Using an Abstract Representation to Specialize Functional Logic Programs*

Elvira Albert[1], Michael Hanus[2], and Germán Vidal[1]

[1] DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain
{ealbert,gvidal}@dsic.upv.es
[2] Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. This paper introduces a novel approach for the specialization of functional logic languages. We consider a maximally simplified abstract representation of programs (which still contains all the necessary information) and define a non-standard semantics for these programs. Both things mixed together allow us to design a simple and concise partial evaluation method for modern functional logic languages, avoiding several limitations of previous approaches. Moreover, since these languages can be automatically translated into the abstract representation, our technique is widely applicable. In order to assess the practicality of our approach, we have developed a partial evaluation tool for the multi-paradigm language Curry. The partial evaluator is written in Curry itself and has been tested on an extensive benchmark suite (even a meta-interpreter). To the best of our knowledge, this is the first purely declarative partial evaluator for a functional logic language.

1 Introduction

Partial evaluation (PE) is a source-to-source program transformation technique for specializing programs w.r.t. parts of their input (hence also called *program specialization*). PE has been studied, among others, in the context of functional programming (e.g., [9, 21]), logic programming (e.g., [12, 24]), and functional logic programming (e.g., [4, 22]). While the aim of traditional partial evaluation is to specialize programs w.r.t. some known data, several PE techniques are able to go beyond this goal, achieving more powerful program optimizations. This is the case of a number of PE methods for functional programs (e.g., positive supercompilation [27]), logic programs (e.g., partial deduction [24]), and functional logic programs (e.g., narrowing-driven PE [4]). A common pattern of these techniques is that they are able to achieve optimizations regardless of whether known data are provided (e.g., they can eliminate some intermediate data structures, similarly to Wadler's deforestation [28]). In some sense, these techniques are stronger *theorem provers* than traditional PE approaches.

* This work has been partially supported by CICYT TIC 98-0445-C03-01, by Acción Integrada hispano-alemana HA1997-0073, and by the DFG under grant Ha 2457/1-1.

Recent proposals of multi-paradigm declarative languages amalgamate the most important features of functional, logic and concurrent programming (see [14] for a survey). The operational semantics of these languages is usually based on a combination of two different operational principles: narrowing and residuation [15]. The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation (by rewriting). On the other hand, the *narrowing* mechanism allows the instantiation of variables in input expressions and, then, applies reduction steps to the function calls of the instantiated expression. Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* [6] is currently the best narrowing strategy for functional logic programs. The formulation of needed narrowing is based on the use of *definitional trees* [5], which define a strategy to evaluate functions by applying narrowing steps.

In this work, we are concerned with the PE of functional logic languages. The first approach to this topic was the narrowing-driven PE of [4], which considered functional logic languages with an operational semantics based solely on narrowing. Recently, [2] introduced an extension of this basic framework in order to consider also the residuation principle. Using the terminology of [13], the narrowing-driven PE methods of [2, 4] are able to produce both *polyvariant* and *polygenetic* specializations, i.e., they can produce different specializations for the same function definition and can also combine distinct original function definitions into a comprehensive specialized function. This means that narrowing-driven PE has the same potential for specialization as *positive supercompilation* [27] and *conjunctive partial deduction* [10] (a comparison can be found in [4]).

Despite its power, the narrowing-driven approach to PE suffers from several limitations: (i) Firstly, in the context of *lazy* functional logic languages, expressions in *head normal form* (i.e., rooted by a constructor symbol) cannot be evaluated at PE time. This restriction is imposed because the *backpropagation* of bindings to the left-hand sides of residual rules can incorrectly restrict the domain of functions (see Example 2). (ii) Secondly, if one intends to develop a PE scheme for a realistic multi-paradigm declarative language, several high-level constructs have to be considered: higher-order functions, constraints, program annotations, calls to external functions, etc. A complex operational calculus is required to properly deal with these additional features of modern languages. It is well-known that a partial evaluator normally includes an interpreter of the language. Therefore, as the operational semantics becomes more elaborated, the associated PE techniques become (more powerful but) also increasingly more complex. (iii) Finally, an interesting application of PE is the generation of compilers and compiler generators [21]. For this purpose, the partial evaluator must be self-applicable, i.e., able to partially evaluate itself. This becomes difficult in the presence of high-level constructs such as those mentioned in (ii). As advised in [21], *it is essential to cut the language down to the bare bones* in order to achieve self-application.

In order to overcome the aforementioned problems, a promising approach successfully tested in other contexts (e.g., [7, 25]) is to consider programs written in

a maximally simplified programming language, into which programs written in a higher-level language can be automatically translated. Recently, [18] introduced an explicit representation of the structure of definitional trees (used to guide the needed narrowing strategy) in the rewrite rules. This provides more explicit control and leads to a calculus simpler than standard needed narrowing. Moreover, source programs can be automatically translated to the new representation.[1] In this work, we consider a very simple abstract representation of functional logic programs which is based on the one introduced in [18]. As opposed to [18], our abstract representation includes also information about the evaluation type of functions: *flexible* —which enables narrowing steps— or *rigid* —which forces delayed evaluation by rewriting. Then, we define a *non-standard* semantics which is specially well-suited to perform computations at PE time. This is a crucial difference with previous approaches [2, 4], where the same mechanism is used both for program execution and for PE. The use of an abstract representation, together with the new calculus, allows us to design a simple and concise automatic PE method for modern functional logic languages, breaking the limitations of previous approaches.

Finally, since truly lazy functional logic languages can be automatically translated into the abstract representation (which still contains all the necessary information about programs), our technique is widely applicable. Following this scheme, partially evaluated programs will be also written in the abstract representation. Since existing compilers use a similar representation for intermediate code, this is not a restriction. Rather, our specialization process can be seen as an optimization phase (transparent to the user) performed during the compilation of the program. In order to assess the practicality of our approach, we have developed a PE tool for the multi-paradigm language Curry [19]. The partial evaluator is written in Curry itself and has been tested on an extensive set of benchmarks (even a meta-interpreter). To the best of our knowledge, this is the first purely declarative partial evaluator for a functional logic language.

The structure of this paper is as follows. After providing some preliminary definitions in Sect. 2, we present our approach for the PE of functional logic languages based on the use of an abstract representation in Sect. 3. We also discuss the limitations of using the standard semantics during PE and, then, introduce a more suitable semantics. Section 4 presents a fully automatic PE algorithm based on the previous ideas, and Sect. 5 shows some benchmarks performed with an implementation of the partial evaluator. Finally, Sect. 6 concludes and discusses some directions for future work. More details and missing proofs can be found in [3].

2 Preliminaries

In this section we recall, for the sake of completeness, some basic notions from term rewriting [11] and functional logic programming [14]. We consider a (*many-*

[1] Indeed, it constitutes the basis of a recent proposal for an standard intermediate language, FlatCurry, for the compilation of Curry programs [20].

sorted) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. There is at least one sort *Bool* containing the constructors `True` and `False`. The set of *terms* and *constructor terms* with *variables* (e.g., x, y, z) from $\mathcal{V}$ are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term is *linear* if it does not contain multiple occurrences of any variable. We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$. We denote by $root(t)$ the symbol at the root of the term t. A *position* p in a term t is denoted by a sequence of natural numbers. Positions are ordered by: $u \leq v$, if $\exists w$ such that $u.w = v$. The *subterm* of t at position p is denoted by $t|_p$, and $t[s]_p$ is the result of *replacing the subterm* $t|_p$ by the term s.

We denote a *substitution* σ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (where $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. By abuse, $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of σ. Also, $\mathcal{R}an(\theta) = \{\theta(x) \mid x \in \mathcal{D}om(\theta)\}$. A substitution σ is a *constructor* substitution, if $\sigma(x)$ is a constructor term $\forall x \in \mathcal{D}om(\sigma)$. The identity substitution is denoted by $\{\ \}$. Given a substitution θ and a set $V \subseteq \mathcal{V}$, we denote the substitution obtained from θ by restricting its domain to V by $\theta_{|V}$. We write $\theta = \sigma\ [V]$ if $\theta_{|V} = \sigma_{|V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$. A term t' is an *instance* of t if $\exists \sigma$ with $t' = \sigma(t)$.

A set of rewrite rules $l = r$ such that $l \notin \mathcal{V}$, 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* and the *right-hand side* of the rule, respectively. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l = r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. Given a relation $\rightarrow$, we denote by $\rightarrow^+$ the transitive closure of $\rightarrow$, and by $\rightarrow^*$ the transitive and reflexive closure of $\rightarrow$. A (constructor) *head normal form* is either a variable or a term rooted by a constructor symbol. To evaluate terms containing variables, narrowing nondeterministically instantiates the variables so that a rewrite step is possible. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \rightarrow_{p,R} t'$. 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 $\sigma = \{\ \}$.) In functional programming, one is interested in the computed *value* whereas logic programming emphasizes the different bindings (*answers*). In an integrated setting, given a narrowing derivation $t_0 \leadsto^*_\sigma t_n$, we say that t_n is the computed value and σ is the computed answer for t_0.

3 Using an Abstract Representation for PE

In this section, we present an appropriate abstract representation for modern functional logic languages. We also provide a non-standard operational semantics which is specially well-suited to perform computations during partial evaluation.

First, let us briefly recall the basis of the narrowing-driven approach to PE of [4]. Informally speaking, given a particular narrowing strategy $\leadsto$, the (paramet-

ric) notions of resultant and partial evaluation are defined as follows. A *resultant* is a program rule of the form: $\sigma(s) = t$ associated to a narrowing derivation: $s \leadsto_\sigma^+ t$. A *partial evaluation* for a term s in a program $\mathcal{R}$ is computed by constructing a finite (possibly incomplete) narrowing *tree* for this term, and then extracting the resultants associated to the root-to-leaf derivations of the tree. Depending on the considered class of programs (and the associated narrowing strategy), a PE might require a post-processing of renaming to recover the same class of programs. An intrinsic feature of the narrowing-driven approach is the use of the same operational mechanism for both execution and PE.

3.1 The Abstract Representation

Recent approaches to functional logic programming consider inductively sequential systems as programs and a combination of needed narrowing and residuation as operational semantics [15, 19]. The precise mechanism (narrowing or residuation) for each function is specified by *evaluation annotations*, which are similar to coroutining declarations in Prolog, where the programmer specifies conditions under which a call is ready for a resolution step. Functions to be evaluated in a deterministic manner are declared as *rigid* (which forces deferred evaluation by rewriting), while functions providing for nondeterministic evaluation steps are declared as *flexible* (which enables narrowing steps).

Similarly to [18], we present an abstract representation for programs in which the definitional trees (used to guide the needed narrowing strategy) are made explicit by means of case constructs. Moreover, here we distinguish two kinds of case expressions in order to make also explicit the flexible/rigid evaluation annotations. In particular, we assume that all functions are defined by one rule whose left-hand side contains only variables as parameters and the right-hand side contains case expressions for pattern-matching. Thanks to this new representation, we can define a simple operational semantics, which will become essential to simplify the definition of the associated PE scheme. The syntax for programs in the abstract representation is summarized as follows:

$\mathcal{R} ::= D_1 \ldots D_m$	$t ::= v$	(variable)
$D ::= f(v_1, \ldots, v_n) = t$	$\mid \; c(t_1, \ldots, t_n)$	(constructor)
	$\mid \; f(t_1, \ldots, t_n)$	(function call)
$p ::= c(v_1, \ldots, v_n)$	$\mid \; \mathit{case}\ t_0\ \mathit{of}\ \{p_1 \to t_1; \ldots; p_n \to t_n\}$	(rigid case)
	$\mid \; \mathit{fcase}\ t_0\ \mathit{of}\ \{p_1 \to t_1; \ldots; p_n \to t_n\}$	(flexible case)

where $\mathcal{R}$ denotes a program, D a function definition, p a pattern and t an arbitrary expression. A program $\mathcal{R}$ consists of a sequence of function definitions D such that the left-hand side is linear and has only variable arguments, i.e., pattern matching is compiled into case expressions. The right-hand side of each function definition is a term t composed by variables, constructors, function calls, and case expressions. The form of a case expression is: $(f)\mathit{case}\ t\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to t_1, \ldots, c_k(\overline{x_{n_k}}) \to t_k\}$, where t is a term, $c_1, \ldots, c_k$ are different constructors of the type of t, and $t_1, \ldots, t_k$ are terms (possibly containing case expressions).

The variables $\overline{x_{n_i}}$ are called *pattern variables* and are local variables which occur only in the corresponding subexpression t_i. The difference between *case* and *fcase* shows up when the argument t is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression (which corresponds to narrowing). Functions defined only by *fcase* (resp. *case*) expressions are called *flexible* (resp. *rigid*). Thus, flexible functions act as generators (like predicates in logic programming) and rigid functions act as consumers. Concurrency is expressed by a built-in operator "&" which evaluates its two arguments concurrently. This operator can be defined by the rule: `True` & `True` = `True` and, hence, in the following we simply consider it as an ordinary function symbol.

Example 1. Consider the rules defining the (rigid) function " $\leqslant$ ":[2]

$$\begin{array}{rcll} \texttt{0} & \leqslant & \texttt{n} & = \texttt{True} \\ \texttt{(Succ m)} & \leqslant & \texttt{0} & = \texttt{False} \\ \texttt{(Succ m)} & \leqslant & \texttt{(Succ n)} & = \texttt{m} \leqslant \texttt{n} \end{array}$$

By using case expressions, they can be represented by the following rewrite rule:

$$\begin{array}{l} \texttt{x} \leqslant \texttt{y} = \texttt{ case x of } \{\texttt{0} \rightarrow \texttt{True;} \\ \qquad\qquad\qquad\quad \texttt{(Succ } x_1\texttt{)} \rightarrow \texttt{case y of } \{\texttt{0} \rightarrow \texttt{ False;} \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{(Succ } y_1\texttt{)} \rightarrow \texttt{ } x_1 \leqslant y_1\} \} \end{array}$$

Due to the presence of fresh pattern variables in the right-hand side of the rule, this is not a standard rewrite rule. Nevertheless, the reduction of a case expression binds these pattern variables so that they disappear during a concrete evaluation (see [18]).

3.2 The Residualizing Semantics

An automatic transformation from inductively sequential programs to programs using case expressions is introduced in [18]. They also provide an appropriate operational semantics for these programs: the LNT calculus (Lazy Narrowing with definitional Trees), which is equivalent to needed narrowing over inductively sequential programs. In this work, we consider functional logic languages with a more general operational principle, namely a combination of (needed) narrowing and residuation. Nevertheless, the translation method of [18] could be easily extended to cover programs containing evaluation annotations; namely, flexible (resp. rigid) functions are translated by using only *fcase* (resp. *case*) expressions. Moreover, the LNT calculus of [18] can be also extended to correctly evaluate *case*/*fcase* expressions. In the following, we refer to the LNT calculus to mean the LNT calculus of [18] extended to cope with *case*/*fcase* expressions (the formal definition can be found in [3]).

Unfortunately, by using the standard semantics during PE, we would have the same problems of previous approaches (see Sect. 1). In particular, one of the

[2] Although we consider in this work a first-order language, we use a curried notation in the examples (as is usual in functional languages).

main problems comes from the *backpropagation* of variable bindings to the left-hand sides of residual rules. In the context of lazy (call-by-name) functional logic languages, this can provoke an incorrect restriction on the domain of functions (regarding the ability to compute head normal forms) and, thus, the loss of correctness for the transformation whenever some term in head normal form is evaluated during PE. The following example illustrates this point.

Example 2. Consider the following program:

```
isZero 0              =  True
nonEmptyList (x : xs) =  True
foo x                 =  isZero x : []
```

Here we use "[]" and ":" as constructors of lists, and "0" and "Succ" to define natural numbers. Then, given the (unique) computation for `foo y`:

$$\texttt{foo y} \;\leadsto_{\{\}}\; \texttt{(isZero y)} : [] \;\leadsto_{\{\texttt{y}\mapsto\texttt{0}\}}\; \texttt{True} : []$$

where `(isZero y) : []` is in head normal form, we get the residual rule:

```
foo 0 = True : []
```

However, the expression `nonEmptyList (foo (Succ 0))` can be evaluated to `True` in the original program (reduced functions are underlined):

$$\begin{array}{l}\texttt{nonEmptyList (}\underline{\texttt{foo}}\texttt{ (Succ 0))} \leadsto_{\{\}} \underline{\texttt{nonEmptyList}}\texttt{ (isZero (Succ 0) : [])} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \leadsto_{\{\}} \texttt{True}\end{array}$$

whereas it is not possible if the residual rule for `foo` is used (together with the original definitions for `isZero` and `nonEmptyList`).

The restriction on forbidding the evaluation of head normal forms can drastically reduce the optimization power of the transformation in some cases. Therefore, we propose a *residualizing* version of the LNT calculus which allows us to avoid this restriction. In the new calculus, variable bindings are encoded by case expressions (and are considered "residual" code). The inference rules of the new calculus, RLNT (Residualizing LNT), can be seen in Fig. 1. Let us explain the inference rules defining the one-step relation $\Rightarrow$. We note that the symbols "$[\![$" and "$]\!]$" in an expression like $[\![t]\!]$ are purely syntactical (i.e., they do not denote "the value of t"). Indeed, they are only used to *guide* the inference rules and, most importantly, to mark which part of an expression can be still evaluated (within the square brackets) and which part must be definitively residualized (not within the square brackets). Let us briefly describe the rules of the calculus:

HNF. The HNF (Head Normal Form) rules are used to evaluate terms in head normal form. If the expression is a variable or a constructor constant, the square brackets are removed and the evaluation process stops. Otherwise, the evaluation proceeds with the arguments. This evaluation can be made in a don't care nondeterministic manner. Note, though, that this source of nondeterminism can be easily avoided by considering a fixed selection rule, e.g., by selecting the leftmost argument which is not a constructor term.

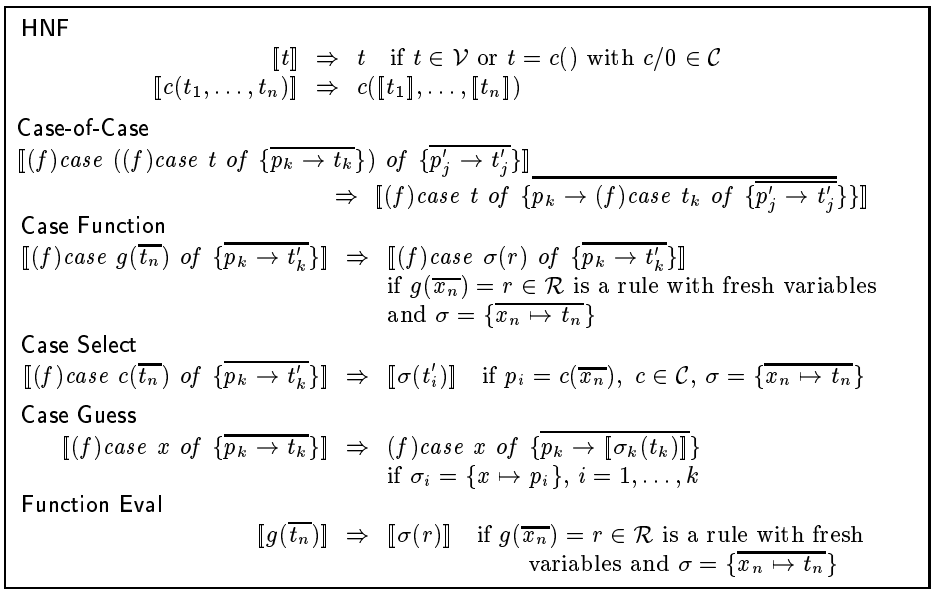

Fig. 1. RLNT Calculus

Case-of-Case. This rule moves the outer case inside the branches of the inner one. Rigorously speaking, this rule can be expanded into four rules (with the different combinations for *case* and *fcase* expressions), but we keep the above (less formal) presentation for simplicity. Observe that the outer case expression may be duplicated several times, but each copy is now (possibly) scrutinizing a known value, and so the Case Select rule can be applied to eliminate some case constructs.

Case Function. This rule can be only applied when the argument of the case is operation-rooted. In this case, it allows the *unfolding* of the function call.

Case Guess. It represents the main difference w.r.t. the standard LNT calculus. In order to imitate the instantiation of variables in needed narrowing steps, this rule is defined in the standard LNT calculus as follows:

$$[\![fcase\ x\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{\sigma} [\![\sigma(t_i)]\!] \quad \text{if } \sigma = \{x \mapsto p_i\},\ i = 1,\ldots,k$$

However, in this case, we would inherit the limitations of previous approaches. Therefore, it has been modified in order not to backpropagate the bindings of variables. In particular, we "residualize" the case structure and continue with the evaluation of the different branches (by applying the corresponding substitution in order to propagate bindings forward in the computation). Note that, due to this modification, no distinction between flexible and rigid case expressions is needed in the RLNT calculus.

Function Eval. This rule performs the unfolding of a function call. As in proof procedures for logic programming, we assume that we take a program rule with fresh variables in each such evaluation step.

In contrast to the standard LNT calculus, the inference system of Fig. 1 is completely deterministic, i.e., there is no don't know nondeterminism involved in the computations. This means that only one derivation can be issued from a given term (thus, there is no need to introduce a notion of RLNT "tree").

Example 3. Consider the well-known function `app` to concatenate two lists:

$$\begin{array}{l}\texttt{app x y} = \texttt{case x of } \{\ [] \quad\ \to \texttt{y}\ ; \\ \qquad\qquad\qquad\quad (\texttt{a} : \texttt{b}) \to \texttt{a} : (\texttt{app b y})\ \ \}\end{array}$$

Given the call `app (app x y) z` to concatenate three lists, we have the following (partial) derivation using the rules of the RLNT calculus:

$$\begin{array}{l}[\![\texttt{app (app x y) z}]\!] \\ \Rightarrow [\![\texttt{case (app x y) of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}]\!] \\ \Rightarrow [\![\texttt{case (case x of } \{[] \to \texttt{y};\ (\texttt{a}' : \texttt{b}') \to (\texttt{a}' : \texttt{app b}' \texttt{ y})\}) \\ \qquad\quad \texttt{of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}]\!] \\ \Rightarrow [\![\texttt{case x of } \{\ [] \quad\ \to \texttt{case y of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}; \\ \qquad\qquad\quad (\texttt{a}' : \texttt{b}') \to \texttt{case } (\texttt{a}' : \texttt{app b}' \texttt{ y}) \texttt{ of } \{[] : \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}]\!] \\ \Rightarrow \texttt{case x of } \{\ [] \quad\ \to [\![\texttt{case y of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}]\!]; \\ \qquad\qquad\quad (\texttt{a}' : \texttt{b}') \to [\![\texttt{case } (\texttt{a}' : \texttt{app b}' \texttt{ y}) \texttt{ of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}]\!] \\ \Rightarrow^* \texttt{case x of } \{\ [] \quad\ \to \texttt{case y of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : [\![\texttt{app b z}]\!])\}; \\ \qquad\qquad\quad (\texttt{a}' : \texttt{b}') \to [\![\texttt{case } (\texttt{a}' : \texttt{app b}' \texttt{ y}) \texttt{ of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}]\!] \\ \Rightarrow^* \texttt{case x of } \{\ [] \quad\ \to \texttt{case y of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : [\![\texttt{app b z}]\!])\}; \\ \qquad\qquad\quad (\texttt{a}' : \texttt{b}') \to (\texttt{a}' : [\![\texttt{app } (\texttt{app b}' \texttt{ y}) \texttt{ z}]\!])\}\end{array}$$

The resulting RLNT calculus shares many similarities with the driving mechanism of [27] and Wadler's deforestation [28] (although we obtained it independently by refining the original LNT calculus to avoid the backpropagation of bindings). The main differences w.r.t. the driving mechanism are that we include the Case-of-Case rule and that driving is defined also for `if_then_else` constructs (which can be expressed in our representation by means of case expressions). The main difference w.r.t. deforestation is revealed in the Case Guess rule, where the patterns p_i are substituted in the different branches, like in the driving transformation. Although it may seem only a slight difference, situations may arise during transformation in which our calculus (as well as the driving mechanism) takes advantage of the sharing between different arguments while deforestation may not (see [27]).

A common restriction in related program transformations is to forbid the unfolding of function calls using program rules whose right-hand side is not linear. This avoids the duplication of calls under an eager (call-by-value) semantics or under a lazy (call-by-name) semantics implementing the *sharing* of common variables. Since our computation model is based on a lazy semantics, which does not consider the sharing of variables, we cannot incur into the risk of duplicated computations. Nevertheless, if sharing is considered (as in, e.g., the language Curry), this restriction can be implemented by requiring right-linear program rules to apply the Case Function and Function Eval rules.

Regarding the PE of programs with *flexible/rigid* evaluation annotations, [2] introduced a special treatment in order to correctly infer the evaluation annotations for residual definitions. Within this approach, one is forced to split resultants by introducing several intermediate functions in order not to mix bindings which come from the evaluation of flexible and rigid functions. Moreover, to avoid the creation of a large number of intermediate functions, only the computation of a single needed narrowing step for suspended expressions is allowed. Now, by using case expressions (instead of functions defined by patterns as in [2]), we are able to proceed the specialization of suspended expressions beyond a single needed narrowing step without being forced to split the associated resultant (and hence without increasing the size of the residual program). This is justified by the fact that case constructs preserve the rigid or flexible nature of the functions which instantiate the variables.[3] The following example is taken from [2] and illustrates that the use of case constructs to represent function definitions simplifies the residual program.

Example 4. Consider a program and its PE for the term `f x (g y (h z))`, according to the technique introduced in [2]:

```
f 0 (Succ 0) = 0          % flex     f' 0 Y Z          = f'_1 Y Z  % flex
g 0 0        = (Succ 0)   % rigid    f'_1 (Succ 0) Z   = f'_2 Z    % rigid
h 0          = 0          % flex     f'_2 0            = f'_3      % flex
                                     f'_3              = 0         % flex
```

where `f x (g y (h z))` is renamed as `f' x y z`. The original program can be translated to our abstract representation as follows:

```
f x y  =  fcase x of {0 → fcase y of {(Succ 0) → 0}}
g x y  =  case x of {0 → case y of {0 → (Succ 0)}}
  h x  =  fcase x of {0 → 0}
```

The following PE for `f x (g y (h z))`, constructed by using the rules of the RLNT calculus, avoids the introduction of three intermediate rules and, thus, is notably simplified:

```
f' x y z = fcase x of {0 → case y of {(Succ 0) → fcase z of {0 → 0}}}
```

The next result establishes a precise equivalence between the standard semantics (the LNT calculus) and its residualizing version. In the following, we denote by $\Rightarrow_{Guess}$ the application of the following rule from the standard semantics:

$$[\![fcase\ x\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{\sigma}_{Guess} [\![\sigma(t_i)]\!] \quad \text{if } \sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k$$

Furthermore, we denote by $del_{sq}(t)$ the expression which results from t by deleting all the occurrences of "$[\![$" and "$]\!]$" (if any).

Theorem 1. *Let t be a term, $V \supseteq \mathcal{V}ar(t)$ a finite set of variables, d a constructor term, and $\mathcal{R}$ a program in the abstract representation. For each LNT*

[3] Indeed, the treatment for *case/fcase* expressions is the same in the RLNT calculus.

derivation $[\![t]\!] \stackrel{*}{\Rightarrow}^{\sigma} d$ *for* t *w.r.t.* $\mathcal{R}$ *computing the answer* σ*, there exists a RLNT derivation* $[\![t]\!] \Rightarrow^{*} t'$ *for* t *w.r.t.* $\mathcal{R}$ *such that there is a finite sequence* $[\![del_{sq}(t')]\!] \Rightarrow^{\sigma_1}_{Guess} \ldots \Rightarrow^{\sigma_n}_{Guess} d$*, where* $\sigma_n \circ \ldots \circ \sigma_1 = \sigma\ [V]$*, and vice versa.*

Roughly speaking, for each (successful) LNT derivation from t to a constructor term d computing σ, there is a corresponding RLNT derivation from t to t' in which the computed substitution σ is encoded in t' by case expressions and can be obtained by a (finite) sequence of $\Rightarrow_{Guess}$ steps (deriving the same value d).

4 Control Issues for Partial Evaluation

Following [12], a simple *on-line* PE algorithm can proceed as follows. Given a term t and a program $\mathcal{R}$, we compute a finite (possibly incomplete) RLNT derivation $t \Rightarrow^{+} s$ for t w.r.t. $\mathcal{R}$.[4] Then, this process is iteratively repeated for any subterm which occurs in the expression s and which is not *closed* w.r.t. the set of terms already evaluated. Informally, the *closedness* condition guarantees that each call which might occur during the execution of the residual program is covered by some program rule. If this process terminates, it computes a set of partially evaluated terms S such that the closedness condition is satisfied and, moreover, it uniquely determines the associated residual program.

First, we formalize the notion of closedness adjusted to our abstract representation.

Definition 1. *Let* S *be a set of terms and* t *be a term. We say that* t *is* S*-closed if* $closed(S,t)$ *holds, where the relation "closed" is defined inductively as follows:*

$$closed(S,t) = \begin{cases} true & \text{if } t \in \mathcal{V} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \text{if } t = c(t_1,\ldots,t_n),\ c \in \mathcal{C} \\ closed(t') \wedge \bigwedge_{i\in\{1,\ldots,k\}} closed(t_i) & \text{if } t = (f)case\ t'\ of\ \{\overline{p_k \to t_k}\} \\ \bigwedge_{t' \in \mathcal{R}an(\theta)} closed(S,t') & \text{if } \exists s \in S \text{ such that } t = \theta(s) \end{cases}$$

A set of terms T *is* S*-closed, written* $closed(S,T)$*, if* $closed(S,t)$ *holds for all* $t \in T$*.*

According to this definition, variables are always closed, while an operation-rooted term is S-closed if it is an instance of some term in S and the terms in the matching substitution are recursively S-closed. On the other hand, for constructor-rooted terms and for case expressions, we have two nondeterministic ways to proceed: either by checking the closedness of their arguments or by proceeding as in the case of an operation-rooted term. For instance, a case expression such as *case* t *of* $\{p_1 \to t_1, \ldots, p_k \to t_k\}$ can be proved closed w.r.t. S either by checking that the set $\{t, t_1, \ldots, t_k\}$ is S-closed[5] or by testing whether the whole case expression is an instance of some term in S.

[4] Note that, since the RLNT calculus is deterministic, there is no branching. Thus, only a single derivation can be computed from a term.

[5] Patterns are not considered here since they are constructor terms and hence closed by definition.

Example 5. Let us consider the following set of terms:

$$S = \{\texttt{app a b}, \texttt{case (app a b) of } \{[] \to \texttt{z}; (\texttt{x} : \texttt{y}) \to \texttt{(app y z)}\}\ \} .$$

The following expression $\texttt{case (app a}'\ \texttt{b}')\ \texttt{of } \{[] \to \texttt{z}'; (\texttt{x}' : \texttt{y}') \to \texttt{(app y}'\ \texttt{z}')\}$ can be proved S-closed using the first element of the set (by checking that the subterms $\texttt{app a}'\ \texttt{b}'$ and $\texttt{app y}'\ \texttt{z}'$ are instances of $\texttt{app a b}$) or by testing that the whole expression is an instance of the second element of the set.

The PE algorithm outlined above involves two control issues: the so-called *local* control, which concerns the computation of partial evaluations for single terms, and the *global* control, which ensures the termination of the iterative process but still guaranteeing that the closedness condition is eventually reached. Following [12], we present a PE procedure which is parameterized by:

- An unfolding rule $\mathcal{U}$ (local control), which determines how to stop RLNT derivations. Formally, $\mathcal{U}$ is a (total) function from terms to terms such that, whenever $\mathcal{U}(s) = t$, then there exists a *finite* RLNT derivation $[\![s]\!] \Rightarrow^+ t$.
- An abstraction operator *abstract* (global control), which keeps the set of partially evaluated terms finite. It takes two sets of terms S and T (which represent the current partially evaluated terms and the terms to be added to this set, respectively) and returns a *safe* approximation of $S \cup T$. Here, by "safe" we mean that each term in $S \cup T$ is closed w.r.t. the result of $abstract(S, T)$.

Definition 2. *Let $\mathcal{R}$ be a program and T a finite set of expressions. We define the PE function $\mathcal{P}$ as follows:*

$$\mathcal{P}(\mathcal{R}, T) = S \quad \textit{if } abstract(\{\}, T) \longmapsto_{\mathcal{P}}^* S \quad \textit{and} \quad S \mapsto\!\!\!\!\to_{\mathcal{P}} S$$

where $\mapsto\!\!\!\!\to_{\mathcal{P}}$ is defined as the smallest relation satisfying

$$\frac{S' = \{s' \mid s \in S \ \wedge\ \mathcal{U}(s) = s'\}}{S \mapsto\!\!\!\!\to_{\mathcal{P}} abstract(S, S')}$$

We note that the function $\mathcal{P}$ does not compute a partially evaluated program, but a set of terms S from which a S-closed PE can be uniquely constructed using the unfolding rule $\mathcal{U}$. To be precise, for each term $s \in S$ with $\mathcal{U}(s) = t$, we produce a residual rule $s = t$. Moreover, in order to ensure that the residual program fulfills the syntax of our abstract representation, a renaming of the partially evaluated calls is necessary. This can be done by applying a standard post-processing renaming transformation. We do not present the details of this transformation here but refer to [3].

As for local control, a number of well-known techniques can be applied for ensuring the finiteness of RLNT derivations, e.g., depth-bounds, loop-checks, well-founded (or well-quasi) orderings (see, e.g., [8, 23, 26]). For instance, an unfolding rule based on the use of the homeomorphic embedding ordering has been proposed in [4].

As for global control, an abstraction operator should essentially distinguish the same cases as in the closedness definition. Intuitively, the reason is that the

abstraction operator must first check whether a term is closed and, if not, try to add this term (or some of its subterms) to the set. Therefore, given a call $abstract(S, \{t\})$, an abstraction operator usually distinguishes three main cases depending on t:

- if t is constructor-rooted, it tries to add the arguments of t;
- if it is operation-rooted and is an instance of some term in S, it tries to add the terms in the matching substitution;
- otherwise (an operation-rooted term which is not an instance of any term in S), it is simply added to S (or *generalized* in order to keep the set S finite).

Our particular abstraction operator uses a *quasi-ordering*, namely the homeomorphic embedding relation $\trianglelefteq$ (see, e.g., [23]), to ensure termination and generalizes those calls which do not satisfy this ordering by using the *msg* (*most specific generalization*) between terms.[6]

As opposed to previous abstraction operators [4], here we need to give a special treatment to case expressions. Of course, if one considers the *case* symbol as an ordinary constructor symbol, the extension would be straightforward. Unfortunately, this will often provoke a serious loss of specialization, as the following example illustrates.

Example 6. Let us consider again the program `app` and the RLNT derivation of Example 3:

$$[\![\texttt{app (app x y) z}]\!]$$
$$\Rightarrow^* [\![\texttt{case (case x of } \{[] \to \texttt{y};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b y})\}) \texttt{ of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}]\!]$$
$$\Rightarrow^* \texttt{case x of } \{\ [] \to \texttt{case y of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : [\![\texttt{app b z}]\!])\};\ (\texttt{a}' : \texttt{b}') \to (\texttt{a}' : [\![\texttt{app (app b}' \texttt{ y) z}]\!])\}$$

If one considers an unfolding rule which stops the derivation at the intermediate case expression, then the abstraction operator will attempt to add only the operation-rooted subterms `app b' y` and `app b y` to the set of terms to be specialized. This will prevent us from obtaining an efficient (recursive) residual function for the original term, since we will never reach again an expression containing `app (app x y) z` (see Example 7).

On the other hand, by treating case expressions as operation-rooted terms, the problem is not solved. For instance, if we consider that the unfolding rule returns the last term of the above derivation, then it is not convenient to add the whole term to the current set. Here, the best choice would be to treat the *case* symbol as a constructor symbol. Moreover, a similar situation arises when considering constructor-rooted terms, since the RLNT calculus has no restrictions to evaluate terms in head normal form.

[6] A *generalization* of the set of terms $S = \{t_1, \ldots, t_n\}$ is a pair $\langle t, \{\theta_1, \ldots, \theta_n\}\rangle$ such that, $\forall i \in \{1, \ldots, n\}$, $\theta_i(t) = t_i$. The pair $\langle t, \{\theta_1, \ldots, \theta_n\}\rangle$ is the *most specific generalization* of S, written $msg(S)$, if $\langle t, \{\theta_1, \ldots, \theta_n\}\rangle$ is a generalization and for every other generalization $\langle t', \{\theta'_1, \ldots, \theta'_n\}\rangle$ of S, t' is more general than t.

Luckily, the RLNT calculus gives us some leeway. The key idea is to take into account the position of the square brackets of the calculus: an expression within square brackets should be added to the set of partially evaluated terms (if possible), while expressions which are not within square brackets should be definitively residualized (i.e., ignored by the abstraction operator, except for operation-rooted terms).

Definition 3. *Given two finite sets of terms, T and S, we define:*[7]

$$abstract(S,T) \;=\; \begin{cases} S & \text{if } T = \varnothing \\ abs(\ldots abs(S,t_1),\ldots,t_n) & \text{if } T = \{t_1,\ldots,t_n\}, n \geq 1 \end{cases}$$

The function $abs(S,t)$ distinguishes the following cases:

$$abs(S,t) \;=\; \begin{cases} S & \text{if } t \in \mathcal{V} \\ abstract(S,\{t_1,\ldots,t_n\}) & \text{if } t = c(t_1,\ldots,t_n),\ c \in \mathcal{C} \\ abstract(S,\{t',t_1,\ldots,t_n\}) & \text{if } t = (f)case\ t'\ of\ \{\overline{p_n \to t_n}\} \\ try_add(S,t) & \text{if } t = f(t_1,\ldots,t_n),\ f \in \mathcal{F} \\ try_add(S,t') & \text{if } t = [\![t']\!] \end{cases}$$

Finally, the function $try_add(S,t)$ is defined as follows:

$$try_add(S,t) \;=\; \begin{cases} abstract(S \setminus \{s\}, \{s'\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)) & \\ \qquad \text{if } \exists s \in S.\ root(s) = root(t) \text{ and } s \trianglelefteq t, & \\ \qquad \text{where } \langle s', \{\theta_1,\theta_2\}\rangle = msg(\{s,t\}) & \\ S \cup \{t\} \qquad \text{otherwise} & \end{cases}$$

Let us informally explain this definition. Given a set of terms S, in order to add a new term t, the abstraction operator *abs* distinguishes the following cases:

- variables are disregarded;
- if t is rooted by a constructor symbol or by a case symbol, then it recursively inspects the arguments;
- if t is rooted by a defined function symbol or it is enclosed within square brackets, then the abstraction operator tries to add it to S with *try_add* (even if it is constructor-rooted or a case expression). Now, if t does not embed any *comparable* (i.e., with the same root symbol) term in S, then t is simply added to S. Otherwise, if t embeds some comparable term of S, say s, then the *msg* of s and t is computed, say $\langle s', \{\theta_1,\theta_2\}\rangle$, and it finally attempts to add s' as well as the terms in θ_1 and θ_2 to the set resulting from removing s from S.

Let us consider an example to illustrate the complete PE process.

Example 7. Consider the program $\mathcal{R}_{\texttt{app}}$ which contains the rule defining the function `app`. In order to compute $\mathcal{P}(\mathcal{R}_{\texttt{app}}, \{\texttt{app (app x y) z}\})$, we start with:

$$\texttt{S}_0 = abstract(\{\}, \{\texttt{app (app x y) z}\}) = \{\texttt{app (app x y) z}\}$$

[7] The particular order in which the elements of T are added to S by *abstract* cannot affect correctness but can degrade the effectiveness of the algorithm. A more precise treatment can be easily given by using sequences instead of sets of terms.

For the first iteration, we assume that:

$$\mathcal{U}(\texttt{app (app x y) z}) =$$
$$\texttt{case x of } \{\ [] \rightarrow \texttt{case y of } \{[] \rightarrow \texttt{z};\ (\texttt{a} : \texttt{b}) \rightarrow (\texttt{a} : [\![\texttt{app b z}]\!])\};$$
$$(\texttt{a}' : \texttt{b}') \rightarrow (\texttt{a}' : [\![\texttt{app (app b}' \texttt{ y) z}]\!])\}$$

(see derivation in Example 3). Then, we compute:

$$\mathtt{S}_1 = \mathit{abstract}(\mathtt{S}_0, \{\mathcal{U}(\texttt{app (app x y) z})\}) = \{\texttt{app (app x y) z}, \texttt{app b z}\}$$

For the next iteration, we assume that:

$$\mathcal{U}(\texttt{app b z}) = \texttt{case b of } \{[] \rightarrow \texttt{z};\ (\texttt{c} : \texttt{d}) \rightarrow \texttt{c} : [\![\texttt{app d z}]\!]\ \}$$

Therefore, $\mathit{abstract}(\mathtt{S}_1, \{\mathcal{U}(\texttt{app b z})\}) = \mathtt{S}_1$ and the process finishes. The associated residual rules are (after renaming the original expression by `dapp x y z`):

$$\begin{aligned}
\texttt{dapp x y z} &= \texttt{case x of } \{\ [] \rightarrow \texttt{case y of } \{[] \rightarrow \texttt{z};\\
&\qquad\qquad (\texttt{a} : \texttt{b}) \rightarrow (\texttt{a} : \texttt{app b z})\};\\
&\qquad (\texttt{a}' : \texttt{b}') \rightarrow (\texttt{a}' : \texttt{dapp b}' \texttt{ y z})\}\\
\texttt{app b z} &= \texttt{case b of } \{\ [] \rightarrow \texttt{z};\ (\texttt{c} : \texttt{d}) \rightarrow (\texttt{c} : \texttt{app d z})\}
\end{aligned}$$

Note that the optimized function `dapp` is able to concatenate three lists by traversing the first list only once, which is not possible in the original program.

The following proposition states that the operator *abstract* of Def. 3 is *safe*.

Proposition 1. *Given two finite sets of terms, T and S, if $S' = \mathit{abstract}(S, T)$, then for all $t \in (S \cup T)$, t is closed with respect to S'.*

Finally, we establish the termination of the complete PE process:

Theorem 2. *Let $\mathcal{R}$ be a program and S a finite set of terms. The computation of $\mathcal{P}(\mathcal{R}, S)$ terminates using a finite unfolding rule and the abstraction operator of Def. 3.*

5 Experimental Evaluation

In order to assess the practicality of the ideas presented in this work, the implementation of a partial evaluator for the multi-paradigm declarative language Curry has been undertaken.[8] Curry [19] integrates features from logic (logic variables, partial data structures, built-in search), functional (higher-order functions, demand-driven evaluation) and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). Furthermore, Curry is a complete programming language which is able to implement distributed applications (e.g. Internet servers [16]) or graphical user interfaces at a high-level [17]. In order to develop an effective PE tool for Curry, one has to extend the basic PE scheme to cover all high-level features. This extension becomes impractical

[8] It is publicly available at `http://www.dsic.upv.es/users/elp/soft.html`.

Benchmark	mix	original	specialized	speedup
allones	470	430	290	1.48
double_app	510	370	320	1.16
double_flip	750	550	400	1.37
kmp	1440	730	35	20.9
length_app	690	310	290	1.07

Table 1. Benchmark results

within previous frameworks for the PE of functional logic languages due to the complexity of the resulting semantics. By using an abstract representation and translating high-level programs to this notation (see [20]), the extension becomes simple and effective. A detailed description of the concrete manner in which each feature is treated can be found in [3]. Moreover, as opposed to previous partial evaluators for Curry (e.g., INDY [1]), it is completely written in Curry. To the best of our knowledge, this is the first purely declarative partial evaluator for a functional logic language.

Firstly, we have benchmarked several examples which are typical from partial deduction and from the literature of functional program transformations. Table 1 shows the results obtained from some selected benchmarks (a complete description can be found, e.g., in [4]). For each benchmark, we show the specialization time including the reading and writing of programs (column mix), the timings for the original and specialized programs (columns original and specialized), and the speedups achieved (column speedup). Times are expressed in milliseconds and are the average of 10 executions on a Sun Ultra-10. Runtime input goals were chosen to give a reasonably long overall time. All benchmarks have been specialized w.r.t. function calls containing no static data, except for the kmp example (what explains the larger speedup produced). Speedups are similar to those obtained by previous partial evaluators, e.g., INDY [1]. Indeed, these benchmarks were used in [4] to illustrate the power of the narrowing-driven approach (and are not affected by the discussed limitations). This indicates that our new scheme for PE is a conservative extension of previous approaches on comparable examples. Note, though, that our partial evaluator is applicable to a wider class of programs (including higher-order, constraints, several built-in's, etc), while INDY is not.

Secondly, we have considered the PE of the collection of programs in the Curry library (see `http://www.informatik.uni-kiel.de/~curry`). Here, our interest was to check the ability of the partial evaluator to deal with realistic programs which make extensive use of all the features of the Curry language. Our partial evaluator has been successfully applied to all the examples producing in some cases significant improvements. We refer to [3] for the source code of some benchmarks. Finally, we have also considered the PE of a meta-interpreter w.r.t. a source program. Although the partial evaluator successfully specialized it, regarding improvement in efficiency, the results were not so satisfactory. To improve this situation, we plan to develop a binding-time analysis to determine,

for each expression, whether it can be definitively evaluated at PE time (hence, it should not be generalized by the abstraction operator) or whether this decision must be taken online. This kind of (*off-line*) analysis would be also useful to reduce specialization times.

Altogether, the experimental evaluation is encouraging and gives a good impression of the specialization achieved by our partial evaluator.

6 Conclusions

In this work, we introduce a novel approach for the PE of truly lazy functional logic languages. The new scheme is carefully designed for an abstract representation in which high-level programs can be automatically translated. We have shown how a non-standard (residualizing) semantics can avoid several limitations of previous frameworks. The implementation of a fully automatic PE tool for the language Curry has been undertaken and tested on an extensive benchmark suite. To the best of our knowledge, this is the first purely declarative partial evaluator for a functional logic language. Moreover, since Curry is an extension of both logic and (lazy) functional languages, we think that our PE scheme can be easily adapted to other declarative languages.

¿From the experimental results, we conclude that our partial evaluator is indeed suitable for "real" Curry programs. Anyway, there is still room for further improvements. For instance, although self-application is already (theoretically) possible, the definition of a precise binding-time analysis seems mandatory to achieve an *effective* self-applicable partial evaluator. On the other hand, we have not considered a formal treatment to measuring the *effectiveness* of our partial evaluator. Another promising direction for future work is the development of abstract criteria to formally measure the potential benefit of our PE algorithm.

References

1. E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical report, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
2. E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. of the 6th Int'l Conf. on Logic for Programming and Automated Reasoning, LPAR'99*, pages 376–395. Springer LNAI 1705, 1999.
3. E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. Technical report, UPV, 2000. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
4. 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.
5. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.
6. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 2000 (to appear). Previous version in *Proc. of POPL'94*, pages 268–279.

7. A. Bondorf. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of TAPSOFT'89*, pages 81–95. Springer LNCS 352, 1989.
8. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.
9. C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 493–501, 1993.
10. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
11. 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.
12. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM, New York, 1993.
13. R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation. Int'l Dagstuhl Seminar*, pages 137–160. Springer LNCS 1110, 1996.
14. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
15. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of POPL'97*, pages 80–93. ACM, New York, 1997.
16. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of PPDP'99*, pages 376–395. Springer LNCS 1702, 1999.
17. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *Int'l Workshop on Practical Aspects of Declarative Languages*, pages 47–62. Springer LNCS 1753, 2000.
18. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
19. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
20. M. Hanus, S. Antoy, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.2: User Manual. Available at `http://www.informatik.uni-kiel.de/~pakcs`, 2000.
21. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
22. Laura Lafave. *A Constraint-based Partial Evaluator for Functional Logic Programs and its Application.* PhD thesis, Department of Computer Science, University of Bristol, 1998.
23. M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In G. Levi, editor, *Proc. of SAS'98*, pages 230–245. Springer LNCS 1503, 1998.
24. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
25. A.P. Nemytykh, V.A. Pinchuk, and V.F. Turchin. A Self-Applicable Supercompiler. In *Proc. of Dagstuhl Sem. on Part. Evaluation*, pages 322–337. Springer LNCS 1110, 1996.
26. M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. MIT Press, 1995.
27. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
28. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.