Automatic Optimization of Multi-Paradigm Declarative Programs [⋆]

Ginés Moreno

Dep. Informática, UCLM, 02071 Albacete, Spain. gmoreno@info-ab.uclm.es

Abstract. This paper investigates the optimization by fold/unfold of functional-logic programs with operational semantics based on needed narrowing. Transformation sequences are automatically guided by tupling, a powerful strategy that avoids multiple accesses to data structures and redundant sub-computations. We systematically decompose in detail the internal structure of tupling in three low-level transformation phases (definition introduction, unfolding and abstraction with folding) that constitute the core of our automatic tupling algorithm. The resulting strategy is (strongly) correct and complete, efficient, elegant and realistic. In addition (and most important), our technique preserves the natural structure of multi-paradigm declarative programs, which contrasts with prior pure functional approaches that produce corrupt integrated programs with (forbidden) overlapping rules.

1 Introduction

Functional logic programming languages combine the operational methods and advantages of the most important declarative programming paradigms, namely functional and logic programming. The operational principle of such languages is usually based on *narrowing*. A *narrowing step* instantiates variables in an expression and applies a reduction step to a redex of the instantiated expression. Needed narrowing is the currently best narrowing strategy for first-order (inductively sequential) functional logic programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions [6], and it can be efficiently implemented by pattern matching and unification.

The fold/unfold transformation approach was first introduced in [10] to optimize functional programs and then used for logic programs [21]. 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. The first attempt to introduce these ideas in an integrated language

[⋆] This work has been partially supported by CICYT under grant TIC 2001-2705-C03-03 and by Acción Integrada Hispano-Italiana HI2000-0161.

is presented in [3], where we investigated fold/unfold rules in the context of a strict (*call-by-value*) functional logic language. A transformation methodology for lazy (*call-by-name*) functional logic programs was introduced in [4]; this work extends the transformation rules of [21] for logic programs in order to cope with lazy functional logic programs (based on needed narrowing). The use of narrowing empowers the fold/unfold system by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework [10] which introduces an instance of an existing equation) into unfolding by means of unification. [4] also proves that the original structure of programs is preserved through the transformation sequence, which is a key point for proving the correctness and the effective applicability of the transformation system. These ideas have been implemented in the prototype SYNTH ([2]) which has been successfully tested with several applications in the field of Artificial Intelligence ([1, 17]).

There exists a large class of program optimizations which can be achieved by fold/unfold transformations and are not possible by using a fully automatic method (such as, e.g., partial evaluation). Typical instances of this class are the strategies that perform *tupling* (also known as *pairing*) [10, 13], which merges separate (nonnested) function calls with some common arguments (i.e., they share the same variables) into a single call to a (possibly new) recursive function which returns a tuple of the results of the separate calls, thus avoiding either multiple accesses to the same data structures or common subcomputations, similarly to the idea of *sharing* which is used in graph rewriting to improve the efficiency of computations in time and space [7]. In this paper, we propose a fully automatic tupling algorithm where eureka generation is done simultaneously at transformation time at a very low cost. In contrast with prior non-automatic approaches (tupling has only been semi-automated to some extent [11, 12]) where eurekas are generated by a complicate pre-process that uses complex data structures and/or produces redundant computations, our approach is fully automatic and covers most practical cases. Our method deals with particular (non trivial) features of integrated (functional-logic) languages and includes refined tests for termination of each transformation phase. More exactly, we have identified three syntactic conditions to stop the search for regularities during the unfolding phase[1].

The structure of the paper is as follows. After recalling some basic definitions, we introduce the basic transformation rules and illustrate its use by means of interesting tupling examples in Section 2. The next three sections describe the different transformation phases that constitute the core of our tupling algorithm. Finally, Section 6 concludes. More details can be found in [16].

Preliminaries We assume familiarity with basic notions from term rewriting [14] and functional logic programming [15]. 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

[1] The method is not universal but, as said in [19], "one cannot hope to construct a universal technique for finding a suitable regularity whenever there is one".

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. A term is *operation-rooted* (*constructor-rooted*) if it has an operation (constructor) symbol at the root. A *position* p in a term t is represented by a sequence of natural numbers. Positions are ordered by the *prefix* ordering: $p \leq q$, if $\exists w$ such that $p.w = q$. Positions p, q are *disjoint* if neither $p \leq q$ nor $q \leq p$. Given a term t, we let $\mathcal{FP}os(t)$ denote the set of non-variable positions of t. $t|_p$ 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. For a sequence of (pairwise disjoint) positions $P = \overline{p_n}$, we let $t[\overline{s_n}]_P = (((t[s_1]_{p_1})[s_2]_{p_2}) \ldots [s_n]_{p_n})$. By abuse, we denote $t[\overline{s_n}]_P$ by $t[s]_P$ when $s_1 = \ldots = s_n = s$, as well as $((t[s_1]_{P_1}) \ldots [s_n]_{P_n})$ by $t[\overline{s_n}]_{\overline{P_n}}$. 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. In the remainder of this paper, a functional logic *program* is a left-linear CB-TRS. 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 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 $\sigma(s) \to_{p,R} t$. We denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). Modern functional logic languages are based on *needed narrowing* and *inductively sequential* programs.

2 Tupling by Fold/Unfold

Originally introduced in [10, 13] for optimizing functional programs, the tupling strategy is very effective when several functions require the computation of the same subexpression. In this case, it is possible to tuple together those functions and to avoid either multiple accesses to data structures or common subcomputations [20]. Firstly, we recall from [4] the basic definitions of the transformation rules. Programs constructed by using the following set of rules are inductively sequential. Moreover, the transformations are strongly correct w.r.t. goals containing (*old*) function symbols from the initial program.

Definition 1. *Let $\mathcal{R}_0$ be an inductively sequential program (the original program). A* transformation sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k > 0$ *is constructed by applying the following transformation rules:*

Definition Introduction: *We may get program $\mathcal{R}_{k+1}$ by adding to $\mathcal{R}_k$ a new rule ("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 $\mathcal{V}ar(r) = \overline{x}$.*

Unfolding: *Let $R = (l \to r) \in \mathcal{R}_k$ be a rule where r is an operation-rooted term or $\mathcal{R}_k$ is completely defined. Then, $\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{\theta(l) \to r' \mid r \leadsto_\theta r'$ in $\mathcal{R}_k\}$. An unfolding step where $\theta = id$ is called a normalizing step.*

Folding: *Let $R = (l \to r) \in \mathcal{R}_k$ be a non definition rule, $R' = (l' \to r') \in \mathcal{R}_j$, $0 \leq j \leq k$, a definition rule*[2] *and p a position in r such that $r|_p = \theta(r')$ and $r|_p$ is not a constructor term. Then, $\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{l \to r[\theta(l')]_p\}$.*

Abstraction: *Let $R = (l \to r) \in \mathcal{R}_k$ be a rule and let $\overline{P_j}$ be sequences of disjoint positions in $\mathcal{FP}os(r)$ such that $r|_p = e_i$ for all p in P_i, $i = 1, \ldots, j$, i.e., $r = r[\overline{e_j}]_{\overline{P_j}}$. We may get program $\mathcal{R}_{k+1}$ from $\mathcal{R}_k$ by replacing R with $l \to r[\overline{z_j}]_{\overline{P_j}}$* where *$\langle z_1, \ldots, z_j \rangle = \langle e_1, \ldots, e_j \rangle$ (where $\overline{z_j}$ are fresh variables).*

The following well-known example uses the previous set of transformation rules for optimizing a program following a tupling strategy. The process is similar to [10, 11, 20] for pure functional logic programs, with the advantage in our case that we avoid the use of an explicit *instantiation rule* before applying unfolding steps. This is possible thanks to the systematic instantiation of calls performed implicitly by our unfolding rule by virtue of the logic component of the needed narrowing mechanism [4].

Example 1. The fibonacci numbers can be computed by the original program $\mathcal{R}_0 = \{R_1 : \texttt{fib(0)} \to \texttt{s(0)},\ R_2 : \texttt{fib(s(0))} \to \texttt{s(0)},\ R_3 : \texttt{fib(s(s(X)))} \to \texttt{fib(s(X))} + \texttt{fib(X)}\}$ (together with the rules for addition $+$). Note that this program has an exponential complexity, which can be reduced to linear by applying the tupling strategy as follows:

1. Definition introduction: (R_4) $\texttt{new(X)} \to \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$
2. Unfold rule R_4 (narrowing the needed redex $\texttt{fib(s(X))}$):
   (R_5) $\texttt{new(0)} \to \langle \texttt{s(0)}, \texttt{fib(0)} \rangle$, (R_6) $\texttt{new(s(X))} \to \langle \texttt{fib(s(X))} + \texttt{fib(X)}, \texttt{fib(s(X))} \rangle$
3. Unfold (normalize) rule R_5: (R_7) $\texttt{new(0)} \to \langle \texttt{s(0)}, \texttt{s(0)} \rangle$
4. Abstract R_6: (R_8) $\texttt{new(s(X))} \to \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle$ `where` $\langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$
5. Fold R_8 using R_4: (R_9) $\texttt{new(s(X))} \to \langle \texttt{Z}_1 + \texttt{Z}_2, \texttt{Z}_1 \rangle$ `where` $\langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(X)}$
6. Abstract R_3: (R_{10}) $\texttt{fib(s(s(X)))} \to \texttt{Z}_1 + \texttt{Z}_2$ `where` $\langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \langle \texttt{fib(s(X))}, \texttt{fib(X)} \rangle$
7. Fold R_{10} with R_4: (R_{11}) $\texttt{fib(s(s(X)))} \to \texttt{Z}_1 + \texttt{Z}_2$ `where` $\langle \texttt{Z}_1, \texttt{Z}_2 \rangle = \texttt{new(X)}$

Now, the (enhanced) transformed program $\mathcal{R}_7$ (with linear complexity thanks to the use of the recursive function `new`), is composed by rules R_1, R_2, R_7, R_9 and R_{11}.

The classical instantiation rule used in pure functional transformation systems is problematic since it uses most general unifiers of expressions (it is commonly called "minimal instantiation" [11]) and, for that reason, it is rarely considered explicitly in the literature. Moreover, in a functional-logic setting the use of an unfolding rule that (implicitly) performs minimal instantiation may generate corrupt programs that could not be executed by needed narrowing. The reader must note the importance of this fact, since it directly implies that tupling algorithms (that performs minimal instantiation) developed for pure functional programs are not applicable in our framework, as illustrates the following example.

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

Example 2. Consider a functional-logic program containing the following well-known set of non overlapping rules: $\{\ldots,\ \texttt{double(0)} \to \texttt{0},\quad \texttt{double(s(X))} \to \texttt{s(s(double(X)))},$ $\texttt{leq(0,X)} \to \texttt{true},\quad \texttt{leq(s(X),0)} \to \texttt{false},\quad \texttt{leq(s(X),s(Y))} \to \texttt{leq(X,Y)}, \ldots\}$. Assume now that a tupling strategy is started and after some definition introduction and unfolding steps we obtain the following rule: $\texttt{new}(\ldots,\texttt{X},\texttt{Y},\ldots) \to \langle \ldots, \underline{\texttt{leq(X,double(Y))}}, \ldots\rangle$. Then, if we apply an unfolding step (over the underlined term) with implicitly performs minimal instantiation before (lazily) reducing it, we obtain:

$$\begin{aligned}
\texttt{new}(\ldots,\texttt{0},\texttt{Y},\ldots) &\to \langle \ldots, \texttt{true}, \ldots\rangle\\
\texttt{new}(\ldots,\texttt{X},\texttt{0},\ldots) &\to \langle \ldots, \texttt{leq(X,0)}, \ldots\rangle\\
\texttt{new}(\ldots,\texttt{X},\texttt{s(Y)},\ldots) &\to \langle \ldots, \texttt{leq(X,s(s(double(Y))))}, \ldots\rangle
\end{aligned}$$

And now observe that there exist overlapping rules that loose the program structure. This loss prevents for further computations with needed narrowing. Fortunately (as proved in [4]), our unfolding rule always generates a valid set of program rules by using appropriate (non most general) unifiers before reducing a term. In the example, it suffices with replacing the occurrences of variable `X` by term `s(X)` in each rule (as our needed narrowing based unfolding rule does), which is the key point to restore the required structure of the transformed program.

Since tupling has not been automated in general in the specialized literature, our proposal consists of decomposing it in three stages and try to automate each one of them in order to generate a fully automatic tupling algorithm. Each stage may consists of several steps done with the transformation rules presented in section 2. We focus our attention separately in the following transformation phases: definition introduction, unfolding and abstraction+folding.

3 Definition Introduction Phase

This phase[3] corresponds to the so-called eureka generation phase, which is the key point for a transformation strategy to proceed. The problem of achieving an appropriate set of eureka definitions is well-known in the literature related to fold/unfold transformations [10, 19, 21, 5]. For the case of the composition strategy, eureka definitions can be easily identified since they correspond to nested calls. On the other hand, the problem of finding good eureka definitions for the tupling strategy is much more difficult, mainly due to the fact that the calls to be tupled are not nested and they may be arbitrarily distributed in the right hand side of a rule. Sophisticated static analysis have been developed in the literature using dependencies graphs ([11, 18]), m-dags ([8]), symbolic trace trees [9] and other intrincated structures. The main problems appearing in such approaches are that the analysis are not as general as wanted (they can fail even although the program admits tupling optimizations), they are time and space consuming and/or they may duplicate some work too[4]. In order to avoid these risks, our approach generates eureka definition following a very simple strategy (Table 1)

[3] Sometimes called "tupling" [19], but we reserve this word for the whole algorithm.
[4] This fact is observed during the so-called "program extraction phase" in [19]. This kind of post-processing can be made directly (which requires to store in memory

that obtains similar levels of generality than previous approaches and covers most practical cases. The main advantages are that the analysis is terminating, easy and quickly, and does not perform redundant calculus (like unfolding, instantiations, etc.) that properly corresponds to subsequent phases.

Table 1. Definition_Introduction Phase

`INPUT:`	`Original Program` $\mathcal{R}$ `and Program Rule` $R = (l \to r) \in \mathcal{R}$
`OUTPUT:`	`Definition Rule (Eureka)` R_{def}
`BODY:`	1. `Let` $T = \langle t_1, \ldots, t_n \rangle$ $(n > 1)$ `be a tuple where` $\{t_1, \ldots, t_n\}$ `is the set of operation-rooted subterms of` r `that are` *innermost* `(i.e.,` t_i `does not contain operation-rooted subterms) such that each one of them shares at least a common variable with at least one more subterm in` T 2. `Apply the DEFINITION INTRODUCTION RULE to generate :` $R_{def} = (f_{new}(\overline{x}) \to T)$ `where` f_{new} `is a new function symbol not appearing` $\mathcal{R}$`, and` $\overline{x}$ `is the set of variables of` T

As illustrated by step 1 in Example 1, our eureka generator proceeds as the algorithm in Table 1 shows. Observe that the input of the algorithm is the original program $\mathcal{R}$ and a selected rule $R \in \mathcal{R}$ which definition is intended to be optimized by tupling. In the worst case, every rule in the program could be used as input, but only those that generate appropriate eureka definitions should be considered afterwards in the global tupling algorithm.

One final remark: it is not clear in general neither the number nor the occurrences of calls to be tupled, but some intuitions exist. Similarly to most classical approaches in the literature, we require that only terms sharing common variables be tupled in the rhs of the eureka definition [11, 19]. On the other hand, since it is not usual that terms to be tupled contain operation rooted terms as parameters, we cope with this fact in our definition by requiring that only operation-rooted subterms at innermost positions of r be collected. On the contrary, the considered subterms would contain nested calls which should be more appropriately transformed by composition instead of tupling (see [5] for details).

4 Unfolding Phase

During this phase, that corresponds to the so-called *symbolic computation* in many approaches (see a representative list in [19]), the eureka definition R_{def} =

elaborated data structures, as is the case of symbolic computation trees) or via transformation rules similarly to our method, but with the disadvantage in that case that many folding/unfolding steps done at "eureka generation time" must be redundantly redone afterwards at "transformation time" or during the program extraction phase.

$(f_{new}(\overline{x}) \to T)$ generated in the previous phase, is unfolded possibly several times (at least once) using the original program $\mathcal{R}$, and returning a new program $\mathcal{R}_{unf}$ which represents the unfolded definitions of f_{new} . Since the rhs of the rule to be transformed is a tuple of terms (T), the subterm to be unfolded can be decided in a "don't care" non-deterministic way. In our case, we follow a criterium that is rather usual in the literature ([10, 11]), and we give priority to such subterms where recursion parameters are less general than others, as occurs in step 2 of Example 1 (where we unfold the term `fib(s(X))` instead of `fib(X)`). Moreover, we impose a new condition: each unfolding step must be followed by normalizing steps as much as possible, as illustrates step 3 in Example 1.

Table 2. Unfolding Phase

```
INPUT:    Original Program $\mathcal{R}$ and Definition Rule (Eureka) $R_{def}$
OUTPUT:   Unfolded Program $\mathcal{R}_{unf}$

BODY:     1. Let $\mathcal{R}_{unf} = \{R_{def}\}$ be a program
          2. Repeat    $\mathcal{R}_{unf}$=UNFOLD($\mathcal{R}_{unf}, \mathcal{R}$)
             until every rule $R' \in \mathcal{R}_{unf}$ verifies TEST($R', R_{def}$)>0
```

As shown in Table 2, the unfolding phase basically consists of a repeated application of the unfolding rule defined in Section 2. This is done by calling function `UNFOLD` with the set of rules that must be eventually unfolded. In each iteration, once a rule in $\mathcal{R}_{unf}$ is unfolded, it is removed and replaced with the rules obtained after unfolding it in the resulting program $\mathcal{R}_{unf}$, which is dynamically actualized in our algorithm. Initially, program $\mathcal{R}_{unf}$ only contains the definition rule $R_{def} = (f_{new}(\overline{x}) \to T)$. We observe that, once the process has been started, any rule obtained by application of the unfolding rule has the form $R' = (\theta(f_{new}(\overline{x})) \to T')$. In order to stop the process, we must check if each rule $R' \in \mathcal{R}_{unf}$ verifies one of the following conditions:

- **Stopping condition 1:** If there are no subterms in T' sharing common variables[5] or they can not be narrowed anymore, then rule R' represents a case base definition for the new symbol f_{new}. This fact is illustrated by step 3 and rule R_7 in Example 1.
- **Stopping condition 2:** There exists a substitution θ and a tuple T'' that packs the set of different innermost operation-rooted subterms that share common variables in T' (without counting repetitions), such that $\theta(T) = T''$[6]. Observe that rule R_6 verifies this condition in Example 1 since, for

[5] This novel stopping condition produces good terminating results during the search for regularities since it is more relaxed than other ones considered in the literature.

[6] This condition is related to the so-called *similarity*, *regularity* or *foldability* conditions in other approaches ([19]) since it suffices to enable subsequent abstraction+folding steps which may lead to efficient recursive definitions of f_{new}.

$\theta = id$ there exists (in its rhs) two an one instances respectively of the terms ocurring in the original tuple, that is, $\theta(T) = T = T'' = \langle \mathtt{f(s(X))}, \mathtt{f(X)} \rangle$.

In algorithm of Table 2 these terminating conditions are checked by function `TEST`, which obviously requires the rules whose rhs's are the intended tuples T and T'. Codes 0, 1 and 2 are returned by `TEST` when none, the first or the second stopping condition hold, respectively. We assume that, when `TEST` returns code 0 forever, the `repeat` loop is eventually aborted and then, the unfolding process returns an empty program (that will abort the whole tupling process too).

5 Abstraction+Folding Phase

This phase follows the algorithm shown in Table 3 and is used not only for obtaining efficient recursive definitions of the new symbol f_{new} (initially defined by the eureka $\mathcal{R}_{def}$), but also for redefine old function symbols in terms of the optimized definition of f_{new}. This fact depends on the rule R to be abstracted and folded, which may belong to the unfolded program obtained in the previous phase ($\mathcal{R}_{unf}$), or to the original program ($\mathcal{R}$), respectively. In both cases, the algorithm acts in the same way returning the abstracted and folded program $\mathcal{R}'$.

Table 3. Abstraction+Folding Phase

```
INPUT:   Program $\mathcal{R}_{aux} = \mathcal{R}_{unf} \cup \mathcal{R}$ and Definition Rule (Eureka) $R_{def}$
OUTPUT:  Abstracted+Folded Program $\mathcal{R}'$

BODY:     1. Let $\mathcal{R}' = \mathcal{R}_{aux}$ be a program and let $R'$ be an empty rule
          2. For every rule $R \in \mathcal{R}_{aux}$ verifying TEST($R, R_{def}$)=2
                  $R'$=ABSTRACT($R, R_{def}$)
                  $R'$=FOLD($R', R_{def}$)
                  $\mathcal{R}'$=$\mathcal{R}' - \{R\} \cup \{R'\}$
```

Firstly we consider the case $R \in \mathcal{R}_{unf}$. Remember that $R_{def} = (f_{new}(\overline{x}) \to T)$ where $T = \langle t_1, \ldots, t_n \rangle$. If $R = (\sigma(f_{new}(\overline{x})) \to r')$ satisfies `TEST`(R, R_{def})=2, then there exists sequences of disjoint positions[7] $\overline{P_j}$ in $\mathcal{FP}os(r')$ where $r'|_p = \theta(t_j)$ for all p in P_j, $j = 1, \ldots, n$, i.e., $r' = r'[\theta(\overline{t_j})]_{\overline{P_j}}$. Hence, it is possible to apply the abstraction rule described in Section 2 to R. This step is done by function `ABSTRACT`(R, R_{def}), that abstracts R accordingly to tuple T in rule R_{def} and generates the new rule R': $\sigma(f_{new}(\overline{x})) \to r'[\overline{z_j}]_{\overline{P_j}}$ *where* $\langle z_1, \ldots, z_n \rangle = \theta(\langle t_1, \ldots, t_n \rangle)$, being $\overline{z_j}$ are fresh variables. This fact is illustrated by step 4 and rule R_8 in Example 1.

Observe now that this last rule can be folded by using the eureka definition R_{def}, since all the applicability conditions required by our folding rule

[7] This positions obviously correspond to the set of innermost operation-rooted subterms that share common variables in r'.

(see Definition 1) are fulfilled. This fact is done by the call to `FOLD`(R', R_{def}) which returns the new rule: $\sigma(f_{new}(\overline{x})) \;\; \rightarrow \;\; r'[\overline{z_j}]_{\overline{P_j}} \;\; where \; \langle z_1, \ldots, z_n \rangle = \theta(f_{new}(\overline{x}))$. Note that this rule (illustrated by rule R_9 and step 5 in Example 1), as any other rule generated (and accumulated in the resulting program $\mathcal{R}'$) in this phase, corresponds to a recursive definition of f_{new}, as desired.

The case when $R \in \mathcal{R}$ is perfectly analogous. The goal now is to reuse as much as possible the optimized definition of f_{new} into the original program $\mathcal{R}$, as illustrate steps 6 and 7, and rules R_{10} and R_{11} in Example 1.

6 Conclusions and Further Research

Tupling is a powerful optimization strategy which can be achieved by fold/unfold transformations and produces better gains in efficiency than other simpler-automatic transformations. As it is well-known in the literature, tupling is very complicated and automatic tupling algorithms either result in high runtime cost (which prevents them from being employed in a real system), or they succeed only for a restricted class of programs [11, 12]. Our approach drops out some of these limitations by automating a realistic and practicable tupling algorithm that optimizes functional-logic programs. Compared with prior approaches in the field of pure functional programming, our method is less involved (we do not require complicate structures for generating eureka definitions), more efficient (i.e., redundant computations are avoided), and deals with special particularities of the integrated paradigm (i.e., transformed rules are non overlapping).

For the future, we are interested in to estimate the gains in efficiency produced at transformation time. In this sense, we want to associate a cost/gain label to each local transformation step when building a transformation sequence. We think that this action will allow to drive more accurately the transformation process, since it will help to define deterministic heuristics and automatic strategies.

Acknowledgements. My thanks to María Alpuente and Moreno Falaschi for helpful discussions on transformation strategies. I am specially grateful to German Vidal for critical comments that helped to improve this paper.

References

1. M. Alpuente, M. Falaschi, C. Ferri, G. Moreno, and G. Vidal. Un sistema de transformación para programas multiparadigma. *Revista Iberoamericana de Inteligencia Artificial*, X/99(8):27–35, 1999.
2. M. Alpuente, M. Falaschi, C. Ferri, G. Moreno, G. Vidal, and I. Ziliotto. The Transformation System SYNTH. Technical Report DSIC-II/16/99, UPV, 1999. Available in URL: `http://www.dsic.upv.es/users/elp/papers.html`.
3. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.

4. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, pages 147–162. Springer LNCS 1722, 1999.
5. M. Alpuente, M. Falaschi, 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.
6. 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.
7. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
8. R.S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–418, 1980.
9. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.
10. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
11. 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.
12. W. Chin, A. Goh, and S. Khoo. Effective Optimisation of Multiple Traversals in Lazy Languages. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA (Technical Report BRICS-NS-99-1)*, pages 119–130. University of Aarhus, DK, 1999.
13. 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.
14. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.
15. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
16. G. Moreno. Automatic Tupling for Functional–Logic Programs. Technical Report DIAB-02-07-24, UCLM, 2002. Available in URL: `http://www.info-ab.uclm.es/personal/gmoreno/gmoreno.htm`.
17. G. Moreno. Transformation Rules and Strategies for Functional-Logic Programs. *AI Communications, IO Press (Amsterdam)*, 15(2):3, 2002.
18. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
19. A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 355–385. Springer LNCS 1110, 1996.
20. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
21. 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.