can be proven in $\mathcal{R}_0$:

$$e[\sigma(r[\theta(l')]_p)]_q \rightarrow_{q.p,R'} e[\sigma(r[\theta(r')]_p)]_q = e[\sigma(r[r_{|p}]_p)]_q = e[\sigma(r)]_q$$

Since $\mathcal{R}_0$ is inductively sequential, it is confluent, and thus $e[\sigma(r[\theta(l')]_p)]_q \rightarrow_{q.p,R'} e[\sigma(r)]_q \rightarrow^* true$. Applying invariant (I1) again, we have $e[\sigma(r)]_q \rightarrow^* true$ in $\mathcal{R}_i$. Finally, since $R = (l \rightarrow r) \in \mathcal{R}_i$ and $e|_q = \sigma(l)$ we have $e \rightarrow_{q,R} e[\sigma(r)]_q \rightarrow^* true$ in $\mathcal{R}_i$, which completes the proof.

Lemma 19 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$ be a virtual transformation sequence and e an equation. If invariants (I1) and (I2) hold in $\mathcal{R}_i$ then: if $e \rightarrow^* true$ in $\mathcal{R}_i$ then $e \rightarrow^* true$ in $\mathcal{R}_{i+1}$.*

PROOF. First we define the well-founded ordering $\gg$ on equations (e, e') that can be reduced to *true* in $\mathcal{R}_0$ (and also in $\mathcal{R}_i$, by invariant (I1)) as follows. We say that $e \gg e'$ iff

(1) $rank(e) > rank(e')$ or
(2) $rank(e) = rank(e')$ and there is a eureka rule $R \in \mathcal{R}_0$ such that $e \to_R e'$.

We consider that *true* is the minimal element under the well-founded ordering $\gg$. Now we proceed with the proof of the lemma. Let $\mathcal{D} = [e \to^* true]$ be a needed reduction sequence in $\mathcal{R}_i$. Since the invariant (I2) holds in $\mathcal{R}_i$ we have that, for every reduction step $e_j \to_R e_{j+1}$ in $\mathcal{D}$, $rank(e_j) = rank(e_{j+1})$ if R is a eureka rule, and $rank(e_j) > rank(e_{j+1})$ otherwise. This implies that $e_j \gg e_{j+1}$. Now, we construct a reduction sequence for e to *true* in $\mathcal{R}_{i+1}$ by induction on the well-founded ordering.

The base case obviously holds since $e = true$. Otherwise, $\mathcal{D}$ is of the form $\mathcal{D} = [e \to_R e' \to^* true]$ in $\mathcal{R}_i$, where $e \gg e'$ by invariant (I2). If $R \in \mathcal{R}_{i+1}$, then the claim follows by the inductive hypothesis. Otherwise, we distinguish two cases:

(1) If R is unfolded in $\mathcal{R}_i$, we have that there exists the following needed reduction sequence in $\mathcal{R}_i$ (by a similar argument as in the proof of Lemma 15 in Appendix B): $\mathcal{D} = [e \to_{p,R} e' \to_{p.q,R'} e'' \to^* true]$ in $\mathcal{R}_i$, such that the result of unfolding $R = (l \to r) \in \mathcal{R}_i$ using $R' \in \mathcal{R}_i$ at position $q \in Pos(r)$ is the new rule $R^* \in R_{i+1}$, and $e \to_{p,R^*} e''$. Observe that R and R' are not eureka rules at the same time, since a eureka rule always defines a new function symbol by means of old function symbols. Moreover, since the invariant (I2) holds in $\mathcal{R}_i$ we have that $e \gg e' \gg e''$. By the inductive hypothesis, $e'' \to^* true$ in $\mathcal{R}_{i+1}$, and hence $e \to_{p,R^*} e'' \to^* true$ in $\mathcal{R}_{i+1}$.

(2) If $R = (l \to r)$ is folded in $\mathcal{R}_i$, by Proposition 16, we know that R is not a eureka rule. Since $e \gg e'$ and R is not a eureka rule, by (I2) we conclude that $rank(e) > rank(e')$.

Moreover there exists a folding rule (which is a eureka rule) $R' = (l' \to r') \in \mathcal{R}_0$ such that the result of folding R using R' is the rule $R^* = (l \to r[\theta(l')]_q) \in \mathcal{R}_{i+1}$ (where $r|_q = \theta(r')$ for some position $q \in Pos(r)$).

Since the first step of $\mathcal{D}$ is given with R, then $e' = e[\sigma(r)]_p$, where $e|_p = \sigma(l)$ for some position $p \in Pos(e)$. By (I1), $e' \to^* true$ in $\mathcal{R}_0$. Since $R' \in \mathcal{R}_0$ and $e' = s[\sigma(r)]_p = e[\sigma(r[r_{|q}]_q)]_p = e[\sigma(r[\theta(r')]_q)]_p$, then the following reduction step can be proven in $\mathcal{R}_0$:

$$e'' = e[\sigma(r[\theta(l')]_q)]_p \to_{p.q,R'} e[\sigma(r[\theta(r')]_q)]_p = e'$$

Thus, $e'' \to_{p.q,R'} e' \to^* true$ in $\mathcal{R}_0$ and, by invariant (I1), $e'' \to^* true$ in $\mathcal{R}_i$ too.

Since R' is a eureka rule and $e'' \to_{p.q,R'} e'$, we have that $rank(e'') =$

$rank(e')$. Also, since we have previously seen that $rank(e) > rank(e')$, we conclude that $rank(e) > rank(e'')$ and, thus, $e \gg e''$. By the inductive hypothesis, we have $e'' \rightarrow^* true$ in $\mathcal{R}_{i+1}$. Finally, since $e \rightarrow_{p,R^*} e[\sigma(r[\theta(l')]_q)]_p = e''$, we have $e \rightarrow_{p,R^*} e'' \rightarrow^* true$ in $\mathcal{R}_{i+1}$, which completes the proof.

The following proposition is useful to prove Lemma 21.

Proposition 20 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$, be a virtual transformation sequence. Let e be an equation such that $e \rightarrow^* true$ in $\mathcal{R}_{i+1}$. If $e|_p$ is a needed redex in e w.r.t. $\mathcal{R}_{i+1}$, then $e|_p$ is a needed redex in e w.r.t. $\mathcal{R}_i$.*

PROOF. (sketch) It follows by structural induction on e, since unfolding can only introduce new needed redexes in e (due to the instantiation of some left-hand sides of the program) and folding does not change the needed redexes in e (since left-hand sides remain unchanged).

Lemma 21 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$ be a virtual transformation sequence. If the invariants (I1) and (I2) hold in $\mathcal{R}_i$ then (I2) holds in $\mathcal{R}_{i+1}$.*

PROOF. We prove that (I2) is preserved in every needed reduction sequence $\mathcal{D} = [e \rightarrow^* true]$ in $\mathcal{R}_{i+1}$. The proof is made by induction on the well-founded ordering $\gg$ defined previously.

The base case is immediate since $e = true$. Otherwise, we have that $\mathcal{D} = [e \rightarrow_{p,R^*} e' \rightarrow^* true]$ is a needed reduction sequence in $\mathcal{R}_{i+1}$. Now, we distinguish three cases:

(1) If $\mathcal{R}^* \in \mathcal{R}_i$, by (I1), we have $e \rightarrow_{p,R^*} e' \rightarrow^* true$ in $\mathcal{R}_i$. By Proposition 20, $e \rightarrow_{p,R^*} e'$ is a needed rewriting step in $\mathcal{R}_i$. Combining this with (I2), we have that $e \rightarrow_{p,R^*} e' \rightarrow^* true$ is a needed reduction sequence in $\mathcal{R}_i$ and $e \gg e'$. Then the claim follows by inductive hypothesis.

(2) Otherwise, if $R^* \in \mathcal{R}_{i+1}$ is the result of unfolding $R = (l \rightarrow r) \in \mathcal{R}_i$ using $R' \in \mathcal{R}_i$ at some position $q \in Pos(r)$, there exists the following needed reduction sequence (preserving (I2)) in $\mathcal{R}_i$ (by a similar argument as in the proof of Lemma 15 in Appendix B):

$$e \rightarrow_{p,R} e'' \rightarrow_{p.q,R'} e' \rightarrow^* t$$

Observe that R and R' are not eureka rules at the same time, since a eureka rule always defines a new function symbol by means of old function symbols. Moreover, since invariant (I2) holds in $\mathcal{R}_i$, we have $e \gg e'' \gg e'$. Combining this with the fact that R and R' are not eureka

rules simultaneously, we conclude that $rank(e) > rank(e')$. Putting all pieces together:

(a) $e \to_{p,R^*} e'$ is a needed reduction step given with the (non eureka) rule $R^* \in \mathcal{R}_{i+1}$ verifying that $rank(e) > rank(e')$, which preserves (I2), and

(b) by the inductive hypothesis, since $e \gg e'$, we know that $e' \to^* true$ is a needed reduction sequence preserving (I2) in $\mathcal{R}_{i+1}$.

(3) Finally, if none of the previous cases hold, we have that $R^* \in \mathcal{R}_{i+1}$ has been obtained by a folding step, i.e., there exists a folded rule $R = (l \to r) \in \mathcal{R}_i$ and a folding rule $R' = (l \to r) \in \mathcal{R}_0$ such that $R^* = (l \to r[\theta(l')]_q) \in \mathcal{R}_{i+1}$ (where $r|_q = \theta(r')$ for some position $q \in Pos(r)$). By the conditions in Definition 8 and Proposition 16, we know that R is not a eureka rule whereas R' is a eureka rule.

By (I1), we have $e \to_{p,R} e'' \to^* true$ in $\mathcal{R}_i$, where (for some substitution σ such that $e|_p = \sigma(l)$)

$$e'' = e[\sigma(r)]_p = e[\sigma(r[r_{|q}]_q)]_p = e[\sigma(r[\theta(r')]_q)]_p$$

By Proposition 20, $e \to_{p,R} e''$ is a needed rewriting step in $\mathcal{R}_i$. Then, since (I2) holds in $\mathcal{R}_i$, we conclude that $e \gg e'$. Since $e \gg e''$ and R is not a eureka rule, we have that $rank(e) > rank(e'')$.

By (I1) we have that e'' can be reduced to $true$ in $\mathcal{R}_0$. Moreover, there exists also the following rewriting sequence in $\mathcal{R}_0$:

$$e' = e[\sigma(r[\theta(l')]_q)]_p \to_{p.q,R'} e[\sigma(r[\theta(r')]_q)]_p = e'' \to^* true$$

Since $e' \to_{p.q,R'} e''$ is a reduction step given with a eureka rule in $\mathcal{R}_0$, we have that $rank(e'') = rank(e')$. Combining this with the fact that $rank(e) > rank(e'')$ we conclude that $rank(e) > rank(e')$. Hence $e > > e'$ and, by the inductive hypothesis, $e \to_{p,R^*} e' \to^* true$ is a needed reduction sequence preserving (I2) in $\mathcal{R}_{i+1}$.

The strong correctness of the transformation system is implied by the following proposition:

Proposition 22 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, be a virtual transformation sequence. Then, invariants (I1) and (I2) hold in $\mathcal{R}_i$, for $i = 0, \ldots, n$.*

PROOF. It is a direct consequence of Lemmata 17, 18, 19, and 21.

Now we can state and prove the strong correctness of the transformation system w.r.t. *virtual* transformation sequences. This result follows from Proposition 22 and the correctness and optimality of needed narrowing (Theorem 2).

Theorem 23 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, be a virtual transformation sequence. Let e be an equation with no new function symbol and $V \supseteq \mathcal{V}ar(e)$ a finite set of variables. Then, $e \leadsto^*_\sigma$ true in $\mathcal{R}_0$ iff $e \leadsto^*_{\sigma'}$ true in $\mathcal{R}_n$, with $\sigma' = \sigma\ [V]$ (up to variable renaming).*

PROOF. We distinguish two cases:

Soundness and Completeness. Firstly, we prove the soundness. Since $e \leadsto^*_\sigma$ *true* in $\mathcal{R}_n$, by the soundness of needed narrowing (claim 1 of Theorem 2), we have that $\sigma(e) \to^*$ *true* in $\mathcal{R}_n$. By the preservation of the invariant (I1) (Proposition 22) there is a reduction sequence $\sigma(e) \to^*$ *true* in $\mathcal{R}_0$. Now, by the completeness of needed narrowing (claim 2 of Theorem 2), there exists a needed narrowing derivation $e \leadsto^*_{\sigma'}$ *true* in $\mathcal{R}_0$ such that $\sigma' \leq \sigma\ [\mathcal{V}ar(e)]$, as we wanted to prove. The completeness proof is perfectly analogous.

Strong Soundness and Strong Completeness. We begin with the proof of strong soundness, which is based on contradiction. Assume that there exists a substitution σ' computed by needed narrowing for e in $\mathcal{R}_n$ such that there is no substitution θ computed by needed narrowing for e in $\mathcal{R}_0$ with $\theta = \sigma'\ [\mathcal{V}ar(e)]$ (up to variable renaming). This fact together with the previous soundness result, implies that there exists a substitution σ computed by needed narrowing for e in $\mathcal{R}_0$ such that $\sigma < \sigma'\ [\mathcal{V}ar(e)]$. Moreover, by the previous completeness result, there exists a substitution θ' computed by needed narrowing for e in $\mathcal{R}_n$ such that $\theta' \leq \sigma\ [\mathcal{V}ar(e)]$. Since $\theta' \leq \sigma\ [\mathcal{V}ar(e)]$ and $\sigma < \sigma'\ [\mathcal{V}ar(e)]$, we have that $\theta' < \sigma'\ [\mathcal{V}ar(e)]$, which contradicts the independence of solutions computed by needed narrowing (claim 3 of Theorem 2). The strong completeness proof is perfectly analogous.

Finally, since each transformation sequence can be transformed into an equivalent *virtual* transformation sequence (following Definition 12) which produces the same output program, we have the following corollary of Theorem 23.

Corollary 24 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, be a transformation sequence. Let e be an equation with no new function symbol and $V \supseteq \mathcal{V}ar(e)$ a finite set of variables. Then, $e \leadsto^*_\sigma$ true in $\mathcal{R}_0$ iff $e \leadsto^*_{\sigma'}$ true in $\mathcal{R}_n$, with $\sigma' = \sigma\ [V]$ (up to variable renaming).*

5 Transformation Strategies

In this section we illustrate the usefulness of the transformation rules introduced so far. For this purpose, we show how the well-known strategies of *composition* and *tupling* can easily be reformulated in our setting.

Input: a program $\mathcal{R}$ and a term t

Initialization: $T := \{t\};\ U := \{\ \}$

Repeat

(1) Select a term $t \in T$ and introduce a new definition of the form:
$R\ =\ aux_t(x_1, \ldots, x_k)\ \rightarrow\ t$
where $\mathcal{V}ar(t) = \{x_1, \ldots, x_k\}$ and aux_t is a new function symbol
(2) $T := T \setminus \{t\};\ U := U \cup \{t\}$
(3) Unfold rule R until each unfolded rule, $l \rightarrow r$, fulfills one of the following conditions:
- r is in head normal form or
- r contains a subterm which is an instance of t.

(4) $C := \{r_1, \ldots, r_n\}$, where $l_i \rightarrow r_i$, $i = 1, \ldots, n$, are the unfolded rules obtained in step (3)
(5) $T := new\_calls(T, U, C)$

Until $T = \{\ \}$

Folding: Fold all instances of the terms in U

Fig. 1. Composition Algorithm

5.1 Composition

The composition strategy was originally introduced in [14,20] with the aim of avoiding the creation of unnecessary data structures. This strategy and its variants (e.g., Scherlis' internal specialization [43] or Wadler's deforestation [48]) have been mainly applied to functional programs, where *nested* function calls may appear. For instance, if a program contains a call of the form $f(g(t))$, then the output of $g(t)$ is consumed by function "f". In this situation, composition aims at replacing the call $f(g(t))$ with a new call $h(t)$ such that function "h" does not construct an intermediate data structure as "g" does.

There are many possible definitions for the composition strategy. The original formulations [14,20] are quite informal and mainly used to illustrate the application of the basic rules for program optimization. In the following, we present a simple composition algorithm for functional logic programs based on the transformation rules of Section 3.

Let us consider a program $\mathcal{R}$ and an operation-rooted term t which appears in the right-hand side of some rule of $\mathcal{R}$ and contains, at least, two nested function calls. The composition algorithm is shown in Figure 1. Let us explain how this algorithm works:

- First, we initialize sets T and U. During the main loop of the algorithm, set T contains the terms to be unfolded while U records the terms already

$$new\_calls(T, U, C) = \begin{cases} nested\_calls(T) & \text{if } C = \{\,\} \\ new\_calls(T \cup T', U, \{r_2, \ldots, r_n\}) & \text{if } C = \{\overline{r_n}\},\ n \geq 1,\ T' = nc(r_1, T \cup U) \end{cases}$$

$$nc(t, S) = \begin{cases} \{\,\} & \text{if } t \text{ is a variable} \\ \bigcup_{i \in \{1,\ldots,n\}} nc(t_i, S) & \text{if } t = c(\overline{t_n}) \\ \bigcup_{t' \in \mathcal{R}an(\theta)} nc(t', S) & \text{if there is a term } s \in S \text{ such that } t = \theta(s) \\ \{t'\} \cup \bigcup_{t'' \in T} nc(t'', S) & \text{otherwise, where } (t', T) = out(t, S) \end{cases}$$

$$out(t, S) = \begin{cases} (t, \{\,\}) & \text{if } t \text{ is a variable} \\ (c(\overline{t'_n}), T_1 \cup \ldots \cup T_n) & \text{if } t = c(\overline{t_n}) \text{ and } out(t_i, S) = (t'_i, T_i) \\ & \text{for all } i = 1, \ldots, n \\ (x, \{t' \mid t' \in \mathcal{R}an(\theta)\}) & \text{if } \exists s \in S.\ t = \theta(s) \text{ and } x \text{ is a fresh variable} \\ (f(\overline{t'_n}), T_1 \cup \ldots \cup T_n) & \text{otherwise, where } t = f(\overline{t_n}) \text{ and} \\ & out(t_i, S) = (t'_i, T_i) \text{ for all } i = 1, \ldots, n \end{cases}$$

Fig. 2. Auxiliary Function *new_calls*

unfolded in a previous iteration.

- Then, we enter into the main loop of the algorithm. Here, we select an arbitrary term $t \in T$ and introduce a new definition for t. Sets T and U are updated accordingly.
- Now, we unfold the new definition rule until all the right-hand sides of the unfolded rules[3] are in head normal form (i.e., a variable or a constructor-rooted term) or contain (at least) one subterm which is an instance of the selected term t. Note that it may happen—e.g., if we consider non-linear data structures—that different subterms in the rhs's of the rules have the same nested function symbols as t while they are not an instance of t, so that several folding steps might be necessary to obtain a recursive definition. This is not a problem in our composition algorithm since it proceeds iteratively, so that these additional subterms will be considered in the next iterations. If these subterms are eventually transformed so that they become an instance of t, then the final folding phase will also fold them using the new function associated to t.

  This represents a classical unfolding strategy which is guided by the *need for folding* (see [19], where it is called *forced folding*). This strategy enforces the creation of new, recursive definitions to evaluate terms containing nested function calls. Often, this has the effect of eliminating the intermediate data structures which were produced by the inner function and consumed by the outer one, since now we have a new, comprehensive definition (see the examples below).
- Then, we update the current set T with those operation-rooted subterms in

[3] Note that unfolding may generate more than one rule due to the non-deterministic nature of the transformation.

the right-hand sides of the unfolded rules which are not "covered" by some term in $T \cup U$. This process is managed by function *new_calls*, which is shown in Figure 2. It essentially proceeds as follows:

- The right-hand sides in C are inspected sequentially by using function *nc*. Each time we call $nc(r, T \cup U)$, we get the set of subterms of each r_i in C which are not covered by $T \cup U$. Then, these terms are added to the current set T.
- The notion of *coveredness*—which is similar to the notion of closedness in partial evaluation [1,5]—is formalized by function *nc* as follows: variables and constructor terms are ignored (they are always covered); function calls which are instances of some term in $T \cup U$ are discarded, but the inner subterms in the matching substitution are recursively inspected; finally, function calls which are not instances of any term in $T \cup U$ are processed by function "*out*". The auxiliary function *out* takes a term t and a set of terms S and returns a pair with the outermost part of t which is not "covered" by S, together with a set containing the remaining inner subterms (which are recursively inspected with *nc*). Let us illustrate its use with an example. Consider the initial call $nc(t, S)$, with $t = \mathtt{X} + \mathtt{f}(\mathtt{g}(\mathtt{h}(\mathtt{Y})))$ and $S = \{\mathtt{f}(\mathtt{g}(\mathtt{Z}))\}$. Since t is operation-rooted but it is not an instance of the term in S, we first compute $out(t, S)$, which demands the evaluation of $out(\mathtt{X}, S) = (\mathtt{X}, \{\ \})$ and $out(\mathtt{f}(\mathtt{g}(\mathtt{h}(\mathtt{Y}))), S) = (\mathtt{W}, \{\mathtt{h}(\mathtt{Z})\})$. Therefore,

  $$out(t, S) = (\mathtt{X} + \mathtt{W}, \{\mathtt{h}(\mathtt{Z})\})$$

  Finally, since $nc(\mathtt{h}(\mathtt{Z}), S) = \{\mathtt{h}(\mathtt{Z})\}$, we have:

  $$nc(t, S) = \{\mathtt{X} + \mathtt{W}, \mathtt{h}(\mathtt{Z})\}$$

- When *new_calls* is called with an empty set of right-hand sides, it simply returns the subset of terms in the current set T that contains nested function calls, denoted by $nested\_calls(T)$. For instance, in the above example, $nested\_calls(\{\mathtt{X} + \mathtt{W}, \mathtt{h}(\mathtt{Z})\}) = \{\ \}$.

- The previous steps are repeated until T contains no terms to be unfolded. Then, the final phase consists in folding all the terms in the current program which are instances of some term in U using the corresponding definition rules. Note that this process can be non-deterministic; consider, for instance, the term $\mathtt{f}(\mathtt{g}(\mathtt{h}(\mathtt{X})))$ which can be folded w.r.t. $U = \{\mathtt{f}(\mathtt{g}(\mathtt{X})), \mathtt{g}(\mathtt{h}(\mathtt{X}))\}$ in two different ways. To overcome this problem, several solutions have been proposed in other related frameworks (e.g., partial evaluation [1,5]) which can be adapted. We consider this problem outside the scope of this paper.

Let us illustrate our composition algorithm with an example. Consider the following program $\mathcal{R}$ which defines the function `dapp` to concatenate three

lists (by applying the standard function append twice):

$$\begin{array}{lcll} \texttt{dapp}(\texttt{X},\texttt{Y},\texttt{Z}) & \to & \texttt{append}(\texttt{append}(\texttt{X},\texttt{Y}),\texttt{Z}) & (R_1) \\ \texttt{append}([\,],\texttt{X}) & \to & \texttt{X} & (R_2) \\ \texttt{append}([\texttt{H}|\texttt{T}],\texttt{X}) & \to & [\texttt{H}|\texttt{append}(\texttt{T},\texttt{X})] & (R_3) \end{array}$$

This program is rather inefficient since dapp traverses list X twice and constructs an (unnecessary) intermediate list by the inner call append(X, Y). The application of the composition algorithm on $\mathcal{R}$ and append(append(X, Y), Z) gives rise to the following transformation sequence:

(1) First, we introduce the following definition: [4]

$$\texttt{aux}(\texttt{X},\texttt{Y},\texttt{Z}) \to \texttt{append}(\texttt{append}(\texttt{X},\texttt{Y}),\texttt{Z}) \qquad (R_4)$$

and update T and U as follows: $T := \{\ \}, U := \{\texttt{append}(\texttt{append}(\texttt{X},\texttt{Y}),\texttt{Z})\}$.

(2) The unfolding of rule R_4 produces the following rules:

$$\begin{array}{lcll} \texttt{aux}([\,],\texttt{Y},\texttt{Z}) & \to & \texttt{append}(\texttt{Y},\texttt{Z}) & (R_{41}) \\ \texttt{aux}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) & \to & \texttt{append}([\texttt{H}|\texttt{append}(\texttt{T},\texttt{Y})],\texttt{Z}) & (R_{42}) \end{array}$$

Then, both rules are unfolded again, thus producing the rules:

$$\begin{array}{lcll} \texttt{aux}([\,],[\,],\texttt{Z}) & \to & \texttt{Z} & (R_{411}) \\ \texttt{aux}([\,],[\texttt{H}|\texttt{T}],\texttt{Z}) & \to & [\texttt{H}|\texttt{append}(\texttt{T},\texttt{Z})] & (R_{412}) \\ \texttt{aux}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) & \to & [\texttt{H}|\texttt{append}(\texttt{append}(\texttt{T},\texttt{Y}),\texttt{Z})] & (R_{421}) \end{array}$$

and the unfolding process stops, since the right-hand sides of all the unfolded rules are in head normal form.

(3) Now, we update T by using function *new_calls* as follows:

$$\begin{array}{l} \mathit{new\_calls}(T, U, \{\texttt{Z}, [\texttt{H}|\texttt{append}(\texttt{T},\texttt{Z})], [\texttt{H}|\texttt{append}(\texttt{append}(\texttt{T},\texttt{Y}),\texttt{Z})]\}) \\ \quad = \ \mathit{new\_calls}(T, U, \{[\texttt{H}|\texttt{append}(\texttt{T},\texttt{Z})], [\texttt{H}|\texttt{append}(\texttt{append}(\texttt{T},\texttt{Y}),\texttt{Z})]\}) \\ \quad = \ \mathit{new\_calls}(T \cup \{\texttt{append}(\texttt{T},\texttt{Z})\}, U, \{[\texttt{H}|\texttt{append}(\texttt{append}(\texttt{T},\texttt{Y}),\texttt{Z})]\}) \\ \quad = \ \mathit{new\_calls}(T \cup \{\texttt{append}(\texttt{T},\texttt{Z})\}, U, \{\ \}) \\ \quad = \ \mathit{nested\_calls}(T \cup \{\texttt{append}(\texttt{T},\texttt{Z})\}) = \{\ \} \end{array}$$

Since T is empty, the main loop terminates.

(4) Finally, we fold the instances of append(append(X, Y), Z) in rules R_1 and

[4] For simplicity, we write aux rather than $\texttt{aux}_{\texttt{append(append(X,Y),Z)}}$.

R_{421} using rule R_4:

$$\begin{array}{lcll} \texttt{dapp}(\texttt{X},\texttt{Y},\texttt{Z}) & \to & \texttt{aux}(\texttt{X},\texttt{Y},\texttt{Z}) & (R_{1\texttt{f}}) \\ \texttt{aux}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) & \to & [\texttt{H}|\texttt{aux}(\texttt{T},\texttt{Y},\texttt{Z})] & (R_{421\texttt{f}}) \end{array}$$

and the algorithm terminates.

Observe that this transformation sequence terminates with the desired recursive definition for `aux` (rules R_{411}, R_{412} and $R_{421\texttt{f}}$). The transformed program is thus the following:

$$\begin{array}{lcll} \texttt{dapp}(\texttt{X},\texttt{Y},\texttt{Z}) & \to & \texttt{aux}(\texttt{X},\texttt{Y},\texttt{Z}) & (R_{1\texttt{f}}) \\ \texttt{append}([\,],\texttt{X}) & \to & \texttt{X} & (R_2) \\ \texttt{append}([\texttt{H}|\texttt{T}],\texttt{X}) & \to & [\texttt{H}|\texttt{append}(\texttt{T},\texttt{X})] & (R_3) \\ \texttt{aux}([\,],[\,],\texttt{Z}) & \to & \texttt{Z} & (R_{411}) \\ \texttt{aux}([\,],[\texttt{H}|\texttt{T}],\texttt{Z}) & \to & [\texttt{H}|\texttt{append}(\texttt{T},\texttt{Z})] & (R_{412}) \\ \texttt{aux}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) & \to & [\texttt{H}|\texttt{aux}(\texttt{T},\texttt{Y},\texttt{Z})] & (R_{421\texttt{f}}) \end{array}$$

Note that this program is an improvement (both in time and space) w.r.t. the original program $\mathcal{R}$.

It can be argued that most of the efficiency improvements achieved by the composition strategy can be simply obtained by lazy evaluation [22]. Nevertheless, the composition strategy often allows the derivation of programs with improved performance also in the context of lazy evaluation [47]. Laziness is decisive when, given a nested function call $\texttt{f}(\texttt{g}(t))$, the intermediate data structure produced by `g` is infinite but the function `f` can still produce its outcome by knowing only a finite portion of the output of `g`. The following example illustrates the advantages of our transformation rules w.r.t. those of [3] (where unfolding is based on eager narrowing and, thus, the pursued optimization is not possible). Consider the function $\texttt{sum\_prefix}(\texttt{X},\texttt{Y})$ defined in the following program $\mathcal{R}$:

$$\begin{array}{lcll} \texttt{sum\_prefix}(\texttt{X},\texttt{Y}) & \to & \texttt{suml}(\texttt{from}(\texttt{X}),\texttt{Y}) & (R_1) \\ \texttt{suml}(\texttt{L},\texttt{0}) & \to & \texttt{0} & (R_2) \\ \texttt{suml}([\texttt{H}|\texttt{T}],\texttt{s}(\texttt{X})) & \to & \texttt{H} + \texttt{suml}(\texttt{T},\texttt{X}) & (R_3) \\ \texttt{from}(\texttt{X}) & \to & [\texttt{X}|\texttt{from}(\texttt{s}(\texttt{X}))] & (R_4) \\ \texttt{0} + \texttt{X} & \to & \texttt{X} & (R_5) \\ \texttt{s}(\texttt{X}) + \texttt{Y} & \to & \texttt{s}(\texttt{X} + \texttt{Y}) & (R_6) \end{array}$$

where `sum_prefix` returns the sum of the `Y` consecutive natural numbers, starting from `X`. We can improve the efficiency of $\mathcal{R}$ by avoiding the creation

and subsequent use of the intermediate, partial list generated by the call to the function `from`. The application of the composition algorithm on $\mathcal{R}$ and $\mathtt{suml}(\mathtt{from}(\mathtt{X}), \mathtt{Y})$ gives rise to the following transformation sequence:

(1) First, we introduce the following definition:

$$\mathtt{aux}(\mathtt{X}, \mathtt{Y}) \;\to\; \mathtt{suml}(\mathtt{from}(\mathtt{X}), \mathtt{Y}) \qquad (R_7)$$

and update T and U as follows: $T = \{\ \}$, $U = \{\mathtt{suml}(\mathtt{from}(\mathtt{X}), \mathtt{Y})\}$.

(2) Then, we proceed with the unfolding of rule R_7:

$$\begin{array}{lcll} \mathtt{aux}(\mathtt{X}, \mathtt{0}) & \to & \mathtt{0} & (R_{71}) \\ \mathtt{aux}(\mathtt{X}, \mathtt{s}(\mathtt{Z})) & \to & \mathtt{suml}([\mathtt{X}|\mathtt{from}(\mathtt{s}(\mathtt{X}))], \mathtt{s}(\mathtt{Z})) & (R_{72}) \end{array}$$

It is worthwhile to note that the above step performs an additional instantiation, $\mathtt{Y} \mapsto \mathtt{s}(\mathtt{Z})$, which avoids the unfolding of rule R_{72} using rule R_2. This allows us to avoid the creation of useless rules in the transformed program and is a clear advantage of using needed narrowing rather than two independent instantiation and unfolding rules.

(3) By unfolding again rule R_{72} we get the new rule:

$$\mathtt{aux}(\mathtt{X}, \mathtt{s}(\mathtt{Y})) \;\to\; \mathtt{X} + \mathtt{suml}(\mathtt{from}(\mathtt{s}(\mathtt{X})), \mathtt{Y}) \qquad (R_{721})$$

and the termination criterion is fulfilled. Note that this unfolding step would be infeasible with an eager strategy.

(4) Now, we update T by using function *new_calls* as follows:

$$\begin{array}{l} \mathit{new\_calls}(T, U, \{\mathtt{0}, \mathtt{X} + \mathtt{suml}(\mathtt{from}(\mathtt{s}(\mathtt{X})), \mathtt{Y})\}) \\ \quad = \; \mathit{new\_calls}(T, U, \{\mathtt{X} + \mathtt{suml}(\mathtt{from}(\mathtt{s}(\mathtt{X})), \mathtt{Y})\}) \\ \quad = \; \mathit{new\_calls}(T \cup \{\mathtt{X} + \mathtt{Z}\}, U, \{\ \}) \\ \quad = \; \mathit{nested\_calls}(T \cup \{\mathtt{X} + \mathtt{Z}\}) = \{\ \} \end{array}$$

Since T is empty, the main loop terminates.

(5) Finally, we fold the occurrences of $\mathtt{suml}(\mathtt{from}(\ldots))$ in rules R_1 and R_{721} using rule R_7:

$$\begin{array}{lcll} \mathtt{sum\_prefix}(\mathtt{X}, \mathtt{Y}) & \to & \mathtt{aux}(\mathtt{X}, \mathtt{Y}) & (R_{1\mathtt{f}}) \\ \mathtt{aux}(\mathtt{X}, \mathtt{s}(\mathtt{Y})) & \to & \mathtt{X} + \mathtt{aux}(\mathtt{s}(\mathtt{X}), \mathtt{Y}) & (R_{721\mathtt{f}}) \end{array}$$

Therefore, the transformed program contains the following rules:

$$\begin{array}{lcll} \mathtt{sum\_prefix}(\mathtt{X}, \mathtt{Y}) & \to & \mathtt{aux}(\mathtt{X}, \mathtt{Y}) & (R_{1\mathtt{f}}) \\ \mathtt{aux}(\mathtt{X}, \mathtt{0}) & \to & \mathtt{0} & (R_{71}) \\ \mathtt{aux}(\mathtt{X}, \mathtt{s}(\mathtt{Y})) & \to & \mathtt{X} + \mathtt{aux}(\mathtt{s}(\mathtt{X}), \mathtt{Y}) & (R_{721\mathtt{f}}) \end{array}$$

together with the initial definitions for +, `from`, and `suml`.

Let us notice that the use of needed narrowing as a basis for our unfolding rule is essential in the above example. It ensures that no redundant rules are produced by unfolding and it also allows the transformation even in the presence of nonterminating functions (as opposed to [3]).

The main shortcoming of our simple composition algorithm is that the termination of the process is not generally ensured. In particular, we can identify two different termination problems:

- The unfolding process may run forever if we never reach a head normal form or a term containing some instance of the initial call.
- The main loop of the algorithm may be repeated infinitely if the set T is never empty.

In order to solve the first termination problem, the unfolding rule must incorporate some mechanism to stop the generation of new unfolded rules. For this purpose, there exist several well-known techniques in the literature, e.g., depth-bounds, loop-checks [11], well-founded orderings [13], well-quasi orderings [44]. In our composition algorithm, we consider the following strategy which has been successfully tested in the partial evaluation of functional logic programs [5]. First, we need the following auxiliary notion:

Definition 25 (unfolding ancestors) *Let $\mathcal{R}$ be a program and $R_0 \in \mathcal{R}$ be a rule. Let $R_1, \ldots, R_k$ be a sequence of rules such that R_{i+1} is obtained from R_i by performing an unfolding step w.r.t. the needed narrowing redex t_i, for $i = 0, \ldots, k-1$. Then, $t_0, \ldots, t_{k-1}$ are the* unfolding ancestors *of rule R_k.*

Now, we define our unfolding strategy based on a particular well-quasi ordering, the *homeomorphic embedding*. Informally, given two terms, t_1 and t_2, we say that t_2 *embeds* t_1, in symbols $t_1 \trianglelefteq t_2$ if t_1 can be obtained from t_2 by deleting some operators, e.g., $\mathtt{f(a,b)} \trianglelefteq \underline{\mathtt{f}}(\mathtt{g(X,}\underline{\mathtt{a}}\mathtt{)},\mathtt{h(Y,}\underline{\mathtt{b}}\mathtt{)})$ (the embedded symbols are underlined). A relevant property of this ordering is that, given a finite signature, any infinite sequence of terms of the form $t_1, t_2, t_3, \ldots$, must contain terms t_i and t_j such that $i < j$ and $t_i \trianglelefteq t_j$ (see [34] for a detailed description).

Definition 26 (Unfolding based on homeomorphic embedding) *Let $\mathcal{R}_0, \ldots, \mathcal{R}_k$ be a transformation sequence and $R \in \mathcal{R}_k$ be a rule. Then, rule R can be only unfolded iff the selected needed narrowing redex (according to Def. 5) does not embed any unfolding ancestor within the transformation sequence.*

The termination of unfolding with the above rule is an easy consequence of Kruskal's Tree Theorem (see, e.g., [34]).

$$
\begin{array}{l}
new\_calls^\sharp(T,U,C) \\
\quad = \begin{cases}
(nested\_calls(T),U) & \text{if } C = \{\,\} \\
new\_calls(T^\sharp,U^\sharp,\{r_2,\ldots,r_n\}) & \text{if } C = \{\overline{r_n}\},\ n \geq 1,\ T' = nc(r_1, T \cup U), \\
 & \text{and } (T^\sharp,U^\sharp) = abstract(T,U,T')
\end{cases} \\
\\
abstract(T,U,S) \\
\quad = \begin{cases}
(T,U) & \text{if } S = \{\,\} \\
abstract(T',U',\{t_2,\ldots,t_{n-1}\}) & \text{if } S = \{\overline{t_n}\}, (T',U') = abs(T,U,t_n), n \geq 1
\end{cases} \\
\\
abs(T,U,t) \\
\quad = \begin{cases}
(T \cup \{t\},U) & \text{if } \nexists t' \in T \cup U.\ \mathcal{H}ead(t) = \mathcal{H}ead(t') \\
(T \cup \{t\},U) & \text{if } \nexists t' \in T \cup U.\ \mathcal{H}ead(t) = \mathcal{H}ead(t') \wedge t' \trianglelefteq t \\
(T \setminus \{t'\} \cup \{msg(t,t')\},U) & \text{if } \exists t' \in T.\ \mathcal{H}ead(t) = \mathcal{H}ead(t') \wedge t' \trianglelefteq t \\
(T \cup \{msg(t,t')\},U \setminus \{t'\}) & \text{if } \exists t' \in U.\ \mathcal{H}ead(t) = \mathcal{H}ead(t') \wedge t' \trianglelefteq t
\end{cases}
\end{array}
$$

Fig. 3. Auxiliary Function $new\_calls^\sharp$

Regarding the second termination problem, it can also be solved with the help of the homeomorphic embedding ordering. This technique is not new, but an adaptation of notions widely used in the literature (see, e.g., [21,35,44]). To be precise, we modify the fifth step of the composition algorithm in Figure 1 in order to forbid the addition of new terms if they embed some previous term in the set. Moreover, we use the operator msg (*most specific generalization*) to generalize the problematic terms. The msg of two terms is defined as follows. A term t is a *generalization* of the terms t_1 and t_2 if both t_1 and t_2 are instances of t. Furthermore, the term t is the msg of t_1 and t_2, in symbols $msg(t_1, t_2)$, if t is a generalization of t_1 and t_2 and, for any other generalization t' of t_1 and t_2, t is an instance of t'. Step (5) of the composition algorithm is then refined as follows:

(5) $(T,U) := new\_calls^\sharp(T,U,C)$

The definition of the new function $new\_calls^\sharp$ is shown in Figure 3. Basically, it mimics the original function new_calls but additionally uses function *abstract* to control the addition of new terms to the set T. In particular, given sets T and U, in order to add a new term t function *abs* proceeds as follows: if there are no termination problems (the first two cases), the new term is added to T; if it embeds some term either in T or U (the last two cases), then both terms are generalized and the new term is added instead. Function *abs* guarantees that the composition algorithm cannot enter into an infinite loop:

- First, the set $T \cup U$ cannot contain an infinite set of terms with the same outermost function symbol. This is an immediate consequence of the use of an embedding ordering together with the fact that a term can only be

generalized a finite number of times.
- On the other hand, the number of different function symbols is finite (i.e., the functions of the original program), since the right-hand sides cannot contain new function symbols. Indeed, this is the reason to perform folding *after* the main loop terminates, since folding would introduce new function symbols in the right-hand sides of the folded rules, and this would imply a (potentially) infinite signature.

The previous facts guarantee that the number of iterations is kept finite. A similar, though more complex, proof of termination can be found in [5].

Let us illustrate the refined algorithm with an example. Consider the following program $\mathcal{R}$ which defines the function `last` to compute the last element of a list in terms of `reverse` with an accumulating parameter:

$$\begin{array}{lcll}
\texttt{last(X)} & \to & \texttt{first(reverse(X))} & (R_1)\\
\texttt{first([H|T])} & \to & \texttt{H} & (R_2)\\
\texttt{reverse(X)} & \to & \texttt{rev(X,[\,])} & (R_3)\\
\texttt{rev([\,],Acc)} & \to & \texttt{Acc} & (R_4)\\
\texttt{rev([H|T],Acc)} & \to & \texttt{rev(T,[H|Acc])} & (R_5)
\end{array}$$

Now, the application of the refined composition algorithm on program $\mathcal{R}$ and term `first(reverse(X))` gives rise to the following transformation sequence:

(1) First, we introduce the following definition:

$$\texttt{aux}_1\texttt{(X)} \to \texttt{first(reverse(X))} \qquad (R_6)$$

and update T and U as follows: $T := \{\ \}$, $U := \{\texttt{first(reverse(X))}\}$.

(2) Then, unfolding of rule R_6 produces the following rule:

$$\texttt{aux}_1\texttt{(X)} \to \texttt{first(rev(X,[\,]))} \qquad (R_{61})$$

Since the termination criteria do not hold, we unfold R_{61} again:

$$\begin{array}{lcll}
\texttt{aux}_1\texttt{([\,])} & \to & \texttt{first([\,])} & (R_{611})\\
\texttt{aux}_1\texttt{([H|T])} & \to & \texttt{first(rev(T,[H]))} & (R_{612})
\end{array}$$

Rule R_{611} cannot be unfolded again since the right-hand side does not match the definition of `first` (indeed, it will lead to a failing derivation, similarly to the original program). On the other hand, rule R_{612} cannot be unfolded since the selected narrowing redex, `rev(T,[H])`, embeds the unfolding ancestor `rev(X,[\,])`.

(3) Now, we update T and U by using function $new\_calls^\sharp$ as follows:

$$\begin{aligned} &new\_calls^\sharp(T, U, \{\mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))\}) \\ &\quad = \ new\_calls(T^\sharp, U^\sharp, \{\ \}) \ = \ (nested\_calls(T^\sharp), U^\sharp) \end{aligned}$$

where

$$nc(\mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}])), T, U) = \{\mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))\}$$

since $\mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))$ is not an instance of $\mathtt{first}(\mathtt{reverse}(\mathtt{X}))$, and

$$\begin{aligned} &abstract(T, U, \{\mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))\}) \\ &\quad = \ abs(T, U, \mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))) \ = \ (\{\mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))\}, U) \end{aligned}$$

Therefore, $T = nested\_calls(T^\sharp) = \{\mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))\}$ and U is not changed. Since T is not empty, we start a new iteration of the main loop.

(4) In the second iteration of the algorithm, we introduce a new definition:

$$\mathtt{aux_2}(\mathtt{T}, \mathtt{H}) \ \rightarrow \ \mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}])) \qquad (R_7)$$

with $T = \{\ \}$ and $U = \{\mathtt{first}(\mathtt{reverse}(\mathtt{X})), \mathtt{first}(\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))\}$.

(5) The unfolding of the new rule produces the following rules:

$$\begin{aligned} &\mathtt{aux_2}([\,], \mathtt{H}) &&\rightarrow \ \mathtt{first}([\mathtt{H}]) &&(R_{71}) \\ &\mathtt{aux_2}([\mathtt{H}'|\mathtt{T}'], \mathtt{H}) &&\rightarrow \ \mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}])) &&(R_{72}) \end{aligned}$$

While rule R_{72} cannot be further unfolded—the redex $\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}])$ embeds the unfolding ancestor $\mathtt{rev}(\mathtt{T}, [\mathtt{H}]))$—, rule R_{71} is unfolded one more time:

$$\mathtt{aux_2}([\,], \mathtt{H}) \ \rightarrow \ \mathtt{H} \qquad (R_{711})$$

and the unfolding phase terminates.

(6) Now, we update T and U as follows:

$$\begin{aligned} &new\_calls^\sharp(T, U, \{\mathtt{H}, \mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}]))\}) \\ &\quad = \ new\_calls(T^\sharp, U^\sharp, \{\ \}) \ = \ (nested\_calls(T^\sharp), U^\sharp) \end{aligned}$$

where

$$nc(\mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}]))))), T, U) = \{\mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}]))\}$$

since $\mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}]))$ is not an instance of any term in $T \cup U$, and

$$\begin{aligned} &abstract(T, U, \{\mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}]))\}) \\ &\quad = \ abs(T, U, \mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}]))) \\ &\quad = \ (\{\mathtt{first}(\mathtt{rev}(\mathtt{T}, \mathtt{W}))\}, \{\mathtt{first}(\mathtt{reverse}(\mathtt{X}))\}) \end{aligned}$$

Therefore, we have $T = nested\_calls(T^\sharp) = \{\mathtt{first}(\mathtt{rev}(\mathtt{T}, \mathtt{W}))\}$, where $\mathtt{first}(\mathtt{rev}(\mathtt{T}, \mathtt{W}))$ is the msg of the new term $\mathtt{first}(\mathtt{rev}(\mathtt{T}', [\mathtt{H}', \mathtt{H}]))$ and

the previous term $\mathtt{first}(\mathtt{rev}(\mathtt{T},[\mathtt{H}]))$ in U. Also, the problematic term has been removed from U and, hence, $U = \{\mathtt{first}(\mathtt{reverse}(\mathtt{X}))\}$. Since T is not empty, we start a new iteration of the main loop.

(7) In the third iteration of the algorithm, we introduce a new definition:

$$\mathtt{aux}_3(\mathtt{T},\mathtt{W}) \;\to\; \mathtt{first}(\mathtt{rev}(\mathtt{T},\mathtt{W})) \qquad (R_8)$$

with $T = \{\ \}$ and $U = \{\mathtt{first}(\mathtt{reverse}(\mathtt{X})), \mathtt{first}(\mathtt{rev}(\mathtt{T},\mathtt{W}))\}$.

(8) The unfolding of the new rule produces the following rules:

$$\begin{array}{lcll} \mathtt{aux}_3([\,],\mathtt{W}) & \to & \mathtt{first}(\mathtt{rev}([\,],\mathtt{W})) & (R_{81}) \\ \mathtt{aux}_3([\mathtt{H}'|\mathtt{T}'],\mathtt{W}) & \to & \mathtt{first}(\mathtt{rev}(\mathtt{T}',[\mathtt{H}|\mathtt{W}])) & (R_{82}) \end{array}$$

While rule R_{82} cannot be further unfolded—the redex $\mathtt{first}(\mathtt{rev}(\mathtt{T}',[\mathtt{H}|\mathtt{W}]))$ embeds the unfolding ancestor $\mathtt{first}(\mathtt{rev}(\mathtt{T},\mathtt{W}))$—, rule R_{81} is unfolded twice:

$$\mathtt{aux}_3([\,],[\mathtt{W}|\mathtt{WS}]) \;\to\; \mathtt{W} \qquad (R_{811})$$

and the unfolding phase terminates.

(9) Now, we update T and U as follows:

$$\begin{array}{l} \mathit{new\_calls}^\sharp(T,U,\{\mathtt{W},\mathtt{first}(\mathtt{rev}(\mathtt{T}',[\mathtt{H}|\mathtt{W}]))\}) \\ \quad = \; \mathit{new\_calls}(T^\sharp,U^\sharp,\{\ \}) \;=\; (\mathit{nested\_calls}(T^\sharp),U^\sharp) \end{array}$$

where

$$\mathit{nc}(\mathtt{first}(\mathtt{rev}(\mathtt{T}',[\mathtt{H}|\mathtt{W}])),T,U) = \{\ \}$$

since $\mathtt{first}(\mathtt{rev}(\mathtt{T}',[\mathtt{H}|\mathtt{W}]))$ is an instance of $\mathtt{first}(\mathtt{rev}(\mathtt{T},\mathtt{W}))$ in U. Therefore, T and U are not changed and the main loop terminates.

(10) Finally, we fold the instances $U = \{\mathtt{first}(\mathtt{reverse}(\mathtt{X})), \mathtt{first}(\mathtt{rev}(\mathtt{T},\mathtt{W}))\}$ in rules R_1, R_{612}, and R_{82} using rules R_6 and R_8:[5]

$$\begin{array}{lcll} \mathtt{last}(\mathtt{X}) & \to & \mathtt{aux}_1(\mathtt{X}) & (R_{1\mathtt{f}}) \\ \mathtt{aux}_1([\mathtt{H}|\mathtt{T}]) & \to & \mathtt{aux}_3(\mathtt{T},[\mathtt{H}]) & (R_{612\mathtt{f}}) \\ \mathtt{aux}_3([\mathtt{H}'|\mathtt{T}'],\mathtt{W}) & \to & \mathtt{aux}_3(\mathtt{T}',[\mathtt{H}|\mathtt{W}]) & (R_{82\mathtt{f}}) \end{array}$$

and the algorithm terminates.

[5] We ignore the rules for $\mathtt{aux}_2$ since they are not needed anymore due to the generalization process.

The final program of the transformation is as follows:

$$
\begin{array}{lcll}
\texttt{last(X)} & \rightarrow & \texttt{aux}_1\texttt{(X)} & (R_{1\texttt{f}}) \\
\texttt{aux}_1\texttt{([\,])} & \rightarrow & \texttt{first([\,])} & (R_{611}) \\
\texttt{aux}_1\texttt{([H|T])} & \rightarrow & \texttt{aux}_3\texttt{(T,[H])} & (R_{612\texttt{f}}) \\
\texttt{aux}_3\texttt{([\,],[W|WS])} & \rightarrow & \texttt{W} & (R_{811}) \\
\texttt{aux}_3\texttt{([H'|T'],W)} & \rightarrow & \texttt{aux}_3\texttt{(T',[H|W])} & (R_{82\texttt{f}})
\end{array}
$$

together with the definitions of `first`, `reverse`, `rev`, and $\texttt{aux}_2$. Let us note that this is a difficult example where several transformation strategies fail to terminate due to the use of an accumulating parameter in function `rev`.

Let us conclude this section by discussing some related work. The composition algorithm presented in this section shares many similarities with partial evaluation algorithms for functional logic programs (e.g., [5,6]). In fact, the narrowing-driven approach to partial evaluation is able to achieve deforestation, similarly to positive supercompilation [45] of functional programs or conjunctive partial deduction [21] of logic programs. Our composition algorithm is as powerful as the narrowing-driven partial evaluator of [6] regarding the composition of nested function calls. Indeed, the main difference between these techniques lies in the fact that in the composition algorithm we only consider the generation of new functions for those terms that contain nested function calls. Apart from this difference, both techniques perform the same basic steps: a main loop consisting in introducing new, residual definitions and unfolding these definitions, followed by a final folding phase. Moreover, both techniques employ the same control strategies to ensure the termination of the process: the embedding ordering and the operator msg. This means that our composition algorithm is also comparable to positive supercompilation or conjunctive partial deduction regarding the ability to perform deforestation.

5.2 Tupling

While other transformation techniques (like partial evaluation or composition) have been widely investigated, tupling is a less known—but equally powerful—transformation strategy based on the fold/unfold methodology. The tupling strategy was originally introduced in [14,20] to optimize functional programs. Essentially, it proceeds by grouping calls with common arguments together so that their results are computed only once. When the strategy succeeds, multiple traversals of data structures can be avoided, thus transforming a nonlinear recursive program into a linear recursive one [41]. Unfortunately, the tupling strategy is more involved than the composition strategy. It often requires complex analyses in order to extract appropriate tuples of calls to

be grouped together [15], to guarantee termination on non-trivial classes of programs (e.g., the synchronization analysis in [17]). The development of a fully automatic tupling algorithm is out of the scope of this paper. In the following, we develop the backbone of the tupling strategy in our setting, and illustrate its use by means of an example.

The input to the tupling strategy is a program containing a rule, $l \to r$, whose right-hand side has the form $r = r[\overline{e_k}]_{\overline{P_k}}$, where $r|_p = e_i$ for all $p \in P_i$, $i = 1, \ldots, k$, and $e_1, \ldots, e_k$ are operation-rooted terms which share, at least, one variable x. Then, the strategy proceeds in four transformation phases:

Eureka generation. First, we introduce a new rule of the form:

$$aux(x, y_1, \ldots, y_n) \ \to \ \langle e_1, \ldots, e_k \rangle$$

where $\mathcal{V}ar(e_1, \ldots, e_k) = \{x, y_1, \ldots, y_n\}$ and $x \in \mathcal{V}ar(e_i)$ for all $i = 1, \ldots, k$.

Unfolding. Here, similarly to the composition strategy, we apply unfolding steps until we reach a right-hand side containing instances of $e_1, \ldots, e_k$ (if possible) or stop the unfolding process if there is a risk of nontermination. If the process succeeds, we end up with a rule of the form:

$$aux(t_0, t_1, \ldots, t_n) \ \to \ \theta(C'[\overline{e_k}]_{\overline{Q_k}})$$

for some substitution θ and context C'.

Abstraction. By using abstraction, the above rule is transformed into

$$aux(t_0, t_1, \ldots, t_n) \ \to \ \theta(C'[\overline{z_k}]_{\overline{Q_k}} \text{ where } \langle z_1, \ldots, z_k \rangle = \langle e_1, \ldots, e_k \rangle)$$

where $z_1, \ldots, z_k$ are fresh variables. A similar transformation is applied to the original rule $l \to r$:

$$l \ \to \ r[\overline{z_k}]_{\overline{P_k}} \text{ where } \langle z_1, \ldots, z_k \rangle = \langle e_1, \ldots, e_k \rangle$$

Folding. Finally, we apply folding steps in order to transform the above rules as follows:

$$\begin{aligned} l &\to r[\overline{z_k}]_{\overline{P_k}} \ \text{ where } \langle z_1, \ldots, z_k \rangle = aux(x, y_1, \ldots, y_n) \\ aux(t_0, t_1, \ldots, t_n) &\to \theta(C'[\overline{z_k}]_{\overline{Q_k}} \ \text{ where } \langle z_1, \ldots, z_k \rangle = aux(x, y_1, \ldots, y_n)) \end{aligned}$$

The following example illustrates the tupling strategy. Consider the function `fib` to compute the Fibonacci numbers:

$$\begin{array}{lcll} \texttt{fib(0)} & \to & \texttt{s(0)} & (R_1) \\ \texttt{fib(s(0))} & \to & \texttt{s(0)} & (R_2) \\ \texttt{fib(s(s(X)))} & \to & \texttt{fib(s(X))} + \texttt{fib(X)} & (R_3) \end{array}$$

This is a typical function which has an exponential complexity that can be reduced to a linear one by applying the tupling strategy. The process starts by introducing the following definition:

$$\texttt{aux(X)} \;\rightarrow\; \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle \qquad (R_4)$$

Now, by unfolding the redex $\texttt{fib(s(X))}$ of rule R_4 we get:

$$\begin{array}{lll} \texttt{aux(0)} & \rightarrow & \langle \texttt{s(0)}, \texttt{fib(s(0))} \rangle \qquad (R_{41}) \\ \texttt{aux(s(X))} & \rightarrow & \langle \texttt{fib(s(X))} + \texttt{fib(X)}, \texttt{fib(s(X))} \rangle \qquad (R_{42}) \end{array}$$

We unfold once again rule R_{41} in order to remove the call to `fib`:

$$\texttt{aux(0)} \;\rightarrow\; \langle \texttt{s(0)}, \texttt{s(0)} \rangle \qquad (R_{411})$$

Then, abstraction is applied to rules R_3 and R_{42} as follows:

$$\begin{array}{lll} \texttt{fib(s(s(X)))} & \rightarrow & \texttt{Z}_1 + \texttt{Z}_2 \text{ where } \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle \qquad (R_{3a}) \\ \texttt{aux(s(X))} & \rightarrow & \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle \text{ where } \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle \qquad (R_{42a}) \end{array}$$

Finally, the right-hand sides of both rules are folded using the original definition of function `aux`:

$$\begin{array}{lll} \texttt{fib(s(s(X)))} & \rightarrow & \texttt{Z}_1 + \texttt{Z}_2 \text{ where } \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{aux(X)} \qquad (R_{3af}) \\ \texttt{aux(s(X))} & \rightarrow & \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle \text{ where } \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{aux(X)} \qquad (R_{42af}) \end{array}$$

Therefore, the transformed (linear) definition of function `fib` is as follows:

$$\begin{array}{lll} \texttt{fib(0)} & \rightarrow & \texttt{s(0)} \qquad (R_1) \\ \texttt{fib(s(0))} & \rightarrow & \texttt{s(0)} \qquad (R_2) \\ \texttt{fib(s(s(X)))} & \rightarrow & \texttt{Z}_1 + \texttt{Z}_2 \text{ where } \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{aux(X)} \qquad (R_{3af}) \\ \texttt{aux(0)} & \rightarrow & \langle \texttt{s(0)}, \texttt{s(0)} \rangle \qquad (R_{411}) \\ \texttt{aux(s(X))} & \rightarrow & \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle \text{ where } \langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{aux(X)} \qquad (R_{42af}) \end{array}$$

As we mentioned before, the difficult part of the tupling algorithm is the extraction of appropriate tuples of program calls. For this purpose, we could adapt existing tupling analyses for functional programs like those presented in [15,17].

This is an interesting topic for further research. Some preliminary results can be found in [38]. In this proposal, all nested function calls are supposed to be first (successfully) optimized by composition so that the considered program only contains non-nested function calls. This restriction greatly simplifies both the eureka generation phase and the unfolding process (in particular, it helps to stop the unfolding of function calls).

6 The Transformation System SYNTH

The basic rules presented so far have been implemented by a prototype system SYNTH [2], which is publicly available at

`http://www.dsic.upv.es/users/elp/soft.html`

It includes a parser for the language Curry, a modern multi-paradigm declarative language based on needed narrowing which extends Haskell with features from logic and concurrent programming [26,27]. It also includes a fully automatic composition strategy based on some (seemingly reasonable) heuristics.

The SYNTH system is written in SICStus Prolog v3.6 and the complete implementation consists of about 680 clauses (2450 lines of code). The transformer is expressed by 330 clauses (including the user interface and the code needed to handle the *ground representation*), the parser and other utilities by 190 clauses, needed narrowing is implemented by 90 clauses and definitional trees are constructed by means of 70 clauses. The implementation only considers unconditional programs. Conditional programs are treated by using the predefined functions `and`, `if_then_else`, and `case_of`, which are reduced by standard defining rules (see, e.g., [39]).

Language syntax follows mainly that of the language Curry. Programs consist of a set of datatype and function declarations. For instance, the datatype declaration

```
data treeInt = Leaf Int
             | Tree treeInt Int treeInt
```

introduces the datatype `treeInt` of binary trees whose nodes are labelled with integers.

A function is defined by a list of defining equations. For instance, the function declaration

```
append [] y = y
```

```
append (x:xs) y = x : append xs y
```

defines the well-known concatenation of lists.

The main differences w.r.t. the standard Curry syntax (as defined in [27]) are the following:

- conditional equations are not allowed;
- variables begin with an uppercase and (constructor and defined) functions begin with a lowercase (as in Prolog);
- type declarations for defined functions are not allowed (SYNTH does not include type checking);
- the `let` construction is not allowed, although local declarations can be given using the `where` construction.

We have recently incorporated a graphical interface (written in Java) to the system that enhances the interaction with the user. The SYNTH main screen is divided into four areas, as shown in Figure 4:

(1) In the upper row of the screen, a *toolbar* presents the following options:
   (a) `File`: contains the typical options for opening, loading and saving a file, as well as cleaning and exiting the system.
   (b) `Edit`: apart from the facilities for cutting, copying, pasting, finding and replacing, it is possible to syntactically analyze a program and to show/edit any program in a given transformation sequence; also, the system is able to show the definitional trees for each defined function symbol in the program.
   (c) `Rules`: allows the user to apply any of the transformation rules presented in Section 3.
   (d) `Strategies`: starts the application of the composition—in both an automatic or semiautomatic way—and tupling strategies.
   (e) `Options`: for configuring visual features of the system, like colors, fonts, etc.
   (f) `Help`: contains additional information about the system.

(2) The *edit window* appears in the central area of the screen. In this window, it is possible to simultaneously edit different programs belonging to a given transformation sequence, as shown in Figure 4.

(3) The *messages window* appears below the previous one, and it is used by the system to show information about the operation that is being performed (parsing, folding, etc).

(4) Finally, the right column of the screen is used for dynamically listing the *set of allowed options* during a transformation process. This facility displays the sequence of applicable icons that can also be selected (in a less direct way) through the toolbar. For instance, the icon associated to the unfolding rule only appears when a rule has been marked inside a

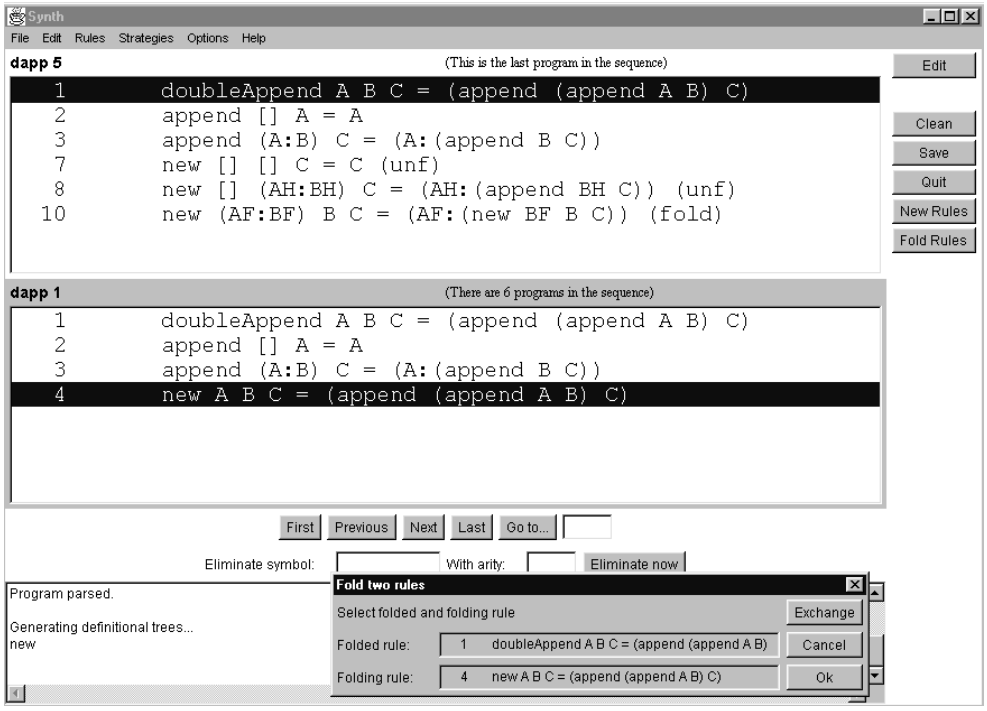


Fig. 4. Folding transformation in the SYNTH environment

loaded program.

The system automatically assigns a number (starting from 0) to each program generated in a transformation sequence. Program rules are also numbered and labelled with the name of the transformation rule that produced them. Hence, it is easy to identify the sequence of transformation steps performed along a transformation process as well as to visualize the sequence of programs by simply clicking one of the options (`First`, `Previous`, `Next`, `Last` or `Go to`) which are shown between the edit window and the messages window. In Figure 4, a folding operation has just been performed using two rules belonging to different programs (`dapp 5` and `dapp 1`, respectively). In this example, an additional window is displayed (right-down corner) which shows the folded and folding program rules involved in the transformation.

As for transformation heuristics, SYNTH includes a fully automatic composition strategy based on some (apparently effective) heuristics. We plan to extend it in order to also deal with tupling automatically (by using, e.g., the analysis method of [15]).

The transformation system allows us to choose between two different heuristics to achieve composition. The first strategy (option `comp`) is semi-automatic: the user is asked to select the rule where the considered nested call appears, and has to provide also an initial definition rule for the transformation to start. The second strategy (option `acomp`) is completely automatic: first a nested call within the program is automatically chosen and, then, an appropriate defini-

Table 1
Benchmark results

Programs	Rw_1	RT_1	Rw_2	RT_2	RT_1/RT_2	Speedups
`doubleappend`	3	1320	6	1190	1.109	9.8%
`sumprefix`	4	1850	6	1030	1.796	44.3%
`lengthapp`	5	1090	7	980	1.112	10.1%
`doubleflip`	3	1520	5	1400	1.086	7.9%
`fibonacci`	3	1410	5	400	3.525	71.6%
`factlist`	4	1480	6	245	6.040	83.4%
`average`	5	1430	7	1080	1.324	24.5%

tion rule for a new function is introduced. After these initial actions, the system proceeds automatically in both strategies. SYNTH builds an incomplete, finite search tree, while looking for a node which can deliver a recursive definition of the new function by a final folding step. Finiteness is ensured either by imposing depth-bounds to the tree or by means of the (homeomorphic) *embedding* test 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). Once the required node is found, the original rule is replaced by the transformed one; otherwise, the set of unfoldings which correspond to the non-failed branches of the tree is returned.

Table 1 summarizes our benchmark results. The first two columns measure the number of rewrite rules (Rw_1) and the absolute runtimes (RT_1) for the original programs. The other columns show the number of rewrite rules (Rw_2)—which includes the rules of the original program since we apply no definition elimination steps—, the absolute runtimes (RT_2), and the speedups achieved for the transformed programs (we include both the speedup factor RT_1/RT_2 and the percentage $((RT_1 - RT_2)/RT_1) \times 100$). All the programs have been executed by using the Curry environment PAKCS [25], which compiles Curry programs to Prolog. Times are expressed in milliseconds and are the average of 10 executions (for sufficiently large input goals).

Our (automatic) composition strategy performs well w.r.t. the first four benchmarks, which are typical functional programs to illustrate deforestation [48]: `dapp` and `sumprefix` are described in Section 5, while the source code of `lengthapp` and `doubleflip` are listed in Table 2. In these examples, composition is able to perform an effective optimization (with speedups ranging between 7.9% and 44.3%). Regarding the last three benchmarks, `fibonacci` has already been described in Section 5 while `factlist` and `average` are shown in Table 2. In all these cases a tupling strategy was necessary to succeed, as

Table 2
Original and transformed source programs

lengthapp	after composition
$\mathtt{lengthapp(L1,L2)} \rightarrow \mathtt{len(app(L1,L2))}$ $\mathtt{len(nil)} \rightarrow \mathtt{0}$ $\mathtt{len(H:T)} \rightarrow \mathtt{s(len(T))}$ $\mathtt{app(nil,L)} \rightarrow \mathtt{L}$ $\mathtt{app(H:T,L)} \rightarrow \mathtt{H:app(T,L)}$	$\mathtt{lengthapp(L1,L2)} \rightarrow \mathtt{new(L1,L2)}$ $\mathtt{new(nil,L)} \rightarrow \mathtt{len(L)}$ $\mathtt{new(H:T,L)} \rightarrow \mathtt{s(new(T,L))}$

doubleflip	after composition
$\mathtt{doubleflip(T)} \rightarrow \mathtt{f(f(T))}$ $\mathtt{f(leaf(N))} \rightarrow \mathtt{leaf(N)}$ $\mathtt{f(tree(L,N,R)} \rightarrow \mathtt{tree(f(R),N,f(L))}$	$\mathtt{doubleflip(T)} \rightarrow \mathtt{n(T)}$ $\mathtt{n(leaf(N))} \rightarrow \mathtt{leaf(N)}$ $\mathtt{n(tree(L,N,R)} \rightarrow \mathtt{tree(n(L),N,n(R))}$

factlist	after tupling
$\mathtt{flist(0)} \rightarrow \mathtt{nil}$ $\mathtt{flist(s(N))} \rightarrow \mathtt{fact(s(N)):flist(N)}$ $\mathtt{fact(0)} \rightarrow \mathtt{1}$ $\mathtt{fact(s(N))} \rightarrow \mathtt{s(N)} * \mathtt{fact(N)}$	$\mathtt{flist(0)} \rightarrow \mathtt{nil}$ $\mathtt{flist(s(N))} \rightarrow \mathtt{U:V\ where}$ $\langle \mathtt{U,V} \rangle = \mathtt{n(N)}$ $\mathtt{n(0)} \rightarrow \langle \mathtt{s(0),nil} \rangle$ $\mathtt{n(s(N))} \rightarrow \langle \mathtt{s(s(N))} * \mathtt{U,U:V} \rangle \ \mathtt{where}$ $\langle \mathtt{U,V} \rangle = \mathtt{n(N)}$

average	after tupling
$\mathtt{average(L)} \rightarrow \mathtt{sum(L)/length(L)}$ $\mathtt{sum(nil)} \rightarrow \mathtt{0}$ $\mathtt{sum(H:T)} \rightarrow \mathtt{H} + \mathtt{sum(T)}$ $\mathtt{length(nil)} \rightarrow \mathtt{0}$ $\mathtt{length(H:T)} \rightarrow \mathtt{s(length(T))}$	$\mathtt{average(L)} \rightarrow \mathtt{U/V\ where}$ $\langle \mathtt{U,V} \rangle = \mathtt{new(L)}$ $\mathtt{new(nil)} \rightarrow \langle \mathtt{0,0} \rangle$ $\mathtt{new(H:T)} \rightarrow \langle \mathtt{H} + \mathtt{U,s(V)} \rangle \ \mathtt{where}$ $\langle \mathtt{U,V} \rangle = \mathtt{new(T)}$

expected. As Table 1 shows, the tupling strategy obtains better efficiency improvements w.r.t. the composition strategy (with speedups ranging between 24.5% and 83.4%) although, in this case, the transformation sequence must be guided by the user.

In general, transformed programs cannot be guaranteed to be faster than the original ones, since there is a trade-off between the smaller amount of computation needed after the transformation (when guided by appropriate strategies) and the larger number of derived rules. Nevertheless, our experiments seem to substantiate that the smaller computations make up for the overhead of checking the applicability of the larger number of rules.

7 Conclusions

The definition of a fold/unfold framework for the optimization of functional logic programs was an open problem marked in [41] as pending research. We have presented a transformation methodology for lazy functional logic programs preserving the semantics of both values and answers computed by an efficient (currently the best) operational mechanism. For proving correctness, we extensively exploit the existing results from Huet and Levy's theory of needed reductions [28] and the wide literature about completeness of needed narrowing [9] (rather than striving an ad-hoc proof). We have shown that the transformation process keeps the inductively sequential structure of programs. We have also shown that the transformation process can be guided by appropriate strategies which lead to effective improvements. The experiments show that our transformation framework combines in a useful and effective way the systematic instantiation of calls during unfolding (by virtue of the logic component of the needed narrowing mechanism) with the power of the abstraction transformations (thanks to the functional dimension). We have also presented an implementation which allows us to perform automatically the composition strategy as well as to perform all basic transformations in a semi-automated way. The results of this work can be applied to optimize a large class of kernel (i.e., non-concurrent) Curry programs.

Acknowledgements

We thank Ivan Ziliotto and César Ferri for having implemented the transformer SYNTH and for the evaluation of the benchmarks. We thank the anonymous referees for their accurate and useful remarks and suggestions, which helped to improve our paper.

References

[1] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1):1–34, 2002.

[2] M. Alpuente, M. Falaschi, C. Ferri, G. Moreno, G. Vidal, and I. Ziliotto. The Transformation System SYNTH. Technical Report DSIC-II/16/99, UPV, 1999. Accessible from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[3] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.

[4] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, pages 147–162. Springer LNCS 1722, 1999.

[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. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In *Proceedings of the Fourth ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 273–283, N.Y., 1999. ACM Press.

[7] 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.

[8] 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, Cambridge, Mass., 1997.

[9] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Journal of the ACM*, volume 47(4), pages 776–822, 2000.

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

[11] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[12] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

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

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

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

[16] W. Chin, A. Goh, and S. Khoo. Effective Optimisation of Multiple Traversals in Lazy Languages. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA (Technical Report BRICS-NS-99-1)*, pages 119–130. University of Aarhus, DK, 1999.

[17] W.-N. Chin, S.-C. Khoo, and T.-W. Lee. Synchronisation Analysis to Stop Tupling. In *Proc. of the 7th European Symposium on Programming (ESOP'98)*, pages 75–89. Springer LNCS 1381, 1998.

[18] B. Courcelle. Equivalences and transformations of regular systems – applications to recursive program schemes and grammars. *Theoretical Computer Science*, 42:1–122, 1986.

[19] J. Darlington. An Experimental Program Transformation and Synthesis System. *Artificial Intelligence*, 16(1):1–46, 1981.

[20] 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.

[21] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.

[22] M. S. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. In *IFIP 87*, pages 165–195, 1987.

[23] P. A. Gardner and J. C. Shepherdson. Unfold/fold Transformation of Logic Programs. In J.L Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. The MIT Press, Cambridge, MA, 1991.

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

[25] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.5: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2003.

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

[27] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.

[28] 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. The MIT Press, Cambridge, MA, 1992.

[29] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture (Nancy,France)*, pages 190–203. Springer LNCS 201, 1985.

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

[31] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. *Theoretical Computer Science*, 75:139–156, 1990.

[32] J.W. Klop and A. Middeldorp. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195, 1991.

[33] L. Kott. Unfold/fold program transformation. In M.Nivat and J.C. Reynolds, editors, *Algebraic methods in semantics*, chapter 12, pages 411–434. Cambridge University Press, 1985.

[34] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In T. Mogensen, D. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation — Essays dedicated to Neil Jones*, pages 379–403. Springer LNCS 2566, 2002.

[35] M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.

[36] M.J. Maher. A Transformation System for Deductive Database Modules with Perfect Model Semantics. *Theoretical Computer Science*, 110(2):377–403, 1993.

[37] 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.

[38] G. Moreno. Automatic Optimization of Multi-Paradigm Declarative Programs. In F. J. Garijo, J. C. Riquelme, and M. Toro, editors, *Proc. of the 8th Ibero–American Conference on Artificial Intelligence, IBERAMIA'02*, pages 131–140. Springer LNAI 2527, 2002.

[39] 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.

[40] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[41] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[42] D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[43] W.L. Scherlis. Program Improvement by Internal Specialization. In *Proc. of 8th Annual ACM Symp. on Principles of Programming Languages*, pages 41–49. ACM Press, New York, 1981.

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

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

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

[47] P.L. Wadler. *Listlessness is better than Laziness.* Computer Science Department, CMU-CS-85-171, Carnegie Mellon Univertsity, Pittsburgh, PA, 1985. Ph.D. Thesis.

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

A Proof of Theorem 11

First, we state that transformation sequences preserve the inductively sequential structure of programs. In order to prove it, we need the following technical results from [6]. The first proposition shows that each substitution in a needed narrowing step instantiates only variables occurring in the initial term.

Proposition 27 [6] *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* $x \in \mathcal{V}ar(\varphi_{i-1} \circ \cdots \circ \varphi_1(t))$.

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

Lemma 28 [6] *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* $c \neq c'$.

Finally, we proceed with the proof of Theorem 11, which states that transformation sequences preserve the inductively sequential structure of programs.

Theorem 11 *Let* $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, *be a transformation sequence such that* $\mathcal{R}_0$ *is an inductively sequential program. Then,* $\mathcal{R}_i$ *is inductively sequential, for* $i = 1, \ldots, n$.

PROOF. We prove the claim by induction on i. Since the case $i = 0$ is obvious, we proceed with the inductive case. By the induction hypothesis, we have that $\mathcal{R}_{i-1}$ is inductively sequential. Then, for every defined function symbol f/n in $\mathcal{R}_{i-1}$, there exists a definitional tree of f in $\mathcal{R}_{i-1}$. Now we prove that this result also holds in $\mathcal{R}_i$. We distinguish the following cases depending on the transformation rule applied to $\mathcal{R}_{i-1}$:

Definition introduction. By Definition 3, the program $\mathcal{R}_i$ has been obtained by adding to program $\mathcal{R}_{i-1}$ a new rule of the form $R = (f(\overline{t_n}) \to r)$, such that f is a new function symbol not occurring in the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_{i-1}$ and $f(\overline{t_n})$ is a linear pattern. Trivially (see, e.g., [6]), a function definition which consists of a single rule whose lhs is a linear pattern has always an associated definitional tree. Hence, the inductive sequentiality

of $\mathcal{R}_i$ follows by the induction hypothesis.

Definition elimination. By Definition 4, the program $\mathcal{R}_i$ has been obtained by deleting from program $\mathcal{R}_{i-1}$ all rules in the definition of a given function f. Then, the inductive sequentiality of $\mathcal{R}_i$ follows trivially by the induction hypothesis.

Folding. By Definition 8, the program $\mathcal{R}_i$ has been obtained from $\mathcal{R}_{i-1}$ by removing a rule $R = (l \to r) \in \mathcal{R}_{i-1}$ and adding a new rule of the form $R' = (l \to r[\theta(l')]_p)$ (which cannot contain extra variables due to the restriction imposed in Definition 3). Since the left-hand sides of both programs $\mathcal{R}_{i-1}$ and $\mathcal{R}_i$ coincide, the claim follows by the induction hypothesis.

Unfolding. By Definition 5, the program $\mathcal{R}_i$ has been obtained from $\mathcal{R}_{i-1}$ by removing a rule $R = (l \to r) \in \mathcal{R}_{i-1}$ and adding a new set of rules of the form:

$$\begin{array}{l} \varphi_1(l) \to r_1 \\ \vdots \\ \varphi_m(l) \to r_m \end{array}$$

where $r \leadsto_{\varphi_i} r_i$, $i = 1, \ldots, m$, are all the one-step needed narrowing derivations issuing from r in $\mathcal{R}_{i-1}$. Assume that l is rooted with the defined function f, and f is defined in $\mathcal{R}_{i-1}$ by a set of rules $\{R_j = (l_j \to r_j) \mid j = 0, \ldots, n, n > 0\}$. By the induction hypothesis, there exists a definitional tree $\mathcal{P}$ for f in $\mathcal{R}_{i-1}$. Then, we know that the root of $\mathcal{P}$ is the pattern $f(\overline{x_p})$ (where p is the arity of f) and $S = \{l_1, \ldots, l_n\}$ is the set of leaves of $\mathcal{P}$, where obviously $l \in S$ and $l \in \mathcal{P}$. In order to show the inductive sequentiality of $\mathcal{R}_i$, it suffices to show that there exists a definitional tree $\mathcal{P}'$ for the set

$$S' = (S \backslash \{l\}) \cup \{\varphi_1(l), \ldots, \varphi_m(l)\} \ .$$

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

$$\mathcal{P}' = \mathcal{P} \cup \{\varphi_{ij} \circ \cdots \circ \varphi_{i1}(l) \mid 1 \leq i \leq m, 1 \leq j \leq k_i\} \ .$$

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, that is, $pattern(\mathcal{P}) = pattern(\mathcal{P}')$, 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 disjoint by Lemma 28, the substitutions $\varphi_1, \ldots, \varphi_m$ are pairwise disjoint. Thus, the replacement of the element l in S by the set $\{\varphi_1(l), \ldots, \varphi_m(l)\}$ 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 $\mathcal{P}'$, $\pi = \varphi_{ij} \circ \cdots \circ \varphi_{i1}(l)$ 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 $\pi = \varphi_{i1}(l)$. It is $\varphi_{i1} \neq id$ (otherwise $\pi = l \in \mathcal{P}$). Thus, by Proposition 27, $\varphi_{i1} = \{x \mapsto c(\overline{x_n})\}$ with $x \in \mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Due to the linearity of the initial pattern l and all substituted terms (cf. Proposition 27), l has a single occurrence o of the variable x and, therefore, $\pi = l[c(\overline{x_n})]_o$, i.e., l is the unique parent of π.

Induction step ($j > 1$): We assume that the parent property holds for $\pi' = \varphi_{i,j-1} \circ \cdots \circ \varphi_{i1}(l)$. Let $\varphi_{ij} \neq id$ (otherwise the induction step is trivial). By Proposition 27, $\varphi_{ij} = \{x \mapsto c(\overline{x_n})\}$ with $x \in \mathcal{V}ar(\varphi_{i,j-1} \circ \cdots \circ \varphi_{i1}(l))$ (since $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$). 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) $\pi \in \mathcal{P} \backslash \{l\}$: Then the induction property holds for π since it already holds in $\mathcal{P}$ and only instances of l are added in $\mathcal{P}'$.

(2) $\pi = \varphi_{ij} \circ \cdots \circ \varphi_{i1}(l)$ 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 $\pi = \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1}(l)$). By Proposition 27, $\varphi_{i,j+1} = \{x \mapsto c(\overline{x_n})\}$ and π has a single occurrence of the variable x (due to the linearity of the initial pattern and all substituted terms). Therefore, $\pi' = \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1}(l)$ is a child of π. Consider another child $\pi'' = \varphi_{i'j'} \circ \cdots \circ \varphi_{i'1}(l)$ 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 28, there exists some k with $\varphi_{i'k} \circ \cdots \circ \varphi_{i'1} = \varphi_{il} \circ \cdots \circ \varphi_{i1}$, $\varphi_{i',k+1} = \{x' \mapsto c'(\cdots)\}$, and $\varphi_{i,k+1} = \{x' \mapsto c''(\cdots)\}$ 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 $\varphi_{i',j'} = \cdots = \varphi_{i',k+2} = id$ (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.

B Proof of Lemma 15

We first prove the soundness of unfolding. To this end, we need the following auxiliary result.

Lemma 29 *Let $\mathcal{R}$ be an inductively sequential program and $R, R' \in \mathcal{R}$, with $R = (l \to r)$. Let $r \leadsto_{q,R',\sigma} r'$ be a needed narrowing step such that the result of unfolding R using R' in $\mathcal{R}$ is the rule $R^* = (\sigma(l) \to r')$. If $t \to_{p,R^*} t'$ for some position $p \in \mathcal{P}os(t)$, then $t \to_{p,R} t'' \to_{p.q,R'} t'$.*

PROOF. Given the needed narrowing step $r \leadsto_{q,R',\sigma} r'$, by the soundness of needed narrowing (claim 1 of Theorem 2), we have $\sigma(r) \to_{q,R'} r'$. Since $t \to_{p,R^*} t'$, there exists a substitution θ such that $\theta(\sigma(l)) = t|_p$ and $t' = t[\theta(r')]_p$. Since $\sigma(r) \to_{q,R'} r'$, by the stability of rewriting,[6] we have $\theta(\sigma(r)) \to_{q,R'} \theta(r')$. Therefore $t = t[\theta(\sigma(l))]_p \to_{p,R} t[\theta(\sigma(r))]_p \to_{p.q,R'} t[\theta(r')]_p = t'$.

The soundness of the unfolding transformation can now be proved.

Lemma 30 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$, be a transformation sequence such that $\mathcal{R}_{i+1}$ has been obtained by unfolding, and let e be an equation. If $e \to^* true$ in $\mathcal{R}_{i+1}$ then $e \to^* true$ in $\mathcal{R}_i$.*

PROOF. We prove the claim by induction on the number n of rewrite steps in $\mathcal{D} = [e \to^* true]$:

Base case. Let $n = 0$. Then, we have $e = true$ and the claim follows trivially.
Inductive case. If $n > 0$ we have that $\mathcal{D}$ is not an empty reduction sequence. Let $\mathcal{D} = [e \to_{p,R^*} e' \to^* true]$. If $R^* \in \mathcal{R}_i$, then the claim follows by the inductive hypothesis. Otherwise, R^* is the result of an unfolding transformation. In this case, by Definition 5, there are rules $R = (l \to r), R' \in \mathcal{R}_i$ such that $r \leadsto_{q,R',\sigma} r'$ is a needed narrowing step w.r.t. $\mathcal{R}_i$ and the result of unfolding R using R' in $\mathcal{R}_i$ is the rule $R^* = (\sigma(l) \to r') \in \mathcal{R}_{i+1}$. Moreover, there exists a substitution θ such that $e|_p = \theta(\sigma(l))$. By Theorem 11, $\mathcal{R}_i$ is inductively sequential, and thus, by Lemma 29, we have

$$e \to_{p,R} e[\theta(\sigma(r))]_p \to_{p.q,R'} e[\theta(r')]_p = e'$$

in $\mathcal{R}_i$. Now, the claim follows by applying the inductive hypothesis to the derivation $e' \to^* true$ in $\mathcal{R}_{i+1}$.

[6] Stability means that $t \to^* t'$ implies $\theta(t) \to \theta(t')$ for all terms t, t' and substitution θ.

Let us now consider the completeness of the unfolding rule. Firstly, we recall the notion of *descendant*. Let $A = (t \rightarrow_{u,l\rightarrow r} t')$ be a reduction step of some term t into t' at position u with rule $l \rightarrow r$. The set of *descendants* [28] 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 descendants of a position v by a reduction sequence B is defined inductively as follows

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

Given a set of positions P, we let $P\backslash B = \bigcup_{p\in P}\ p\backslash B$.

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 [37]. In the remainder of this section, we consider *outermost-needed* reduction sequences which are formally defined by means of the following function φ from terms and definitional trees to sets of tuples (position, rule) as the least set satisfying the following properties. We consider two cases for $\mathcal{P}$:

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

$$\varphi(t, \mathcal{P}) = \{(\Lambda, \pi \rightarrow r)\}\ .$$

(2) If π is a branch, consider the inductive position o of π and some 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 $t|_o$:

$$\lambda(t, \mathcal{P}) \ni \begin{cases} (p, R) & \text{if } t|_o = c_i(\overline{t_n}) \text{ and } (p, R, \sigma) \in \lambda(t, \mathcal{P}_i); \\ (o.p, R) & \text{if } t|_o = f(\overline{t_n}) \text{ for } f \in \mathcal{F} \text{ and} \\ & \quad (p, R) = \varphi(t|_o, \mathcal{P}') \text{ where } \mathcal{P}' \\ & \quad \text{is a definitional tree for } f. \end{cases}$$

To compute outermost-needed steps for an operation-rooted term t, we take a definitional tree $\mathcal{P}$ for the root of t and compute $\varphi(t, \mathcal{P})$. Then, for all

$(p, R) \in \varphi(t, \mathcal{P})$, $t \rightarrow_{p,R} t'$ is an *outermost-needed rewrite step*. Outermost-needed reduction and needed narrowing are trivially equivalent for terms without variables.

We also need the following results from [6].

Theorem 31 [6] *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 redexes $\sigma(s)|_p$ 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 $\sigma' \circ \sigma_n \circ \cdots \circ \sigma_1 = \sigma$ $[V]$.*

Lemma 32 [6] *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 of s is root-needed in t.*

The following lemma is auxiliary.

Lemma 33 *Let $\mathcal{R}$ be an inductively sequential program and t be a term. Let $t|_p = \sigma(l)$ be an innermost root-needed redex in t, where $R = (l \rightarrow r) \in \mathcal{R}$, r is operation-rooted, and $p \in \mathcal{P}os(t)$. Given the reduction step $A = [t \rightarrow_{p,R} t[\sigma(r)]_p]$, for every outermost-needed redex $\sigma(r)|_q$ of $\sigma(r)$, we have $p \in \mathcal{NVP}os(r)$.*

PROOF. By contradiction. Assume that there exists an outermost-needed redex $\sigma(r)|_q$ of $\sigma(r)$ and $q \notin \mathcal{NVP}os(r)$. Since $\sigma(r)$ is operation-rooted and $t|_p$ is a root-needed redex in t, then $\sigma(r)$ must contain at least one root-needed redex in t. By Lemma 32, $\sigma(r)|_q$ is a root-needed redex in t. Let $p' \in \mathcal{P}os(t)$ be a position such that $q \in p'\backslash A$, i.e., p' is an *antecedent* of q w.r.t. A. Then, the subterm $t|_{p'}$, $p < p'$, must be a root-needed redex in t, thus contradicting the hypothesis that $t|_p$ is an innermost root-needed redex in t.

The following lemma states the completeness of the unfolding rule (at the level of rewrite sequences). It essentially establishes that each rewrite sequence for an equation in the original program can be mimicked in the unfolded program.

Lemma 34 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$, be a transformation sequence such that $\mathcal{R}_{i+1}$ has been obtained by unfolding, and let e be an equation. If $e \rightarrow^* true$ in $\mathcal{R}_i$ then $e \rightarrow^* true$ in $\mathcal{R}_{i+1}$.*

PROOF. Let $B_1, \ldots, B_m$ be all possible needed reduction sequences from e to $true$, and let k_i be the number of contracted redexes in B_i, $i = 1, \ldots, m$. We

prove the claim by induction on the maximum number $n = max(k_1, \ldots, k_m)$ of contracted needed redexes which are necessary to reduce e to $true$:

$n = 0$**.** This case is trivial since $e = true$.

$n > 0$**.** Since e contains at least one needed redex, then there exists a rule $R = (l \to r) \in \mathcal{R}_i$ such that $e|_p = \theta(l)$ is an innermost needed redex of e. Let us consider the following reduction sequence

$$e \to_{p,R} e[\theta(r)]_p \to^* true$$

in $\mathcal{R}_i$. Now, if R belongs to both programs $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$, then the claim follows by the inductive hypothesis.

Otherwise, R has been unfolded in $\mathcal{R}_i$. Then, by Definition 5, the rhs r of the rule R is not in head normal form and hence $\theta(r)$ is an operation-rooted term. Let $\theta(r)|_q$, $q \in \mathcal{P}os(\theta(r))$, be the outermost needed redex of $\theta(r)$. By Lemma 33, we have that $q \in \mathcal{P}os(r)$. Now, consider an arbitrary reduction step $\theta(r) \to_{q,R'} r'$, $R' \in \mathcal{R}_i$, which contracts the outermost needed redex of r. Then, by Theorem 31, the needed narrowing step $r \leadsto_{q,R',\sigma} r''$ can be proven in $\mathcal{R}_i$, and there is a substitution σ' such that $\sigma'(r'') = r'$ and $\sigma' \circ \sigma = \theta\ [Var(r)]$. By Definition 5, the rule $R^* = (\sigma(l) \to r'')$ belongs to $\mathcal{R}_{i+1}$ (i.e., it is the result of unfolding the rule R using R').

Now, consider the reduction sequence

$$e \to_{p,R} e[\theta(r)]_p \to_{p.q,R'} e[r']_p \to^* true$$

in $\mathcal{R}_i$. Finally, since the following step can be done in $\mathcal{R}_{i+1}$ using the unfolded rule R^*:

$$e = e[\theta(l)]_p = e[\sigma'(\sigma(l))]_p \to_{p,R^*} e[\sigma'(r'')]_p = e[r']_p$$

the claim follows by applying the inductive hypothesis to the subsequence $e[r']_p \to^* true$ in $\mathcal{R}_i$ (whose maximum number of contracted needed redexes is strictly lesser than n).

Finally, Lemma 15 follows trivially by Lemmata 30 and 34.