An Automatic Composition Algorithm for Functional Logic Programs

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

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

Abstract. Functional logic languages with a complete operational semantics are based on narrowing, which combines the instantiation of variables with the reduction of expressions. In this paper, we investigate the relationship between partial evaluation and more general transformations based on folding/unfolding. First, we show that the transformations obtained by partial evaluators can be also achieved by folding/unfolding using a particular kind of eurekas which can be mechanically attained. Then, we propose an algorithm (based on folding/unfolding) which starts with the automatic eureka generation and is able to perform program composition, i.e., it is able to produce a single function definition for some nested functions of the original program. This avoids the construction of intermediate data structures that are produced by the inner function and consumed as inputs by the outer function. As opposed to both partial evaluation and (general) fold/unfold transformations, strong correctness of the transformed programs holds w.r.t. goals which contain calls to the *old* function symbols –i.e., from the original program– as well as to the *new* ones –i.e., introduced during the transformation.

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 [5], and it can be efficiently implemented by pattern matching and unification.

The aim of *partial evaluation* (PE) is to specialize a given program w.r.t. part of its input data (hence also called *program specialization*). PE has been widely applied in the field of functional programming [8, 13] and logic programming [11, 14]. Although the objectives are similar, the general methods are often different due to the distinct underlying computation models. Narrowing-driven PE [3] is the first generic algorithm for the specialization of functional logic languages.

The method is parametric w.r.t. the narrowing strategy which is used for the automatic construction of partial trees. The method is formalized within the theoretical framework established in [14] for the PE of logic programs (also known as *partial deduction*), although a number of concepts have been generalized to deal with the functional component of the language (e.g., nested function calls, different evaluation strategies, etc.). This approach has better opportunities for optimization thanks to the functional dimension (e.g., by the inclusion of deterministic evaluation steps). Also, since unification is embedded into narrowing, it is able to automatically propagate syntactic information on the partial input (term structure) and not only constant values, similarly to partial deduction.

The fold/unfold transformation approach was first introduced in [6] to optimize functional programs and then used for logic programs [17]. 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. A transformation methodology for lazy (*call-by-name*) functional logic programs was introduced in [2]; this work extends the transformation rules of [17] 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 [6] which introduces an instance of an existing equation) into unfolding by means of unification. [2] also demonstrates that the (inductively sequential) structure of programs is preserved through the transformation sequence $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, which is a key point for proving the correctness of the transformation system as well as for being effectively applicable.

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). However, a weak point of the fold/unfold transformation approach is the achievement of the appropriate set of (*eureka*) definitions which make it possible the optimizations to be pursued [6, 17, 15].

In this paper, we propose an automatic composition algorithm (based on folding/unfolding) which uses a pre-process based on partial evaluation to ascertain the eurekas which guarantee that the optimization will be attained. On the theoretical side, we first prove in Sect. 3 that fold/unfold transformations are able to achieve the effects of PE. A new result w.r.t. both partial evaluation and (general) fold/unfold transformations is that strong correctness of the transformed programs holds w.r.t. goals which contain calls to the *old* function symbols –i.e., from the original program– as well as to the *new* ones –i.e., introduced during the transformation. On the practical side, we define in Sect. 4 an automatic composition algorithm which uses partial evaluation as a (sort of) static analysis phase which delivers the intended eurekas.

2 Preliminaries

We assume familiarity with basic notions from term rewriting [10] and functional logic programming [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* (e.g., x, y, z) 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. A term is *operation-rooted* (*constructor-rooted*) if it has an operation (constructor) symbol at the root. A *position* p in a term t is represented by a sequence of natural numbers. (Λ denotes the empty sequence, i.e., the root position). $t|_p$ 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. 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 $s = t[\sigma(r)]_p$. Given a relation $\to$, we denote by $\to^+$ its transitive closure and by $\to^*$ its reflexive and transitive closure.

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 $\sigma = id$). Narrowing sequences can be represented by a (possibly infinite) finitely branching tree, which is called *narrowing tree*. Modern functional logic languages are based on *needed narrowing* and *inductively sequential* programs.

3 Partial Evaluation via Fold/Unfold

In this section, we recall from [4] the notion of partial evaluation of lazy functional logic programs. Specialized program rules are constructed from narrowing derivations using the notion of *resultant*: given a narrowing derivation $s \leadsto^*_\sigma t$, its associated resultant is the rule $\sigma(s) \to t$. The *(pre–)partial evaluation* of a term s is obtained by constructing a (possibly incomplete) narrowing tree for s and then extracting the specialized definitions (the resultants) from the non–failing, root–to–leaf paths of the tree.

Definition 1 (pre–partial evaluation). *Let $\mathcal{R}$ be a TRS and s be a term. Let $\mathcal{T}$ be a finite (possibly incomplete) narrowing tree for s in $\mathcal{R}$ such that no constructor-rooted term has been narrowed. Let $\{t_1, \ldots, t_n\}$ be the terms in the leaves of $\mathcal{T}$. Then, the set of resultants $\{\sigma_i(s) \to t_i \mid s \leadsto^+_{\sigma_i} t_i,$ for $i = 1, \ldots, n\}$ is called a pre–partial evaluation of s in $\mathcal{R}$. The pre–partial evaluation of a set of terms S in $\mathcal{R}$ is defined as the union of the pre–partial evaluations for the terms of S in $\mathcal{R}$.*

Given a call s and a program $\mathcal{R}$, there exists in general an infinite number of different pre–partial evaluations of s in $\mathcal{R}$. A fixed rule for generating resultants called an *unfolding rule* is assumed, which determines the expressions to be narrowed (by using a fixed narrowing strategy) and which decides how to stop the construction of narrowing trees (see [1, 3] for the definition of concrete unfolding rules).

A recursive *closedness* condition is needed in order to guarantee that each call which might occur during the execution of the resulting program is covered by some program rule. Informally, a term t rooted by a defined function symbol is closed w.r.t. a set of calls S (alternatively, t is S-closed), if it is an instance of a term s in S: $t = \theta(s)$, and the terms in the matching substitution θ are recursively closed by S.

In order to apply a partial evaluator based on needed narrowing and to guarantee that the resulting program is inductively sequential whenever the source program is, we have to ensure that the set of specialized terms contains only linear patterns with distinct root symbols. We note that, whenever the specialized call is not a linear pattern, lhs's of resultants may not be linear patterns either and hence resultants may not be program rules. In order to produce program rules, it is necessary to consider a post-processing renaming transformation which introduces a new function symbol for each specialized call and then replaces each call in the specialized program by a call to the corresponding renamed function. In particular, the lhs's of specialized program rules (which are constructor instances of the specialized terms) are replaced by instances of the corresponding new linear patterns by renaming.

Definition 2 (independent renaming). *An independent renaming ρ for a set of terms S is a mapping from terms to terms defined as follows: for $s \in S$, $\rho(s) = f_s(\overline{x_n})$, where $\overline{x_n}$ are the distinct variables in s in the order of their first occurrence, and f_s is a new function symbol, which does not occur in $\mathcal{R}$ or S and is different from the root symbol of any other $\rho(s')$, with $s' \in S$ and $s' \neq s$. We let $\rho(S)$ denote the set $S' = \{\rho(s) \mid s \in S\}$.*

Now, the notion of partial evaluation can be formulated as the renaming (using a given independent renaming ρ) of the set of resultants obtained by a pre-partial evaluation. This PE method preserves the structure of programs and it is strongly correct w.r.t. the renamed goals (see [4] for details). Let us illustrate these definitions with an example.

Example 1. Consider the well-known function `append` to concatenate two lists:[1]

[1] We use "`nil`" and "`:`" as constructors of lists.

$$
\begin{array}{lll}
\mathtt{append(nil, Y_s)} & \rightarrow \mathtt{Y_s} & (R_1) \\
\mathtt{append(X : X_s, Y_s)} & \rightarrow \mathtt{X : append(X_s, Y_s)} & (R_2)
\end{array}
$$

and the set $S = \{\mathtt{append(X_s, Y_s)}, \mathtt{append(append(X_s, Y_s), Z_s)}\}$. An independent renaming ρ for S is the mapping:

$$
\begin{array}{llll}
\{ & \mathtt{append(X_s, Y_s)} & \mapsto \mathtt{app(X_s, Y_s)}, & \\
 & \mathtt{append(append(X_s, Y_s), Z_s)} & \mapsto \mathtt{dapp(X_s, Y_s, Z_s)} & \}.
\end{array}
$$

A possible partial evaluation $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. S (under ρ) is:

$$
\begin{array}{ll}
\mathtt{dapp(nil, Y_s, Z_s)} & \rightarrow \mathtt{app(Y_s, Z_s)} \\
\mathtt{dapp(X : X_s, Y_s, Z_s)} & \rightarrow \mathtt{X : dapp(X_s, Y_s, Z_s)} \\
\mathtt{app(nil, Y_s)} & \rightarrow \mathtt{Y_s} \\
\mathtt{app(X : X_s, Y_s)} & \rightarrow \mathtt{X : app(X_s, Y_s)}
\end{array}
$$

In the following, we show that (a subset of) the folding/unfolding transformation rules described in [2] are able to achieve the effects of PE. The proof (see Appendix B) is not trivial, since: 1) the PE method applies to a single program whereas, in the fold/unfold framework, the programs considered for each transformation step are different, and 2) we do not consider a complete transformation system (such as [2, 6]) but three simple rules.

Firstly, we recall from [2] the definition of the transformation rules.

Definition 3. *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 (the "definition rule") of the form* $f(\overline{t_n}) \rightarrow r$, *provided the following conditions hold:*

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

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

Unfolding: *Let* $R = (l \rightarrow r) \in \mathcal{R}_k$ *be a rule where* r *is an operation-rooted term. Then,* $\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{\theta(l) \rightarrow r' \mid r \leadsto_\theta r' \text{ in } \mathcal{R}_k\}$.

Folding: *Let* $R = (l \rightarrow r) \in \mathcal{R}_k$ *be a non definition rule,* $R' = (l' \rightarrow r') \in \mathcal{R}_j$, $0 \leq j \leq k$, *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,* $\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{l \rightarrow r[\theta(l')]_p\}$.

Programs in a transformation sequence constructed by using the previous set of rules are inductively sequential. Moreover, the transformations are strongly correct w.r.t. goals which do not contain new function symbols. The following result represents the main theoretical result of this section.

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

Theorem 1. *Let $\mathcal{R}$ be an inductively sequential program, S a finite set of operation-rooted terms, and ρ an independent renaming of S. If $\mathcal{R}'$ is a PE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}'$ is S'-closed (where $S' = \rho(S)$) then, there exists a transformation sequence starting from $\mathcal{R}$ and ending with $\mathcal{R} \cup \mathcal{R}'$, such that only definition introduction, unfolding and folding steps (in this order) have been performed.*

The following example illustrates how folding/unfolding transformations are able to specialize an input program for a given goal. The main idea is to consider the renaming which is used in the PE process as the (inverted) eurekas which start the transformation process.

Example 2. Consider the program defining the function `append` of Example 1 as the initial program $\mathcal{R}_0$ of a transformation sequence. In order to mimic the partial evaluation process shown in Example 1, we consider the same independent renaming, ρ, and apply the definition introduction rule twice in order to produce two definition rules for the new symbols of ρ:

$$\mathcal{R}_1 = \mathcal{R}_0 \cup \left\{ \begin{array}{lll} \mathtt{app(X_s, Y_s)} & \to \mathtt{append(X_s, Y_s)} & (R_3) \\ \mathtt{dapp(X_s, Y_s, Z_s)} & \to \mathtt{append(append(X_s, Y_s), Z_s)} & (R_4) \end{array} \right\}$$

Now, by unfolding R_3 (using rules R_1 and R_2, i.e., the original definition of `append`), we get:

$$\mathcal{R}_2 = \mathcal{R}_0 \cup \left\{ \begin{array}{lll} \mathtt{app(nil, Y_s)} & \to \mathtt{Y_s} & (R_5) \\ \mathtt{app(X : X_s, Y_s)} & \to \mathtt{X : append(X_s, Y_s)} & (R_6) \\ \mathtt{dapp(X_s, Y_s, Z_s)} & \to \mathtt{append(append(X_s, Y_s), Z_s)} & (R_4) \end{array} \right\}$$

By unfolding R_4 (using rules R_1 and R_2):

$$\mathcal{R}_3 = \mathcal{R}_0 \cup \left\{ \begin{array}{lll} \mathtt{app(nil, Y_s)} & \to \mathtt{Y_s} & (R_5) \\ \mathtt{app(X : X_s, Y_s)} & \to \mathtt{X : append(X_s, Y_s)} & (R_6) \\ \mathtt{dapp(nil, Y_s, Z_s)} & \to \mathtt{append(Y_s, Z_s)} & (R_7) \\ \mathtt{dapp(X : X_s, Y_s, Z_s)} & \to \mathtt{append(X : append(X_s, Y_s), Z_s)} & (R_8) \end{array} \right\}$$

The unfolding of R_8 (using rule R_2) produces:

$$\mathcal{R}_4 = \mathcal{R}_0 \cup \left\{ \begin{array}{lll} \mathtt{app(nil, Y_s)} & \to \mathtt{Y_s} & (R_5) \\ \mathtt{app(X : X_s, Y_s)} & \to \mathtt{X : append(X_s, Y_s)} & (R_6) \\ \mathtt{dapp(nil, Y_s, Z_s)} & \to \mathtt{append(Y_s, Z_s)} & (R_7) \\ \mathtt{dapp(X : X_s, Y_s, Z_s)} & \to \mathtt{X : append(append(X_s, Y_s), Z_s)} & (R_9) \end{array} \right\}$$

Finally, by folding R_6 and R_7 w.r.t. the definition rule R_3 and folding R_9 w.r.t the definition rule R_4, we obtain the desired final program $\mathcal{R}_5 = \mathcal{R}_0 \cup \mathcal{R}'$, where $\mathcal{R}'$ is exactly the program obtained by PE in Example 1.

4 An Automatic Composition Algorithm

The composition strategy was originally introduced in [6, 9] for the optimization of pure functional programs. Variants of this composition strategy are the *internal specialization* technique [16] and the *deforestation* method [18]. By using the composition strategy, 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 a nested expression like `f(g(t))`.

Now, we introduce an algorithm to perform composition:

The Automatic Composition Algorithm

(1) Partial evaluation phase: Let $\mathcal{R}$ be an inductively sequential program. As the input of the PE method, we consider the set T of calls which contain the expressions (with nested functions) to be composed. Let $S \supseteq T$ be a finite set of terms and ρ be an independent renaming for S such that the PE $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. S (under ρ) is S'-closed, where $S' = \{\rho(s) \mid s \in S\}$.

(2) Definition introduction phase: By repeated application of the definition introduction rule, we add to $\mathcal{R}$ the following set of definition rules: $\mathcal{R}_{def} = \{\rho(s) \to s \mid s \in S\}$. Note that this phase ends with the program $\mathcal{R} \cup \mathcal{R}_{def}$.

(3) Unfolding phase: Each definition rule $R = (\rho(s) \to s) \in \mathcal{R}_{def}$ is unfolded (at least once) until a rule whose rhs contains an instance of some term in S is reached. This way, we produce a set of transformed rules of the form: $\mathcal{R}_{unf} = \{\theta(\rho(s)) \to r \mid (\rho(s) \to s) \in \mathcal{R}_{def}\}$, where each term r is obtained by unfolding the corresponding rhs s a number of times (computing the substitution θ). At the end of this phase, we get $\mathcal{R} \cup \mathcal{R}_{unf}$.

(4) Folding phase: In order to obtain efficient recursive definitions of the new symbols defined by the rules of $\mathcal{R}_{def}$, we use these rules to fold (if possible) the rhs's of the unfolded rules of $\mathcal{R}_{unf}$. After this process, the set of rules $\mathcal{R}_{unf}$ transforms into $\mathcal{R}_{fold}$, and thus we get $\mathcal{R} \cup \mathcal{R}_{fold}$.

(5) Post-folding phase: Finally, we also fold every instance of each term s occurring in the rhs's of the rules of $\mathcal{R}$ by using the appropriate definition rules of $\mathcal{R}_{def}$, which finally produces the transformed program $\mathcal{R}_{post} \cup \mathcal{R}_{fold}$.

There are several important remarks regarding the previous algorithm. A key point is the way in which eureka definitions are derived. The problem of achieving an appropriate set of eureka definitions is well-known in the literature related to fold/unfold transformations [6, 15, 17]. In our algorithm, we perform a finite, static analysis (by PE) that generates an appropriate set of definition rules at the beginning of the transformation (phases 1 and 2). Termination of the global process is ensured by the closedness condition, which guarantees that the search for regularities during the unfolding process will always succeed, producing efficient and recursive definitions. Observe that this is not generally true for the composition strategy, where the pursued optimization might be frustrated by infinite unfolding.

A relevant property of the programs produced by the automatic composition algorithm is the strong correctness w.r.t. goals containing calls to both the old and the new defined function symbols. This is an improvement w.r.t. the correctness results of the PE method as well as the general fold/unfold techniques: in programs specialized by PE only calls to the new function symbols can be safely executed, whereas in programs optimized by folding/unfolding only goals constructed with the old function symbols are safe (see discussion below). On the other hand, our composition strategy obtains an optimization of the input program which does not reduce its domain of application (as opposed to PE).

The outcome of step 4 of our automatic composition algorithm is a program of the form $\mathcal{R} \cup \mathcal{R}_{fold}$, where $\mathcal{R}_{fold}$ can be seen as an specialization of $\mathcal{R}$ where no calls to old function symbols appear. Since $\mathcal{R}_{fold}$ is self-contained (i.e., every call is covered by the set of rules themselves), then the strong correctness of the PE method ensures that goals which contain calls to the new function symbols can be safely executed in the transformed program. On the other hand, since the algorithm constructs a particular kind of transformation sequence, the strong correctness results of [2] apply. This guarantees that a goal containing old function symbols computes the same set of answer substitutions (and values) in the initial program $\mathcal{R}$ and in the final program $\mathcal{R}_{post} \cup \mathcal{R}_{fold}$, where they (hopefully) run faster.

There is a final remark about the structure of the transformed programs. Looking at their syntax, we distinguish two disjoint set of rules:

- $\mathcal{R}_{fold}$ represents the set of rules defining the new function symbols, where no calls to rules outside from $\mathcal{R}_{fold}$ appear, i.e., the specialized component is "self-contained".
- On the other hand, $\mathcal{R}_{post}$ coincides with the set of rules defining the set of old function symbols. This component is very similar to the original program $\mathcal{R}$, except for certain calls to the old function symbols which have been transformed (by virtue of the folding steps) into calls to functions of the $\mathcal{R}_{fold}$ specialized component.

5 Further Research

We conclude by mentioning some ideas for further research. There are some optimizations which can be achieved by fold/unfold transformations and are not possible by means of PE, e.g., *tupling* [6]. This technique eliminates parallel traversals of identical data structures by merging separate (non-nested) loops together into a new recursive function defined on *tuples* (pairs), thus avoiding the traversal of the involved data structures more than once. Generally, 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. Although PE is not able to produce the tupling optimization by itself, we think that it can be also used (with some modifications) as a pre-processing phase for an automatic tupling strategy. Inspired by the analysis of Chin [7], we plan to investigate how to use the PE method as a tupling eureka generator.

References

1. E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In G. Levi, editor, *Proc. of Static Analysis Symposium, SAS'98*, pages 262–277. Springer LNCS 1503, 1998.

2. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming, FLOPS'99*, pages 147–162. Springer LNCS 1722, 1999.
3. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
4. 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.
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. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
7. 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.
8. C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.
9. 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.
10. 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.
11. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.
12. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
14. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
15. 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.
16. 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.
17. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of 2nd Int'l Conf. on Logic Programming*, pages 127–139, 1984.
18. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Needed Narrowing

In this appendix we recall the formal definition of inductively sequential programs and needed narrowing. A detailed description can be found in [5].

Needed narrowing is defined on inductively sequential programs. To provide a precise definition of this class of programs and the needed narrowing strategy, we introduce definitional trees.

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 $\pi' < \pi'' < \pi$.

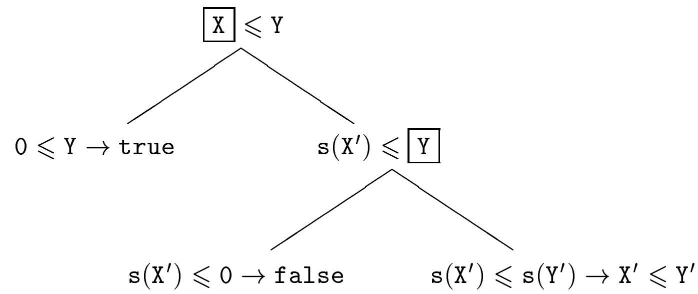
Induction property: Given $\pi \in \mathcal{P} \backslash S$, there is a position o of $pattern(\mathcal{P})$ with $pattern(\mathcal{P})|_o \in \mathcal{X}$ (called the *inductive position*), and constructors $c_1, \ldots, c_n \in \mathcal{C}$ with $c_i \neq c_j$ for $i \neq j$, such that, for all $\pi_1, \ldots, \pi_n$ 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 $1 \leq i \leq n$.

If $\mathcal{R}$ is an orthogonal term rewriting system and f/n a defined function, we call $\mathcal{P}$ a *definitional tree of* f if $pattern(\mathcal{P}) = f(\overline{x_n})$ for distinct variables $\overline{x_n}$ and the leaves of $\mathcal{P}$ are all and only variants of the left-hand sides of the rules in $\mathcal{R}$ defining f. Due to the orthogonality of $\mathcal{R}$, we can assign to each leaf a unique rule defining f. A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential. 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.

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

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

can be illustrated as follows:



For the definition of needed narrowing we assume that t is an operation-rooted term and $\mathcal{P}$ is a definitional tree with $pattern(\mathcal{P}) = \pi$ such that $\pi \leq t$. We define a function λ from terms and definitional trees to sets of tuples (position, rule, substitution) as the least set satisfying the following properties. We consider two cases for $\mathcal{P}$:[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 all patterns are instances of π_i. Then we consider the following cases for the subterm $t|_o$:

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

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). Thus, each needed narrowing step can be represented as $(p, R, \varphi_k \circ \cdots \circ \varphi_1)$, where each φ_j is either the identity or the replacement of a single variable computed in each recursive step (see [4]. This is also called the *canonical representation* of a needed narrowing step.

As in proof procedures for logic programming, we assume that the definitional trees always contain new variables if they are used in a narrowing step. This

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

implies that all computed substitutions are idempotent (we will implicitly assume this property in the following).

To compute needed narrowing steps for an operation-rooted term t, we take a definitional tree $\mathcal{P}$ for the root of t and compute $\lambda(t, \mathcal{P})$. Then, for all $(p, R, \sigma) \in \lambda(t, \mathcal{P})$, $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step.*

B Proof of Theorem 1

Let us first formalize the notions which we need for the proof.

Definition 4 (closedness). *Let S be a finite set of terms. We say that a term t is S-closed if $closed(S,t)$ holds, where the predicate closed is defined inductively as follows:*

$$closed(S,t) \Leftrightarrow \begin{cases} true & \textit{if } t \in \mathcal{X} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \textit{if } t = c(\overline{t_n}),\ c \in (\mathcal{C} \cup \{\approx, \wedge\}), \\ & \quad n \geq 0 \\ \displaystyle\bigwedge_{x \mapsto t' \in \theta} closed(S,t') & \textit{if } \exists \theta, \exists s \in S \ \textit{s.t.}\ \theta(s) = t \end{cases}$$

We say that a set of terms T is S-closed, written $closed(S,T)$, if $closed(S,t)$ holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is S-closed if $closed(S, \mathcal{R}_{calls})$ holds. Here we denote by $\mathcal{R}_{calls}$ the set of the rhs's of the rules in $\mathcal{R}$.

Definition 5 (partial evaluation). *Let $\mathcal{R}$ be a TRS, S be a finite set of terms and $\mathcal{R}'$ be a pre–partial evaluation of $\mathcal{R}$ w.r.t. S such that $\mathcal{R}'$ is S-closed. Let ρ be an independent renaming of S. We define the partial evaluation $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. S (under ρ) as follows:*

$$\mathcal{R}'' = \bigcup_{s \in S} \{\theta(\rho(s)) \to ren_\rho(r) \mid \theta(s) \to r \in \mathcal{R}' \ \textit{is a resultant for } s \textit{ in } \mathcal{R}\}$$

where the nondeterministic renaming function ren_ρ is defined inductively as follows:

$$ren_\rho(t) = \begin{cases} t & \textit{if } t \in \mathcal{X} \\ c(\overline{ren_\rho(t_n)}) & \textit{if } t = c(\overline{t_n}),\ c \in (\mathcal{C} \cup \{\approx, \wedge\}),\ n \geq 0 \\ \theta'(\rho(s)) & \textit{if } \exists \theta, \exists s \in S \ \textit{such that } t = \theta(s) \textit{ and} \\ & \quad \theta' = \{x \mapsto ren_\rho(\theta(x)) \mid x \in \mathcal{D}om(\theta)\}. \end{cases}$$

Now we proceed with the proof of Theorem 1.

Proof. By definition of PE, we have that every rule $R \in \mathcal{R}'$ has the form $R = (\theta(\rho(s)) \to ren_\rho(r))$, where $s \in S$ is an operation rooted term such that $s \leadsto^+_\theta r$ in $\mathcal{R}$. Formally, if $\mathcal{R}'$ is a PE of $\mathcal{R}$ w.r.t. S under ρ then:

$$\mathcal{R}' = \{\theta(\rho(s)) \to ren_\rho(r) \mid s \in S \text{ and } s \leadsto^+_\theta r \text{ in } \mathcal{R}\}$$

We prove the theorem by building up a transformation sequence leading to $\mathcal{R} \cup \mathcal{R}'$. We consider the following three phases (where $0 < i < j < k$):

$$(\mathcal{R}_0 = \mathcal{R}, \ldots, \mathcal{R}_i = \mathcal{R} \cup \mathcal{R}_{def}, \ldots, \mathcal{R}_j = \mathcal{R} \cup \mathcal{R}_{unf}, \ldots, \mathcal{R}_k = \mathcal{R} \cup \mathcal{R}_{fold},)$$

– *DEFINITION INTRODUCTION STEPS: from* $\mathcal{R}_0 = \mathcal{R}$ *to* $\mathcal{R}_i = \mathcal{R} \cup \mathcal{R}_{def}$.
This phase shows the relationship between the independent renaming used in the PE method and the set of *eurekas* or definition rules needed to start a fold/unfold transformation sequence. For each term $s \in S$ we have that $\rho(s) = f_s(\overline{x_n})$ is a mapping included in the independent renaming ρ, where $\overline{x_n}$ are the distinct variables in s in the order of their first occurrence, and f_s is a new function symbol, which does not occur in $\mathcal{R}$ or S and is different from the root symbol of any other $\rho(s')$, with $s' \in S$ and $s' \neq s$. Hence, for each $s \in S$ we can add to $\mathcal{R}$ a definition rule of the form $\rho(s) \to s$ by repeated application of the definition introduction rule (note that the requirements of Definition 3 are fulfilled). This process ends with the program: $\mathcal{R}_i = \mathcal{R} \cup \mathcal{R}_{def}$, where $\mathcal{R}_{def} = \{\rho(s) \to s \mid s \in S\}$.

– *UNFOLDING STEPS: from* $\mathcal{R}_i = \mathcal{R} \cup \mathcal{R}_{def}$ *to* $\mathcal{R}_j = \mathcal{R} \cup \mathcal{R}_{unf}$.
During the unfolding process, no rules of $\mathcal{R}$ are unfolded, which guarantees that those rules are never removed from the derived programs. This fact is essential in order to achieve that the intended unfolding for each rule belong to $\mathcal{R}_{def}$. During the process we unfold (possibly several times) every rule in $\mathcal{R}_{def}$ using only rules belonging to $\mathcal{R}$, obtaining as outcome a program of the form $\mathcal{R} \cup \mathcal{R}_{unf}$, where $\mathcal{R}_{unf}$ is the result of unfolding the rules of $\mathcal{R}_{def}$ using only rules of $\mathcal{R}$. Observe that, if $R = (\rho(s) \to s) \in \mathcal{R}_{def}$, then only new function symbols (and possibly variables) appear in its lhs, whereas only old function symbols (and possibly constructor and variables) appear in its rhs. This implies that, if R' is a rule used to unfold the rule R then $R' \in \mathcal{R}$ and $R' \notin \mathcal{R}_{def}$. Moreover, this property holds along the whole unfolding sequence $(\mathcal{R}_i \ldots \mathcal{R}_j)$, since unfolded rules never include new function symbols in its rhs's (because the rules used to unfold R belong to $\mathcal{R}$, which does not contain new function symbols).
Now, remember that in the pre-PE of $s \in S$ w.r.t. $\mathcal{R}$, a finite (possibly incomplete) narrowing tree $\mathcal{T}$ for s in $\mathcal{R}$ is generated such that no term in the tree is narrowed beyond its constructor rooted form (as also occurs with the unfolding rule). Moreover, if r is a leaf of $\mathcal{T}$, then there exists a needed narrowing derivation of the form $D : [s \leadsto_\theta^n r]$, $n >= 1$, using $\mathcal{R}$, such that $\theta(s) \to r$ is the corresponding resultant. Now, we prove that for any resultant of this form there exists one (and only one) rule $(\theta(\rho(s)) \to r) \in \mathcal{R}_{unf}$ obtained by repeatedly unfolding the rule $R = (\rho(s) \to s) \in \mathcal{R}_{def}$. More exactly, if $D : [s = s_0 \leadsto_{R_1,\theta_1} s_1 \leadsto \ldots \leadsto_{R_n,\theta_n} s_n = r]$, (with $\theta_n \circ \ldots \circ \theta_1 = \theta$) is the branch of $\mathcal{T}$ that generates the resultant $\theta(s) \to r$, we prove that by applying n times the unfolding rule on $R = (\rho(s) \to s) \in \mathcal{R}_{def}$ (using $R_1, \ldots, R_n$ in every unfolding step), we obtain the intended rule $(\theta(\rho(s)) \to r) \in \mathcal{R}_{unf}$. The proof is made by induction on the length n of the derivation D used to generate the resultant.

$n = 1$. Then, $D : [s = s_0 \leadsto_{R_1,\theta_1} s_1 = r]$, where $\theta_1 = \theta$. Since $R_1 \in \mathcal{R}$ then, by definition of unfolding of $R = (\rho(s) \to s) \in \mathcal{R}_{def}$ w.r.t $\mathcal{R} \cup \mathcal{R}_{def}$, we have that the rule $(\theta(\rho(s)) \to r) \in \mathcal{R}_h$, $(i < h < j)$. Now, by simply considering that this new rule is not unfolded in the remaining unfolding chain, we conclude that it belongs to $\mathcal{R}_{unf}$, and hence to $\mathcal{R}_j$, as we wanted to prove.

$n > 1$. After applying $n-1$ unfolding steps over $R = (\rho(s) \to s) \in \mathcal{R}_{def}$ we can assume (by the induction hypothesis) the presence of the rule $R = ((\theta_{n-1} \circ \ldots \circ \theta_0)(\rho(s)) \to s_{n-1})$ in the program $\mathcal{R}_h$, $(i < h < j)$, to be subsequently unfolded. By considering the last step in D, we have that $s_{n-1} \leadsto_{R_n, \theta_n} s_n$ is a needed narrowing step on s_{n-1} that uses the rule $R_n \in \mathcal{R}$. Hence, by unfolding $R \in \mathcal{R}_h$ using $R_n \in \mathcal{R}$ we have that $((\theta_n \circ \theta_{n-1} \circ \ldots \circ \theta_0)(\rho(s)) \to s_n)$, or equivalently $\theta(\rho(s)) \to r$, belongs to $\mathcal{R}_{h+1}$. Similarly to the base case, we now assume that this rule is not unfolded in the subsequent steps, so it belongs to $\mathcal{R}_{unf}$, and hence to $\mathcal{R}_j$, and the conclusion follows.

Observe that the unfolding process constitutes an alternative way to construct the unfolding trees used in the PE method, by means of repeated application of the unfolding rule in a transformation sequence. To summarize, if we call $\mathcal{R}_{unf} = \{\theta(\rho(s)) \to r \mid (\rho(s) \to s) \in \mathcal{R}_{def}\}$ then, this (sub)sequence ends with the desired program: $\mathcal{R}_j = \mathcal{R} \cup \mathcal{R}_{unf}$.

– *FOLDING STEPS: from* $\mathcal{R}_j = \mathcal{R} \cup \mathcal{R}_{unf}$ *to* $\mathcal{R}_k = \mathcal{R} \cup \mathcal{R}_{fold}$.
In this final transformation (sub)sequence, we relate the post-processing renaming of a PE process with the folding rule of the transformation system. More exactly, our goal is to transform every rule of the form $R = (\theta(\rho(s)) \to r) \in \mathcal{R}_{unf}$, in another rule of the form $R' = (\theta(\rho(s)) \to ren_\rho(r))$ by means of folding steps. Observe that this suffices to complete our proofs, since we can simply fold the rules of $\mathcal{R}_{unf}$ using the definition rules of $\mathcal{R}_{def}$ (which is possible since is allowed the folding operation to use rules from a previous program as folding rules). Thus, we obtain the intended program $\mathcal{R}_{fold}$, which coincides with the program obtained by PE. First, let us note that the renaming of r w.r.t. ρ can be obtained in an alternative way by appropriately folding (when it is the case) those subterms of r which are instances of the rhs of some definition rule. We consider the three possible cases considered in the renaming function ren_ρ of Definition 5.

1. if $r \in \mathcal{X}$, then r is not altered, hence no folding step is performed.
2. if $r = c(\overline{t_n}), c \in (\mathcal{C} \cup \{\approx, \wedge\})$, $n \geq 0$, then $ren_\rho(r) = c(\overline{ren_\rho(t_n)})$, which enables the recursive folding of the subterms $\overline{t_n}$.
3. let σ, let $s \in S$ such that $r = \sigma(s)$ and consider $\sigma' = \{x \mapsto ren_\rho(\sigma(x)) \mid x \in \mathcal{D}om(\sigma)\}$, with $ren_\rho(r) = \sigma'(\rho(s))$[4]. Now observe that the following conditions of the definition of the folding transformation are fulfilled:
   (a) r is not a constructor term;
   (b) $R \in \mathcal{R}_{unf}$ is the result of at least one unfolding within the sequence $\mathcal{R}_i, \ldots, \mathcal{R}_j$; and
   (c) since $s \in S$, then there exists a definition rule of the form $R_{def} = (\rho(s) \to s) \in \mathcal{R}_{def}$.

[4] Observe that this is guaranteed by the closedness condition (see Definition 4).

Then, we can fold the rule R using the rule R_{def} obtaining the new rule $\theta(\rho(s)) \to \sigma(\rho(s))$. Finally, observe that the renaming of the terms introduced by σ in $\sigma(\rho(s))$ produces the desired call $\sigma'(\rho(s))$.

Also note that, for a given set of definition rules, the form of the program $\mathcal{R}_{fold}$ may depend on the strategy which selects the definition rule $\rho(s) \to s$ to use as folding rule for the considered subterm $\sigma(s)$ (as occurs, for example, with $\mathtt{append(append(X_s, Y_s), Z_s)}$), since there may exist, in general, more than one term s in the rhs of the definition rules which cover the considered call. This kind of nondeterminism corresponds to the nondeterminism of the post-processing renaming of the PE method. Nevertheless, we are able to exactly mimic the concrete renaming that uses the mapping $s \to \rho(s)$ with the corresponding folding step by using the definition rule $\rho(s) \to s$, which guarantees the exact correspondence between the renamed program of the PE transformation and the folded one in the fold/unfold transformation.

To summarize, if we call $\mathcal{R}_{fold} = \{\theta(\rho(s)) \to ren_\rho(r) \mid (\theta(\rho(s)) \to r) \in \mathcal{R}_{unf}\}$, then, the last phase ends with the program: $\mathcal{R}_k = \mathcal{R} \cup \mathcal{R}_{fold} = \mathcal{R} \cup \mathcal{R}'$, as we wanted to prove.