University of Castilla-La Mancha

A publication of the

Department of Computer Science

A Safe Transformation System for Optimizing Functional Programs

by

Ginés Moreno

Technical Report **#DIAB-02-07-27** September 2002

DEPARTAMENTO DE INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete - 02071 - Spain
Phone +34.967.599200, Fax +34.967.599224

A Safe Transformation System for Optimizing Functional Programs *

Ginés Moreno

Department of Computer Science, University of Castilla-La Mancha,
02071 Albacete, Spain. gmoreno@info-ab.uclm.es

Abstract. In this paper 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 a pure functional setting, we take advantage of previous experiences in the (more general) context on functional-logic programming, where instantiation is naturally embedded into an unfolding rule based on narrowing. Inspired in the so called needed narrowing strategy, our instantiation rule inherits its best properties and optimices previous approaches defined in the literature (that require more transformation effort) by anticipating bindings and generating redexes at different positions of a term in one go.

Keywords: 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). Our programming language is similar to the one presented in [8, 17] 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 integrated functional-logic programs that combine the operational methods and advantages of both functional and logic programming styles [4]. If the operational principle of functional languages is based on rewriting, integrated programs are usually executed by *narrowing*, a generalization of rewriting that instantiates variables in an expression and applies a reduction step to a redex of the instantiated expression. Needed narrowing extends the Huet and Lévy's notion of a needed

* This work has been partially supported by CICYT under grant TIC 2001-2705-C03-03, by Acción Integrada Hispano-Italiana HI2000-0161, and by the Valencian Research Council under grant GV01-424.

reduction [13], 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 number of computed solutions and it can be efficiently implemented by pattern matching and unification [5].

The fold/unfold transformation approach was first introduced in [8] to optimize functional programs and then used for logic programs [22] and functional-logic programs [1, 2]. 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. This fact contrast with logic and functional-logic transformation systems where the presence of this rule is avoided: 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 [21] where the role of instantiation is assumed to be played by certain distribution laws for case expressions (which are even more problematic than instantiation). 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.

Similarly to the instantiation process generated implicitly by most unfolding rules for logic ([22, 16]) and functional-logic programs ([1]), classical instantiation rules for pure functional programs ([8, 9, 17]) 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. In any case and as said in [21], unrestricted instantiation is problematic because it is not even locally equivalence preserving, since it can force premature evaluation of expressions (a problem addressed in some detail in [20]) 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[1] 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 (*call-by-name*) functional-logic programs 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

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

not only of values, but also of computed answer substitutions). Inspired in all these nice results, we show in this paper how to extrapolate the rules presented in [2] to a pure functional setting and, in particular, we built a safe instantiation rule that drops out 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 and its properties 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 funcional programs in Section 5. Section 6 shows some representative examples of program optimization and finally Section 7 concludes.

2 Functional and 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) term rewriting system and they only differs from its different operational semantics which are based on rewriting and narrowing, respectively. We assume familiarity with basic notions from term rewriting, functional and functional-logic programming [6, 12]. 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. A term is *linear* if it does not contain multiple occurrences of one variable. $t|_p$ 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.

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 left-hand side is a pattern. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position and a most-general unifier σ such that $\sigma(l|_p) = \sigma(l')$. A left-linear TRS with no overlapping rules is called *orthogonal*. In the remainder of this paper, functional programs are a subclass of left-linear CB orthogonal TRS's called inductively sequential TRS's. To provide a precise definition of this class of programs we introduce definitional trees [4]. 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 $\pi \in \mathcal{P}$, $\pi \neq \mathit{pattern}(\mathcal{P})$, 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 $\pi' < \pi'' < \pi$.

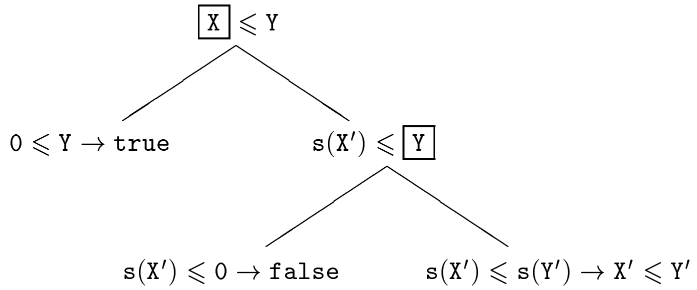
Induction property: Given $\pi \in \mathcal{P} \backslash S$, there is a position o of $\mathit{pattern}(\mathcal{P})$ with $\mathit{pattern}(\mathcal{P})|_o \in \mathcal{X}$ (the *inductive position*), and constructors $c_1, \ldots, c_n \in \mathcal{C}$ with $c_i \neq c_j$ for $i \neq j$, such that, 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 $1 \leq i \leq n$.

The definitional trees are similar to standard matching trees of functional programming[2]. However, differently from left-to-right matching trees used in either Hope, Miranda, or Haskell, definitional trees deal with *dependencies* between arguments of functional patterns. Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves contain all (and only) the rules used to define f and whose inner nodes contain information to guide the (optimal) pattern matching during the evaluation of expressions. Each inner node contains a *pattern* and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\overline{x_n})$, where $\overline{x_n}$ are different variables.

It is often convenient and simplifies the understanding to provide a graphic representation of definitional trees, where each node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules. For instance, the definitional tree for the function "$\leqslant$":

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

can be illustrated as follows:



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

[2] Definitional trees can also be encoded using *case expressions*, another well-known technique to implement pattern matching in functional languages [18].

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 functional programs is based on *rewriting*. 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 $s = t[\sigma(r)]_p$. The instantiated lhs $\sigma(l)$ is called a *redex*. 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-name*) 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, $s \leadsto_{p,R,\sigma} t$ is a *narrowing step* if p is a non-variable position in s and $\sigma(s) \to_{p,R} t$. Modern functional-logic languages are based on *needed narrowing* and *inductively sequential* programs (a detailed description can be found in [5]).

3 Instantiation

Following [8], the instantiation rule is used in pure functional transformation systems to introduce, or more appropriately, to replace substitutions 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 directly does not offer neither correctness results, nor consider instantiation explicitly [8, 21]. In this section we face this problem by defining a safe instantiation rule inspired in the needed narrowing based unfolding transformation for functional-logic programs 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*. Function λ is not only useful for defining needed narrowing, but also for being used in our instantiation rule. We consider two cases for $\mathcal{P}$[3]:

1. If π is a leaf, i.e., $\mathcal{P} = \{\pi\}$, and $\pi \to r$ is a variant of a rewrite rule, then
$$\lambda(t, \mathcal{P}) = \{(\Lambda, \pi \to r, id)\} \ .$$
2. If π is a branch, consider the inductive position o of π and some child $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. Let $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ be the definitional tree where

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

all patterns are instances of π_i. Then we consider the following cases for the subterm $t|_o$:

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

Informally speaking, needed narrowing applies a rule, if possible (case 1), or checks the subterm corresponding to the inductive 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 needed narrowing. 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).

3.1 The Instantiation Rule

Now, we re-use function λ for defining our instantiation rule. Basically, 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:

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

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`:

$$\begin{array}{rll} \mathtt{test(X,Y)} \to \mathtt{X} \ \leqslant\ \mathtt{double(Y)} & & (R_4) \\ \mathtt{double(0)} \to 0 & & (R_5) \\ \mathtt{double(s(X))} \to \mathtt{s(s(double(X)))} & & (R_6) \end{array}$$

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

$$\begin{array}{rll} \mathtt{test(0,Y)} \to 0 \ \leqslant\ \mathtt{double(Y)} & & (R_7) \\ \mathtt{test(s(X'),0)} \to \mathtt{s(X')} \ \leqslant\ \mathtt{double(0)} & & (R_8) \\ \mathtt{test(s(X'),s(Y'))} \to \mathtt{s(X')} \ \leqslant\ \mathtt{double(s(Y'))} & & (R_9) \end{array}$$

In contrast with the usual instantiation rule used in functional programming and similarly to needed narrowing, we observe two important properties enjoyed by our transformation rule:

1. It is able to instantiate more than an unique subterm in one go. In fact, subterms at positions Λ and 2 have been considered in the example, and 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).
2. It uses more specific unifiers than mgu's. This is represented by the extra binding $\mathtt{X} \mapsto \mathtt{s(X')}$ 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 $\mathtt{test(X, 0)}$ and $\mathtt{test(X, s(Y'))}$ that overlap with the lhs of rule R_7.

Experiences in functional-logic programming shows that the preservation of the program structure is the key point to prove the correctness of the transformation system based on needed narrowing presented in [2]. Since the instantiation rule is not explicitly used in [2], in the next section we formally establish this property specially formulated for this (purely functional) transformation in an isolated way.

3.2 Preservation of Program Structure

In order to prove this nice property of the instantiation rule, we need the following technical results from [3]. The first proposition shows that each substitution in a needed narrowing step instantiates only variables occurring in the initial term.

Proposition 1. [3] *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 (computing different substitutions) there is always a variable which is instantiated to different constructors:

Lemma 1. [3] *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*
  1. *$\phi_j \circ \cdots \circ \phi_1 = \phi'_j \circ \cdots \circ \phi'_1$, and*
  2. *$\phi_{j+1} = \{x \mapsto c(\cdots)\}$ and $\phi'_{j+1} = \{x \mapsto c'(\cdots)\}$ with $c \neq c'$.*

Now, we present the main technical result of this work.

Theorem 1. *The application of the instantiation rule to an inductively sequential program generates an inductively sequential program too.*

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 $r \rightsquigarrow_{\phi_i} r_i$, $i = 1, \ldots, m$, 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 $\{R_j = (l_j \to r_j) \mid j = 0, \ldots, n, n > 0\}$. By the induction hypothesis, 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 $r \rightsquigarrow_{\phi_i} r_i$ 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 are identical for both definitional trees, that is, $pattern(\mathcal{P}) = pattern(\mathcal{P}')$, since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.

Leaves property: The maximal elements of $\mathcal{P}$ are S. Since all substitutions computed by needed narrowing along different derivations are independent by Lemma 1, 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:

1. $\pi \in \mathcal{P}$: Then the parent property trivially holds since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.
2. $\pi \notin \mathcal{P}$: By definition of $\mathcal{P}'$, $\pi = \phi_{ij} \circ \cdots \circ \phi_{i1}(l)$ 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:

1. $\pi \in \mathcal{P} \backslash \{l\}$: Then the induction property holds for π since it already holds in $\mathcal{P}$ and only instances of l are added in $\mathcal{P}'$.
2. $\pi = \phi_{ij} \circ \cdots \circ \phi_{i1}(l)$ 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 1, 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.

In the next section we focus on the counterpart of the instantiation rule, that is, the unfolding transformation.

4 Unfolding

The unfolding of a rule in a functional program basically consists of replacing it by a new rule obtained by the application of a rewriting step to a redex in its rhs. This transformation is strongly related to the previous one, since instantiation is usually used in a pure functional program to generate redexes in the rhs of its 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: $\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{\sigma(l) \to \sigma(r)\} \cup \{\sigma(l) \to r' \mid \sigma(r) \to_{p,R} r'\}$.

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 than instantiation is naturally embedded by unfolding in a functional-logic setting.

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:

$$\begin{array}{rll} \mathtt{test(s(X'),0)} \rightarrow \mathtt{s(X')} \leqslant \mathtt{0} & & (R_{11}) \\ \mathtt{test(s(X'),s(Y'))} \rightarrow \mathtt{s(X')} \leqslant \mathtt{s(s(double(Y')))} & & (R_{12}) \end{array}$$

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

$$\begin{array}{rll} \mathtt{test(s(X'),0)} \rightarrow \mathtt{false} & & (R_{13}) \\ \mathtt{test(s(X'),s(Y'))} \rightarrow \mathtt{X'} \leqslant \mathtt{s(double(Y'))} & & (R_{14}) \end{array}$$

Let us see now the effects produced in our original program by using an unfolded rule preceded by the classical instantiation steps used in pure 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 lately 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:

$$\begin{array}{rll} \mathtt{test(X,0)} \rightarrow \mathtt{X} \leqslant \mathtt{double(0)} & & (R'_7) \\ \mathtt{test(X,s(Y'))} \rightarrow \mathtt{X} \leqslant \mathtt{double(s(Y'))} & & (R'_8) \end{array}$$

This process is called "minimal" instantiation in [9], 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:

$$\begin{array}{rll} \mathtt{test(X,0)} \rightarrow \mathtt{X} \leqslant \mathtt{0} & & (R'_9) \\ \mathtt{test(X,s(Y'))} \rightarrow \mathtt{X} \leqslant \mathtt{s(s(double(Y')))} & & (R'_{10}) \end{array}$$

Now, we can perform two instantiation steps, one for rule R'_9 ant other 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:

$$\begin{array}{rll} \mathtt{test(0,0)} \rightarrow \mathtt{0} \leqslant \mathtt{0} & & (R'_{11}) \\ \mathtt{test(0,s(Y'))} \rightarrow \mathtt{0} \leqslant \mathtt{s(s(double(Y')))} & & (R'_{12}) \end{array}$$

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

$$\begin{array}{rll} \mathtt{test(0,0)} \rightarrow \mathtt{true} & & (R'_{13}) \\ \mathtt{test(0,s(Y'))} \rightarrow \mathtt{true} & & (R'_{14}) \end{array}$$

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 in the sense that we need less instantiation and unfolding steps, apart from producing smaller (correct) programs. Our transformation rules inherit this nice properties from the optimality of needed narrowing w.r.t. the length of derivations and the number of computed solutions [5].

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 a functional setting the folding operation proceeds in a contrary direction to the usual reduction steps, that is, reduction is performed against a reversed ("folding") rule. Now we adapt the folding operation of [2] to our pure functional context. Basically, 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: $\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{l \to r\} \ \cup \ \{l \to r[\theta(l')]_p\}$.

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 logic and functional-logic programs, which have been defined in a similar "functional style" (see, e.g., [7, 17, 22, 2]), and can still produce at a very low cost many powerful optimizations. Many attempts have been also made to define a folding transformation in a (pure) functional context [8, 10, 14]. A *marked* folding for a lazy (higher-order) functional language has been presented in [21], which preserves the semantics of (ground constructor) values under applicability conditions which are similar to ours[4]. Some other works in the literature have advocated different styles of folding, for instance *disjunctive* and *reversible* folding [19, 11, 16, 1]. The first class of folding is able to simultaneously fold several rules in one go, while the second one (which usually is also disjunctive) 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 (as our rule does), which drastically reduces the power of the transformation.

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*

[4] In particular, both definitions ensure that the number of folding steps performed on a rule is never greater than the number of unfolding steps, which is crucial to ensure the correctness of the transformation.

$\mathcal{R}_0, \ldots, \mathcal{R}_k$, 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 $f(\overline{x}) \to r$, 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 in two senses:

1. by relaxing the (strong) condition that the "folding" rule belong to the last program in the sequence[5], and
2. by guarantying that "self folding" (i.e., the possibility to unsafely fold a rule by itself [16]) is disallowed.

Finally, the *non-erasing* condition, i.e. $\mathcal{V}ar(r) = \overline{x}$, required in the definition introduction rule 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 (forbidden) extra variables in the rhs of the resulting rule.

Let us illustrate our definitions. Assume a program containing the following rule, together with de definitions for "+":

$$\mathtt{twice(X,Y)} \to \mathtt{(X+Y)+(X+Y)} \qquad (R_1)$$

Now, we apply the definition introduction rule by adding the following eureka to the program:

$$\mathtt{new(Z)} \to \mathtt{Z+Z} \qquad (R_2)$$

Finally, if we fold R_1 using R_2, we obtain:

$$\mathtt{twice(X,Y)} \to \mathtt{new(X+Y)} \qquad (R_3)$$

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 in the one presented by [21]) of the following rule, which is expressed by means of a local declaration built with the `where` construct typical of functional languages:

$$\mathtt{twice(X,Y)} \to \mathtt{Z+Z\ where\ Z=X+Y}$$

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 [8, 21] (often known as *where–abstraction* rule [17]). The use of this transformation is mandatory in order to obtain the benefits of the powerful tupling strategy [8, 9, 15]. Moreover, as noted by [17], the use of the where–abstraction rule has the extra advantage that, in the

[5] A large number of proposals also allow that the folded and the folding rule to belong to different programs (see, e.g., [7, 16, 17, 22, 2]), which in general is crucial to achieve an effective optimization.

call-by-value mode of execution, the evaluation of the abstracted expression 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. In [2] we formalize and abstraction rule that allows the abstraction of different expressions in one go.

6 More examples

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

Example 1. The function $\mathtt{sum\_prefix(X,Y)}$ defined in the following program $\mathcal{R}_0$ returns the sum of the Y consecutive natural numbers, starting from 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 from:

1. Definition introduction:
$$\mathtt{aux(X,Y)} \to \mathtt{suml(from(X),Y)} \quad (R_7)$$
2. Now, since function λ applied to $\mathtt{suml(from(X),Y)}$ returns the tuples $\{(\lambda, R_2, \{\mathtt{Y} \mapsto \mathtt{0}\}), (1, R_4, id)\}$, we apply the instantiation rule as follows:
$$\begin{array}{rll} \mathtt{aux(X,0)} \to & \underline{\mathtt{suml(from(X),0)}} & (R_8) \\ \mathtt{aux(X,s(Y))} \to & \mathtt{suml(\underline{from(X)},s(Y))} & (R_9) \end{array}$$
3. By unfolding the underlined subtems on rules R_8 and R_9 using rules R_2 and R_4, we obtain:
$$\begin{array}{rll} \mathtt{aux(X,0)} \to & \mathtt{0} & (R_{10}) \\ \mathtt{aux(X,s(Y))} \to & \mathtt{suml([X|from(s(X))],s(Y))} & (R_{11}) \end{array}$$
4. Instantiation of rule R_{11} using the tuple (λ, R_3, id):
$$\mathtt{aux(X,s(Y))} \to \underline{\mathtt{suml([X|from(s(X))],s(Y))}} \quad (R_{12})$$

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

$$\mathtt{aux(X, s(Y))} \rightarrow \mathtt{X} + \underline{\mathtt{suml(from(s(X)), Y)}} \quad (R_{13})$$

6. Folding of the underlined subterm $\mathtt{suml(from(s(X)), Y)}$ (which is an instance of the lhs of rule R_7) in rule R_{10} using the eureka R_7:

$$\mathtt{aux(X, s(Y))} \rightarrow \mathtt{X} + \mathtt{aux(s(X), Y)} \quad (R_{14})$$

7. Finally, folding of the rhs of rule R_1 using R_7:

$$\mathtt{sum\_prefix(X, Y)} \rightarrow \mathtt{aux(X, Y)} \quad (R_{15})$$

Then, the transformed and enhanced final program is the following:

$$\begin{array}{rll} \mathtt{sum\_prefix(X, Y)} & \rightarrow \mathtt{aux(X, Y)} & (R_{15}) \\ \mathtt{aux(X, 0)} & \rightarrow \mathtt{0} & (R_{10}) \\ \mathtt{aux(X, s(Y))} & \rightarrow \mathtt{X} + \mathtt{aux(s(X), Y)} & (R_{14}) \end{array}$$

(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 even in the presence of nonterminating functions.

Example 2. 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:

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

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

$$\begin{array}{rll} \mathtt{new(0)} & \rightarrow (\underline{\mathtt{fib(s(0))}}, \mathtt{fib(0)})) & (R_5) \\ \mathtt{new(s(X'))} & \rightarrow (\underline{\mathtt{fib(s(s(X')))}}, \mathtt{fib(s(X'))}) & (R_6) \end{array}$$

3. Unfolding of rules R_5 and R_6 reducing the underlined subterms with rules R_2 and R_3.

$$\begin{array}{rll} \mathtt{new(0)} & \rightarrow \mathtt{(s(0), fib(0))} & (R_7) \\ \mathtt{new(s(X'))} & \rightarrow \mathtt{(fib(s(X'))} + \mathtt{fib(X'), fib(s(X')))} & (R_8) \end{array}$$

4. Instantiation of rule R_7 by using the tuple $(2, R_1, id)$)

$$\texttt{new(0)} \rightarrow (\texttt{s(0)}, \underline{\texttt{fib(0)}}) \quad (R_9)$$

5. Unfolding of rule R_9 with R_1:

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

6. Abstraction of R_8:
   - Definition introduction:

$$\texttt{new\_aux}((\texttt{Z}_1, \texttt{Z}_2)) \rightarrow (\texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1) \quad (R_{11})$$

   - Folding of R_8 w.r.t. rule R_{11}:

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

7. Folding of $(\texttt{fib(s(X}')), \texttt{fib(X}'))$ in rule R_{12} using R_4:

$$\texttt{new(s(X}')) \rightarrow \texttt{new\_aux(new(X}')) \quad (R_{13})$$

8. Abstraction of R_3:
   - Definition introduction:

$$\texttt{fib\_aux}((\texttt{Z}_1, \texttt{Z}_2)) \rightarrow \texttt{Z}_1 + \texttt{Z}_2 \quad (R_{14})$$

   - Folding of R_3 w.r.t. rule R_{14}:

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

9. Folding of $(\texttt{fib(s(X))}, \texttt{fib(X)})$ in rule R_{15} using again rule R_4:

$$\texttt{fib(s(s(X)))} \rightarrow \texttt{fib\_aux(new(X))} \quad (R_{16})$$

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

$$\begin{array}{rll}
\texttt{fib(0)} & \rightarrow \texttt{s(0)} & (R_1) \\
\texttt{fib(s(0))} & \rightarrow \texttt{s(0)} & (R_2) \\
\texttt{fib(s(s(X)))} & \rightarrow \texttt{Z}_1 + \texttt{Z}_2 \ \texttt{where} \ (\texttt{Z}_1, \texttt{Z}_2) = \texttt{new(X)} & (R_{14}, R_{16}) \\
\texttt{new(0)} & \rightarrow (\texttt{s(0)}, \texttt{s(0)}) & (R_{10}) \\
\texttt{new(s(X))} & \rightarrow (\texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1) \ \texttt{where} \ (\texttt{Z}_1, \texttt{Z}_2) = \texttt{new(X)} & (R_{11}, R_{13})
\end{array}$$

where rules (R_{14}, R_{16}) and (R_{11}, R_{13}) are expressed by using local declarations for readability.

7 Conclusions

We have defined an instantiation rule that combined with other transformation rules for unfolding, folding, abstracting and introducing new definitions, is able to optimize functional programs. The main novelty of our approach is that it is inspired in the kind of instantiation that needed narrowing implicitly produces before reducing a given term. As a nice consequence, our transformation inherits the best properties of needed narrowing and enhances previous approaches in several senses. For instance, it preserves the structure of transformed programs, it is able to use specific (non more general) unifiers and it produces redexes at different positions of a given rule, 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 in a functional-logic setting, 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). This directly implies that the whole transformation system specially tailored for functional programs is also semantics preserving in this context, which guarantees its effective use in real tools and its possible integration with other (correct) transformation rules defined in the literature (i.e., algebraic replacement, disjunctive folding, etc.).

Acknowledgements. My thanks to Germán Vidal, María Alpuente and Moreno Falaschi for collaborative work on folding/unfolding of functional-logic programs and helpful discussions on its extensions.

References

1. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.
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. 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.
4. 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.
5. 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.
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. *Journal of Logic Programming*, 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, 1993*, pages 119–132. ACM, New York, 1993.
10. B. Courcelle. Recursive Applicative Program Schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 459–492. Elsevier, Amsterdam, 1990.
11. 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.
12. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. 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.
14. L. Kott. Unfold/fold program transformation. In M.Nivat and J.C. Reynolds, editors, *Algebraic methods in semantics*, chapter 12, pages 411–434. Cambridge University Press, 1985.
15. G. Moreno. Automatic Optimization of Multi-Paradigm Declarative Programs. In *Proc. of 8th Ibero-American Conference on Artificial Intelligence, IBERAMIA'02, Sevilla (Spain).* To appear in Springer LNCS, 2002.
16. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
17. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
18. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
19. 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 the International Conference on Principles and Practice of Declarative Programming, PPDP'99, Paris, France*, pages 141–157. Springer LNCS 1702, 1999.
20. 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.
21. 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.
22. 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.