A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages[*†]

Elvira Albert[§] Michael Hanus[¶] Germán Vidal[§]

Abstract

We present a practical partial evaluation scheme for multi-paradigm declarative languages combining features from functional, logic, and concurrent programming. In contrast to previous approaches, we consider an intermediate representation for programs into which source programs can be automatically translated. The use of this simplified representation, together with the appropriate control issues, make our partial evaluation scheme practically applicable to modern multi-paradigm declarative languages like Curry. An implementation of a partial evaluator for Curry programs has been undertaken. The partial evaluator allows the specialization of programs containing higher-order functions, calls to external functions, concurrent constraints, etc. Our partial evaluation tool is integrated in the PAKCS programming environment for the language Curry as a source-to-source transformation on intermediate programs. The partial evaluator is written in Curry itself. To the best of our knowledge, this is the first purely declarative partial evaluator for a multi-paradigm functional logic language.

1 Introduction

A partial evaluator is a program transformer which takes a program and part of its input data—the so-called *static* data—and tries to perform as many computations as possible with the given data. A partial evaluator returns a new, residual program—a *specialized* version of the original one—which hopefully runs more efficiently than the original program since those computations that depend only on the static data have been performed once and for all at partial evaluation time. Two main approaches exist when designing a partial evaluation

[*]A preliminary version of this work appeared in the Proceedings of FLOPS 2001 [7].

[†]This work has been partially supported by CICYT TIC 2001-2705-C03-01, by Acción Integrada Hispano-Alemana HA2001-0059, by Acción Integrada Hispano-Austriaca HU2001-0019, by Acción Integrada Hispano-Italiana HI2000-0161, and by the DFG under grant Ha 2457/1-2.

[§]DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, E-46022 Valencia, Spain. Email: `{ealbert,gvidal}@dsic.upv.es`

[¶]Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany. Email: `mh@informatik.uni-kiel.de`

tool: *online*, monolithic partial evaluators and *offline*, staged partial evaluators. Online transformers are essentially non-standard interpreters; they specialize programs by evaluating expressions while sufficient information is available and by generating residual code otherwise. The transformation is viewed as a one-phase process in which specialize/residualize decisions are taken on the fly. In contrast, offline partial evaluators have two separate phases: a static analysis—usually a binding-time analysis [36]—to detect which constructs of the source program can be evaluated with the known input data, followed by the proper specialization phase which is guided by the information gathered by the analysis. While partial evaluation of functional programs has mainly stressed the offline approach [35], partial evaluators for logic programs are usually online (e.g., [37, 39, 47]).

This work is concerned with the design and development of a practical *online* partial evaluation scheme which is applicable to realistic multi-paradigm declarative languages. The most recent proposal for multi-paradigm declarative programming advocates the integration of features from functional, logic, and concurrent programming [29, 31]. The resulting language includes the most important features of the distinct paradigms, e.g., lazy evaluation, higher-order functions, non-deterministic computations, concurrent evaluation of constraints with synchronization on logical variables, and a unified computation model which integrates *narrowing* and *residuation* [29]. Our partial evaluator takes a multi-paradigm declarative program and a set of partially instantiated calls and returns a new program optimized for all the instances of these calls. Optimization is mainly achieved by performing those computations that depend only on the known input data and by removing *redundant* computations. Apart from its ability to specialize programs, a powerful optimization achieved by our partial evaluator is the transformation of higher-order functions into first-order functions. This reduces the execution time and space requirements w.r.t. existing (e.g., [32]) implementations (see the example in the following section).

We propose a partial evaluation scheme which is based on a simple intermediate language into which programs written in a higher-level source language can be automatically translated. Our scheme relies on a non-standard semantics, the RLNT calculus [1, 6], which does not propagate bindings backwards but represents them within the evaluated expression by *residual* case expressions with a variable argument. For example, the expression

$\texttt{fcase x of } \{ \texttt{0} \mapsto \texttt{0} + \texttt{1} \}$

denotes the term $\texttt{0} + \texttt{1}$ with the associated binding $\{\texttt{x} \mapsto \texttt{0}\}$. We will show that both ingredients mixed together make our approach suitable to design practical partial evaluators for modern multi-paradigm functional logic languages like Curry [31] and Toy [34]. With this in mind, our main contributions are:

(1) *We properly extend the RLNT calculus to cover all the facilities provided by multi-paradigm declarative languages.* The original RLNT calculus was defined for a simple intermediate language, containing only function calls, constructors and case expressions [6]. The extension to cover the additional language features (e.g., higher-order functions, calls to external

functions, concurrent constraints, guarded rules, etc.) is far from being trivial, since the RLNT calculus does not compute bindings but represents them by residual case expressions. We will show how, under certain conditions, it is possible to float out case expressions so that bindings are correctly propagated, which is crucial to achieve a good level of specialization.

(2) *We define appropriate control strategies which take into account the particularities of the considered language and (non-standard) semantics.* The main novelty of our partial evaluation algorithm is the use of the extended RLNT calculus mentioned above to perform computations during partial evaluation. Within this calculus, we use the symbols "⟦" and "⟧" to enclose those parts of an expression which need further evaluation. These symbols turn out to be very helpful to *guide* the whole partial evaluation process, thus permitting a smooth transition between the different control levels of the algorithm.

(3) *We provide a complete implementation of the partial evaluator for the language Curry.* The practicality of our approach is witnessed by implementing a partial evaluator for Curry [31] written in Curry itself. Our partial evaluator is able to transform Curry programs including higher-order functions, calls to external functions, concurrent constraints, etc., in contrast to other existing partial evaluators (e.g., INDY [3]). The developed tool has been integrated into the PAKCS [32] programming environment for Curry as a fully automatic source-to-source transformation on intermediate programs.

The structure of this paper is as follows. In Section 2 we present an overview of the partial evaluation scheme. Section 3 presents the intermediate representation for programs and Section 4 introduces a non-standard semantics for such programs which is specially well-suited to perform computations at partial evaluation time. Control issues are dealt with in Section 5, while Section 6 presents an experimental evaluation of a partial evaluator for Curry programs. Some related works are discussed in Section 7 before we conclude in Section 8.

2 Overview of the Partial Evaluator

We start with an overview of the partial evaluator. Our partial evaluation tool constructs optimized, residual versions for some "parts" of an input program. Those "parts" to be partially evaluated are annotated in the program by means of the function `PEVAL`. For example, assume that we have a program (see below) including the following function definition:

```
main xs ys = (map (iter (+1) 2) xs) ++ ys
```

Then, we can annotate the expression "`map (iter (+1) 2) xs`" as follows:

```
main xs ys = (PEVAL (map (iter (+1) 2) xs)) ++ ys
```

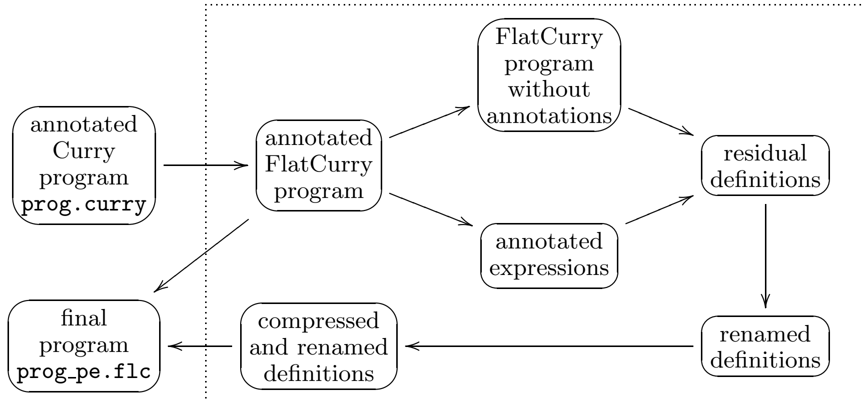

Figure 1: Overview of the Partial Evaluation Process

From a semantic point of view, `PEVAL` is the identity function. In other words, it is a technical device for the user to indicate which expressions of the source program should be optimized by partial evaluation. After annotating the program, the partial evaluation process is fully automatic (i.e., there is no need for user interaction) and always terminating. The definition of appropriate heuristics to annotate program calls is outside the scope of partial evaluation techniques and an interesting topic for future research (see also Section 8).

An implementation of the partial evaluator has been incorporated into the latest distribution of the PAKCS [32] programming environment for Curry. The process consists of the following phases (depicted in Figure 1):

- The process starts with a Curry program, `prog.curry`, where the expressions to be partially evaluated are annotated with `PEVAL`, as shown above. The function `PEVAL` is defined in the standard prelude as "`PEVAL x = x`."
- The source program is then translated into the FlatCurry format, a standard intermediate representation which is used during the compilation of Curry programs [16, 32, 43] (this representation is described in detail in Section 3).
- The process continues by extracting the set of annotated expressions and by creating a copy of the FlatCurry program without the `PEVAL` annotations. Both the program and the set of expressions are the input for the proper partial evaluation process. We do not further describe this process here since this is the subject of Section 5.
- The output of the partial evaluation process is a set of new, residual function definitions which are semantically equivalent and (potentially) more efficient than the original functions to execute the annotated expressions of the source program. These new definitions do not generally fulfill the syntax of FlatCurry and, therefore, a post-processing of *renaming* is ap-

plied. This is also useful to remove some redundant symbols from the residual definitions.

- Frequently, residual definitions contain a number of "useless" functions which are only used to pass control between two program points. Therefore, we finish the transformation process by applying a *compression* phase which is useful to produce more compact and legible definitions.
- The final program is obtained from the original one as follows: first, residual definitions are added to the original program; then, each marked expression (`PEVAL t`) of the original program is replaced by `t'`, where `t'` is the renaming of the expression `t` according to the post-processing renaming mentioned above. The optimized program is saved in a new file `prog_pe.flc` and loaded into the environment.

The partially evaluated program will be stored in FlatCurry format, in contrast to the original program which was written in Curry. This is not a restriction since FlatCurry programs are directly executable by the compiler of PAKCS and can be inspected by a standard display utility in the PAKCS environment.

Let us show a typical session with the partial evaluator. Here we consider the optimization of a program containing several calls to higher-order functions (since it is common to use higher-order combinators such as `map`, `foldr`, etc. in functional logic programs). Although the use of such functions makes programs concise, some overhead is introduced at run time. Hence, we apply our partial evaluator to optimize calls to these functions. As a concrete example, consider the following (annotated) Curry program:

```
main xs ys = (PEVAL (map (iter (+1) 2) xs)) ++ ys

iter f n = if n==0 then f else iter (comp f f) (n-1)
comp f g x = f (g x)

bench = let l,r free in
            l =:= [1..20000] &> r =:= evalTime (main l [])
```

stored in the file `prog.curry`. Here, function `comp` is a higher-order function to compose two input functions, while `iter` composes a given function 2^{n} times. Thus, given two input lists, `xs` and `ys`, function `main` adds 4 to each element of `xs`—the annotated expression—and then concatenates the result with the second list `ys`. For the remaining built-in functions of the language, e.g., `if then else`, `let`, `++` (concatenation), `&>` (sequential conjunction of constraints), etc., we refer the reader to the Curry report [31]. In order to measure the improvement achieved by the process, we have also included function `bench`, where `evalTime` is a system utility to measure the execution time of a function call. First, we load the program into PAKCS and execute function `bench`:

```
prelude> :l prog
Parsing prog.curry...
...
```

```
{compiled /tmp/prog.pl in module user, 1300 msec 832 bytes}
prog> bench
Runtime: 930 msec.
Elapsed time: 1174 msec.
Number of garbage collections: 14
```

Now, we run the partial evaluation tool and show the result of the process:

```
prog> :peval
Writing specialized program into "prog_pe.flc"...
Loading partially evaluated program "prog_pe"...
prog_pe> :show
No source program file available, generating source from
FlatCurry...

main xs ys = (map_pe0 xs) ++ ys

iter f n = if n==0 then f else iter (comp f f) (n-1)
comp f g x = f (g x)

bench = let l,r free in
            l =:= [1..20000] &> r =:= evalTime (main l [])

map_pe0 [] = []
map_pe0 (x : xs) = ((((x + 1) + 1) + 1) + 1) : map_pe0 xs
```

Only two modifications have been performed over the original program: the annotated expression has been replaced by a call to the new function `map_pe0` and the residual (first-order) definition of `map_pe0` has been added. In order to check the improvement achieved, we execute function `bench` again:

```
prog_pe> bench
EVALTIME: Runtime: 130 msec.
EVALTIME: Elapsed time: 177 msec.
EVALTIME: Number of garbage collections: 2
```

Thus, the new program runs approximately 7 times faster than the original one. The reason is that it has a first-order definition and is completely "deforested" [54] in contrast to the original definition. More experimental results can be found in Section 6.

3 The Flat Representation

This section begins by informally describing the considered source language. Then, we formally define an intermediate representation for source programs.

We consider multi-paradigm languages which integrate the most important features of functional and logic programming. To make things concrete, we

mainly follow the syntax of the language Curry [31]. In our source language, functions are defined by a sequence of rules (or equations) of the form

$f\ t_1 \ldots t_n$ = e

where $t_1, \ldots, t_n$ are *constructor* terms and the right-hand side e is an expression. Constructor terms may contain variables and constructor symbols, i.e., symbols which are not defined by the program rules. Functions can be also defined by *conditional equations* which have the form

$f\ t_1 \ldots t_n$ | c = e

where the condition (or *guard*) c can be either a Boolean function or a constraint. Elementary constraints are `success`, which is always satisfied, and *equational constraints* e_1 `=:=` e_2 between two expressions. The latter is satisfied if both expressions are reducible to a same ground constructor term (i.e., we consider the so-called *strict equality* [27, 45]). Operationally, an equational constraint e_1 `=:=` e_2 is solved by evaluating e_1 and e_2 to unifiable constructor terms. Higher-order features include partial function applications and lambda abstractions. Function application is denoted by juxtaposition of the function and its argument. The evaluation of higher-order calls containing free variables as functions is not allowed (i.e., such calls are suspended to avoid the use of higher-order unification [33]). Finally, we also consider the use of functions which are not defined in the user's program (*external* functions), like arithmetic operators, usual higher-order functions (`map`, `foldr`, etc.), basic input/output facilities, etc.

Example 3.1 *Consider the following rules defining a function to concatenate two lists (where* `[]` *denotes the empty list and* `x:xs` *a list with first element* `x` *and tail* `xs`*):*

```
conc eval flex
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys
```

The evaluation annotation "`eval flex`*" declares* `conc` *as a flexible function which can be also used to solve equations over functional expressions (see below). For instance, the equation "*`conc p s =:= [1,2,3]`*" is solved by instantiating the variables* `p` *and* `s` *to lists so that their concatenation yields the list* `[1,2,3]`*. Thus, we can define a constraint which is satisfied if* `p` *is a prefix of the list* `xs` *as follows ("*`let s free in`*" denotes the declaration of a local variable and corresponds to existential quantification):*

```
prefix p xs = let s free in  conc p s =:= xs
```

In order to show an example for higher-order programming, we define a higher-order constraint `satisfyAll` *which takes a unary constraint* `c` *and a list* `xs` *as input and is satisfied if all elements of* `xs` *satisfy the constraint* `c` *(the infix operator* `&` *denotes the conjunction of constraints):*

```
satisfyAll _ []     = success
satisfyAll c (x:xs) = c x & satisfyAll c xs
```

Now we can combine this definition with our previous definition of a prefix to compute a common prefix of a list of strings (strings are considered as lists of characters):

```
commonPrefix p xs = satisfyAll (prefix p) xs
```

For instance, the solutions for the constraint

```
commonPrefix p ["abc", "abda", "abab"]
```

are the instantiations `""`*,* `"a"`*, or* `"ab"` *for the variable* `p`*.*

The basic operational semantics of our source language is based on a combination of (needed) narrowing and residuation [29]. The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation. Residuation preserves the deterministic nature of functions and naturally supports concurrent computations. On the other hand, the *narrowing* mechanism allows the instantiation of free variables in input expressions and then applies reduction steps to the function calls of the instantiated expression. This instantiation is usually computed by unifying a subterm of the entire expression with the left-hand side of some program rule. To avoid unnecessary computations and to deal with infinite data structures, demand-driven generation of the search space has recently been advocated by a flurry of outside-in, lazy narrowing strategies (see, e.g., [15, 27, 41, 45]). Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* [15] is currently the best lazy narrowing strategy for functional logic programs.

The precise mechanism—narrowing or residuation—for each function is specified by *evaluation annotations*. The annotation of a function as *rigid* forces the delayed evaluation by rewriting, while functions annotated as *flexible* can be evaluated in a non-deterministic manner by applying narrowing steps. To provide concurrent computation threads, expressions can be combined by the *concurrent conjunction operator* "`&`," i.e., the expression e_1 `&` e_2 can be reduced by reducing either e_1 or e_2. The basic semantics can be properly extended to cover all the above features (see, e.g., [31, Appendix D] where the operational semantics of the language Curry is provided).

Example 3.2 *Residuation and concurrent computations are useful to model so-called* passive constraints *which can reduce the search space whenever their arguments are sufficiently known. For instance, consider the following definition:*

```
digit eval flex
digit 0 = success
...
digit 9 = success

arith x y = x+x=:=y & x*x=:=y & digit x
```

Here, `digit` *is a flexible function (constraint), i.e., a goal like* `digit x` *can be satisfied by instantiating* `x` *to one of the values* `0`*,...,*`9`*. On the other hand,*

the standard arithmetic functions like `+` *or* `*` *are rigid to avoid an unrestricted instantiation of their arguments. A goal like "*`arith x y`*" is evaluated by a reduction step to*

```
x+x=:=y & x*x=:=y & digit x
```

In this goal the constraints `x+x=:=y` *and* `x*x=:=y` *cannot be further evaluated since both functions* `+` *and* `*` *are rigid and their arguments are not yet known. The only evaluable part is* `digit x` *which can be non-deterministically reduced to* `success`*. For instance, consider the choice* $\{\mathtt{x} \mapsto 2\}$ *which leads to the following goal:*

```
2+2=:=y & 2*2=:=y & success
```

Now we can evaluate the subexpressions `2+2` *and* `2*2` *followed by solving both equational constraints which instantiates* `y` *to* `4`*. Thus, one computed solution is* $\{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 4\}$*. Another choice for* `digit x` *yields the second solution* $\{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0\}$*, and all further choices fail.*

As we mentioned before, online partial evaluators normally include a (non-standard) interpreter [21]. Hence, as the operational semantics gets more elaborated, the associated partial evaluation methods get also more complex. A promising approach successfully applied in other contexts (e.g., [18, 28, 46]) is to consider programs written in an intermediate programming language with a simple operational semantics and to automatically translate source-level programs into this intermediate language. Recently, Hanus and Prehofer [33] introduced such a simplified representation of functional logic programs based on the idea to "compile" definitional trees [14] (these are hierarchical structures used to guide the needed narrowing strategy) into specific rewrite rules. This provides more explicit control and leads to a calculus simpler than standard needed narrowing. This representation has been also extended to include information about the evaluation type of functions (flexible or rigid) in [1, 6]. In the following, we extend it in order to cover all the facilities we mentioned above. The syntax of the resulting representation is depicted in Figure 2. Following the terminology of [33], we refer to this representation as "the *flat* representation for programs."

Within this (first-order) representation, a flat program $\mathcal{R}$ consists of a sequence of function definitions D such that each function is defined by one rule whose left-hand side contains only different variables as parameters. The right-hand side is an expression e composed by variables (e.g., x, y, z,...), constructors (e.g., a, b, c,...), function calls (e.g., f, g,...), and case expressions for pattern matching. Additionally, we also allow external functions, higher-order features like partial application and an application of a functional expression to an argument, constraints (possibly containing existentially quantified variables), guarded expressions (to represent conditional rules, where the list of variables are the local variables which are visible in the guard and the right-hand side), and disjunctions (to represent functions with overlapping left-hand sides). Source-level programs (e.g., Curry programs) can be automatically translated to our flat representation; indeed, it essentially coincides with the standard

$$
\begin{array}{lcll}
\mathcal{R} & ::= & D_1 \ldots D_m & \\
D & ::= & f(x_1, \ldots, x_n) = e & \\
e & ::= & x & \text{(variable)} \\
 & | & c(e_1, \ldots, e_n) & \text{(constructor)} \\
 & | & f(e_1, \ldots, e_n) & \text{(function call)} \\
 & | & case\ e_0\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(rigid case)} \\
 & | & fcase\ e_0\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(flexible case)} \\
 & | & external(e) & \text{(external function call)} \\
 & | & partcall(f, e_1, \ldots, e_k) & \text{(partial application)} \\
 & | & apply(e_1, e_2) & \text{(application)} \\
 & | & constr([x_1, \ldots, x_n], e) & \text{(constraint)} \\
 & | & guarded([x_1, \ldots, x_n], e_1, e_2) & \text{(guarded expression)} \\
 & | & or(e_1, e_2) & \text{(disjunction)} \\
p & ::= & c(x_1, \ldots, x_n) &
\end{array}
$$

Figure 2: The Flat Representation for Programs

intermediate representation, FlatCurry, used during the compilation of Curry programs [16, 32, 43].

We distinguish two kinds of case expressions in order to cope with both flexible and rigid functions. The form of a case expression is the following:[1]

$$(f)case\ e\ of\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$$

where e is an expression, $c_1, \ldots, c_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are expressions, possibly containing $(f)case$'s. The variables $\overline{x_{n_i}}$ are *local* variables which occur only in the corresponding subexpression e_i. The difference between *case* and *fcase* only shows up when the argument e is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* non-deterministically binds this variable to a pattern in a branch of the case expression (which corresponds to narrowing). Functions defined only by *fcase* or *case* expressions are called *flexible* or *rigid*, respectively.

Example 3.3 *Consider the functions defined in Example 3.1. The function* `conc` *can be translated into the following flat representation:*[2]

```
conc xs ys = fcase xs of {[]    → ys ;
                          z:zs → z : conc zs ys }
```

Intuitively, the translation process is guided by the patterns in the left-hand sides of the rewrite rules. A precise definition can be found in [33]. As a further example, we show the translation of constraints and higher-order functions by the flat representation of the functions `prefix`*,* `satisfyAll`*, and* `commonPrefix`*:*

```
prefix p xs = constr [s] (conc p s =:= xs)
```

[1] We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

[2] Although we consider a first-order representation for flat programs, we use a curried notation in concrete examples.

```
satisfyAll c zs = case zs of
                    {[]   → success ;
                     x:xs → (apply c x) & (satisfyAll c xs) }

commonPrefix p xs = satisfyAll (partcall prefix p) xs
```

The standard operational semantics of flat programs is based on the LNT calculus [33] (Lazy Narrowing with definitional Trees). It was originally defined for flat programs whose right-hand sides contain only variables, constructors, function calls, and flexible case expressions. The extension of the LNT calculus to cope with flexible/rigid case expressions can be found in [1, 6].

4 The Residualizing Semantics

Following previous partial evaluation methods for functional logic programs [12, 13], residual rules are constructed from partial computations w.r.t. source programs as follows. Given an expression e and a (possibly incomplete) standard evaluation $e \Rightarrow^*_\sigma e'$ computing the substitution σ, we derive a residual rule—a *resultant*—of the form: $\sigma(e) = e'$. However, the backpropagation of bindings to the left-hand sides of residual rules (e.g., the instantiation of e by σ), introduces several problems:

- In general, the left-hand sides of resultants may become instantiated, which is not allowed by the syntax of Figure 2 (where only variable arguments are accepted in the left-hand sides of the rules). Therefore, some kind of post-processing is mandatory in order to recover a legal program.
- Evaluation annotations are complex to determine for residual functions or may even not exist (e.g., when bindings coming from the evaluation of both flexible and rigid functions have been computed, see [4]). Furthermore, the unrestricted propagation of bindings may destroy the *floundering* behaviour of the original program (i.e., it may introduce new suspensions which were not present in the original program and vice versa).
- As pointed out in [11, 13], the completeness of the transformation is only ensured if terms in *head normal form* (e.g., rooted by a constructor symbol) are not evaluated during partial evaluation, which may restrict the power of the method.

In order to overcome the above drawbacks, we propose the use of a non-standard, *residualizing* semantics on flat programs to perform partial computations. The main particularity of the new semantics, the RLNT calculus (which stands for Residualizing LNT calculus), is that bindings are not propagated backwards but represented by *case expressions with a variable argument*. Firstly, we will recall the basic RLNT calculus [6] to deal with expressions containing only variables, constructors, defined functions and case expressions. Later, we will introduce the appropriate extensions to cover all the remaining features of the flat representation.

HNF

$$[\![e]\!] \;\Rightarrow\; e \quad \text{if } e \in \mathcal{X} \text{ or } e = c() \text{ with } c \in \mathcal{C}$$

$$[\![c(e_1,\ldots,e_n)]\!] \;\Rightarrow\; c([\![e_1]\!],\ldots,[\![e_n]\!])$$

Case Eval

$$[\![(f)case\ e\ of\ \{\overline{p_k \to e_k}\}]\!] \;\Rightarrow\; \begin{cases} [\![(f)case\ e'\ of\ \{\overline{p_k \to e_k}\}]\!] & \text{if } [\![e]\!] \Rightarrow [\![e']\!] \\ (f)case\ e\ of\ \{\overline{p_k \to e_k}\} & \text{otherwise} \end{cases}$$
$$\text{if } e \neq case\ x\ of\ \{\ldots\},\ e \notin \mathcal{X} \text{ and } root(e) \notin \mathcal{C}$$

Case Select

$$[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}]\!] \;\Rightarrow\; [\![\sigma(e'_i)]\!] \quad \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C},\ \sigma = \{\overline{x_n \mapsto e_n}\}$$

Function Eval

$$[\![g(\overline{e_n})]\!] \;\Rightarrow\; [\![\sigma(r)]\!] \quad \text{if } g(\overline{x_n}) = r \in \mathcal{R} \text{ is a function definition with fresh variables and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Case Guess

$$[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!] \;\Rightarrow\; (f)case\ x\ of\ \{\overline{p_k \to [\![\sigma_k(e_k)]\!]}\}$$
$$\text{if } \sigma_i = \{x \mapsto p_i\},\ i = 1,\ldots,k$$

Case-of-Case

$$[\![(f)case\ ((f)case\ x\ of\ \{\overline{p_k \to e_k}\})\ of\ \{\overline{p'_j \to e'_j}\}]\!]$$
$$\Rightarrow\; [\![(f)case\ x\ of\ \{\overline{p_k \to (f)case\ e_k\ of\ \{\overline{p'_j \to e'_j}\}}\}]\!]$$

Figure 3: Basic RLNT Calculus

The basic inference rules of the RLNT calculus are depicted in Figure 3. Here, we denote by $\mathcal{X}$ the set of variables, by $\mathcal{C}$ the set of constructors, and by $\mathcal{F}$ the set of defined functions or operations. Given an expression e, function *root* returns the outermost function symbol of e (or it is undefined if e is a variable). The symbols "$[\![$" and "$]\!]$" in an expression like $[\![e]\!]$ do not denote a semantic function but are only used to identify which part of an expression should be still evaluated. Indeed, the introduction of these symbols allows the RLNT calculus to ignore those parts of an expression which are definitely residual (outside square brackets). Furthermore, as we will see in Section 5, they are also helpful to achieve a smooth transition between the local and global levels of the partial evaluation algorithm. We briefly describe the six rules of the basic RLNT calculus.

HNF. These rules can be applied when the considered term is in head normal form, i.e., it is a variable or a constructor-rooted term. They simply return the same term (if it is a value) or continue with the evaluation of all the arguments in parallel. This is safe in our context since the RLNT calculus does not compute bindings and, thus, there is no need to propagate bindings between arguments.

Case Eval. This rule initiates the evaluation of the case argument by creating a call for this subterm. If there is some progress towards the evaluation of the

case argument, i.e., the expression is not *suspended* (which is denoted by the fact that square brackets are removed), then we proceed to evaluate the resulting case expression; otherwise, the complete expression is suspended and we return the original case expression (without square brackets). Note that this rule excludes the evaluation of a case expression whose argument is either a variable, a constructor-rooted term, or another case expression with a variable argument, since it is dealt with in other rules (namely, in Case Guess, Case Select and Case-of-Case, respectively).

Case Select. It selects the appropriate branch of the current case expression and continues with the evaluation of this branch (here we assume for the sake of simplicity that the branches of a case expression are complete, i.e., there is always a matching branch; this can always be achieved by completing case expressions with "failure" branches of the form $p_k \rightarrow \texttt{fail}$).

Function Eval. This rule simply performs the unfolding of a function call. It is a purely functional unfolding since all arguments in the left-hand sides of the rules are variables.

Case Guess. This rule represents the main difference w.r.t. the standard semantics (i.e., the LNT calculus of [33]). In the standard semantics, these expressions are evaluated by means of the following rule:

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

Since we want to avoid the backpropagation of bindings, in the RLNT calculus 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). In other words, the new rule imitates the instantiation of variables in the standard evaluation of a flexible case but keeps the case structure. Due to this modification, no distinction between flexible and rigid case expressions is needed in the RLNT calculus. Moreover, the resulting calculus does not compute *answers*. Rather, they are represented in the derived expressions by means of case expressions with variable arguments. As a consequence, the calculus becomes deterministic, i.e., there is no don't-know non-determinism involved in the computations. This means that only one derivation can be issued from a given expression and, thus, there is no need to introduce a notion of RLNT "tree."

Case-of-Case. An undesirable effect of the residualizing Case Guess rule is that nested case expressions may suspend unnecessarily. Take, for instance, the expression:

$$\begin{array}{ll} [\![\texttt{case (case x of \{} & \texttt{0} \rightarrow \texttt{True;} \\ & \texttt{(Succ y)} \rightarrow \texttt{False\})} \;\; \texttt{of \{True} \rightarrow \texttt{C x\}}]\!] \end{array}$$

The evaluation of this expression suspends since the outer case can be only evaluated if the argument is a variable (Case Guess), a function call (Case Eval) or a constructor-rooted term (Case Select). To avoid such premature suspensions, the Case-of-Case rule moves the outer case inside the branches of the inner one and, thus, the evaluation of some branches can now proceed (similar rules can be found in the Glasgow Haskell Compiler as well as in Wadler's deforestation [54]). By using the Case-of-Case rule, the above expression can be reduced to:

$$[\![\texttt{case x of} \quad \{\texttt{0} \rightarrow \texttt{case True of } \{\texttt{True} \rightarrow \texttt{C x}\} \\ \qquad\qquad\qquad (\texttt{Succ y}) \rightarrow \texttt{case False of } \{\texttt{True} \rightarrow \texttt{C x}\}]\!]$$

which can be further simplified with the Case Guess and Case Select rules.[3] Rigorously speaking, this rule can be expanded into four rules (with the different combinations for *case* and *fcase*), 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.

Now, we present the extension of the basic RLNT rules in order to deal with all the remaining features of flat programs.

In principle, this extension could be done in a simple way. The naive idea is to treat all the additional features of the language as constructor symbols during partial evaluation. This is safe since these features are dealt with by the language environment and, thus, there is no special need to include associated residual definitions in partially evaluated programs. However, in realistic programs, the presence of these additional features is perfectly common, hence it is an unacceptable restriction just to residualize them. Moreover, our experimental tests have shown that no specialization is obtained in most cases if we follow this simple approach.

On the other hand, in order to allow the evaluation of the additional features at partial evaluation time—instead of residualizing them—, the RLNT calculus should be properly extended. Unfortunately, the extension of the basic RLNT calculus depicted in Figure 3 with the *standard semantics* (e.g., along the lines of [31, Appendix D]) is not a good solution either. The problem stems from the fact that the RLNT calculus only propagates bindings forward into the branches of a case expression. However, there are a number of functions, like equalities, (concurrent) conjunctions, some arithmetic functions, etc., in which the propagation of bindings between their arguments is crucial to achieve a good level of specialization. In order to propagate bindings[4] between different arguments, we permit to lift some flexible case expressions from argument positions to the top

[3] For the sake of simplicity, we have not included a failure rule for non-matching case expressions like "`case False of {True → C x}`" since such situations can be simply handled by replacing the corresponding expressions by "fail."

[4] Recall that bindings are represented by case expressions with a variable argument.

level while propagating the corresponding bindings to the remaining arguments. For example, the expression

$$[\![(\texttt{x} =:= \texttt{1}) \ \& \ (\texttt{fcase x of } \{\texttt{1} \to \texttt{success}\})]\!]$$

can be transformed into

$$[\![\texttt{fcase x of } \{\texttt{1} \to (\texttt{x} =:= \texttt{1} \ \& \ \texttt{success})\}]\!]$$

The transformed expression can be now evaluated by the Case Guess rule, thus propagating the binding $\{\texttt{x} \mapsto \texttt{1}\}$ to the first conjunct:

$$\texttt{fcase x of } \{\texttt{1} \to [\![\texttt{1} =:= \texttt{1} \ \& \ \texttt{success}]\!]\}$$

This transformation cannot be applied over arbitrary expressions since the intended semantics is only preserved when the involved functions fulfill some conditions. For instance, consider the following expression:

$$\texttt{case x of } \{\texttt{0} \mapsto \texttt{0}\} =:= \texttt{fcase x of } \{\texttt{0} \mapsto \texttt{0}\}$$

By assuming a fixed left-to-right evaluation of the equality arguments, the computation of this expression must suspend according to the standard semantics. However, by floating out the rightmost case expression, we obtain the derived expression:

$$\texttt{fcase x of } \{\texttt{0} \mapsto (\texttt{0} =:= \texttt{case 0 of } \{\texttt{0} \mapsto \texttt{0}\})\}$$

which no longer suspends. On the other hand, consider the following concurrent conjunction:

$$\texttt{case x of } \{\texttt{0} \mapsto \texttt{success}\} \ \& \ \texttt{fcase x of } \{\texttt{0} \mapsto \texttt{success}\}$$

which can be solved to `success` according to the standard semantics: first, we evaluate the second conjunct and propagate the corresponding binding to the first conjunct, and then we evaluate the (instantiated) first conjunct. In this case, by floating out the leftmost case expression, we obtain:

$$\texttt{case x of } \{\texttt{0} \mapsto (\texttt{success} \ \& \ \texttt{fcase x of } \{\texttt{0} \mapsto \texttt{success}\})\}$$

which gives rise to a suspension (with the standard semantics). To summarize, we need to distinguish two different kinds of functions:

1. functions in which the evaluation order of the arguments is fixed, and
2. functions in which the evaluation order of the arguments is dynamic and should be decided at run time.

Examples of the first kind of functions are: equalities, sequential conjunctions, constructor-rooted terms, calls to external functions (like arithmetic operators), etc. In many implementations, the evaluation of such functions proceeds by a fixed left-to-right evaluation of the arguments. Thus, we assume in the following a fixed left-to-right evaluation for these functions. As for the second kind of functions, we only find the concurrent conjunction operator. The evaluation of each kind of functions is performed as follows:

1. Fixed left-to-right evaluation: we can proceed with the evaluation of any argument, but only bindings coming from the evaluation of the leftmost one (which is not a constructor term) can be floated out.

2. Dynamic evaluation: we can proceed with the evaluation of any argument, but only flexible bindings (*fcase*) can be floated out.

The following sections present the appropriate treatment for each feature of the flat language according to the above restrictions.

4.1 Non User-Defined Functions

Flat programs may contain calls to functions which are not defined in the user's program code. Examples of these functions are those which are defined in the *prelude* of the language (i.e., a set of standard datatype and function definitions which are added to each program) or those which are implemented in another language.

Regarding the first kind of functions, we find some utility functions like `length`, `head`, and `tail`, the Boolean operators `&&` (sequential conjunction) and `||` (disjunction), the higher-order functions `map`, `foldr`, etc. An explicit treatment of these functions is not necessary since we have their definitions available at partial evaluation time (i.e., by loading them from the prelude file). Hence, we adopt a standard treatment by considering such functions as any other user-defined function.

Regarding the second kind of functions (which are denoted with the *external* construct), there are a number of arithmetic operators (e.g., $+$, $-$, $*$), the basic I/O facilities (e.g., `putChar`, `readChar`), etc. Such functions are executed only if all arguments are evaluated to ground constructor terms.[5] The same restriction seems reasonable when computing the partial evaluation of an external function. Therefore, their evaluation is formalized by the following rules:[6]

$[\![external(f(\overline{e_n}))]\!] \;\Rightarrow\; ext_call(f(\overline{e_n}))$
if $e_1, \ldots, e_n$ are ground constructor terms;

$[\![external(f(\overline{e_n}))]\!] \;\Rightarrow [\![(f)case\ x\ of\ \{\overline{p_k \to external(f(\overline{e_{i-1}}, e'_k, e_{i+1}, \ldots, e_n))}\}]\!]$
if $e_i = (f)case\ x\ of\ \{\overline{p_k \to e'_k}\}$ and it is the leftmost argument which is not ground constructor;

$[\![external(f(\overline{e_n}))]\!] \;\Rightarrow\; [\![external(f(e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n))]\!]$
if $\exists i \in \{1, \ldots, n\}$ such that $[\![e_i]\!] \;\Rightarrow\; e''_i$, $e'_i = del_{sq}(e''_i)$, and $e_i \neq e'_i$;

$[\![external(f(\overline{e_n}))]\!] \;\Rightarrow\; external(f(\overline{e_n}))$ otherwise.

[5] There are few exceptions to this general rule but typical external functions (like arithmetic operators) fulfill this condition. We assume it for the sake of simplicity.

[6] Since these rules are non-deterministic, we always require that the different cases are tried in their textual order and the arguments are evaluated from left to right.

Function $ext_call(e)$ evaluates e according to its predefined semantics. Also, we denote by $del_{sq}(e)$ the expression which results from deleting all occurrences of "$[\![$" and "$]\!]$" from e. We use it to test syntactic equality between expressions without taking into account the relative positions of "$[\![$" and "$]\!]$". Let us informally explain the rules above. Firstly, we try to execute the external function. If this is not possible because some argument is not a ground constructor term, we try to continue with the evaluation of some argument. This can be done by either floating out the leftmost argument which is not a constructor term (if it has a "binding" at the outermost position) or by applying the calculus recursively to some argument. Finally, if none of these rules is applicable, the computation suspends.

The only exception to the above rules are I/O actions. Current functional (logic) languages consider the monadic approach to I/O. Thus, these functions act on the current "state of the outside world." They are residualized since this state is not known at partial evaluation time.

4.2 Constraints

The treatment for constraints heavily depends on the associated constraint solver. As we mentioned in Section 3, we only consider *equational* constraints of the form $e_1 =:= e_2$ (which are solvable when both sides are reducible to unifiable constructor terms). This notion of equality is incorporated in our calculus by the following rules:

$[\![e_1 =:= e_2]\!] \;\Rightarrow\; case_\sigma(\mathtt{success})$
if e_1 and e_2 are unifiable constructor terms and $\sigma = mgu(e_1, e_2)$;

$[\![(f)case\ x\ of\ \{\overline{p_k \to e'_k}\} =:= e_2]\!] \;\Rightarrow\; [\![(f)case\ x\ of\ \{\overline{p_k \to e'_k =:= e_2}\}]\!]$

$[\![e_1 =:= (f)case\ x\ of\ \{\overline{p_k \to e'_k}\}]\!] \;\Rightarrow\; [\![(f)case\ x\ of\ \{\overline{p_k \to e_1 =:= e'_k}\}]\!]$
if e_1 is a constructor term;

$[\![e_1 =:= e_2]\!] \;\Rightarrow\; [\![e'_1 =:= e_2]\!]$ if $[\![e_1]\!] \;\Rightarrow\; e''_1$, $e'_1 = del_{sq}(e''_1)$, and $e_1 \neq e'_1$;

$[\![e_1 =:= e_2]\!] \;\Rightarrow\; [\![e_1 =:= e'_2]\!]$ if $[\![e_2]\!] \;\Rightarrow\; e''_2$, $e'_2 = del_{sq}(e''_2)$, and $e_2 \neq e'_2$;

$[\![e_1 =:= e_2]\!] \;\Rightarrow\; e_1 =:= e'_2$ otherwise.

Here, we use $case_\sigma(\mathtt{success})$ as a shorthand for denoting the encoding of σ by nested (flexible) case expressions with `success` at the final branch. For example, the expression $[\![\mathtt{C\ x\ 2} =:= \mathtt{C\ 1\ y}]\!]$, whose *mgu* (*most general unifier*) is $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$, is evaluated to: $\mathtt{fcase\ x\ of}\ \{1 \to \mathtt{fcase\ y\ of}\ \{2 \to \mathtt{success}\}\}$. This simple treatment of constraints is not sufficient in practical programs since they are often used in concurrent conjunctions, written as $c_1 \,\&\ldots\&\, c_n$ using the concurrent conjunction operator "&." In this case, constraints may instantiate variables and the corresponding bindings should be propagated to the remaining conjuncts. This is the only case of dynamic evaluation (according to the standard semantics) and, thus, we can only float out flexible bindings:

$[\![c_1 \ \& \ \ldots \ \& \ c_n]\!] \ \Rightarrow \ \texttt{success}$
if $c_i = \texttt{success}$ for all $i \in \{1,\ldots,n\}$;

$[\![c_1 \ \& \ \ldots \ \& \ c_n]\!] \ \Rightarrow \ [\![\mathit{fcase} \ x \ \mathit{of} \ \{\overline{p_k \to (c_1 \& \ldots \& c_{i-1} \ \& \ e_k \ \& \ c_{i+1} \& \ldots \& c_n)}\}]\!]$
if $c_i = \mathit{fcase} \ x \ \mathit{of} \ \{\overline{p_k \to e_k}\}$ for some $i \in \{1,\ldots,n\}$;

$[\![c_1 \ \& \ \ldots \ \& \ c_n]\!] \ \Rightarrow \ [\![c_1 \ \& \ldots \ \& \ c_{i-1} \ \& \ c'_i \ \& \ c_{i+1} \ \& \ \ldots \ \& \ c_n]\!]$
if $\exists i \in \{1,\ldots,n\}$ such that $[\![c_i]\!] \Rightarrow c''_i$, $c'_i = \mathit{del}_{sq}(c''_i)$, and $c_i \neq c'_i$;

$[\![c_1 \ \& \ \ldots \ \& \ c_n]\!] \ \Rightarrow \ c_1 \ \& \ \ldots \ \& \ c_n$ otherwise.

Equational constraints can also contain local existentially quantified variables. In this case, they take the form $\mathit{constr}(\mathit{vars}, c)$, where vars are the existentially quantified variables in the constraint c. We treat these constraints as follows:

$[\![\mathit{constr}(\mathit{vars}, c)]\!] \ \Rightarrow \ \texttt{success}$ if $c = \texttt{success}$;

$[\![\mathit{constr}(\mathit{vars}, c)]\!] \ \Rightarrow \ [\![(f)\mathit{case} \ x \ \mathit{of} \ \{\overline{p_k \to \mathit{constr}(\mathit{vars}, e_k)}\}]\!]$
if $c = (f)\mathit{case} \ x \ \mathit{of} \ \{\overline{p_k \to e_k}\}$;

$[\![\mathit{constr}(\mathit{vars}, c)]\!] \ \Rightarrow \ [\![\mathit{constr}(\mathit{vars}, c')]\!]$
if $[\![c]\!] \ \Rightarrow \ c''$, $c' = \mathit{del}_{sq}(c'')$, and $c \neq c'$;

$[\![\mathit{constr}(\mathit{vars}, c)]\!] \ \Rightarrow \ \mathit{constr}(\mathit{vars}, c)$ otherwise.

Note that the second rule above moves all bindings to the top level, even those for the local variables in vars. One might think that it is better to remove those case expressions denoting bindings for the variables in var. However, case expressions with a variable argument are also useful to encode nondeterministic branches and, thus, they cannot be simply discarded. Alternatively, we could include the condition $x \notin \mathit{vars}$ in the second rule above. Unfortunately, this condition also prevents us from floating out bindings for the free variables of the constraint when they are inside a case expression whose variable argument belongs to vars. Therefore, we adopt the simpler solution of permitting the lifting of all bindings. Trivially, when the name of a local variable appears outside the scope of the constraint, it may become bound incorrectly. Nevertheless, this situation can be easily solved in practical implementations by renaming local variables with fresh names so that they do not appear elsewhere in the considered expression.

4.3 Guarded Expressions

Functions in the source language can be defined by conditional rules of the form

$$f \ t_1 \ldots t_n \mid c \ = \ e$$

where c is a constraint. Conditional rules are represented in the flat representation by the *guarded* construct. At partial evaluation time, we are interested in inspecting not only the guard but also the right-hand side of the guard. However, only bindings produced from the evaluation of the guard can be floated out (since, according to the standard evaluation order, we never evaluate the

right-hand side until the guard has been satisfied). The corresponding rules are the following:

$[\![guarded(vars, gc, e)]\!] \;\Rightarrow\; [\![e]\!] \quad$ if $gc = \texttt{success}$;

$[\![guarded(vars, gc, e)]\!] \;\Rightarrow\; [\![(f)case\ x\ of\ \{\overline{p_k \to guarded(vars, e_k, e)}\}]\!]$
 if $gc = (f)case\ x\ of\ \{\overline{p_k \to e_k}\}$;

$[\![guarded(vars, gc, e)]\!] \;\Rightarrow\; [\![guarded(vars, gc', e)]\!]$
 if $[\![gc]\!] \;\Rightarrow\; gc''$, $gc' = del_{sq}(gc'')$, and $gc \neq gc'$;

$[\![guarded(vars, gc, e)]\!] \;\Rightarrow\; [\![guarded(vars, gc, e')]\!]$
 if $[\![e]\!] \;\Rightarrow\; e''$, $e' = del_{sq}(e'')$, and $e \neq e'$;

$[\![guarded(vars, gc, e)]\!] \;\Rightarrow\; guarded(vars, gc, e) \quad$ otherwise.

Observe that, in the first rule above, we can safely remove the *guarded* construct (and forget the names of local variables) when gc is `success` since, in this case, all local variables in e must be bound. On the other hand, the same considerations as in the end of the previous section about the lifting of local variables (in the second rule) apply.

4.4 Higher-Order Functions

Since we do not consider higher-order unification, the standard semantics covers the usual higher-order features of functional languages by adding the following axiom [31]:

$$[\![apply(f(e_1, \ldots, e_m), e)]\!] \Rightarrow [\![f(e_1, \ldots, e_m, e)]\!]$$

if f has arity $n > m$. Thus, an application is evaluated by simply adding the argument to the partial call. In the flat representation, we distinguish partial applications from total functions; namely, partial applications are represented by means of the *partcall* symbol. We treat higher-order features as follows:

$[\![apply(e_1, e_2)]\!] \;\Rightarrow\; [\![f(\overline{c_k}, e_2)]\!] \quad$ if $e_1 = partcall(f, \overline{c_k})$, $k + 1 = ar(f)$;

$[\![apply(e_1, e_2)]\!] \;\Rightarrow\; [\![partcall(f, \overline{c_k}, e_2)]\!] \quad$ if $e_1 = partcall(f, \overline{c_k})$, $k + 1 < ar(f)$;

$[\![apply(e_1, e_2)]\!] \;\Rightarrow\; [\![(f)case\ x\ of\ \{\overline{p_k \to apply(e'_k, e_2)}\}]\!]$
 if $e_1 = (f)case\ x\ of\ \{\overline{p_k \to e'_k}\}$;

$[\![apply(e_1, e_2)]\!] \;\Rightarrow\; [\![apply(e'_1, e_2)]\!] \quad$ if $[\![e_1]\!] \;\Rightarrow\; e''_1$, $e'_1 = del_{sq}(e''_1)$, and $e_1 \neq e'_1$;

$[\![apply(e_1, e_2)]\!] \;\Rightarrow\; [\![apply(e_1, e'_2)]\!] \quad$ if $[\![e_2]\!] \;\Rightarrow\; e''_2$, $e'_2 = del_{sq}(e''_2)$, and $e_2 \neq e'_2$;

$[\![apply(e_1, e_2)]\!] \;\Rightarrow\; apply(e_1, e_2) \quad$ otherwise.

Here, we denote by $ar(f)$ the arity of the function f. Roughly speaking, we allow a partial function to become a total function by adding the missing argument, if possible. If the function does not have the right number of arguments yet, we maintain it as a partial function. In the remaining cases, we evaluate the *apply* arguments in hopes of achieving a partial call after evaluation. In principle, partial calls cannot be evaluated. Nevertheless, we can proceed with the evaluation of their arguments as follows:

$$[\![partcall(f, e_1, \ldots, e_n)]\!] \;\Rightarrow\; partcall(f, [\![e_1]\!], \ldots, [\![e_n]\!])$$

4.5 Overlapping Left-Hand Sides

Overlapping left-hand sides in source programs produce a disjunction where the different alternatives have to be considered. Similarly, we treat *or* expressions in the flat representation as follows:

$$[\![or(e_1, e_2)]\!] \;\Rightarrow\; or([\![e_1]\!], [\![e_2]\!])$$

The correctness of the partial evaluation scheme based on the rules of Figure 3 relies on the proof of equivalence between the RLNT calculus and the original LNT calculus (as formally shown in [8]). The extension of the correctness result to consider the additional features of multi-paradigm declarative languages is similarly based on the equivalence between the treatment of each particular feature during partial evaluation (as defined throughout this section) and its standard evaluation (as described in [31]). Due to the lack of a formal semantics for these features, we do not include here such a formal proof. However, note that we mainly follow the directions described in [31] except from the lifting of case expressions (which is properly justified at the beginning of this section).

5 Control Issues

In this section we present the basic partial evaluation algorithm. It essentially follows the procedure of [12], which is parametric w.r.t. an *unfolding rule* used to determine when and how to terminate the construction of partial computations and an *abstraction operator* used to guarantee that the number of partially evaluated terms is kept finite. The basic algorithm proceeds as follows (see Figure 4, where we denote by $\mathcal{R}_{calls}$ the set of terms in the right-hand sides of the rules of $\mathcal{R}$). Given an input program and a set of terms, the first step consists in applying an unfolding rule to compute a finite RLNT derivation for each term of the set; then, it returns the set of residual rules or *resultants* (i.e., a program) associated to these RLNT derivations (namely, for each RLNT derivation of the form $e \Rightarrow^* e'$, we compute a resultant $e = e'$). Then, an abstraction operator is applied to properly add the terms in the right-hand sides of resultants to the set of terms already partially evaluated; the abstraction phase yields a new set of terms which may need further evaluation and, thus, the process is iteratively repeated while new terms are introduced. Observe that the algorithm does not

Input: a program $\mathcal{R}$ and a set of terms T
Output: a set of terms S
Initialization: $i := 0;\ T_0 := T$
Repeat
$\mathcal{R}' := unfold(T_i, \mathcal{R})$;
$T_{i+1} := abstract(T_i, \mathcal{R}'_{calls})$;
$i := i + 1$;
Until $T_i = T_{i-1}$ (modulo renaming)
Return: $S := T_i$

Figure 4: Basic Algorithm for Partial Evaluation

return a partially evaluated program but a set of terms which unambiguously determines the associated partial evaluation. In particular, by applying once the same unfolding rule, we generate the corresponding resultants which form the residual program. Additionally, a standard renaming transformation is applied over the residual program (see, e.g., [11]). Finally, we perform a novel post-unfolding transformation to remove redundant code from the partially evaluated program (see below).

The procedure follows the style of Gallagher's partial deduction method [25] and two control levels are clearly distinguished: the *local level*—which is managed by an unfolding rule—and the *global level*—which is controlled by an abstraction operator. Trivially, in order to ensure the termination of the algorithm, we must ensure both *local* and *global* termination, i.e., partial RLNT derivations must be finite and the iterative construction of RLNT derivations must eventually terminate. The remaining of this section provides some insights on both control levels as well as on their termination.

5.1 Local Control

As for local control, the main novelty w.r.t. previous partial evaluators for functional logic programs is the use of a non-standard semantics, the RNLT calculus, to perform computations during partial evaluation. In particular, for each expression e, the unfolding rule performs a finite RLNT computation of the form $e \Rightarrow^* e'$ and, then, returns the residual rule $e = e'$.

In order to ensure the local termination of the algorithm, the unfolding rule must incorporate some mechanism to stop the construction of RLNT derivations. For this purpose, there exist several well-known techniques in the literature, e.g., depth-bounds, loop-checks [17], well-founded orderings [19], well-quasi orderings [50], etc. In our partial evaluator, we have experimented with three (terminating) unfolding rules:

One-step unfolding. This is a "cheap" unfolding rule in which only the unfolding of one function call is allowed (the positive supercompiler of [35] employs a similar strategy).

Unfolding based on a well-founded ordering. We define a simple well-founded

order, $>_S$, on expressions. Basically, an expression e is greater than e', in symbols $e >_S e'$, if they have the same outermost function symbol and the *size* (i.e., the number of symbols) in e is greater than the number of symbols in e'. Thus, this unfolding rule allows us to expand a RLNT derivation as long as the derived expression is not greater (according to $>_S$) than the previous expression in the computation.

Unfolding based on an embedding ordering. In previous approaches (e.g., [2, 12]), unfolding rules have been defined by using a particular type of well-quasi ordering: *homeomorphic embedding* (see [40] for a detailed description). Informally, term t_1 *embeds* term t_2 if t_2 can be obtained from t_1 by deleting some operators, e.g., $\texttt{Succ}\ (\underline{\texttt{Succ}}\ ((\underline{\texttt{u}} + \texttt{w})\underline{\times}(\underline{\texttt{u}+}(\texttt{Succ}\ \underline{\texttt{v}}))))$ embeds $\texttt{Succ}\ (\texttt{u} \times (\texttt{u} + \texttt{v}))$. Unfolding rules based on the embedding ordering allow the expansion of derivations until reaching a term which embeds some of the previous terms in the same derivation. However, in the presence of an infinite signature (e.g., integers), this unfolding rule can lead to non-terminating computations. For example, consider the following source program which generates a list of natural numbers within two given limits:

$$\texttt{enum a b} \ = \ \texttt{if a} > \texttt{b then } [\,] \texttt{ else } (\texttt{a} : \texttt{enum } (\texttt{a} + \texttt{1})\ \texttt{b})$$

During its specialization w.r.t. the call `enum 1 n`, the following calls are produced: `enum 1 n`, `enum 2 n`, `enum 3 n`, ..., and no call embeds some previous call. We overcome this problem by considering integers as lists of characters (between "0" and "9") when testing for embedding. In this way, the number of different symbols becomes finite.

Let us say that, in combination with our global control (cf. Section 5.2), the best results were achieved by using the one-step unfolding rule. In practice, it generates optimal recursive functions in many cases.[7] As a counterpart, many redundant functions may appear in the residual program. This does not mean that we incur in a "code explosion" problem since this kind of redundant rules can be easily removed with a simple post-unfolding compression phase. The notion of redundancy that we detect and remove is similar to the one identified by [49] in the field of supercompilation. In particular, we care about *intermediate* functions (i.e., functions which are called from a single program point), which are only used to pass control from one program point to another. Roughly speaking, our post-unfolding transformation allows the additional unfolding of function calls provided that the unfolded function is not recursive—to ensure termination—and that it appears only once in the program—hence it is actually an intermediate function.

5.2 Global Control

Global control cannot be managed with the same flexibility as the local control if we want to preserve the correctness of the method. At the local level,

[7] In fact, the experiments in Section 6 have been performed with the one-step unfolding rule.

this flexibility allows us to safely stop the construction of RLNT derivations at any point. In contrast, we cannot stop the iterative construction of partial derivations until all the function calls in the generated resultants are "closed" w.r.t. the corresponding set of partially evaluated terms. Roughly speaking, a term is *closed* w.r.t. a set of terms if it is an instance of some term in the set and the terms in the matching substitution are recursively closed (see [12] for more details). This condition is necessary to ensure the correctness of the partial evaluation process [12]. On the other hand, it may also happen that this condition is never reached and, in this case, the iterative process runs forever. Therefore, global control usually includes some kind of generalization to enforce the termination of the process. The most popular generalization operator is the *msg* (*most specific generalization*) between terms.

In our partial evaluator, the abstraction operator may also make use of the same orderings explained in Section 5.1 to decide when to generalize and when to continue with the iterative construction of RLNT derivations.[8] Following [12], our abstraction operator *abstract* takes two sets of terms (the terms already partially evaluated T_i and the terms to be added to this set, $\mathcal{R}'_{calls}$, as shown in Figure 4) and returns a *safe* approximation of $T_i \cup \mathcal{R}'_{calls}$. By safe we mean that each term in $T_i \cup \mathcal{R}'_{calls}$ is closed w.r.t. the set of terms resulting from $abstract(T_i, \mathcal{R}'_{calls})$. To be precise, in order to add a new term, t, to the current set of partially evaluated terms, T_i, the abstraction operator essentially proceeds as follows:

- variables are disregarded;
- if $t = c(t_1, \ldots, t_n)$ is not enclosed within square brackets and c is not a defined function symbol, then it tries to add $t_1, \ldots, t_n$ to the set;
- if t is rooted by a defined function symbol or it is enclosed within square brackets, then one of the following actions is performed:
    1. add t to the current set T_i,
    2. discard the term t, or
    3. compute the *msg* between t and some term $t' \in T_i$, say t'', and then try to add both t'' as well as the terms in the matching substitutions (i.e., terms in σ and in σ', with $\sigma(t'') = t$ and $\sigma'(t'') = t'$).

As in the unfolding rule, the concrete action may depend on the use of some ordering between terms. In particular, we have implemented three different abstraction operators. Our first experiments used a (non-terminating) abstraction operator which simply adds the new terms to the current set of partially evaluated terms, i.e., it always takes action (1) above. Then, we implemented an abstraction operator which uses a well-founded ordering on terms to decide what action to take: if the new term is smaller than the last term added to the

[8] For simplicity, in Figure 4, we considered that the current collection of partially evaluated terms is represented by means of a set. A more precise treatment can be easily given by using sequences of terms [12] or *global trees* [44] instead of sets.

set (with the same outermost symbol), then it is also added; otherwise, they are generalized by using the *msg* operator, as shown in action (3). However, the best results—while still guaranteeing termination in all cases—were obtained with an abstraction operator based on an embedding ordering.[9] It proceeds in a similar way as the previous abstraction operator but replacing the use of a well-founded order by the use of an embedding ordering.

The main novelty of our abstraction operator w.r.t. previous operators (e.g., [2, 12]) is that it is *guided* by the RLNT calculus. The key idea is to take into account the position of the square brackets in expressions. This simple extension turns out to be crucial to achieve a good level of specialization in many cases. In particular, the position of square brackets in an expression allows the abstraction operator to know which parts of the expression have been definitely residualized—hence they can be safely ignored—and which parts need further evaluation—and should be added to the current set of partially evaluated terms, if possible. Thanks to the use of this "RLNT-based" abstraction operator, there is no loss of information when passing control from the local level to the global one (in contrast to previous abstraction operators which were "blind" at this respect). The combination of the abstraction operator based on an embedding ordering with the one-step unfolding rule gives rise to efficient residual programs in many cases, while still guaranteeing termination.

6 Experimental Results

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. Curry [31] 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 [30]) or graphical user interfaces at a high-level. The sources of the partial evaluator are publicly available from `http://www.dsic.upv.es/users/elp/peval/`. This tool has been also integrated into the latest distribution of the PAKCS compiler [32] for Curry (see `http://www.informatik.uni-kiel.de/~pakcs/`). As opposed to previous partial evaluators for Curry (e.g., Indy [3]), our partial evaluator is completely written in Curry and fully integrated into a compiler system. To the best of our knowledge, this is the first purely declarative partial evaluator for a functional logic language.

Let us show some results from an experimental evaluation of the implemented partial evaluator. We have only considered the partial evaluation of "small" functions, i.e., functions which are not computationally expensive. In practice, this suffices to obtain good improvements in many programs, e.g., by removing unnecessary data structures, by composing nested functions into

[9] Experiments in Section 6 use this latter abstraction operator.

a comprehensive new function, or by transforming higher-order calls into first-order ones. In order to allow an *effective* partial evaluation of complex programs (e.g., a meta-interpreter), some improvements are still necessary.

Firstly, we have benchmarked several examples which are typical from partial deduction (see, e.g., [38, 40]) and from the literature of functional program transformations, such as positive supercompilation [51], fold/unfold transformations [20, 22], and deforestation [54]. These are simple functions which do not exploit advanced features of practical programming languages. For instance, the benchmark `allones`, which transforms all elements of a list into `1`, is defined in Curry as follows:

```
data Nat = Z | S Nat

allones Z     = []
allones (S x) = 1 : allones x

len []     = Z
len (x:xs) = S (len xs)

main xs = PEVAL (allones (len xs))
```

Table 1 shows the results obtained from the following selected benchmarks:[10] `allones`; `double_app`, the well-known concatenation of three lists; `double_flip`, which flips a tree structure twice, then returning the original tree back; `kmp`, a string pattern matcher; `length_app`, which computes the length of the list resulting from concatenating two lists.

For each benchmark, we show the execution time and heap usage for the original and specialized calls as well as the speedups achieved. Times are expressed in milliseconds and are the average of 10 executions on a 1.3 GHz Linux-PC (AMD Athlon with 256 KB cache). Runtime input goals were chosen to give a reasonably long overall time. The programs were executed with the Curry→Prolog compiler [16] of PAKCS. We do not show the exact times for performing the partial evaluation since this also includes the parsing of input files, writing of output files, etc. Nevertheless, we want to emphasize that the implementation of the partial evaluator in a high-level declarative programming language has no serious consequences w.r.t. its execution time, since all examples are partially evaluated in less than a second including loading the partial evaluator, reading and writing of the intermediate program files. The only exception is the pattern matcher `kmp` which needs an overall time of 1.5 seconds. In all the benchmarks, the calls to be partially evaluated contained no static data, except for the `kmp` example (what explains the larger speedup produced). We did not include the speedup achieved by the previous partial evaluator Indy [3], since the generated residual programs are identical on these benchmarks. This indicates that our new partial evaluation scheme is a conservative extension of previous approaches on comparable examples. Note, though, that our par-

[10] The sources can be found at `http://www.dsic.upv.es/users/elp/peval/`.

Benchmark	original		specialized		speedup
	time	heap	time	heap	
allones	500	8000244	370	4800228	1.35
double_app	570	9603852	440	6403828	1.30
double_flip	710	13631652	550	8388788	1.29
kmp	700	9839908	50	320196	14.0
length_app	400	8824248	280	5624228	1.43

Table 1: Benchmark Results (I)

tial evaluator is applicable to a wider class of programs (including higher-order functions, constraints, several built-in's, etc), while these practical features were not treated in previous approaches to the partial evaluation of functional logic languages.

One of the most useful features of functional languages are higher-order functions since they improve code reuse and modularity in programming. Thus, such features are often used in practical Curry programs (much more than in Prolog programs, which are based on first-order logic and offer only weak features for higher-order programming). Furthermore, almost every practical program uses built-in arithmetic functions which are available in Curry as external functions (but, for instance, not in purely narrowing-based functional logic languages).

The following functions `map` (for applying a function to each element of a list) and `foldr` (for accumulating all list elements) are often used in functional (logic) programs:

```
map _ []     = []
map f (x:xs) = f x : map f xs

foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

For instance, the expression "`foldr (+) 0 [1,2,3]`" is the sum of all elements of the list `[1,2,3]`. Due to the special handling of higher-order features (*apply* and *partcall*) and built-in functions (*external*), our partial evaluator is able to reduce occurrences of this expression to the value `6`. However, instead of such constant expressions, programs may often contain calls to higher-order functions which are partially instantiated. For instance, the expression "`foldr (+) 0 xs`" is transformed by our partial evaluator into "`f xs`," where `f` is a first-order function defined by the rules:

```
f []     = 0
f (x:xs) = x + f xs
```

Calls to this residual function run three times faster (in the Curry→Prolog compiler [16] of PAKCS [32]) than calls to the original definitions; also, memory usage has been reduced significantly (see Table 2, first row). Similarly, the expression "`foldr (+) 0 (map (+1) xs)`" is successfully transformed into "`foo xs`," where the (efficient) first-order function `foo` is defined by the rules:

Benchmark	original		specialized		speedup
	time	heap	time	heap	
`foldr (+) 0 xs`	120	2219196	40	619180	3.00
`foldr (+) 0 (map (+1) xs)`	220	4059208	60	859180	3.67
`foldr (+) 0 (map square xs)`	265	4970732	100	1770704	2.65
`foldr (++) [] xs` (*concat*)	160	2560244	70	560228	2.29
`filter (>100) (map (*3) xs)`	430	6639932	270	3599896	1.59
`any (>10000) xs`	100	1920432	20	120228	5.00
`map (iter (+1) 2) xs`	920	17120280	100	1440228	9.20

Table 2: Benchmark Results (II)

```
foo []     = 0
foo (x:xs) = (x+1) + foo xs
```

Similar results are also obtained for functions defined by a chain of other higher-order functions, like in the expression "`any (>10000) xs`" where the function `any` is defined by "`any p xs = or (map p xs)`" and the function `or` is defined by "`or xs = foldr (||) False xs`" ("`||`" denotes the Boolean disjunction). This expression is replaced by "`any' xs`" where the following first-order definition for `any'` is generated:

```
any' []     = False
any' (x:xs) = if x>10000 then True else any' xs
```

Note that our partial evaluator neither requires function definitions in a specific format (like "foldr/build" in short cut deforestation [26]) nor it is restricted to "higher-order macros" (as in [54]), but can handle arbitrary higher-order functions. For instance, the higher-order function

```
iter f n = if n==0  then f  else iter (f . f) (n-1)
```

which modifies its higher-order argument in each recursive call (`f . f` denotes function composition) can be successfully handled by our partial evaluator (compare example in Section 2).

Table 2 shows the results of specializing some calls to higher-order (like `foldr` or `map`) and built-in functions (like the addition "`+`" or the comparison operator "`>`") with our partial evaluator. For each benchmark, we show the execution time and heap usage for the original and specialized calls and the speedups measured in the same computer environment of Table 1. The input list `xs` contains 20,000 elements in each call.

As shown in Example 3.2, an important feature of Curry is the use of (concurrent) constraints. Due to the extension of our partial evaluator to cover constraints and concurrent conjunctions, constraints might be completely evaluated at partial evaluation time. Actually, our partial evaluator returns for the call "`arith x y`" w.r.t. the program of Example 3.2 the following residual function `arith'`:

```
arith' eval flex
arith' 0 0 = success
```

```
arith' 2 4 = success
```

This example shows that our partial evaluator is also able to transform a concurrent program into a purely sequential one.

Before concluding this section, let us mention that our experimental tests have been performed on a concrete language and compiler by selecting a particular set of benchmark programs. Clearly, there is no guarantee of improvement in general. A first step towards a machine-independent way of assessing the effectiveness of partial evaluators has been taken by [5, 53].

7 Related Work

There exists an active line of research on partial evaluation techniques for narrowing-based, multi-paradigm declarative languages (see [10] for a survey). The original framework [12] is defined for languages whose operational semantics is based solely on narrowing—also known as narrowing-driven partial evaluation—although it has been extended to deal with residuation in [4]. The INDY partial evaluator [3] is a prototype implementation based on the narrowing-driven methodology. The system is written in Prolog and only accepts unconditional term rewriting systems as programs.

In principle, the narrowing-driven approach to partial evaluation has the same potential for specialization as *positive supercompilation* [51] of functional programs and *conjunctive partial deduction* [23] of logic programs (the interested reader is referred to [10, 12] for a more detailed comparison with these approaches). Unfortunately, the use of INDY within a realistic functional logic language (e.g., Curry [31] or Toy [42]) becomes impractical since there are many features of these languages (like higher-order functions, constraints, built-in's, etc.) which are not covered neither by INDY nor by the underlying partial evaluation framework. Our new partial evaluation scheme overcomes the limitations of previous approaches by including a safe treatment for the aforementioned language features. Additionally, the treatment of flexible/rigid functions is simplified in comparison to [4]. This makes the new approach more practically applicable.

Part of this success can be attributed to the use of an intermediate simplified language. This idea, though, is not new in the literature. For instance, the (self-applicable) partial evaluator developed by Bondorf [18] is based on such an approach. His intermediate representation, called Tree, shares many similarities with the flat representation introduced in Section 3. Basically, both languages rely on programs in the form of term rewriting systems where pattern matching is expressed by means of case expressions. However, Tree is a first-order language which uses innermost deterministic reduction. As claimed by Bondorf, the use of a lazy evaluation strategy would require a complete revision of his partial evaluator.

In the field of supercompilation [52], there is also interesting research on the use of intermediate representations. Supercompilation is a program transformation technique based on *driving*—a unification-based function evaluation

mechanism closely related to narrowing—which can perform a deep transformation on programs by using a principle similar to partial evaluation. Applications of supercompilation include program specialization, program inversion and theorem proving. From its inception, supercompilation was tied to a specific programming language, called Refal. [46] proposes the use of "flat Refal" as an intermediate language in which the transformation is performed. Flat Refal is a subset of Refal, formed by rules whose right-hand sides do not contain nested function calls (information exchange takes place only through variables). The simplification of the underlying operational mechanism is also the the motivation for using the "minimized S-Graph language" [28] to present driving. In essence, we share with these works the intention behind using a simplified representation for programs as well as the similarities between narrowing and driving. However, the considered languages belong to different paradigms and the resulting partial evaluation algorithms are still different.

8 Conclusions and Future Work

We have introduced a practical partial evaluation scheme for multi-paradigm declarative languages combining features from functional, logic and concurrent programming. A significant difference with previous approaches is that it considers an intermediate representation for programs into which higher level programs can be automatically translated. A version of our partial evaluator has been distributed with the Portland Aachen Kiel Curry System [32] since April 2001. The most successful experiences were achieved by specializing calls involving higher-order functions, obtaining speedups up to a factor of 9, and generic functions with some static data, like a string pattern matcher where a speedup of 14 is obtained.

The experience gained using the partial evaluator to specialize Curry programs provided invaluable feedback. It guided us to define an appropriate extension of the RLNT calculus for the particular features of practical multi-paradigm declarative languages. It also prompted to develop suitable control strategies which take into account the particularities of the considered language and (non-standard) semantics.

There are several ways in which the research reported here could be continued. Let us mention some possibilities for future work. Technically, the partial evaluator is already able to tackle the specialization of complex programs. Indeed, it is able to partially evaluate a meta-interpreter for Curry written in Curry w.r.t. a given program (which should return a "compiled" version of the source program [24]). However, the partially evaluated program is less efficient and much larger than the original one. Traditionally, this specialization tasks are satisfactorily managed by *offline* partial evaluators which rely on a preprocessing phase of analysis (usually, a *binding-time* analysis [36]). We think that our *online* partial evaluator could also benefit from the information gathered by a binding-time analysis. In particular, this will help us to greatly reduce the work done by the partial evaluator, since many decisions will be taken offline

during the analysis phase, thus improving the specialization process.

Another interesting line of research is the definition of appropriate means to help the user to decide which are the best candidates in a program to be partially evaluated. We suppose that the use of *profiling* tools could be helpful for this task. For instance, the profiling scheme of [48] allows the user to associate a *cost center*—to which execution costs are attributed—with each expression of the source program. In this way, the user can detect which are the most expensive cost centers and, thus, the best candidates to be optimized by partial evaluation. Recently, a symbolic profiler for functional logic languages has been defined [9] by using the idea of cost center. Currently, our main concern is to investigate the combination of this symbolic profiler with the partial evaluation technique presented in this paper in order to "guide" the specialization process.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of FLOPS 2001 for their comments on a preliminary version of this work.

References

[1] E. Albert. *Partial Evaluation of Multi-Paradigm Declarative Languages: Foundations, Control, Algorithms and Efficiency.* PhD thesis, DSIC, Universidad Politécnica de Valencia, 2001. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

[2] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In *Proc. of the 5th Int'l Static Analysis Symposium (SAS'98)*, pages 262–277. Springer LNCS 1503, 1998.

[3] E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical Report DSIC-II/12/98, UPV, 1998. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

[4] 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 Programming and Automated Reasoning (LPAR'99)*, pages 376–395. Springer LNAI 1705, 1999.

[5] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2000)*, pages 103–124. Springer LNCS 2042, 2001.

[6] E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of the 7th Int'l Conf.*

on Logic for Programming and Automated Reasoning (LPAR 2000), pages 381–398. Springer LNAI 1955, 2000.

[7] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. In *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*, pages 326–342. Springer LNCS 2024, 2001.

[8] E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. Technical report, UPV, 2002. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

[9] E. Albert and G. Vidal. Source-Level Abstract Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, 2001. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

[10] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.

[11] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.

[12] 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.

[13] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. In P. Lee, editor, *Proc. of ICFP'99*, pages 273–283. ACM, New York, 1999.

[14] 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.

[15] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[16] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the 3rd Int'l Workshop on Frontiers of Combining Systems (FroCoS 2000)*, pages 171–185. Springer LNCS 1794, 2000.

[17] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[18] A. Bondorf. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of Int'l Conf. on Theory and Practice of Software Development, Barcelona, Spain*, pages 81–95. Springer LNCS 352, 1989.

[19] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.

[20] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[21] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'93)*, pages 493–501. ACM, New York, 1993.

[22] J. Darlington. Program transformation. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.

[23] 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.

[24] Yoshihiko Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprint of article in *Systems, Computers, Controls* 1971.

[25] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.

[26] A.J. Gill, J. Launchbury, and S.L. Peyton Jones. A Short Cut to Deforestation. In *Proc. of the Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 223–232, New York, NY, USA, 1993. ACM Press.

[27] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.

[28] R. Glück and A.V. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In *Proc. of 3rd Int'l Workshop on Static Analysis (WSA'93)*, pages 112–123. Springer LNCS 724, 1993.

[29] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.

[30] M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the 1st Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[31] M. Hanus. Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry/`, 2000.

[32] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.2: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000.

[33] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[34] T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*, pages 216–232. Springer LNCS 2024, 2001.

[35] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[36] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[37] A. Lakhotia and L. Sterling. ProMiX: A Prolog Partial Evaluation System. In L. Sterling, editor, *The Practice of Prolog*, chapter 5, pages 137–179. The MIT Press, Cambridge, MA, 1991.

[38] J. Lam and A. Kusalik. A Comparative Analysis of Partial Deductors for Pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1991. Revised April 1991.

[39] M. Leuschel. The ECCE Partial Deduction System. In *Proc. of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, U.P. Madrid, Tech. Rep. CLIP7/97.1, 1997.

[40] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proc. of the 5th Int'l Static Analysis Symposium (SAS'98)*, pages 230–245. Springer LNCS 1503, 1998.

[41] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.

[42] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[43] W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.

[44] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. of the 12th Int'l Conf. on Logic Programming (ICLP'95)*, pages 597–611. MIT Press, 1995.

[45] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[46] A.P. Nemytykh, V.A. Pinchuk, and V.F. Turchin. A Self-Applicable Supercompiler. In *Partial Evaluation. Proceedings*, pages 322–337. Springer LNCS 1110, 1996.

[47] D. Sahlin. The Mixtus Approach to Automatic Partial Evaluation of Full Prolog. In *Proc. of the 1990 North American Conf. on Logic Programming*, pages 377–398. The MIT Press, Cambridge, MA, 1990.

[48] P.M. Sansom and S.L. Peyton-Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, 1997.

[49] M.H. Sørensen. Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation. Technical Report 94/7, Master's Thesis, DIKU, University of Copenhagen, Denmark, 1994.

[50] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of the 1995 Int'l Logic Programming Symposium (ILPS'95)*, pages 465–479. The MIT Press, Cambridge, MA, 1995.

[51] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[52] V.F. Turchin. Program Transformation by Supercompilation. In *Proc. of the Int'l Workshop on Programs as Data Objects 1985*, pages 257–281. Springer LNCS 217, 1986.

[53] Germán Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62, New York, 2002. ACM Press.

[54] P.L. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.