A Narrowing-based Instantiation Rule for Rewriting-based Fold/Unfold Transformations

Ginés Moreno [1,2]

Department of Computer Science
University of Castilla-La Mancha
Albacete 02071, Spain

Abstract

In this paper we show how to transfer some developments done in the field of functional–logic programming (FLP) to a pure functional setting (FP). More exactly, we propose a complete fold/unfold based transformation system for optimizing lazy functional programs. Our main contribution is the definition of a safe instantiation rule which is used to enable effective unfolding steps based on rewriting. Since instantiation has been traditionally considered problematic in FP, we take advantage of previous experiences in the more general setting of FLP where instantiation is naturally embedded into an unfolding rule based on narrowing. Inspired by the so called needed narrowing strategy, our instantiation rule inherits the best properties of this refinement of narrowing. Our proposal optimizes previous approaches (that require more transformation effort) defined in the specialized literature of pure FP by anticipating bindings on unifiers used to instantiate a given program rule and by generating redexes at different positions on instantiated rules in order to enable subsequent unfolding steps. As a consequence, our correct/complete technique avoids redundant rules and preserves the natural structure of programs.

Key words: Rewriting, Narrowing, Instantiation, Fold/Unfold

1 Introduction

In this paper we are concerned with first order (lazy) functional programs expressed as term rewriting systems (TRS's for short). Moreover, since higher order programs can be transformed into first order programs by using transformations (suitable for narrowing too) as explained in [17,9], the results of this paper are available for higher order programs, too. Our programming

[1] This work has been partially supported by CICYT under grant TIC 2001-2705-C03-03 and by Acción Integrada Hispano-Alemana HA03-100.
[2] Email: gmoreno@info-ab.uclm.es

language is similar to the one presented in [13,30] in which programs are written as a set of mutually exclusive recursive equations, but we do not require that the set of equations defining a given function f be exhaustive (i.e., we let f to be undefined for certain elements of its domain). More precisely , we use inductively sequential TRS's, i.e., the mainstream class of pattern-matching functional programs which are based on a case distinction. This is a reasonable class of TRS's not only for modeling pure functional programs but also for representing multiparadigm functional–logic programs that combine the operational methods and advantages of both FP and LP (logic programming) [6]. If the operational principle of FP is based on rewriting, integrated programs are usually executed by *narrowing*, a generalization of rewriting that instantiates variables on a given expression and then, it applies a reduction step to a redex of the instantiated expression. Needed narrowing extends the Huet and Lévy's notion of a needed reduction [23], and is the currently best narrowing strategy for first-order (inductively sequential) integrated programs due to its optimality properties w.r.t. the length of derivations and the disjointness of computed solutions and it can be efficiently implemented by pattern matching and unification [8].

The fold/unfold transformation approach was first introduced in [13] to optimize functional programs (by reformulating function definitions) and then used in LP [35] and FLP [2,27]. This approach is commonly based on the construction, by means of a *strategy*, of a sequence of equivalent programs each obtained from the preceding ones 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, for example: instantiation, definition introduction/elimination, and abstraction.

Instantiation is a purely functional transformation rule used for introducing an instance of an existing equation in order to enable subsequent unfolding steps based on rewriting. For instance, if we want to apply an unfolding step on rule $R_1 : \mathtt{f(X)} \to \mathtt{g(X)}$ by reducing its right hand side (rhs) by using rule $R_2 : \mathtt{g(0)} \to \mathtt{0}$, we firstly need to "instantiate" the whole rule with the binding $\{\mathtt{X} \mapsto \mathtt{0}\}$ obtaining the new (instantiated) rule $R_3 : \mathtt{f(0)} \to \mathtt{g(0)}$. Now, we can perform a rewriting step (note that this reduction step is unfeasible on the original rule R_1) on the rhs of R_3 using R_2, and then we obtain the desired (unfolded) rule $R_4 : \mathtt{f(0)} \to \mathtt{0}$. In contrast with this naive example, we will see in Section 3 that, in general, it is not easy to decide neither the binding to be applied nor the subterm to be converted in a reducible expression in the considered rule.

The instantiation rule is avoided in LP and FLP transformation systems: the use of SLD-resolution or narrowing (respectively) empowers the fold/unfold system by implicitly embedding the instantiation rule into unfolding by means of unification. Moreover, instantiation is not treated explicitly even in some pure functional approaches, as is the case of [33] where the role of instantia-

tion is assumed to be played by certain distribution laws for case expressions. Nevertheless, the need for a transformation that generates redexes on rules is crucial in a functional setting if we really want to apply unfolding steps based on rewriting. In contrast with other methods used in FP which are based on strictness analysis [32], our (needed) narrowing-based technique generates, at a very low cost, useful information not only for performing instantiation steps, but also for guiding the application of subsequent unfolding steps.

Similarly to the instantiation process generated implicitly by most unfolding rules for LP ([35,30]) and FLP ([27]), classical instantiation rules for pure FP ([13,14,30]) consider a unique subterm t in a rule R to be converted in a redex and then, R is instantiated with the most general unifiers obtained by unifying t with every rule in the program. Anyway (and as said in [33]), unrestricted instantiation is problematic because it is not even locally equivalence preserving, since it can force premature evaluation of expressions (a problem noted in [12], and addressed in some detail in [32]) and is better suited to a typed language in which one can ensure that the set of instances is exhaustive. Moreover, the use of mgu's in (the also called "minimal") instantiation rules [3] instead of more precise unifiers may produce sets of non mutually exclusive (i.e., overlapping) rules, which leads to corrupt programs.

All these facts strongly contrast with the transformation methodology for lazy (*non-strict*) FLP based on needed narrowing presented in [2]. The use of needed narrowing inside an unfolding step is able to reduce redexes at different positions in a rule once it as been instantiated by using unifiers which are more precise than the standard most general ones. Moreover, the transformation preserves the original structure (inductive sequentiality) of programs and offers strong correctness results (i.e., it preserves the semantics not only of values, but also of computed answer substitutions). Inspired by all these nice results, we show in this paper how to extrapolate the rules presented in [2] to a pure FP setting and, in particular, we built a safe instantiation rule that transcends the limitations of previous approaches.

The structure of the paper is as follows. After recalling some basic definitions in the next section, we introduce our needed narrowing based instantiation rule in Section 3. Section 4 defines an unfolding rule that reinforces the effects of the previous transformation. By adding new rules for folding, abstracting and introducing new definitions, we obtain a complete transformation system for FP in Section 5. Section 6 illustrates our approach with practical examples and a real implementation (the Synth tool). Finally, Section 7 concludes. More details can be found in [24].

[3] Specially when trying to generate redexes at different positions on instantiated rules.

2 Functional Programs *versus* Functional-Logic Programs

For the purposes of this paper, functional and functional–logic programs are undistinguished by syntactic aspects since both can be seen as (inductively sequential) TRS's and they only differ from its corresponding operational semantics which are based on rewriting and narrowing, respectively. In this section we briefly recall some preliminary concepts and notation subjects. We assume familiarity with basic notions from TRS's, FP and FLP [11,19,10]. In this work we consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of *defined* functions. The set of *constructor terms* with *variables* is obtained by using symbols from $\mathcal{C}$ and $\mathcal{X}$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. 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 $d_1, \ldots, d_n$ are constructor terms (note the difference with the usual notion of pattern in functional programming: a constructor term). A term is *linear* if it does not contain multiple occurrences of one variable. 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). $t|_p$ 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. id denotes the identity substitution. We write $t \leq t'$ (*subsumption*) iff $t' = \sigma(t)$ for some substitution σ.

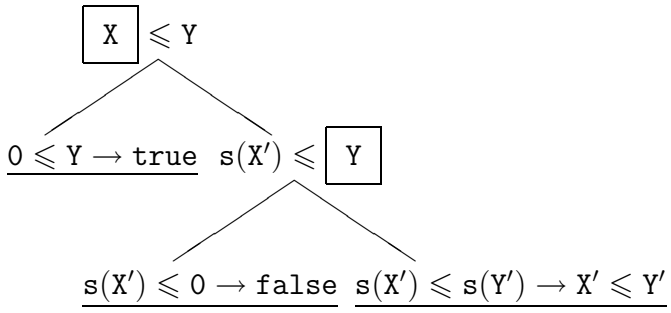
A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (lhs) and the *right-hand side* (rhs) of the rule, respectively. In the remainder of this paper, functional programs are a subclass of TRS's called inductively sequential TRS's. To provide a precise definition of this class of programs we introduce definitional trees [6], which are similar to standard matching trees of FP (definitional trees can also be encoded using *case expressions*, another well-known technique to implement pattern matching in FP [31]). However, differently from left-to-right matching trees, definitional trees deal with *dependencies* between arguments of functional patterns. As a good point, optimality is achieved when definitional trees are used (in this sense, they are closer to *matching dags* or *index trees* for TRSs [23,21]). 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.

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 (underlined) rules. For instance, the definitional tree for the function "$\leqslant$":

$$\begin{array}{rcll} \mathtt{0} & \leqslant & \mathtt{N} & \rightarrow \quad \mathtt{true} \qquad (R_1) \\ \mathtt{s(M)} & \leqslant & \mathtt{0} & \rightarrow \quad \mathtt{false} \qquad (R_2) \\ \mathtt{s(M)} & \leqslant & \mathtt{s(N)} & \rightarrow \quad \mathtt{M} \leqslant \mathtt{N} \qquad (R_3) \end{array}$$

can be illustrated as follows:



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

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

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

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

Induction property: Given $\pi \in \mathcal{P} \backslash S$, there is a position o of π with $\pi|_o \in \mathcal{X}$ (the *inductive position*), 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 $1 \leq i \leq n$.

A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential. An inductively sequential TRS can be viewed as a set of definitional trees, each defining a function symbol. There can be more than one definitional tree for an inductively sequential function. In the following we assume that there is a fixed definitional tree for each defined function.

The operational semantics of FP is based on *rewriting*. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l \rightarrow r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The instantiated lhs $\sigma(l)$ is called a *redex*. $\rightarrow^+$ denotes the

transitive closure of $\rightarrow$ and $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$. By giving priority to *innermost* or, alternatively *outermost* redexes, we obtain two different rewriting principles, corresponding to the so called *eager* (*strict* or *call-by-value*) and *lazy* (*non-strict*, or *call-by-need*) functional programs, respectively.

On the other hand, the operational semantics of integrated languages is usually based on *narrowing*, a combination of variable instantiation and reduction. Formally, $t \leadsto_{p,R,\sigma} s$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \rightarrow_{p,R} s$. Modern functional–logic languages are based on *needed narrowing* and *inductively sequential* programs (a detailed description can be found in [8]).

3 Instantiation

Following [13], the instantiation rule is used in pure functional transformation systems to introduce, or more appropriately, to replace substitution instances of an existing equation in a program. Similarly to the implicit instantiation produced by the narrowing calculus, the goal here is to enable subsequent rewriting steps. This is not an easy task, since there may exist many different alternatives to produce bindings and redexes, and many approaches do not offer correctness results or/and they do not consider instantiation explicitly [13,33]. In this section we face this problem by defining a safe instantiation rule inspired by the needed narrowing based unfolding transformation for FLP of [2]. This last rule mirrors the needed narrowing calculus by implicitly performing instantiation operations that use more precise bindings than mgu's and generate redexes at different positions on rules.

For the definition of needed narrowing we assume that t is a term rooted with a defined function symbol 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) such that, for all $(p, R, \sigma) \in \lambda(t, \mathcal{P})$, $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*[4]. Function λ is not only useful for defining needed narrowing, but also for defining our instantiation rule. We consider two cases for $\mathcal{P}$[5]:

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

(ii) 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

[4] A more precise characterization of this notion is the so called canonical representation $(p, R, \phi_{ik_i} \circ \cdots \circ \phi_{i1}) \in \lambda(t, \mathcal{P})$, where σ is expressed by explicitly composing all the individual bindings computed by (the recursive invocation of) function λ.

[5] This description of a needed narrowing step is slightly different from [8] but it results in the same needed narrowing steps.

cases for the subterm $t|_o$:

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

Informally speaking, needed narrowing applies a rule, if possible (case 1), or checks the subterm corresponding to the inductive positions 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 function λ. Thus, the strategy differs from lazy functional languages only in the instantiation of free variables. Note that we compose in each recursive step during the computation of λ the substitution with the local substitution of this step (which can be the identity).

Now, we re-use function λ for defining our instantiation rule. Given the last (inductively sequential) program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, and a rule $(l \to r) \in \mathcal{R}_k$ where r is rooted by a defined function symbol with definitional tree $\mathcal{P}_r$, we may get program $\mathcal{R}_{k+1}$ by application of the instantiation rule as follows:

$$\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{l \to r\} \quad \cup \quad \{\sigma(l) \to \sigma(r) \mid (p, R, \sigma) \in \lambda(r, \mathcal{P}_r)\}.$$

In order to illustrate our instantiation rule, consider again a program $\mathcal{R}$ with the set of rules defining "$\leqslant$" together with the new rules for `test` and `double`:

$$\texttt{test}(\texttt{X},\texttt{Y}) \to \texttt{X} \leqslant \texttt{double}(\texttt{Y}) \qquad (R_4)$$

$$\texttt{double}(\texttt{0}) \to \texttt{0} \qquad (R_5)$$

$$\texttt{double}(\texttt{s}(\texttt{X})) \to \texttt{s}(\texttt{s}(\texttt{double}(\texttt{X}))) \qquad (R_6)$$

Then, function λ computes the following set for the initial term $\texttt{X} \leqslant \texttt{double}(\texttt{Y})$: $\{(\Lambda, R_1, \{\texttt{X} \mapsto \texttt{0}\}),\ (2, R_5, \{\texttt{X} \mapsto \texttt{s}(\texttt{X}'), \texttt{Y} \mapsto \texttt{0}\}),\ (2, R_6, \{\texttt{X} \mapsto \texttt{s}(\texttt{X}'), \texttt{Y} \mapsto \texttt{s}(\texttt{Y}')\})\}$. Now, by application of the instantiation rule, we replace R_4 in $\mathcal{R}$ by the rules:

$$\texttt{test}(\texttt{0},\texttt{Y}) \to \texttt{0} \leqslant \texttt{double}(\texttt{Y}) \qquad (R_7)$$

$$\texttt{test}(\texttt{s}(\texttt{X}'),\texttt{0}) \to \texttt{s}(\texttt{X}') \leqslant \texttt{double}(\texttt{0}) \qquad (R_8)$$

$$\texttt{test}(\texttt{s}(\texttt{X}'),\texttt{s}(\texttt{Y}')) \to \texttt{s}(\texttt{X}') \leqslant \texttt{double}(\texttt{s}(\texttt{Y}')) \qquad (R_9)$$

In contrast with the usual instantiation rule used in pure FP and similarly to the needed narrowing calculus used in FLP, we observe two important

properties enjoyed by our transformation rule:

(i) It is able to instantiate more than a unique subterm at a time. In fact, subterms at positions Λ and 2 have been considered in the example. Observe that, if, for instance, we only perform the operation on subterm at position Λ, only rule R_7 could be achieved and then, the correctness of the transformation would be lost (i.e., function `test` would remain undefined for other terms different from 0 as first parameter).

(ii) It uses more specific unifiers than mgu's. This is represented by the extra binding $\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')$ applied in rules R_8 and R_9. Observe that the generation of this extra binding is crucial to preserve the structure (inductive sequentiality) of the transformed program. Otherwise, the lhs's of rules R_8 and R_9 would (erroneously) be $\mathtt{test}(\mathtt{X}, \mathtt{0})$ and $\mathtt{test}(\mathtt{X}, \mathtt{s}(\mathtt{Y}'))$ that unify with the lhs of rule R_7.

Experiences in FLP show that the preservation of the program structure is the key point to prove the correctness of a transformation system based on needed narrowing [2,4]. Instantiation is the unique transformation rule used throughout this paper that modifies the head of program rules, thus compromising the natural structure of programs. Moreover, since the instantiation rule is not explicitly used in [2], we now formally establish the following result (proved in the appendix section) specially formulated for this (purely functional) transformation in an isolated way.

Theorem 3.1 *The application of the instantiation rule to an inductively sequential program generates an inductively sequential program too.*

4 Unfolding

In this section we focus on the counterpart of the instantiation rule, that is, the unfolding transformation which basically consists of replacing a rule by a new one obtained by the application of a rewriting step to a redex in its rhs. This transformation is strongly related to the previous one, since it is usually used in pure FP to generate redexes in the rhs of program rules. In our case, we take advantage once again of the information generated by function λ to proceed with our rewriting based unfolding rule. Hence, given the last (inductively sequential) program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, and a (previously instantiated) rule $(\sigma(l) \to \sigma(r)) \in \mathcal{R}_k$ such that $(p, R, \sigma) \in \lambda(r, \mathcal{P}_r)$, then we may get program $\mathcal{R}_{k+1}$ by application of the unfolding rule as follows:

$$\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{\sigma(l) \to \sigma(r)\} \quad \cup \quad \{\sigma(l) \to r' \mid \sigma(r) \to_{p,R} r'\}.$$

Observe that, as defined in Section 2, given a term r, if $(p, R, \sigma) \in \lambda(r, \mathcal{P}_r)$ then $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*. Hence, a program obtained by substituting one of its rules, say $l \to r$, by the set $\{\sigma(l) \to r' \mid r \leadsto_{p,R,\sigma} r'\}$, can be considered as the result of transforming it by applying the appropriate

instantiation and unfolding steps. In fact, the unfolding rule based on needed narrowing presented in [2] is defined in this way, and for this reason it is said that instantiation is naturally embedded by unfolding in FLP.

Continuing now with our example, we can unfold rule R_7 by rewriting the redex at position Λ of its rhs (i.e., the whole term $\mathtt{0} \leqslant \mathtt{double(Y)}$) and using rule R_1, obtaining the new rule:

$$\mathtt{test(0,Y)} \rightarrow \mathtt{true} \quad (R_{10})$$

Similarly, the unfolding of rules R_8 and R_9 (on subterms at position 2 of its rhs's, with rules R_5 and R_6 respectively) generates:

$$\mathtt{test(s(X'),0)} \rightarrow \mathtt{s(X')} \leqslant \mathtt{0} \qquad (R_{11})$$

$$\mathtt{test(s(X'),s(Y'))} \rightarrow \mathtt{s(X')} \leqslant \mathtt{s(s(double(Y')))} \quad (R_{12})$$

Finally, rules R_{11} and R_{12} admit empty instantiations and can be unfolded with rules R_2 and R_3 respectively, obtaining:

$$\mathtt{test(s(X'),0)} \rightarrow \mathtt{false} \qquad (R_{13})$$

$$\mathtt{test(s(X'),s(Y'))} \rightarrow \mathtt{X'} \leqslant \mathtt{s(double(Y'))} \qquad (R_{14})$$

Let us see now the effects that would be produced (in our original program) by the application of unfolding steps preceded by the classical instantiation steps used in traditional functional transformation systems. Starting again with rule R_4, we try to generate a redex in its rhs by instantiation. We have seen in Section 3 that, by applying the binding $\{\mathtt{X} \mapsto \mathtt{0}\}$ to R_4 in order to enable a subsequent unfolding step with rule R_1, we obtain an incorrect program. Hence, we prefer to act on subterm $\mathtt{double(Y)}$ in rule R_4 in order to later reduce its associated redexes with rules R_5 and R_6. This is achieved by applying the ("minimal") bindings $\{\mathtt{Y} \mapsto \mathtt{0}\}$ and $\{\mathtt{Y} \mapsto \mathtt{s(Y')}\}$ to R_4:

$$\mathtt{test(X,0)} \rightarrow \mathtt{X} \leqslant \mathtt{double(0)} \qquad (R'_7)$$

$$\mathtt{test(X,s(Y'))} \rightarrow \mathtt{X} \leqslant \mathtt{double(s(Y'))} \qquad (R'_8)$$

This process is called "minimal" instantiation in [14], since it uses the mgu's obtained by exhaustively unifying the subterm to be converted in a redex with the lhs's of the rules in the program. The corresponding unfolding steps on R'_7 and R'_8 return:

$$\mathtt{test(X,0)} \rightarrow \mathtt{X} \leqslant \mathtt{0} \qquad (R'_9)$$

$$\mathtt{test(X,s(Y'))} \rightarrow \mathtt{X} \leqslant \mathtt{s(s(double(Y')))} \qquad (R'_{10})$$

Now, we can perform two instantiation steps, one for rule R'_9 and one more for rule R'_{10}, using the bindings $\{\mathtt{X} \mapsto \mathtt{0}\}$ and $\{\mathtt{X} \mapsto \mathtt{s(X')}\}$, and obtaining the

rules (previously generated by our method) R_{11} and R_{12} together with:

$$\texttt{test}(\texttt{0},\texttt{0}) \to \texttt{0} \;\leqslant\; \texttt{0} \qquad (R'_{11})$$

$$\texttt{test}(\texttt{0},\texttt{s}(\texttt{Y}')) \to \texttt{0} \;\leqslant\; \texttt{s}(\texttt{s}(\texttt{double}(\texttt{Y}'))) \qquad (R'_{12})$$

Finally, the unfolding of R'_{11} and R'_{12} returns:

$$\texttt{test}(\texttt{0},\texttt{0}) \to \texttt{true} \qquad (R'_{13})$$

$$\texttt{test}(\texttt{0},\texttt{s}(\texttt{Y}')) \to \texttt{true} \qquad (R'_{14})$$

Observe now that our method, instead of producing these last rules, only generates the unique rule R_{10} that subsumes both ones. Moreover, compared with the traditional transformation process, our method is faster apart from producing smaller (correct) programs. Our transformation rules inherit these nice properties from the optimality of needed narrowing (see [8]) with respect to:

(i) the (shortest) length of successful derivations for a given goal which, in our framework, implies that less instantiation and unfolding steps are needed when building a transformation sequence, and

(ii) the independence of (minimal) computed solutions associated to different successful derivations, which guarantees that no redundant rules are produced during the transformation process.

5 Folding and Related Rules

In order to complete our transformation system, let us now 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. Roughly speaking, in FP the folding operation proceeds in the opposite direction of the usual reduction steps, that is, reduction is performed against a reversed program rule. Now we adapt the folding operation of [2] to FP. Given the last (inductively sequential) program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, and two rules belonging to R_k, say $l \to r$ (the "folded" rule) and $l' \to r'$ (the "folding" rule), such that there exists an operation rooted subterm of r which is an instance of r', i.e., $r|_p = \theta(r')$, we may get program $\mathcal{R}_{k+1}$ by application of the folding rule as follows:

$$\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{l \to r\} \;\cup\; \{l \to r[\theta(l')]_p\}.$$

For instance, by folding rule $\texttt{f}(\texttt{s}(\texttt{0})) \to \texttt{s}(\texttt{f}(\texttt{0}))$ with respect to rule $\texttt{g}(\texttt{X}) \to \texttt{f}(\texttt{X})$, we obtain the new rule $\texttt{f}(\texttt{s}(\texttt{0})) \to \texttt{s}(\texttt{g}(\texttt{0}))$. Observe that substitution θ used in our definition is not a unifier but just a matcher. This is similar to many other folding rules for LP and FLP, which have been defined in a similar "functional style" (see, e.g., [30,35,2]), and can still produce at a very low cost many powerful optimizations.

In order to increase the optimization power of our folding rule we now introduce a new kind of transformation called definition introduction rule. So, given the last (inductively sequential) program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, we may get program $\mathcal{R}_{k+1}$ by adding to $\mathcal{R}_k$ a new rule, called *definition rule* or *eureka*, of the form $f(\overline{x}) \rightarrow r$, where f is a *new* function symbol not occurring in the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_k$ and $\mathcal{V}ar(r) = \overline{x}$. Now, we can reformulate our folding rule by imposing that the "folding" rule be an eureka belonging to any program in the transformation sequence whereas the "folded" rule never be an eureka. Note that these new applicability conditions enhances the original definition by relaxing the (strong) condition that the "folding" rule belong to the last program in the sequence, which in general is crucial to achieve effective optimizations [35,30,2]. Moreover, the possibility to unsafely fold a rule by itself ("self folding") is disallowed.

Let us illustrate our definitions. Assume a program containing the set of rules defining "+" together with the following rule R_1 : $\texttt{twice}(\texttt{X},\texttt{Y}) \rightarrow (\texttt{X} + \texttt{Y}) + (\texttt{X} + \texttt{Y})$. Now, we apply the definition introduction rule by adding to the program the following eureka rule R_2: $\texttt{new}(\texttt{Z}) \rightarrow \texttt{Z} + \texttt{Z}$. Finally, if we fold R_1 using R_2, we obtain rule R_3: $\texttt{twice}(\texttt{X},\texttt{Y}) \rightarrow \texttt{new}(\texttt{X} + \texttt{Y})$. Note that the new definition of `twice` enhances the original one, since rules R_2 and R_3 can be seen as a *lambda lifted* version (inspired by the one presented by [33]) of the following rule, which is expressed by means of a local declaration built with the `where` construct typical of functional languages: $\texttt{twice}(\texttt{X},\texttt{Y}) \rightarrow \texttt{Z} + \texttt{Z}$ `where` $\texttt{Z} = \texttt{X}+\texttt{Y}$. This example shows a novel, nice capability of our definition introduction and folding rules: they can appropriately be combined in order to implicitly produce the effects of the so-called *abstraction rule* of [13,33] (often known as *where–abstraction rule* [30]). The use of this transformation is mandatory in order to obtain the benefits of the powerful tupling strategy [13,14,25], as we will see in the following section [6].

The set of rules presented so far is correct/complete (for FP), as formalized in the following theorem, which is an immediate consequence of the strong correctness of the transformation system (for FLP) presented in [2] [7].

Theorem 5.1 *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$ be a transformation sequence constructed by the application of the following transformation rules: definition introduction, instantiation, unfolding and folding. Let t be a term without new function symbols and s a constructor term. Then, $t \rightarrow^* s$ in $\mathcal{R}_0$ iff $t \rightarrow^* s$ in $\mathcal{R}_n$.*

[6] In [2] we formalize an abstraction rule that allows the abstraction of different expressions simultaneously.

[7] We omit the formal proof here since it directly corresponds to the so called *invariant I1* proved there. Proof details can be found in [4].

6 Some examples

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 [16,30]. One of the most relevant quests in applying a transformation strategy is the introduction of new functions, often called in the literature *eureka* definitions. In the following, we illustrate the power of our transformation system by tackling some representative examples regarding the optimizations of composition and tupling [13]. Both strategies have been recently automated for functional–logic programs in [3] and [25] respectively, and they can be easily adapted to functional programming by simple considering the instantiation rule we have just introduced in this paper. See also [1] for an implemented prototype that optimizes both pure functional programs and functional–logic programs.

6.1 Composition

The composition strategy was originally introduced in [13] for the optimization of pure functional programs (see [29,30] for more details). Variants of this composition strategy are the *internal specialization* technique [34] and the *deforestation* method [37]. 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 order to illustrate the composition strategy, consider the following program $\mathcal{R}$ which defines the function `dapp` to concatenate three lists (by applying the standard function `append` twice):

$$\texttt{dapp}(\texttt{X},\texttt{Y},\texttt{Z}) \to \texttt{append}(\texttt{append}(\texttt{X},\texttt{Y}),\texttt{Z}) \quad (R_1)$$

$$\texttt{append}([\,],\texttt{X}) \to \texttt{X} \quad (R_2)$$

$$\texttt{append}([\texttt{H}|\texttt{T}],\texttt{X}) \to [\texttt{H}|\texttt{append}(\texttt{T},\texttt{X})] \quad (R_3)$$

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

(i) First, we introduce the following eureka definition:

$$\texttt{new}(\texttt{X},\texttt{Y},\texttt{Z}) \to \texttt{append}(\texttt{append}(\texttt{X},\texttt{Y}),\texttt{Z}) \quad (R_4)$$

(ii) Now, since function λ applied to the term $\texttt{append}(\texttt{append}(\texttt{X},\texttt{Y}),\texttt{Z})$ returns the following pair of tuples: $(\texttt{1}, R_2, \{\texttt{X} \mapsto [\,]\})$ and $(\texttt{1}, R_3, \{\texttt{X} \mapsto [\texttt{H}|\texttt{T}]\})$, we apply the instantiation rule on rule R_4 as follows:

$$\texttt{new}([\,],\texttt{Y},\texttt{Z}) \to \texttt{append}(\underline{\texttt{append}([\,],\texttt{Y})},\texttt{Z}) \quad (R_5)$$

$$\texttt{new}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) \to \texttt{append}(\underline{\texttt{append}([\texttt{H}|\texttt{T}],\texttt{Y})},\texttt{Z}) \quad (R_6)$$

(iii) The unfolding of the underlined subterms in rules R_5 and R_6 using R_2 and R_3 respectively, generates:

$$\texttt{new}([\,],\texttt{Y},\texttt{Z}) \rightarrow \texttt{append}(\texttt{Y},\texttt{Z}) \qquad (R_7)$$
$$\texttt{new}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) \rightarrow \texttt{append}([\texttt{H}|\texttt{append}(\texttt{T},\texttt{Y})],\texttt{Z}) \quad (R_8)$$

(iv) Since function λ applied to the whole rhs of rule R_8 returns the (unique) tuple (Λ, R_2, id), the application of the instantiation rule returns a new rule R_9 which is identical to R_8 (due to the use of an empty unifier).

(v) Now, we unfold (the whole rhs of) rule R_9 using R_2 obtaining:

$$\texttt{new}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) \rightarrow [\texttt{H}|\texttt{append}(\texttt{append}(\texttt{T},\texttt{Y}),\texttt{Z})] \quad (R_{10})$$

(vi) Finally, we fold the instances of $\texttt{append}(\texttt{append}(\texttt{X},\texttt{Y}),\texttt{Z})$ in rules R_1 and R_{10} using rule R_4:

$$\texttt{dapp}(\texttt{X},\texttt{Y},\texttt{Z}) \rightarrow \texttt{new}(\texttt{X},\texttt{Y},\texttt{Z}) \qquad (R_{11})$$
$$\texttt{new}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) \rightarrow [\texttt{H}|\texttt{new}(\texttt{T},\texttt{Y},\texttt{Z})] \quad (R_{12})$$

Observe that this transformation sequence emerges with the desired recursive definition for `new` (rules R_7 and R_{12}). Hence, the transformed program (which is an improvement w.r.t. the original program $\mathcal{R}$) is the following:

$$\texttt{dapp}(\texttt{X},\texttt{Y},\texttt{Z}) \rightarrow \texttt{new}(\texttt{X},\texttt{Y},\texttt{Z}) \qquad (R_{11})$$
$$\texttt{append}([\,],\texttt{X}) \rightarrow \texttt{X} \qquad (R_2)$$
$$\texttt{append}([\texttt{H}|\texttt{T}],\texttt{X}) \rightarrow [\texttt{H}|\texttt{append}(\texttt{T},\texttt{X})] \quad (R_3)$$
$$\texttt{new}([\,],[\texttt{H}|\texttt{T}],\texttt{Z}) \rightarrow [\texttt{H}|\texttt{append}(\texttt{T},\texttt{Z})] \quad (R_7)$$
$$\texttt{new}([\texttt{H}|\texttt{T}],\texttt{Y},\texttt{Z}) \rightarrow [\texttt{H}|\texttt{new}(\texttt{T},\texttt{Y},\texttt{Z})] \quad (R_{12})$$

It can be argued that most of the efficiency improvements achieved by the composition strategy can be simply obtained by lazy evaluation [16]. Nevertheless, the composition strategy often allows the derivation of programs with improved performance also in the context of lazy evaluation [36]. 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`, as illustrates the following example.

The $\texttt{sum\_prefix}(\texttt{X},\texttt{Y})$ function defined in the following program $\mathcal{R}_0$ returns the sum of the `Y` consecutive natural numbers, starting from `X`. This function first generates an infinite list of consecutive numbers starting from `X`, and then

it adds the first Y numbers of the list:

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

Note that function `from` is non-terminating (which does not affect the correctness of the transformation). 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 function `from`:

(i) Definition introduction:

$$\texttt{new}(\texttt{X},\texttt{Y}) \to \texttt{suml}(\texttt{from}(\texttt{X}),\texttt{Y}) \quad (R_7)$$

(ii) Now, function λ applied to $\texttt{suml}(\texttt{from}(\texttt{X}),\texttt{Y})$ returns the pair of tuples $(\Lambda, R_2, \{\texttt{Y} \mapsto \texttt{0}\})$ and $(\texttt{1}, R_4, \{\texttt{Y} \mapsto \texttt{s}(\texttt{Y}')\})$. It is important to note here that our strategy generates this second tuple with a non empty identifier (as would produce other simpler methods that simply would unfold subterm $\texttt{from}(\texttt{X})$ without instantiating the whole rule,hence damaging the program structure) in order to preserve the inductive sequentiality of the resulting program. Now, we apply the instantiation rule as follows:

$$\begin{aligned}
\texttt{new}(\texttt{X},\texttt{0}) &\to \underline{\texttt{suml}(\texttt{from}(\texttt{X}),\texttt{0})} && (R_8)\\
\texttt{new}(\texttt{X},\texttt{s}(\texttt{Y}')) &\to \texttt{suml}(\underline{\texttt{from}(\texttt{X})},\texttt{s}(\texttt{Y}')) && (R_9)
\end{aligned}$$

(iii) Unfolding of the underlined subtems on rules R_8 and R_9 using R_2 and R_4:

$$\begin{aligned}
\texttt{new}(\texttt{X},\texttt{0}) &\to \texttt{0} && (R_{10})\\
\texttt{new}(\texttt{X},\texttt{s}(\texttt{Y}')) &\to \texttt{suml}([\texttt{X}|\texttt{from}(\texttt{s}(\texttt{X}))],\texttt{s}(\texttt{Y}')) && (R_{11})
\end{aligned}$$

(iv) Instantiation of rule R_{11} using the tuple (Λ, R_3, id):

$$\texttt{new}(\texttt{X},\texttt{s}(\texttt{Y}')) \to \underline{\texttt{suml}([\texttt{X}|\texttt{from}(\texttt{s}(\texttt{X}))],\texttt{s}(\texttt{Y}'))} \quad (R_{12})$$

Observe that since the unifier generated by the call to function λ is empty and hence, it is not relevant for the instantiation step itself, we need the more significant position and rule information to perform the next unfolding step.

(v) Unfolding of the underlined subterm in R_{12} using R_3 (note that this is infeasible with an eager strategy):

$$\texttt{new}(\texttt{X},\texttt{s}(\texttt{Y}')) \to \texttt{X}+\underline{\texttt{suml}(\texttt{from}(\texttt{s}(\texttt{X})),\texttt{Y}')} \quad (R_{13})$$

(vi) Folding of the underlined subterm (which is an instance of the lhs of rule R_7)in rule R_{10} using the eureka rule R_7:

$$\texttt{new}(\texttt{X}, \texttt{s}(\texttt{Y}')) \rightarrow \texttt{X} + \texttt{new}(\texttt{s}(\texttt{X}), \texttt{Y}') \quad (R_{14})$$

(vii) Folding of the rhs of rule R_1 using R_7:

$$\texttt{sum\_prefix}(\texttt{X}, \texttt{Y}) \rightarrow \texttt{new}(\texttt{X}, \texttt{Y}) \quad (R_{15})$$

Then, the transformed and enhanced final program is:

$$\begin{aligned} \texttt{sum\_prefix}(\texttt{X}, \texttt{Y}) &\rightarrow \texttt{new}(\texttt{X}, \texttt{Y}) && (R_{15}) \\ \texttt{new}(\texttt{X}, \texttt{0}) &\rightarrow \texttt{0} && (R_{10}) \\ \texttt{new}(\texttt{X}, \texttt{s}(\texttt{Y}')) &\rightarrow \texttt{X} + \texttt{new}(\texttt{s}(\texttt{X}), \texttt{Y}') && (R_{14}) \end{aligned}$$

(together with the initial definitions for +, `from`, and `suml`). Note that the use of instantiation and unfolding rules based on needed narrowing is essential in the above example. It ensures that no redundant rules are produced and it also allows the transformation to succeed even in the presence of non-terminating functions. Table 1 shows two more examples (`lengthapp` and `doubleflip`) of program optimization by using the composition strategy.

6.2 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 [13,15] to optimize functional programs. Essentially, it proceeds by grouping calls with common arguments together so that their results are computed simultaneously. When the strategy succeeds, multiple traversals of data structures can be avoided, thus transforming a nonlinear recursive program into a linear recursive one [30]. Unfortunately, the tupling strategy is more involved than the composition strategy. In contrast with the composition strategy, where eureka rules always contains nested call in its rhs, the tupling strategy often requires complex analyses in order to extract appropriate tuples of calls to be grouped together [14]. The following well-known example illustrates the tupling strategy.

The fibonacci numbers can be computed in an inefficient way by the following initial program $\mathcal{R}_0$:

$$\begin{aligned} \texttt{fib}(\texttt{0}) &\rightarrow \texttt{s}(\texttt{0}) && (R_1) \\ \texttt{fib}(\texttt{s}(\texttt{0})) &\rightarrow \texttt{s}(\texttt{0}) && (R_2) \\ \texttt{fib}(\texttt{s}(\texttt{s}(\texttt{X}))) &\rightarrow \texttt{fib}(\texttt{s}(\texttt{X})) + \texttt{fib}(\texttt{X}) && (R_3) \end{aligned}$$

(together with the rules for addition +). Observe that this program has an exponential complexity that can be reduced to linear by applying the tupling

strategy as follows:

(i) Definition introduction:

$$\texttt{new(X)} \rightarrow (\texttt{fib(s(X))}, \texttt{fib(X)}) \quad (R_4)$$

(ii) Instantiation of rule R_4 by using the intended tuples $(\texttt{1}, R_2, \{\texttt{X} \mapsto \texttt{0}\})$ and $(\texttt{1}, R_3, \{\texttt{X} \mapsto \texttt{s(X')}\})$:

$$\texttt{new(0)} \rightarrow (\underline{\texttt{fib(s(0))}}, \texttt{fib(0)})) \quad (R_5)$$

$$\texttt{new(s(X'))} \rightarrow (\underline{\texttt{fib(s(s(X')))}}, \texttt{fib(s(X'))}) \quad (R_6)$$

(iii) Unfolding (the underlined subterms in) rules R_5 and R_6 using R_2 and R_3, respectively, obtaining:

$$\texttt{new(0)} \rightarrow (\texttt{s(0)}, \texttt{fib(0)}) \quad (R_7)$$

$$\texttt{new(s(X'))} \rightarrow (\texttt{fib(s(X'))} + \texttt{fib(X')}, \texttt{fib(s(X'))}) \quad (R_8)$$

Before continuing with our transformation sequence we firstly reduce subterm $\texttt{fib(0)}$ in rule R_7 in order to obtain the following rule R_9 which represents a case base definition for `new`:

$$\texttt{new(0)} \rightarrow (\texttt{s(0)}, \texttt{s(0)}) \quad (R_{10})$$

This kind of *normalizing* step (which obviously is a particular case of unfolding step) that does not require a previous instantiation step is performed automatically by the incremental tupling algorithm described in [25].

(iv) In order to proceed with the abstraction of rule R_8, we decompose the process in two low level transformation steps as described in Section 5:

- Definition introduction:

$$\texttt{new\_aux}((\texttt{Z}_1, \texttt{Z}_2)) \rightarrow (\texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1) \quad (R_{10})$$

- Folding R_8 w.r.t. R_{10}:

$$\texttt{new(s(X'))} \rightarrow \texttt{new\_aux}((\texttt{fib(s(X'))}, \texttt{fib(X')})) \quad (R_{11})$$

(v) Folding R_{11} using R_4:

$$\texttt{new(s(X'))} \rightarrow \texttt{new\_aux(new(X'))} \quad (R_{12})$$

(vi) Abstraction of R_3:

- Definition introduction:

$$\texttt{fib\_aux}((\texttt{Z}_1, \texttt{Z}_2)) \rightarrow \texttt{Z}_1 + \texttt{Z}_2 \quad (R_{13})$$

- Folding R_3 w.r.t. R_{13}:

$$\texttt{fib(s(s(X)))} \rightarrow \texttt{fib\_aux}((\texttt{fib(s(X))}, \texttt{fib(X)})) \quad (R_{14})$$

(vii) Folding R_{14} using R_4:

$$\texttt{fib(s(s(X)))} \rightarrow \texttt{fib\_aux(new(X))} \quad (R_{15})$$

Now, the transformed program (with linear complexity thanks to the use of the recursive function `new`), is the following:

$$\begin{aligned}
\texttt{fib}(\texttt{0}) &\rightarrow \texttt{s}(\texttt{0}) \\
\texttt{fib}(\texttt{s}(\texttt{0})) &\rightarrow \texttt{s}(\texttt{0}) \\
\texttt{fib}(\texttt{s}(\texttt{s}(\texttt{X}))) &\rightarrow \texttt{Z}_1 + \texttt{Z}_2 \ \texttt{where}\ (\texttt{Z}_1, \texttt{Z}_2) = \texttt{new}(\texttt{X}) \\
\texttt{new}(\texttt{0}) &\rightarrow (\texttt{s}(\texttt{0}), \texttt{s}(\texttt{0})) \\
\texttt{new}(\texttt{s}(\texttt{X})) &\rightarrow (\texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1)\ \texttt{where}\ (\texttt{Z}_1, \texttt{Z}_2) = \texttt{new}(\texttt{X})
\end{aligned}$$

where the abstracted and folded rules (R_{13}, R_{15}) and (R_{10}, R_{12}) are expressed by using local declarations for readability.

The three last programs shown in Table 1 are also well-known examples of tupling optimizations. In particular, the `hanoi's towers` problem has special significance in this setting since it requires the generation of two eureka definitions in order to be solved. In [26] we have recently proposed in incremental version of tupling which is able to deal with this kind of problems for both pure functional and functional–logic programs.

6.3 The transformation system Synth

The basic rules presented so far [8] together with a fully automatic composition strategy [9] have been implemented by the prototype system Synth [1]. The Synth system is written in SICStus Prolog v3.6 and the complete implementation consists of about 680 clauses (2450 lines of code). The transformator 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. We have recently incorporated a graphical interface (written in Java) to the system that enhances the interaction with the user.

Language syntax follows mainly that of the language Curry, a modern multiparadigm declarative language based on needed narrowing which extends Haskell with features for logic and concurrent programming and which is intended to become a standard in the functional–logic community [20,22]. However, the subset of Curry programs accepted by the system are also Haskell programs, which confirms the adequacy of the tool to deal not only with integrated programs, but also with pure functional programs.

[8] The system performs automatic instantiation during unfolding. Moreover, since transformed programs respect both the Curry and Haskell syntax, and thanks to the correctness results exposed in this paper, they can be safely executed by narrowing and rewriting, respectively.

[9] Nowadays we are also including the incremental version of tupling described in [26], where several eureka definitions are produced in multiple transformation phases hence obtaining powerful optimizations by means of a simple (purely syntactic) and fully automatic method.

Table 1
Examples of transformation strategies

ORIGINAL PROGRAM	FINAL PROGRAM
lengthapp	**Composition**

`lengthapp(L1, L2) → len(app(L1, L2))`
`len([ ]) → 0`
`len([H|T]) → s(len(T))`
`app([ ], L) → L`
`app([H|T], L) → [H|app(T, L)]`
	`lengthapp(L1, L2) → new(L1, L2)`
	`new([ ], L) → len(L)`
	`new([H|T], L) → s(new(T, L))`

doubleflip	**Composition**

`doubleflip(T) → flip(flip(T))`
`flip(leaf(N)) → leaf(N)`
`flip(tree(L, N, R) → tree(flip(R), N, flip(L))`
	`doubleflip(T) → new(T)`
	`new(leaf(N)) → leaf(N)`
	`new(tree(L, N, R) → tree(new(L), N, new(R))`

factlist	**tupling**

`factlist(0) → [ ]`
`factlist(s(N)) → [fact(s(N))|factlist(N)]`
`fact(0) → s(0)`
`fact(s(N)) → s(N) * fact(N)`
	`factlist(0) → [ ]`
	`factlist(s(N)) → [U|V] where (U, V) = new(N)`
	`new(0) → (s(0), [ ])`
	`new(s(N)) → (s(s(N)) * U, U : V) where (U, V) = new(N)`

average	**tupling**

`average(L) → sum(L)/length(L)`
`sum([ ]) → 0`
`sum([H|T]) → H + sum(T)`
`length([ ]) → 0`
`length([H|T]) → s(length(T))`
	`average(L) → U/V where (U, V) = new(L)`
	`new([ ]) → (0, 0)`
	`new([H|T]) → (H + U, s(V)) where (U, V) = new(T)`

hanoi	**tupling**

`h(0, A, B, C) → [ ]`
`h(s(N), A, B, C) → app(h(N, A, C, B), [mov(A, B)|h(N, C, B, A)])`
	`h(0, A, B, C) → [ ]`
	`h(s(N), A, B, C) → app(U, [mov(A, B)|V])`
	`where (U, V) = new(N, A, C, B)`
	`new(0, A, B, C) → ([ ], [ ])`
	`new(s(N), A, B, C) → (app(U, [mov(A, B)|V]),`
	`app(W, [mov(B, C)|U]))`
	`where (U, V, W) = new2(N, A, C, B)`
	`new2(0, A, B, C) → ([ ], [ ], [ ])`
	`new2(s(N), A, B, C) → (app(U, [mov(A, B)|V]),`
	`app(W, [mov(B, C)|U]),`
	`app(V, [mov(C, A)|W]))`
	`where (U, V, W) = new2(N, A, C, B)`

7 Conclusions

We have defined a safe instantiation rule that combined with other transformation rules for unfolding, folding, abstracting and introducing new definitions, is able to optimize lazy functional programs. The main novelty of our approach is that it is inspired by the kind of instantiation that needed narrowing implicitly produces before reducing a given term. As a nice consequence, our transformation methodology inherits the best properties of needed narrowing and enhances previous pure FP approaches in several senses. For instance, it preserves the structure of transformed programs, it is able to use specific (as opposed to most general) unifiers and it produces redexes at different positions

of a given rule after being instantiated with different unifiers, thus minimizing not only the set of transformed rules but also the transformation effort.

Beyond instantiation, the set of rules presented so far has been implemented and proved correct/complete in FLP i.e., all them preserve the semantics of values (corresponding to its functional dimension) and computed answer substitutions (which is associated to its logic counterpart). Since the operational semantics of FLP is based on narrowing, and narrowing subsumes rewriting (i.e., the operational semantics of FP), this directly implies that the whole transformation system specially tailored for FP is also semantics preserving in this context, which guarantees its effective use in real tools. For the future, we plan to extend our transformation system in order to cope with lazy functional programs with different pattern matching semantics and sharing. Moreover, a new, natural and conceptually challenging extension that would greatly expand the applicability of the proposed transformation system is the treatment of non-deterministic functions, following the ideas of [18,7].

Acknowledgements.
Many thanks to María Alpuente, Moreno Falaschi and Germán Vidal for collaborative work on fold/unfold transformations in FLP and helpful discussions on its extensions. I am also grateful to the anonymous referees for their accurate and useful remarks and suggestions, which helped to improve this paper.

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, UPV, 1999. Available in URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[2] 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.

[3] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. An Automatic Composition Algorithm for Functional Logic Programs. In V. Hlaváč, K. G. Jeffery, and J. Wiedermann, editors, *Proc. of the 27th Annual Conference on Current Trends in Theory and Practice of Informatics, SOFSEM'2000*, pages 289–297. Springer LNCS 1963, 2000.

[4] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science*, page (56), 2003. Accepted for publication (to appear).

[5] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In P. Lee, editor, *Proc.*

of 1999 International Conference on Functional Programming, ICFP'99, Paris (France). ACM, New York, 1999.

[6] 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.

[7] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, pages 16–30. Springer LNCS 1298, 1997.

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

[9] S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722, pages 335–350, Tsukuba, Japan, 11 1999. Springer LNCS.

[10] Sergio Antoy. Evaluation strategies for functional logic programming. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier Science Publishers, 2001.

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

[12] R.S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21(1):239–250, 1984.

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

[14] 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.

[15] 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.

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

[17] J.C. González-Moreno. A correctness proof for Warren's HO into FO translation. In *Proc. GULP' 93*, pages 569–585, Gizzeria Lido, IT, Oct. 1993.

[18] J.C. González-Moreno, F.J. López-Fraguas, M.T. Hortalá-González, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

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

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

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

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

[23] 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.

[24] G. Moreno. A Safe Transformation System for Optimizing Functional Programs. Technical Report DIAB-02-07-27, UCLM, 2002. Available in URL: `http://www.info-ab.uclm.es/~personal/gmoreno/gmoreno.htm`.

[25] 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'2002*, pages 131–140. Springer LNAI 2527, 2002.

[26] G. Moreno. Incremental Tupling in a Functional-Logic Setting. In *Proc. of Segundas Jornadas sobre Lenguajes de Programación, PROLE'02, El Escorial (Spain)*, page 16. Universidad Complutense de Madrid, 2002.

[27] G. Moreno. Transformation Rules and Strategies for Functional-Logic Programs. *AI Communications, IO Press (Amsterdam)*, 15(2):3, 2002.

[28] 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.

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

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

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

[32] C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language: An approach to instantiation. In K. Davis and R. J. M. Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 133–141, London, UK, 1990. Springer-Verlag.

[33] 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.

[34] 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.

[35] 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.

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

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

Appendix

In order to prove Theorem 3.1, we need the following technical results from [5]. The first proposition shows that each substitution in a needed narrowing step instantiates only variables occurring in the initial term.

Proposition .1 [5] *If* $(p, R, \phi_k \circ \cdots \circ \phi_1) \in \lambda(t, \mathcal{P})$ *is a needed narrowing step, then, for* $i = 1, \ldots, k$, $\phi_i = id$ *or* $\phi_i = \{x \mapsto c(\overline{x_n})\}$ *(where* $\overline{x_n}$ *are pairwise different variables) with* $x \in \mathcal{V}ar(\phi_{i-1} \circ \cdots \circ \phi_1(t))$.

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

Lemma .2 [5] *Let* t *be a term rooted with a defined function symbol,* $\mathcal{P}$ *a definitional tree with* $pattern(\mathcal{P}) \leq t$ *and* $(p, R, \phi_k \circ \cdots \circ \phi_1), (p', R', \phi'_{k'} \circ \cdots \circ \phi'_1) \in \lambda(t, \mathcal{P})$, $k \leq k'$. *Then, for all* $i \in \{1, \ldots, k\}$,

- *either* $\phi_i \circ \cdots \circ \phi_1 = \phi'_i \circ \cdots \circ \phi'_1$, *or*
- *there exists some* $j < i$ *with*

(i) $\phi_j \circ \cdots \circ \phi_1 = \phi'_j \circ \cdots \circ \phi'_1$, *and*
(ii) $\phi_{j+1} = \{x \mapsto c(\cdots)\}$ *and* $\phi'_{j+1} = \{x \mapsto c'(\cdots)\}$ *with* $c \neq c'$.

Now we proceed with the proof of Theorem 3.1.

Proof. Let $\mathcal{R}'$ be the program obtained by application of the instantiation rule to the inductively sequential program $\mathcal{R}$. By Definition of our instantiation rule, program $\mathcal{R}'$ has been obtained from $\mathcal{R}$ by removing a rule $R = (l \to r) \in \mathcal{R}$ and adding a new set of rules whose lhs's are represented by the set $S_{inst} = \{\phi_1(l), \ldots, \phi_m(l)\}$, where $r \leadsto_{\phi_i} r_i$, $i = 1, \ldots, m$, are all the one-step needed narrowing derivations from r in $\mathcal{R}$. Assume that l is rooted with the defined function f, and f is defined in $\mathcal{R}$ by a set of rules $\{R_j = (l_j \to r_j) \mid j = 0, \ldots, n, n > 0\}$. Since $\mathcal{R}$ is inductively sequential, there exists a definitional tree $\mathcal{P}$ for f in $\mathcal{R}$. 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 $l \in \mathcal{P}$. To show the inductive sequentiality of $\mathcal{R}'$, it suffices to show that there exists a definitional tree $\mathcal{P}'$ for the set

$$S' = (S \backslash \{l\}) \cup S_{inst} \ .$$

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

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

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 (root patterns) are identical for both definitional trees, since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.

Leaves property: The maximal elements of $\mathcal{P}$ are S. Since all substitutions computed by needed narrowing along different derivations are independent by Lemma .2, the substitutions $\phi_1, \ldots, \phi_m$ are pairwise independent. Thus, the replacement of the element l in S by the set $\{\phi_1(l), \ldots, \phi_m(l)\}$ 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:

(i) $\pi \in \mathcal{P}$: Then the parent property trivially holds since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.

(ii) $\pi \notin \mathcal{P}$: By definition of $\mathcal{P}'$, $\pi = \phi_{ij} \circ \cdots \circ \phi_{i1}(l)$ 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 $\pi = \phi_{i1}(l)$. It is $\phi_{i1} \neq id$ (otherwise $\pi = l \in \mathcal{P}$). Thus, by Proposition .1, $\phi_{i1} = \{x \mapsto c(\overline{x_n})\}$ 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 .1), 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 $\pi' = \phi_{i,j-1} \circ \cdots \circ \phi_{i1}(l)$. Let $\phi_{ij} \neq id$ (otherwise the induction step is trivial). By Proposition .1, $\phi_{ij} = \{x \mapsto c(\overline{x_n})\}$ with $x \in \mathcal{V}ar(\phi_{i,j-1} \circ \cdots \circ \phi_{i1}(l))$ (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:

(i) $\pi \in \mathcal{P} \backslash \{l\}$: Then the induction property holds for π since it already holds in $\mathcal{P}$ and only instances of l are added in $\mathcal{P}'$.

(ii) $\pi = \phi_{ij} \circ \cdots \circ \phi_{i1}(l)$ for some $1 \leq i \leq m$ and $0 \leq j < k_i$. Assume $\phi_{i,j+1} \neq id$ (otherwise, do the identical proof with the representation $\pi = \phi_{i,j+1} \circ \cdots \circ \phi_{i1}(l)$). By Proposition .1, $\phi_{i,j+1} = \{x \mapsto c(\overline{x_n})\}$ and π has a single occurrence of the variable x (due to the linearity of the initial pattern and all substituted terms). Therefore, $\pi' = \phi_{i,j+1} \circ \cdots \circ \phi_{i1}(l)$ is a child of π. Consider another child $\pi'' = \phi_{i'j'} \circ \cdots \circ \phi_{i'1}(l)$ of π (other patterns in $\mathcal{P}'$ cannot be children of π due to the induction property for $\mathcal{P}$). Assume $\phi_{i'j'} \circ \cdots \circ \phi_{i'1} \neq \phi_{i,j+1} \circ \cdots \circ \phi_{i1}$ (otherwise, both children are identical). By Lemma .2, there exists some k with $\phi_{i'k} \circ \cdots \circ \phi_{i'1} = \phi_{il} \circ \cdots \circ \phi_{i1}$, $\phi_{i',k+1} = \{x' \mapsto c'(\cdots)\}$, and $\phi_{i,k+1} = \{x' \mapsto c''(\cdots)\}$ 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 $\phi_{i',j'} = \cdots = \phi_{i',k+2} = id$ (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. □