A Transformation System for Lazy Functional Logic Programs*

María Alpuente[1], Moreno Falaschi[2], Ginés Moreno[3], and Germán Vidal[1]

[1] DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.
{alpuente,gvidal}@dsic.upv.es
[2] Dip. Mat. e Informatica, U. Udine, 33100 Udine, Italy. falaschi@dimi.uniud.it
[3] Dep. Informática, UCLM, 02071 Albacete, Spain. gmoreno@info-ab.uclm.es

Abstract. Needed narrowing is a complete operational principle for modern declarative languages which integrate the best features of (lazy) functional and logic programming. We define a transformation methodology for functional logic programs based on needed narrowing. We provide (strong) 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 benefits of our approach.

1 Introduction

Functional logic programming languages combine the operational principles of the most important declarative programming paradigms, namely functional and logic programming (see [14] 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 [5]. 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 was first introduced in [8] to optimize functional programs and then used for logic programs [28]. This approach

* This work has been partially supported by CICYT under grant TIC 98-0445-C03-01.

is commonly based on the construction, by means of a *strategy*, of a sequence of equivalent programs each obtained by the preceding ones 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, for example, 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 possible by using a fully automatic method (such as, e.g., partial evaluation). Typical instances of this class are the strategies that perform *tupling* (also known as *pairing*) [8, 11], 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, thus avoiding 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 [6]. A closely related strategy is *composition* [30] (also known as *fusion*, *deforestation*, or *vertical jamming* [12]), 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, and composing these two calls into one has the effect to avoid the generation of the intermediate data structure. The composition can be made automatically [30], whereas tupling has only been automated to some extent [9, 10].

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 [7, 13, 20, 21, 23, 28], in functional programming the problem of correctness has received surprisingly little attention [26, 27]. Of the very few studies of correctness of fold/unfold transformations in functional programming, the most general and recent work is [26], which defines a simple (syntactic) condition for restricting general fold/unfold transformations and which can be applied to give correctness proofs for several well-known transformation methods, such as the deforestation.

In [2], 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 [8] which introduces an instance of an existing equation) into unfolding by means of unification. However, [2] does not consider a general transformation system (only two rules: fold and unfold), and hence the composition or tupling transformations cannot be achieved. Also, [2] refers to a notion of "reversible" folding, which is strictly weaker than the one which we consider here. 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 paper 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 [28] 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 [4]. Thus, we demonstrate that such a program structure is preserved through the 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 its effective applicability. For instance, by using other variants of narrowing (e.g., lazy narrowing [22]) the structure of the original program is not preserved, thus seriously restricting the applicability of the resulting system. 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. On the practical side, we investigate how the classical and powerful transformation methodologies of *tupling* and *composition* [24] transfer to our framework. We show the advantages of using needed narrowing to achieve composition and tupling in an integrated setting, and illustrate the power of our transformation system by (automatically) optimizing several significative examples using a prototype implementation [1].

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 simple examples in Sec. 3. We also state the correctness of the transformation system and show some results about the structure of transformed programs. Section 4 shows how to achieve the (automatic) composition and tupling strategies in our framework as well as an experimental evaluation of the method on a small set of benchmarks. Section 5 concludes. More details and proofs of all technical results can be found in [3].

2 Preliminaries

We assume familiarity with basic notions of term rewriting [6] and functional logic programming [14]. 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 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, 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 $\theta_{\lceil V}$ the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta_{\lceil V} = \sigma_{\lceil V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. Two substitutions σ and σ' are *independent* (on a set of variables V) iff there exists some $x \in V$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable.

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A TRS $\mathcal{R}$ is left-linear if l is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor based (CB) if each lhs l is a pattern. In the remainder of this paper, a functional logic *program* is a left-linear CB-TRS. Conditions in program rules are treated by using the predefined functions `and`, `if_then_else`, `case_of` which are reduced by standard defining rules [17, 22]. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position $p \in \mathcal{NVP}os(l)$ and a most-general unifier σ such that $\sigma(l|_p) = \sigma(l')$. 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 notation of a computation 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 $\to$.

To evaluate terms containing variables, narrowing non-deterministically instantiates the variables such 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 $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. 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. 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 λ on the narrowing steps issuing from t, without losing completeness. *Needed narrowing* [5] is currently the best known narrowing strategy due to its optimality properties. It extends the Huet and Lévy's notion of a needed reduction [18]. The definition of needed narrowing [5] uses the notion of *definitional tree* [4]. 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. 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.

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*. Informally speaking, needed narrowing applies a rule, if possible, or checks the subterm corresponding to the inductive position of the branch: if it is a variable, it is instantiated to the constructor of a child; if it is already a constructor, we proceed with the corresponding child; if it is a function, we evaluate it by recursively applying needed narrowing (see [5] for a detailed definition).

Example 1. Consider the following set of rules for "$\leqslant$" and "+":

$$
\begin{array}{rclcl@{\qquad}rcl}
\mathtt{0} & \leqslant & \mathtt{N} & \to & \mathtt{true} & \mathtt{0} + \mathtt{N} & \to & \mathtt{N} \\
\mathtt{s(M)} & \leqslant & \mathtt{0} & \to & \mathtt{false} & \mathtt{s(M)} + \mathtt{N} & \to & \mathtt{s(M+N)} \\
\mathtt{s(M)} & \leqslant & \mathtt{s(N)} & \to & \mathtt{M} \leqslant \mathtt{N} & & &
\end{array}
$$

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

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

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

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

Needed narrowing is sound and complete for inductively sequential programs. Moreover, it is *minimal*, i.e., given two distinct needed narrowing derivations $e \leadsto^*_\sigma \mathit{true}$ and $e \leadsto^*_{\sigma'} \mathit{true}$, we have that σ and σ' are independent 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 computed values and answer substitutions. Let us first give the rules for the introduction and elimination of function definitions in a similar style to [28], in which the set of definitions is partitioned into "old" and "new" ones. In the following, we consider a fixed transformation sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq 0$.

Definition 1 (Definition introduction). *We may get program $\mathcal{R}_{k+1}$ by adding to $\mathcal{R}_k$ a new rule (the "definition rule") 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., it 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 definition is virtually always the first step of a transformation sequence. Determining which definitions should be introduced is a task which falls into the realm of *strategies* (see [23] for a survey), which we discuss in Sec. 4.

Definition 2 (Definition elimination). *We may get program $\mathcal{R}_{k+1}$ by deleting from program $\mathcal{R}_k$ all rules defining the function f, say R^f, such that f does 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 [21] and also in [7], 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 [23] not to allow folding steps if a definition elimination step has been performed.

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

Definition 3 (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 $\{\theta(l) \to r' \mid r \leadsto_\theta r'$ 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 not operation-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 2. Consider the following programs:

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

By a needed narrowing step $\mathtt{s(f(X))} \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} \mathtt{s(0)}$ given from the rhs of the second rule of $\mathcal{R}$, we get (by unfolding) the transformed program $\mathcal{R}'$. Now, the goal $\mathtt{h(g(s(0)))} \approx \mathtt{X}$ has the successful needed narrowing derivation in $\mathcal{R}$

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

whereas it fails in the transformed program. Essentially, completeness is lost because the considered 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 $\mathtt{g}$.

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 4 (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 $p \in \mathcal{NVP}os(r)$, fulfilling the following conditions:*

1. *$r|_p$ is not a constructor term;*
2. *either l (the lhs of the folded rule R) is rooted by an old function symbol, or R is the result of at least one unfolding within the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_k$; and*
3. *the folding rule R' is a definition rule.*[1]

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, that is, reductions are performed against the reverse folding rules. Note that the applicability conditions 2 and 3 for the folding rule guarantee that "self folding" (i.e., the possibility to unsafely fold a rule by itself [23]) is disallowed. There are several points regarding our definition of the folding rule which are worth noticing: (i) As a difference w.r.t. 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 Example 3. (ii) In contrast to [2], the substitution θ of Def. 4 is not a unifier but just a matcher. This is similar to many other folding rules for logic programs, which have been defined in a similar "functional style" (see, e.g., [7, 20, 24, 28]). (iii) Finally, the *non-erasing* condition in Def. 1 can now be fully clarified: it avoids to consider a rule $l \to r$, with $\mathcal{V}ar(r) \subset \mathcal{V}ar(l)$, as a folding rule, since it might introduce extra variables in the rhs of the resulting rule.

Many attempts have been also made to define a folding transformation in a (pure) functional context [8, 27]. A *marked* folding for a lazy (higher-order) functional language has been presented in [26], 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*.

[1] A *definition rule* maintains its status only as long as it remains unchanged, i.e., once a definition rule is transformed it is not considered a *definition rule* anymore.

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., [7, 20, 23, 24, 28]), which in general is crucial to achieve an effective optimization. Some other works in the literature have advocated a different style of folding which is *reversible* [13], 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, which drastically reduces the power of the transformation. The folding rule proposed in [2] for a strict functional logic language is reversible and thus its transformational power is very limited. The folding rule introduced in this paper is more powerful and the applicability conditions are less restrictive.[2] 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 for automatizing the *composition* strategy. However, the transformation system must be empowered for achieving the *tupling* optimization, which we attain by extending the transformation system with a rule of abstraction [8, 26] (often known as *where–abstraction* rule [24]). 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 $\mathtt{double\_sum(X,Y)} \ \rightarrow\ \mathtt{sum(sum(X,Y),sum(X,Y))}$ can be transformed into the new rule

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

As noted by [24], 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 do contain extra variables in the right-hand sides. However, as noted in [26], this can be easily 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 $\mathtt{double\_sum(X,Y)} \ \rightarrow\ \mathtt{sum(Z,Z)\ where\ Z} =$ $\mathtt{sum(X,Y)}$, we can transform it (by lambda lifting [19]) into the new pair of rules

$$\begin{array}{r} \mathtt{double\_sum(X,Y)} \rightarrow \mathtt{ds\_aux(sum(X,Y))} \\ \mathtt{ds\_aux(Z)} \rightarrow \mathtt{sum(Z,Z)} \end{array}$$

Note that these rules can be directly generated from the initial definition by a definition introduction ($\mathtt{ds\_aux(Z)} \rightarrow \mathtt{sum(Z,Z)}$) and then by folding the original rule at the expression $\mathtt{sum(sum(X,Y),sum(X,Y))}$ using as folding rule the newly

[2] It would be interesting to study a generalization of our folding rule to a *disjunctive* folding rule, i.e., allowing the folding of multiple recursive rules (see [25]).

generated definition for `ds_aux/1`. The inclusion of an abstraction rule is traditional in functional fold/unfold frameworks [8, 24, 26, 27]. In the case of logic programs, abstraction is only possible by means of the so called *generalization* strategy [24], 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 5 (Abstraction). *Let $R = (f(\overline{t_n}) \to r) \in \mathcal{R}_k$ be a 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 $\mathcal{R}_k$ by replacing R with $\{f(\overline{t_n}) \to f\_aux(\overline{y_m}, \langle e_1, \ldots, e_j\rangle),\quad f\_aux(\overline{y_m}, \langle z_1, \ldots, z_j\rangle) \to r[\overline{z_j}]_{\overline{P_j}}\}$, where $\overline{z_j}$ are fresh variables not occurring in $\overline{t_n}$, f_aux is a fresh function symbol that does not occur in $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, and $\mathcal{V}ar(r[\overline{z_j}]_{\overline{P_j}}) = \{\overline{y_m}, \overline{z_j}\}$.*

Informally, the two rules generated by the abstraction transformation can be understood as a syntactic variant of the following rule:

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

Now we state the main theoretical results for the basic transformations introduced in this section. We state the correctness of transformation sequences constructed from an inductively sequential program by applying the following rules: definition introduction, definition elimination, unfolding, folding, and abstraction. 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 along a transformation sequence [23]. First, we state that transformations preserve the inductively sequential structure of programs.

Theorem 1. *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$ be a transformation sequence. If $\mathcal{R}_0$ is inductivlely sequential, then $\mathcal{R}_i$ is also inductively sequential, for $i = 1, \ldots, n$.*

Sands formalizes a syntactic *improvement* theory [26] which restricts general fold/unfold transformations and can be applied to give correctness proofs for some existing transformation methods (such as deforestation [30]). However, we find it more convenient to stick to the logic programming methods for proving correctness because the narrowing mechanism can be properly seen as a generalization of the SLD-resolution method which implicitly applies instantiation before replacing a call by the corresponding instance of the body. That is, 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 [26]. Unrestricted instantiation is problematic since it does not even preserve local equivalence, and for this reason the instantiation rule is not considered explicitly in [26]. As a consequence, the improvement theorem of [26] does not directly apply to our context.

Our demonstration technique for the correctness result is inspired by the original proof scheme of Tamaki and Sato [28] concerning the least Herbrand model semantics of logic programs (and the subsequent extension of Kawamura and Kanamori [20] for the semantics of computed answers). Intuitively, a fold/unfold transformation system is correct if there are "at least as many folds as there are unfolds" or, equivalently, if "going backward in the computation (as folding does) does not prevail over going forward in the computation (as unfolding does)" [23, 26]. 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 [28], the *weight of a proof tree* in [20], or the notion of *improvement* in [26]. In our context, we have introduced the notion of *rank of a term* in order to measure the computational cost of a given term. The detailed proof scheme can be found in [3]. The strong correctness of the transformation is stated as follows.

Theorem 2. *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 renaming).*

4 Some Experiments

The building blocks of strategic program optimizations are transformation tactics (*strategies*), which are used to guide the process and effect some particular kind of change to the program undergoing transformation [12, 24].

One of the most relevant quests in applying a transformation strategy is the introduction of new functions, often called in the literature *eureka* definitions. Although there is no general theory of strategies which ensures that derived programs are more efficient than the initial ones, some partial results exist. For instance, in the setting of higher-order (non-strict) functional languages, Sands [26] has recently introduced the theory of *improvement* to provide a syntactic method for guiding and constraining the unfold/fold method in [8] so that total correctness and performance improvement are always guaranteed.

In the following, we illustrate the power of our transformation system by tackling some representative examples regarding the optimizations of composition [30] and tupling [8, 11].

4.1 Transformation Strategies

The composition strategy was originally introduced in [8, 11] for the optimization of pure functional programs. Variants of this composition strategy are the *internal specialization* technique [27] and the *deforestation* method [30]. By using the composition strategy (or its variants), one may avoid the construction of intermediate data structures that are produced by some function `g` and consumed as inputs by another function `f`. In some cases, most of the efficiency improvement of the composition strategy can be simply obtained by lazy evaluation [12]. Nevertheless, the composition strategy often allows the derivation of programs with improved performance also in the context of lazy evaluation [29]. Laziness is decisive when, given a nested function call $\mathtt{f(g(X))}$, the intermediate data structure produced by $\mathtt{g}$ is infinite but the function $\mathtt{f}$ can produce its outcome by knowing only a finite portion of the output of $\mathtt{g}$. The following example illustrates the advantages of our transformation rules w.r.t. those of [2].

Example 3. The function $\mathtt{sum\_prefix(X,Y)}$ defined in the following program $\mathcal{R}_0$ returns the sum of the $\mathtt{Y}$ consecutive natural numbers, starting from $\mathtt{X}$:

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

We can improve the efficiency of $\mathcal{R}_0$ by avoiding the creation and subsequent use of the intermediate, partial list generated by the call to the function $\mathtt{from}$:

1. Definition introduction:

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

2. Unfolding of rule R_7 (note that instantiation is automatic):

$$
\begin{array}{rl}
\mathtt{aux(X,0)} \to \mathtt{0} & (R_8) \\
\mathtt{aux(X,s(Y))} \to \mathtt{suml([X|from(s(X))],s(Y))} & (R_9)
\end{array}
$$

3. Unfolding of rule R_9 (note that this is infeasible with an eager strategy):

$$\mathtt{aux(X,s(Y))} \to \mathtt{X+suml(from(s(X)),Y)} \quad (R_{10})$$

4. Folding of $\mathtt{suml(from(s(X)),Y)}$ in rule R_{10} using R_7:

$$\mathtt{aux(X,s(Y))} \to \mathtt{X+aux(s(X),Y)} \quad (R_{11})$$

5. Folding of the rhs of rule R_1 using R_7:

$$\mathtt{sum\_prefix(X,Y)} \to \mathtt{aux(X,Y)} \quad (R_{12})$$

Then, the transformed program $\mathcal{R}_5$ is formed by the following rules:

$$
\begin{array}{rl}
\mathtt{sum\_prefix(X,Y)} \to \mathtt{aux(X,Y)} & (R_{12}) \\
\mathtt{aux(X,0)} \to \mathtt{0} & (R_8) \\
\mathtt{aux(X,s(Y))} \to \mathtt{X+aux(s(X),Y)} & (R_{11})
\end{array}
$$

(together with the initial definitions for $+$, $\mathtt{from}$, and $\mathtt{suml}$).

Note 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 [2]).

The tupling strategy was introduced in [8, 11] to optimize functional programs. The tupling strategy is very effective when several functions require the computation of the same subexpression, in which case we tuple together those functions. By avoiding either multiple accesses to data structures or common subcomputations one often gets linear recursive programs (i.e., programs whose rhs's have at most one recursive call) from nonlinear recursive programs [24]. The following well-known example illustrates the tupling strategy.

Example 4. The fibonacci numbers can be computed by the program $\mathcal{R}_0$:

$$\begin{array}{rll}
\mathtt{fib(0)} \rightarrow \mathtt{s(0)} & & (R_1)\\
\mathtt{fib(s(0))} \rightarrow \mathtt{s(0)} & & (R_2)\\
\mathtt{fib(s(s(X)))} \rightarrow \mathtt{fib(s(X))} + \mathtt{fib(X)} & & (R_3)
\end{array}$$

(together with the rules for addition $+$). Observe that this program has an exponential complexity, which can be reduced to a linear one by applying the tupling strategy as follows:

1. Definition introduction:
$$\mathtt{new(X)} \rightarrow \langle \mathtt{fib(s(X))}, \mathtt{fib(X)} \rangle \quad (R_4)$$
2. Unfolding of rule R_4 (narrowing the needed redex fib(s(X))):
$$\begin{array}{rll}
\mathtt{new(0)} \rightarrow \langle \mathtt{s(0)}, \mathtt{fib(s(0))} \rangle & & (R_5)\\
\mathtt{new(s(X))} \rightarrow \langle \mathtt{fib(s(X))} + \mathtt{fib(X)}, \mathtt{fib(s(X))} \rangle & & (R_6)
\end{array}$$
3. Unfolding of rule R_5 (narrowing the needed redex fib(s(0))):
$$\mathtt{new(0)} \rightarrow \langle \mathtt{s(0)}, \mathtt{s(0)} \rangle \quad (R_7)$$
4. Abstraction of R_6:
$$\begin{array}{rll}
\mathtt{new(s(X))} \rightarrow \mathtt{new_aux}(\langle \mathtt{fib(s(X))}, \mathtt{fib(X)} \rangle) & & (R_8)\\
\mathtt{new_aux}(\langle \mathtt{Z_1}, \mathtt{Z_2} \rangle) \rightarrow \langle \mathtt{Z_1} + \mathtt{Z_2}, \mathtt{Z_1} \rangle & & (R_9)
\end{array}$$
5. Folding of $\langle \mathtt{fib(s(X))}, \mathtt{fib(X)} \rangle$ in rule R_8 using R_4:
$$\mathtt{new(s(X))} \rightarrow \mathtt{new\_aux(new(X))} \quad (R_{10})$$
6. Abstraction of R_3:
$$\begin{array}{rll}
\mathtt{fib(s(s(X)))} \rightarrow \mathtt{fib_aux}(\langle \mathtt{fib(s(X))}, \mathtt{fib(X)} \rangle) & & (R_{11})\\
\mathtt{fib_aux}(\langle \mathtt{Z_1}, \mathtt{Z_2} \rangle) \rightarrow \mathtt{Z_1} + \mathtt{Z_2} & & (R_{12})
\end{array}$$
7. Folding of $\langle \mathtt{fib(s(X))}, \mathtt{fib(X)} \rangle$ in rule R_{11} using again rule R_4:
$$\mathtt{fib(s(s(X)))} \rightarrow \mathtt{fib\_aux(new(X))} \quad (R_{13})$$

Now, the (enhanced) transformed program $\mathcal{R}_7$ (with linear complexity thanks to the use of the recursive function new), is the following:

$$\begin{array}{rll}
\mathtt{fib(0)} \rightarrow \mathtt{s(0)} & & (R_1)\\
\mathtt{fib(s(0))} \rightarrow \mathtt{s(0)} & & (R_2)\\
\mathtt{fib(s(s(X)))} \rightarrow \mathtt{Z_1} + \mathtt{Z_2} \ \mathtt{where} \ \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new(X)} & & (R_{12}, R_{13})\\
\mathtt{new(0)} \rightarrow \langle \mathtt{s(0)}, \mathtt{s(0)} \rangle & & (R_7)\\
\mathtt{new(s(X))} \rightarrow \langle \mathtt{Z_1} + \mathtt{Z_2}, \mathtt{Z_1} \rangle \ \mathtt{where} \ \langle \mathtt{Z_1}, \mathtt{Z_2} \rangle = \mathtt{new(X)} & & (R_9, R_{10})
\end{array}$$

where rules (R_{12}, R_{13}) and (R_9, R_{10}) are expressed by using local declarations for readability.

Table 1. Benchmark results.

Benchmarks	Rw_1	RT_1	$Comp$	Rw_2	RT_2	Speedup (%)
`doubleappend`	3	1.77	0.1	6	1.63	10%
`sumprefix`	8	3.59	0.21	10	3.48	3%
`lengthapp`	7	1.61	0.17	10	1.51	6%
`doubleflip`	3	0.95	0.11	5	0.7	26%
`fibprefix`	11	2.2	0.28	13	2.26	-3%

4.2 Benchmarks

The basic rules presented so far have been implemented by a prototype system SYNTH [1], which is publicly available at `http://www.dsic.upv.es/users/elp/soft.html`. It is written in SICStus Prolog and includes a parser for the language Curry, a modern multiparadigm declarative language based on needed narrowing which is intended to become a standard in the functional logic community [16, 17]. It also includes a fully automatic composition strategy based on some (apparently reasonable) heuristics. The transformation system allows us to choose between two ways to apply the composition strategy. The first way is semi-automatic, since the user has to indicate the rule in which a nested call appears. A second way is completely automatic. It is the transformer which looks for a nested call in one of the rules and introduces a definition rule for a new function to start the process. We are currently extending the system in order to mechanize tupling (e.g., by using the analysis method of [9]).

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 next column ($Comp$) shows the execution times of the (automatic) composition algorithm. The other columns show the number of rewrite rules (Rw_2), the absolute runtimes (RT_2), and the speedups achieved for the transformed programs. All the programs have been executed by using TasteCurry, which is a publicly available interpreter for a subset of Curry [16]. Times are expressed in seconds and are the average of 10 executions. We note that our (automatic) composition strategy performs well w.r.t. the first four benchmarks. They are classical examples in which composition is able to perform an effective optimization (`sumprefix` is described in Example 3, while `doubleappend`, `lengthapp`, and `doubleflip` are typical functional programs to illustrate deforestation [30]). Regarding the last benchmark `fibprefix`, which is similar to `sumprefix` but it sums Fibonacci numbers instead of natural numbers, a slowdown has been produced (due to an incorrect folding, which added a new function call to the recursion). In this case a tupling strategy is mandatory to succeed, as expected.

In general, the 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 in the derived programs.

5 Conclusions

The definition of a fold/unfold framework for the optimization of functional logic programs was an open problem marked in [24] 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 [18] and the wide literature about completeness of needed narrowing [5] (rather than striving an ad-hoc proof). We have shown that the transformation process keeps the inductively sequential structure of programs. We have also illustrated with several examples that the transformation process can be guided by appropriate strategies which lead to effective improvements. Our 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-automatized way. The multi-paradigm language Curry [15, 17] is an extension of Haskell with features for logic and concurrent programming. The results in this paper can be applied to optimize a large class of kernel (i.e., non concurrent) Curry programs.

Acknowledgements

We thank Ivan Ziliotto and Cesar Ferri for having implemented the transformer SYNTH under SICStus Prolog and for the evaluation of the benchmarks.

References

1. M. Alpuente, M. Falaschi, C. Ferri, G. Moreno, G. Vidal, and I. Ziliotto. The Transformation System SYNTH. Technical report DSIC-II/16/99, DSIC, 1999. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
2. 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 ALP'97*, Springer LNCS 1298, pages 1–15, 1997.
3. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. Technical report, DSIC, UPV, 1999. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
4. S. Antoy. Definitional Trees. In *Proc. of ALP'92*, Springer LNCS 632, pages 143–157, 1992.
5. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pages 268–279, 1994.
6. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
7. A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *JLP*, 16:47–87, 1993.

8. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
9. W. Chin. Towards an Automated Tupling Strategy. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132. ACM, 1993.
10. W. Chin, A. Goh, and S. Khoo. Effective Optimisation of Multiple Traversals in Lazy Languages. In *Proc. of PEPM'99 (Technical Report BRICS-NS-99-1)*, pages 119–130. University of Aarhus, DK, 1999.
11. J. Darlington. Program transformation. In *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.
12. M. S. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. In *IFIP'87*, pages 165–195, 1987.
13. 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.
14. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
15. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM POPL*, pages 80–93. ACM, New York, 1997.
16. 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.
17. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.
18. G. Huet and J.J. Lévy. Computations in Orthogonal Rewriting Systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.
19. T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *FPLCA'85*, pages 190–203. Springer LNCS 201, 1985.
20. T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. *TCS*, 75:139–156, 1990.
21. M.J. Maher. A Transformation System for Deductive Database Modules with Perfect Model Semantics. *Theoretical Computer Science*, 110(2):377–403, 1993.
22. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *JLP*, 12(3):191–224, 1992.
23. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19&20:261–320, 1994.
24. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
25. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A Parameterized Unfold/Fold Transformation Framework for Definite Logic Programs. In *Proc. of PPDP'99*, Springer LNCS, 1999. To appear.
26. D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM ToPLaS*, 18(2):175–234, March 1996.
27. W.L. Scherlis. Program Improvement by Internal Specialization. In *Proc. of 8th ACM POPL*, pages 41–49. ACM Press, 1981.
28. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of 2nd ICLP*, pages 127–139, 1984.
29. P.L. Wadler. *Listlessness is better than Laziness.* Computer Science Department, CMU-CS-85-171, Carnegie Mellon Univertsity, Pittsburgh, PA, 1985. Ph.D. Thesis.
30. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.