Partial Evaluation of Functional Logic Programs in Rewriting-based Languages

L. Lafave   J. P. Gallagher

University of Bristol

Department of Computer Science

Partial Evaluation of Functional Logic Programs in Rewriting-based Languages

L. Lafave and J.P. Gallagher

Department of Computer Science, University of Bristol, Bristol BS8 1UB, U.K.
{lafave,john}@cs.bris.ac.uk

Abstract. The aim of this work is to describe a procedure for the partial evaluation of functional logic languages based on rewriting. In this work, we will use the Escher language as an example of a (concurrent) functional logic language which has rewriting as its computational mechanism. Partial evaluation is a program transformation technique which, by performing some computation and abstraction at compile time generates a specialised version of the program. We present an algorithm for the partial evaluation of Escher programs. The algorithm incorporates techniques from the partial evaluation of logic programs and positive supercompliation. We discuss the correctness and termination of the algorithm. Examples of the performance of the algorithm are presented. Finally, we compare our algorithm with the current partial evaluation procedures for narrowing functional-logic languages and traditional logic languages.

1 Introduction

We will explore the partial evaluation of functional logic languages with rewriting as their computational mechanism, design an algorithm which handles the unique terms which can arise, and extend a termination algorithm for global and local control of the procedure. The Escher functional logic language has been chosen as an example of one of these languages to use in this work. Escher combines of the best aspects of functional and logic programming with concurrency. It uses a rewriting simplification technique (which is not narrowing) to evaluate terms which can contain partially instantiated terms or higher-order functions. Escher computes the normal form of an expression by matching its subterms with an instantiated schema statement from the program. Because of its ability to handle the evaluation of non-ground terms, Escher differs from functional languages, and its lack of substitution sets it apart from logic programming languages.

Partial evaluation is a program transformation technique which optimises programs by performing some of the computation at compile-time. Therefore, given some (not all) of the input data and a program, a partial evaluator, which is a meta-program, will compute a specialised version of the original program, called a *residual program*. Given the rest of the input data, the residual program should compute the same answers as the original program run with the entire set of input data.

Futamura first introduced partial evaluation as an optimisation in [Fut71]. Since then, partial evaluation techniques have been developed for functional, logic, and some imperative languages [JGS93]. Using partial evaluators in declarative languages, compilers and compiler generators have been automatically generated. Furthermore, specialising an interpreter with respect to an object program has resulted in a meta-program which interprets a class of object programs. Partial evaluators which are self-applicable can specialise themselves with respect to an interpreter to generate a compiler.

Traditional partial evaluation in functional programming is based on pattern matching and constant-based information propagation; this was shown to be less powerful than partial evaluation in logic programming (called partial deduction) [SGJ94]. Partial evaluation in logic programming uses unification to propagate constant values in addition to further information about the partial input. In general, for a normal logic program and a set of atoms, a (finite) partial SLD-tree is constructed and the specialised definitions for the predicates are extracted from the tree [LS91, BL89]. This technique recently has been modified so conjunctions of predicates can be specialised with respect to each other [LSdW96, GJMS96]. This leads to extra specialisation, since dependencies between the predicates can be accounted for, and intermediate data structures can be optimised away. Another transformation technique called positive supercompilation has adopted the unification-based information propagation procedure for the partial evaluation of functional programs [GS96]. The technique resembles that of partial deduction: a partial process tree is generated using driving and generalisation from which a program is extracted. It has been shown that this technique is equivalent to the partial evaluation of logic languages [GS94], that is, it is more *perfect* in its specialisation than partial evaluation of functional languages.

A partial evaluation procedure for the specialisation of functional logic programs based on narrowing has been presented in [AFV96]. Narrowing is the mechanism for evaluating non-ground terms by using rewrite rules. The free variables in arguments are instantiated via substitutions so that one of the rewrite rules of a program can apply. An overview of narrowing can be found in [Han94]. Alpuente et al. [AFV96] present a general algorithm for the partial evaluation of narrowing functional logic programs and prove the correctness and termination of the algorithm. Their procedure shares the framework of partial evaluation of logic programs; however, changes in the algorithm were necessary to account for the nested functions of functional logic languages. A generalisation scheme similar to the one of positive supercompilation guarantees termination of the algorithm.

In the following paper, we will describe a partial evaluation technique for rewriting functional logic languages, in particular, the Escher language. The technique will use unification-based propagation in order to achieve greater specialisation than is possible in the Escher language. For the control of the algorithm, we will extend recent work in generalisation from positive supercompilation [SG95] and global control of partial deduction [MG95]. The algorithm will also concentrate on dead code elimination. We omit details of pre-processing

renaming and static analysis; the following algorithm is intended only to be a basis for the partial evaluation of these languages.

Outline of Paper

In the next section, we will give a brief introduction to the Escher language. We will then describe a basic algorithm for the partial evaluation of Escher programs in Section 3 and show an example. A refinement of the basic algorithm is presented in Section 4. We will discuss local and global control of the refined algorithm in Section 5. The correctness of this procedure will be discussed in Section 6 and further examples of the performance of the refined algorithm are available in Section 7. Finally, Section 8 will include a discussion of the differences between our algorithm and those for the partial evaluation of narrowing functional-logic languages [AFV96] and other partial evaluation techniques, such as supercompilation [GS96] and conjunctive partial evaluation [LSdW96, GJMS96].

2 Background

2.1 The Escher Language

Escher is a higher-order, typed functional logic language which is based on Church's simple theory of types [Llo95]. Escher uses *rewriting* as its computational mechanism. Rewriting differs from reduction and narrowing; reduction is an operational mechanism for evaluating ground expressions and those non-ground expressions which can be represented by non-ground terms, while narrowing operates on non-ground expressions as well, by finding the substitutions which make the expressions true [Red89, Han94].

Using rewriting, Escher can compute with partially instantiated terms without the need for substitutions or proof-based computations. This differs from narrowing, which uses substitutions to instantiate variables in the non-ground expression, and conditional narrowing, in which conditions have to be proved by narrowing [Han94]. All definitions in the Escher language are written as a set of schema statements. Each schema statement is the shorthand notation for the (infinite) set of all corresponding ground statements. The variables in an Escher schema statement are not variables at the object level, but are syntactic variables that stand for any ground term which can replace that variable. Because the variables are not in the object language, traditional bindings cannot be formed, and therefore, substitution does not apply in Escher.

As an example of the computation mechanism of the language, the following Escher statement from [Llo95] defines a function for splitting lists. This function is written with the intention that the first argument will be a list at the time that this function is called.

Example 1. Splitting a list in Escher.

```
Split([], x, y) =>
    x = [] & y = [].

Split([x | y], v, w) =>
    (v = [] & w = [x | y]) \/
    SOME [z] (v = [x | z] & Split(y, z, w)).
```

The term `Split([1,2], v, w)` will simplify to the boolean term
`(v = [] & w = [1,2]) \/ (v = [1] & w = [2]) \/ (v = [1,2] & w = []).`

The necessity of the first argument for the `Split` function to be a list constructor in order to satisfy the intended interpretation is explicitly noted in Escher by mode declarations. Implicitly, all instances of the schema statements for `Split` will have a list constructor as the outermost function of the first argument; therefore, no terms with, say, a variable in the first argument will match any instantiation of `Split`.

We will now define some of the key concepts behind Escher computations which will aid in our presentation of the algorithms. Some of the following definitions are taken from [Llo95].

From now on, the identifiers "terms" and "programs" refer to Escher terms and programs unless noted otherwise. In the following, R, S, T will denote terms, B, B_i denote terms (or sets of terms) in the body of a schema statement, s_i, t_i denote terms as arguments, F, G, H denote functions, and (syntactic and object level) variables are represented by the lowercase x, y, z. Terms are identical if they are modulo variable renaming, unless noted otherwise.

A function in Escher can either be free or defined. A defined function is one with an explicit definition and an associated equality theory given by the definition. On the other hand, a free function (including constructor and constant functions) are those which have the Clark equality theory of syntactic identity as their "definition".

Definition 1. A *pattern* is a term of the form $F(t_1, \ldots, t_n)$, where F is a free function.

As noted in the last section, an Escher program is a set of definitions, and a definition is a set of schema statements. An individual schema statement is an abbreviation for the (infinite) set of ground instances of the statement. A schema statement has the form $H \Rightarrow B$, where the head of the statement, H, and the body B are arbitrary terms.

Definition 2. A *redex* r of a term t is a selectable subterm of t such that r is identical to the head of some instance of a schema statement.

Let $T[r]_p$ represent a term containing a redex r at position p with surrounding term T.

Definition 3. A term is in Escher *normal form* if it does not contain a redex.

The selection rule for the Escher language constructs the set of redexes of a term and selects the outermost ones. The Escher selection rule is invariant under renaming of bound variables. A term S is obtained from a term $T = T'[r]_p$ by a *computation step* if r is identical to the head H of some instance $H \Rightarrow B$ of a schema statement and S is the term obtained from T by replacing r by B.

Definition 4. A *computation* from a term T is a sequence $\{T_i\}_{i=1}^n$ of terms such that the following conditions are satisfied.

1. $T = T_1$.
2. T_{i+1} is obtained from T_i by a computation step, for $i = 1, \ldots, n-1$.
3. T_n is in Escher normal form.

The term T_1 is called the *goal* of the computation and T_n is called the *answer*.

Let a *partial computation* be a computation whose last term (T_n) may or may not contain a redex. It is important to note that a computation may end with a term which contains user-defined functions.

3 Algorithm for Partial Evaluation in Escher

3.1 Preliminary Concepts

We begin with a few definitions that will be employed in our presentation of the algorithm for the partial evaluation of Escher programs. In general, the following definitions are not part of the Escher language.

Let V be the set of variables and patterns. Let $FV(T)$ be the set of free variables in a term T. Given a set of equations, A, $rhs(A)$ is the set of terms in the bodies of the equations: $rhs(A) = \{R_j \mid L_j \Rightarrow R_j \in A\}$. The *definition* Def_P^G of a k-ary function G in a program P is the set of statements in P with head $G(t_1, \ldots, t_k)$.

A *standard term* is a term with a user-defined function as the outermost function.

Definition 5. For a program P, a term T is a *standard term* if $T = F(s_1, \ldots, s_n)$ and F is an n-ary user-defined function, i.e. $\exists \, \mathrm{Def}_P^F$.

We define a shorthand notation for describing the result of an Escher computation.

Definition 6. $\Rightarrow^\phi$: Let $T_1 \Rightarrow^\phi T_n$ represent the equation resulting from the Escher computation $\{T_i\}_{i=1}^n$.

This differs from $T_1 \Rightarrow^* T_n$ which usually means that T_n is reached from T_1 by arbitrarily many computation steps, but T_n may not be in Escher normal form.

Clearly, $\Rightarrow^\phi$ is not guaranteed to terminate. However, for an arbitrary term T and program P, if the computation of T in P terminates, there exists a statement $T \Rightarrow^\phi T_n$ where T_n is the last term of the computation sequence. We will discuss this further in Section 5.

Definition 7. *subs(T)*: The set of standard sub-functions *subs(T)* of an arbitrary term T is the set of terms $\mathcal{S}$ where $S \in \mathcal{S}$ if S is a subterm of T, S is a standard term, and there exists no S' such that S is a subterm of S' in T and S' is a standard term.

Therefore, the function *subs(T)* extracts the outermost standard terms from the arbitrary term T.

Example 2. $subs((\texttt{F(G(x),H(z))} \ \& \ \texttt{H(w)})) = \{\texttt{F(G(x),H(z))}, \texttt{H(w)}\}$, where F, G, and H are user-defined functions in an Escher program.

3.2 Basic Algorithm

We will now describe the procedure for partially evaluating Escher programs. The following function, *specialise(T,P)*, returns a specialised definition for a term T wrt a program P. It allows for more information propagation than matching since it allows for subterms in the left hand side of the schema statement to be passed to the term via unification.

Definition 8. *specialise(T,P)*: Let T be a term and P be a program. Let $restart(T, P) = S$ be a subterm of T. Let F be the outermost function of S. Then, *specialise(T,P)* is the set of specialised statements:

$$specialise(T, P) = \{T\theta_i \Rightarrow E_j \mid L_i \Rightarrow R_i \in \operatorname{Def}_P^F \ \& \ \theta_i = \lceil S, L_i \rceil \neq \mathit{fail}$$
$$\& \ T\theta_i \Rightarrow^\phi E_j\}$$

If there is no such S in T, $specialise(T, P) = \emptyset$.

To generate the substitution for the *specialise* function, take the most general instance of the schema statement such that the left hand side unifies with the term in question. The expression $\lceil a, b \rceil$ denotes the idempotent most general unifier of a and b if it exists; otherwise, it equals *fail*.

In the above function, the function $restart(T, P)$ is used to find the standard subterm S to restart the partial evaluation; this function can be set to return any subterm of the standard term T, but in this case, we will define $restart(T, P)$ as follows.

Definition 9. *restart(T,P)*: For a term T and a program P, $F(t_1, \ldots, t_n) = restart(T,P)$ is the leftmost outermost *standard subterm* such that there is at least one L_i in Def_P^F such that $\lceil S, L_i \rceil \neq \mathit{fail}$.
    Special cases:

- $restart(T,P) = restart(b,P)$, if $T = \texttt{IF [SOME vars]} \ b \ \texttt{THEN} \ S_1 \ \texttt{ELSE} \ S_2$, and
- $restart(T,P) = restart(S_1,P)$, if $T = S_1 \ \texttt{>>} \ S_2$ or $S_1 \ \texttt{>>>} \ S_2$

Definition 10. Algorithm 1

Input: an Escher program P and an arbitrary term T.
Output: a Escher program P' which is the partial evaluation of P wrt T, and a set of terms B.
Initialisation:
$\quad P_0 := P \cup \{Ans(x_1 \ldots x_n) \Rightarrow T\}, \text{where } FV(T) = (x_1 \ldots x_n)$;
$\quad B_0 := \emptyset$;
$\quad Q_0 := \emptyset$;
$\quad B_1 := \{Ans(x_1 \ldots x_n)\}$;
$\quad i := 1$;

repeat
$\quad A_i = \bigcup \{specialise(S, P_0) \mid S \in (B_i \backslash B_{i-1})\}$;
$\quad Q_i = A_i \cup Q_{i-1}$;
$\quad B_{i+1} = (B_i \cup subs(rhs(A_i)))^\alpha$;
$\quad i = i + 1$;
until $B_i = B_{i-1}$ (modulo variable renaming)
return $B = B_i$;
$P' :=$ the program obtained from the set of statements of Q_{i-1}.

The α operator is intended to generalise the set B_i in order to control recursive unfolding. Control of the partial evaluation procedure will be discussed in Section 5.
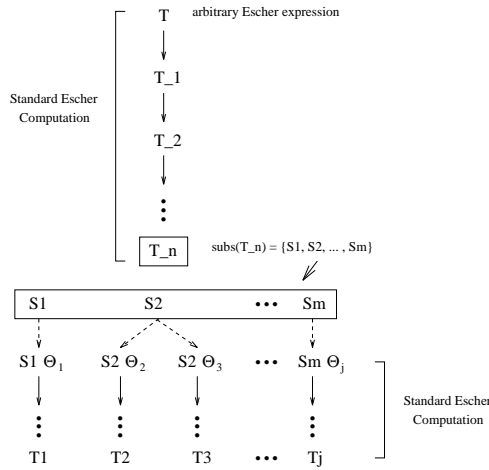

Fig. 1. A graphical description of Algorithm 1.

A graphical description of this procedure is shown in Figure 1. In this figure, the standard Escher computation of T to T_n is represented as a linear path. The computation suspends at T_n. The set of outermost standard subterms of T_n, represented as $\{S_1, \ldots, S_m\}$, is found by means of the *subs* function. Following the algorithm, *specialise* is called with each of these standard subterms. In this example, S_1 and S_m both have subterms which unify with the left hand side of only one schema statement; the subterm in S_2 unifies with two schema statements in its definition. The following statements are added to the set Q:

— T => T_n
— S_iθ_k => Tk for $1 \leq i \leq m$, $1 \leq k \leq j$.

Example 3. We have omitted the headings of the programs in these examples to conserve space; for examples of the full syntax of Escher programs, see [Llo95].

Consider an Escher program P with one function which computes the concatenation of two lists. The definition of Concat in P is:

```
FUNCTION  Concat : List(a) * List(a) -> List(a).
MODE      Concat(?, _).
Concat([], x) =>
     x.
Concat([u | x], y) =>
     [u | Concat(x, y)].
```

We partially evaluate P wrt to the term T = Concat(Concat(x, y), z).

```
B_0 = {};
Q_0 = {};
B_1 = {Ans(x,y,z)};

A_1 = {Ans(x,y,z) => Concat(Concat(x, y), z).};
Q_1 = A_1;
B_2 = {Ans(x,y,z), Concat(Concat(x, y), z)};
```

Following the algorithm, *specialise* uses the subterm S = Concat(x, y) since S unifies with both heads of the schema statements in the definition of Concat.

```
A_2 = {Concat(Concat([], y), z) => Concat(y, z).
        Concat(Concat([u | x], y),z) => [u | Concat(Concat(x, y), z)].};
Q_2 = A_2 \/ Q_1;
B_3 = ({Ans(x,y,z), Concat(Concat(x, y), z)} \/ {Concat(y, z),
          Concat(Concat(x, y), z)})^alpha
    = {Ans(x,y,z), Concat(Concat(x, y), z), Concat(y, z)};

A_3 = {Concat([], x) => x., Concat([u | x], y) => [u | Concat(x, y)].};
Q_3 = A_3 \/ Q_2;
B_4 = B_3 \/ {Concat(x, y)}^alpha
    = {Ans(x,y,z), Concat(Concat(x, y), z), Concat(y, z)}
    = B_3;
```

This computation returns the following specialised definition for `Concat`, which incorporates the nested `Concat` function. After renaming, we obtain an optimised program. The algorithm has performed well to identify the dependencies between the nested function and the outermost function.

```
Concat(Concat([], y), z) =>          Concat([], x) =>
    Concat(y, z).                        x.
Concat(Concat([u | x], y), z) =>     Concat([u | x], y) =>
    [u | Concat(Concat(x, y), z)].       [u | Concat(x, y)].
```

4 Refinement of the Algorithm

The basic algorithm is the backbone of the partial evaluation procedure, but there are two main improvements we can make at this time to achieve greater specialisation of Escher terms.

Although the algorithm specialises the nested `Concat(Concat(x, y), z)` term as shown in the previous section, it does not identify interrelationships between the variables of standard terms across boolean operators. For example, appending three lists in Escher could be achieved by the term `Append(x, y, t) & Append(t, z, w)`, with `Append` defined as in logic languages [1]. The refined algorithm presented in this section takes these cases into account in a way inspired by the recent work in conjunctive partial evaluation [LSdW96].

Secondly, to obtain greater polyvariance, we change the B_i sets of the previous algorithms to trees, similar to the *m-tree* approach of Martens and Gallagher [MG95].

We combine the notation of [GJMS96] and [MG95] to define the m-trees.

Definition 11. A *m-tree* μ is a labelled tree where nodes can either be marked or unmarked. A node N is labelled with a term T_N. For a branch β of the tree, $\mathcal{N}_\beta$ is the set of labels of β and for a leaf L of a tree, β_L is the unique branch containing L.

Definition 12. *extend*(τ, P) Given a m-tree μ and program P, *extend*(μ, P) is:

$\forall$ unmarked leaf nodes $L \in \mu_i$(
  $A_i = specialise(T_L, P_0)$;
  mark L in μ;
  $\forall B \in rhs(A_i)$(add a leaf L' to branch β_L with $T_{L'} = B$));

---

[1] Obviously, concatenating three lists could be done efficiently using nested functions, such as `w = Concat(Concat(x, y), z)`; nevertheless, a functional logic language leaves open the possibility of the above example.

Definition 13. Algorithm 2

Input: an Escher program P and an arbitrary term T.
Output: a Escher program P' which is the partial evaluation of P wrt T, and a m-tree of terms μ.
Initialisation:
$\quad P_0 := P \cup \{Ans(x_1 \ldots x_n) \Rightarrow T\}$, where $FV(T) = (x_1 \ldots x_n)$;
$\quad \mu_0 :=$ the tree containing one node labelled $Ans(x_1, \ldots, x_n)$;
$\quad i := 0$;
repeat
$\quad \mu_{i+1} = (extend(\mu_i, P_0))^\alpha$;
$\quad i = i + 1$;
until $\mu_i = \mu_{i-1}$
return μ_i;
$P' :=$ the program obtained from extracting the set of statements of μ_i.

The *extend* function adds leaves to the m-tree, so that the closedness of the algorithm can be ensured. The tree is generalised by the α operator in the algorithm, which we will define in Section 5. The residual program is extracted from the final tree μ_i in the manner described in [SG95].
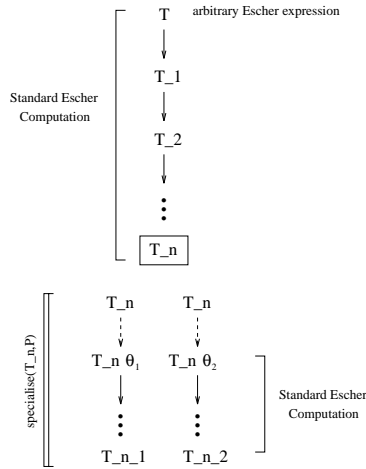

Fig. 2. A graphical description of a computation in Algorithm 2.

Using entire terms in the algorithm simplifies the computation; a description of a computation is shown in Figure 2. The standard Escher computation ends at T_n. The leftmost outermost redex of one of the subterms of T_n is chosen; call

this redex S. The definition of S in P contains two schema statements: $L_i \Rightarrow R_i$ for $i = 1, 2$. These statements are employed in the call to *specialise(T_n, P)*, after which we obtain the statements

- `T_n`θ_i `=>` `T_n_i` for $i = 1, 2$.

5   Termination of the Algorithm

Termination of $\Rightarrow^\phi$

As noted in Section 3, the operator $\Rightarrow^\phi$ as it is defined is not guaranteed to terminate. However, with the addition of an ordering on terms, we can define $\Rightarrow^\phi$ to generate finite Escher computations.

The strict homeomorphic embedding relation, $\trianglelefteq$, [SG95, LM95, GJMS96], orders the terms in such a way. For two terms S_1 and S_2, $S_1 \prec S_2$ indicates that S_2 is a strict instance of S_1.

Definition 14. strict homeomorphic embedding

- For variables x, y, $x \trianglelefteq y$.
- For terms $S, F(t_1, \ldots, t_n)$, $S \trianglelefteq F(t_1, \ldots, t_n)$ if $S \trianglelefteq t_i$ for some i.
- For terms $F(s_1, \ldots, s_n), F(t_1, \ldots, t_n)$, $F(s_1, \ldots, s_n) \trianglelefteq F(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for all i and $F(t_1, \ldots, t_n) \not\prec F(s_1, \ldots, s_n)$.

Definition 15. Given a partial computation $\{T_i\}_{i=1}^m$, the set of *selectable redexes*, L_i, is defined on the length of the computation:

- For $m = 1$, L_m is the set of outermost redexes of T_1.
- For $m > 1$,

$$L_m = \{R \mid R \text{ is an outermost redex of } T_m \ \& \ \not\exists R' \in L_j(R' \trianglelefteq S)\}$$

where $1 \leq j < m$.

A selectable computation step incorporates the selectable redexes into the computation.

Definition 16. A term S is obtained from a term T_j by a *selectable computation step* if the following are satisfied:

1. The set of selectable redexes of T_j, L_j, is a non-empty set.
2. For each α, the selectable redex $R_\alpha \in L_j$ is identical to the head H_α of some instance $H_\alpha \Rightarrow B_\alpha$ of a statement schema.
3. S is the term obtained from T_j by replacing, for each α, the selectable redex R_α by B_α.

We can now redefine our $\Rightarrow^\phi$ operator. A selectable computation $\{T\}_{i=1}^n$ is a sequence of terms obtained using the selectable computation steps where $L_n = \emptyset$.

Definition 17. $\Rightarrow^\phi$: Given a selectable computation $\{T\}_{i=1}^n$, $T_1 \Rightarrow^\phi T_n$.

By Kruskal's Theorem, this is a finite sequence. Therefore, the Escher computation sequences will be finite. This is not much of a restriction for the Escher computations, since the language is defined in such a way that most of the time the computations will stop if a necessary term is not instantiated sufficiently. The abstraction operator is more important for the termination of the algorithm, as this ensures the computation is not restarted with *specialise* having the same or growing terms as arguments each time.

The Abstraction Operator, α

In order to guarantee coveredness for the functions of the specialised definitions, yet ensuring termination of the partial evaluation procedure, the abstraction operator α of the m-tree of the algorithm occasionally must generalise the bodies of the specialised definitions before they are added to the tree. The abstraction operator we will use in this work is one which does not add a leaf L with label T_L to a tree μ if there exists a label S in $\mathcal{N}_{\beta_\mathcal{L}}$ and $S \trianglelefteq T_L$. The leaf is either generalised or folded back into the global tree. This method has been previously described in [SG95, MG95]. The most specific generalisation $\lfloor S, T \rfloor$ is used in order to guarantee termination of this generalisation step, since there exists the wfo $>$ such that $S, T_L > \lfloor S, T_L \rfloor$.

The definitions of instance, generalisation, and msg are included for completeness here; the original definitions can be found in [SG95, GS96].

Definition 18. For terms T, T_1, T_2, an *instance* of T is a term $T\theta$ where θ is a substitution. A *generalisation* of T_1 and T_2 is a triple (T, θ_1, θ_2) where θ_1, θ_2 are substitutions and $T\theta_1 = T_1$ and $T\theta_2 = T_2$. Finally, a *most specific generalisation (msg)* of T_1 and T_2 is a generalisation (T, θ_1, θ_2) of T_1 and T_2 such that for every generalisation $(T', \theta_1', \theta_2')$ of T_1 and T_2, T is an instance of T'.

Now, we can define the α abstraction operator. It has two functions: generalisation and folding. We use *folding* and *generalisation* nodes [SG95] for the m-tree to indicate where the operations occurred. These nodes are represented by the functions $F(x_1, \ldots, x_n)$ and $G(x_1, \ldots, x_n)$ for folding and generalisation respectively. $F(x_1, \ldots, x_n)$ has n children which are the terms of the substitution of the recursive function. The folding node also has a dashed line linking it to a previous node. $G(x_1, \ldots, x_n)$ has $n + 1$ children: the generalised term T of the *msg* and the terms of the substitution $t_1, \ldots, t_n$. These functions are not defined in P and have no effect on the ordering of the m-tree.

Definition 19. μ^α: For a m-tree μ containing branch β with added leaf L with label T_L, a term, μ^α is:

- if there exists an $T_{L'} \in \mathcal{N}_\beta$ on leaf $L'(\neq L)$ such that $T_{L'}\theta = T_L$ and $\theta = \{x_1 = t_1, \ldots, x_m = t_m\}$, then *fold* the tree in the following manner. Replace the label of L with $F(x_1, \ldots, x_m)$, draw a dashed line to L', and add leaves with labels $t_1, \ldots, t_m$ to branch β.

– if there exists an $T_{L'} \in \mathcal{N}_\beta$ on leaf $L'(\neq L)$ such that $T_{L'} \trianglelefteq T_L$, then let $\lfloor T, T_{L'} \rfloor = (T, \theta_1, \theta_2)$ where $\theta_1 = \{x_1 = t_1, \dots, x_m = t_m\}$ and *generalise* the tree in the following manner:

   • if T is not a variable:
      ∗ delete the branch β from after L' to L, including folds in the tree,
      ∗ replace the label of L' with $G(x_1, \dots, x_m)$, and
      ∗ add leaves with labels $T, t_1, \dots, t_m$ to branch β.
   • if T is a variable:
      ∗ split the term $T_L = F(t_1, \dots t_n)$ into $n + 1$ terms: $F(y_1, \dots y_n), t_1, \dots t_n$ where $y_1, \dots, y_n$ do not occur in T_L.
      ∗ replace the label of L with $G(x_1, \dots, x_n)$, and
      ∗ add leaves with labels $F(x_1, \dots x_n), t_1, \dots, t_n$ to branch β.

– if T is either $ITE(t_1, t_2, t_3)$, $ISTE(v, t_1, t_2, t_3)$, $t_1 \gg t_2$, or $t_1 \ggg t_2$, and there does *not* exist a subterm S such that $restart(t_1, P) = S$, add three leaves to β, labelled t_1, t_2, and t_3, or two leaves, labelled t_1 and t_2, respectively.

– if T is a pattern $F(t_1, \dots t_n)$, add leaves to β, labelled $t_1, \dots, t_n$.

– μ otherwise.

6 Correctness of the Procedure

The correctness of the partial evaluation algorithm for Escher follows.

Definition 20. The partial evaluation of a program P wrt a term T is *specialise(T,P)*. The partial evaluation of program P wrt a set of terms $\mathbf{W}$ is $\bigcup \{specialise(T, P) \mid T \in \mathbf{W}\}$.

In order to prove the correctness of this algorithm, we must first give an amended closedness condition.

Definition 21. $\mathbf{W}$-*closed:* Let $\mathbf{W}$ be a finite set of terms. Let R be a set of terms. Then, B is $\mathbf{W}$-closed if, for all terms $S \in subs(B)$, $\exists W \in subs(\mathbf{W})$ such that $W\theta = S$ and the set of terms of θ is $\mathbf{W}$-closed.

In the above definition, $subs(R)$ is shorthand notation for the set resulting after applying $subs$ to each member of the set of terms R. For a program $P = \{L_1 \Rightarrow R_1, \dots, L_n \Rightarrow R_n\}$, P is $\mathbf{W}$-closed if $\{L_1, R_1, \dots, L_n, R_n\}$ is $\mathbf{W}$-closed. The following theorem guarantees the soundness of this procedure.

Theorem 22. *Let P be an Escher program and $\mathbf{W}$ an set of arbitrary terms. Let P' be the partial evaluation of P wrt $\mathbf{W}$. Let T be a term such that $T \in \mathbf{W}$ and θ be a substitution such that the set of terms of θ is $\mathbf{W}$-closed. Let T_n^P be the result of the computation of $T\theta$ in P. Then, if P' is $\mathbf{W}$-closed, the result of the computation of $T\theta$ in P', $T_n^{P'}$, is equivalent to T_n^P (modulo renaming).*

A sketch of the proof is as follows: given a computation sequence $\mathbf{C} = (T\theta = T_1, T_2, \ldots, T_n)$ in P', construct a corresponding sequence in P with the following translations. If the computation step T_j to $T_{j+1}, 1 \le j \le n-1$ is obtained using the statement $L_i \Rightarrow R_i$ in P' and $T_j = L_i\phi$, then one of the following holds:

- If $L_i \Rightarrow R_i$ in P' is obtained during the partial evaluation by the standard Escher computation $L_i = S_1, S_2, \ldots, S_n = R_i$, insert $S_2\phi, \ldots, S_{n-1}\phi$ into the computation sequence $\mathbf{C}$.
- If $L_i \Rightarrow R_i$ in P' is obtained by
  $specialise(T', P)$ with $L_i = T'\theta$, $restart(T', P) = S'$ and $\theta = \lceil S', L_S \rceil$ for some $L'_S \Rightarrow R'_S$ in the definition of S', then insert $T'\theta\phi, S_2\phi, \ldots, S_{n-1}\phi$ into the computation sequence $\mathbf{C}$, where $S_2, \ldots, S_{n-1}$ indicates the standard Escher computation from $T'\theta$ to R_i.

Similar translations exist for the generalisation and folding operations of the abstraction operator. The idea behind the proof can be seen in the following computation sequences for Example 2 of Section 3; the specialised computation follows the original computation, with unnecessary computation steps removed.

Example 4. Given the term `T = Concat(Concat([1, 2], [3]), z)`, the two computation sequences for T in P and P' are respectively:

```
Concat(Concat([1,2],[3]),z) => Concat([1 | Concat([2], [3])], z)
                            => [1 | Concat(Concat([2], [3]), z)]
                            => [1 | Concat([2 | Concat([], [3])], z)]
                            => [1, 2 | Concat(Concat([], [3]), z)]
                            => [1, 2 | Concat([3], z)]
                            => [1, 2, 3 | Concat([], z)]
                            => [1, 2, 3 | z]

Concat(Concat([1,2],[3]),z) => [1 | Concat(Concat([2], [3]), z)]
                            => [1, 2 | Concat(Concat([], [3]), z)]
                            => [1, 2 | Concat([3], z)]
                            => [1, 2, 3 | Concat([], z)]
                            => [1, 2, 3 | z]
```

It is obvious from the generation of the residual program that the following lemma holds.

Lemma 23. *Let P be an Escher program and $\mathbf{W}$ a set of arbitrary terms. Let P' be the partial evaluation of P wrt $\mathbf{W}$. For any $T \in \mathbf{W}$, if there is no computation step from $T\theta$ in P', then there is no computation step from $T\theta$ in P.*

7   Examples

Specialisation of Higher Order Functions

This example presents a partial evaluation involving higher-order functions. The following program includes a definition of the `Map` predicate.

```
MODULE     Relational.

IMPORT     Lists.

CONSTRUCT Person/0.

FUNCTION  Bob, John : One -> Person.

FUNCTION  Age : Person * Integer -> Boolean.
Age(x,y) =>
    (x=Bob & y=24)  \/  (x=John & y=7).

FUNCTION  MapPred : (a * b -> Boolean) * List(a) * List(b) -> Boolean.
MODE      MapPred(_, ?, _).
MapPred(p, [], z) =>
    z = [].
MapPred(p, [x | xs], z) =>
    SOME [y,ys] (p(x, y) & MapPred(p, xs, ys) & z = [y | ys]).
```

Partially evaluating this program wrt to the term

```
T = MapPred(Age, x, z).
```

leads to the following specialised definitions:

```
MapPred(Age, x, z) =>
    F1(x, z).

F1([], z) =>
    z = [].
F1([y | w], z) =>
    SOME [ys1] ((y = Bob) & F1(w, ws1) & z = [24 | ws1]) \/
    SOME [ys1] ((y = John) & F1(w, ws1) & z = [7 | ws1]).
```

Partial Evaluation of a Pattern Matcher

The next example is intended to give an indication of the performance of the
algorithm. This compares the algorithm to others via the Knuth-Morris-Pratt
pattern matching program.

Example 5. The following section of program P is a general, tail-recursive pattern
matcher.

```
FUNCTION  Match : List(a) * List(a) -> Boolean.
MODE      Match(_, _).
Match(p, s) =>
    Loop(p, s, p, s).

FUNCTION  Loop : List(a) * List(a) * List(a) * List(a) -> Boolean.
MODE      Loop(?, ?, _, _).
Loop([], [], op, os) =>
    True.
Loop([], [s | ss], op, os) =>
    True.
Loop([p | pp], [], op, os) =>
    False.
Loop([p | pp], [s | ss], op, os) =>
    IF    p = s
    THEN  Loop(pp, ss, op, os)
    ELSE  Next(op, os).

FUNCTION  Next : List(a) * List(a) -> Boolean.
MODE      Next(_, ?).
Next(op, []) =>
    False.
Next(op, [s | ss]) =>
    Loop(op, ss, op, ss).
```

We partially evaluate the above program with respect to the term

```
T = Match([A, A, B], x)
```

in order to perform the Knuth Morris Pratt (KMP) test. Where the general tail-recursive matcher above will perform the computation in $O(|p||s|)$ time, a residual program is a KMP style pattern matcher if it is a $O(|s|)$ time program [JGS93]. A comparison of the techniques for partial evaluation with respect to this test is available in [SGJ94].
After execution of the partial evaluation of the above program wrt to T (assuming post-unfolding), the residual program is:

```
Ans(x) =>
    F_1(x).

Loop_AAB([]) =>  False.
Loop_AAB([s | ss]) =>
    IF    s = A
    THEN Loop_AB(ss, s)
    ELSE Loop_AAB(ss).

Loop_AB([], s) =>  False.
```

```
Loop_AB([s1 | ss1], s) =>
    IF    s1 = A
    THEN Loop_B(ss1, s, s1)
    ELSE Loop_AAB([s1 | ss1]).

Loop_B([], s, s1) =>  False.
Loop_B([s2 | ss2], s, s') =>
    IF    s2 = B
    THEN Loop_Nil(ss2, s, s1, s2)
    ELSE Loop_AAB([s1, s2 | ss2]).

Loop_Nil([], s, s1, s2) =>  True.
Loop_Nil([s3 | ss3], s, s1, s2) =>  True.
```

We obtain an equivalent level of specialisation as deforestation techniques, although our technique specialises away the `Next` function of the program. Because our algorithm does not employ positive driving, we do not obtain the almost KMP-style pattern matcher that a technique like positive supercompilation would generate [SGJ94]; the specialised pattern matcher still has to call `Loop_AAB` with the partial string every time a match fails.

8   Discussion

The first algorithm for partial evaluation of a functional logic language based on narrowing was presented in [AFV96]. This algorithm shares the framework of partial deduction as formalised by Lloyd and Shepherdson [LS91]. For a functional logic program R and a goal $y = s$ where s is a term and y is a variable not contained in s, a finite narrowing tree μ is constructed. Then, the non-failing leaves of μ are collected and the resultants are formed. The set of resultants is the partial evaluation of s in R (using μ). A revised version of the closedness condition incorporates the possibility of nested functions in functional logic languages. The work is based on computing partial narrowing trees for the terms of the partial evaluation. Homeomorphic embedding is used in this work to ensure local termination of the algorithm.

Because of the close relationship between the languages, our algorithm shares many characteristics with this work. Because of the nested functions, the closedness condition must recurse over the term. The approach to global control of the algorithms are similar, both extracted from positive supercompilation. However, their algorithm considers only atoms, much like our basic algorithm. Therefore, they cannot achieve the level of specialisation we obtain by specialising the entire term.

The actual algorithm for the partial evaluation of Escher programs is generic wrt the selection rule for Escher, the $\Rightarrow^\phi$ operator, and the α abstraction operator. The backbone of the algorithm is based on Gallagher's Basic Algorithm for the specialisation of logic programs.

Our procedure achieves the elimination of intermediate data structures and increased specialisation across boolean terms with interrelationships between variables by specialising the entire term. Therefore, when partially evaluating some function calls, we obtain further specialisation than current algorithms for the partial evaluation of functional logic languages.

We have not reached the same level of specialisation as positive supercompilation [SGJ94], since we have not incorporated all of the propagation of information performed by driving [GS96]. Positive driving uses unification-based propagation over conditionals; this advantage could not be added to our algorithm at this time because of restrictions in the rewriting mechanism. However, current work on the partial evaluation procedure includes incorporating constraints for storing the conditions of IF-THEN-ELSE statements. In this way, positive and negative driving will be available by means of constraint-based propagation; therefore, greater specialisation may be achieved. Furthermore, the constraint solving may be able to occur in the Escher language, by means of constraint handling rules [Frü94]. At the moment, the specialisation which results from this procedure is comparable with that of deforestation [SGJ94].

The unification-based information propagation of our procedure allows for more specialisation than constant-based partial evaluation of functional programs. As described in [JGS93], binding time improvements are necessary to introduce to the original functional program in order to achieve noticeable specialisation of the Match program, when traditional partial evaluation techniques are used.

In addition to work in constraint-based propagation of information, future work includes using recent work in characteristic trees [GB91] and generalisation algorithms [MG95, GS96] to guarantee termination of the algorithm while providing a suitable level of polyvariance. The current version of the α-operator uses techniques similar to the generalisation step of the positive supercompiler; the use of trace terms [GL96] may allow an elegant application of the results of [Leu95].

9 Conclusion

Escher is a new functional logic language which incorporates the best aspects of functional and logic languages into a declarative language with concurrency. Similarly, for the definition of the procedure of the partial evaluation of Escher programs, we have incorporated some of the recent work in program specialisation. We have presented and refined a partial evaluation algorithm for the Escher language. The global and local control issues have been addressed using homeomorphic embedding, and the soundness of the algorithm has been discussed. The algorithm optimises away intermediate data structures, but not yet able to propagate information through a conditional term as in positive driving.

Acknowledgements

References

[AFV96]   M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. R. Nielson, editor, *Proc. of European Symp. on Programming Languages, ESOP'96*, pages 45–61. Springer LNCS 1058, 1996.

[BL89]   K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming, Proc. of the North American Conference*, pages 343–358, Cleveland, 1989. The MIT Press.

[DGT96]   O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *LNCS*, Dagstuhl Castle, Germany, 1996. Springer.

[Frü94]   T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 1994.

[Fut71]   Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 25:45–50, 1971.

[GB91]   J.P. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Gener. Comput.*, 9:305–333, 1991.

[GJMS96]   R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. Technical Report CW 226, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996. Submitted for Publication.

[GL96]   J.P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In Danvy et al. [DGT96], pages 115–136.

[GS94]   R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 165–181. Springer-Verlag, 1994.

[GS96]   R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In Danvy et al. [DGT96].

[Han94]   M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *J. Logic Programming*, 19,20:583–628, 1994.

[JGS93]   N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. Series editor C. A. R. Hoare.

[Leu95]   M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. Technical Report CW216, Katholieke Universiteit Leuven, October 1995.

[Llo95]   J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[LM95]   M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. Technical Report CW220, Katholieke Universiteit Leuven, December 1995.

[LS91]   J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3&4):217–242, October 1991.

[LSdW96] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction; Towards a Maximal Integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.

[MG95]   B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Stirling, editor, *International Conference on Logic Programming*, pages 597–613. MIT Press, 1995.

[Red89]   U. S. Reddy. Rewriting techniques for program synthesis. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 388–403, Chapel Hill, NC, April 1989. Vol. 355 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

[SG95]   M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *International Logic Programming Symposium*, page to appear. MIT Press, 1995.

[SGJ94]   M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP*. Springer-Verlag, 1994.

This article was processed using the LaTeX macro package with LLNCS style