A Practical Partial Evaluator for a Multi-Paradigm Declarative Language*

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

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

Abstract. Partial evaluation is an automatic technique for program optimization which preserves program semantics. The range of its potential applications is extremely large, as witnessed by successful experiences in several fields. This paper summarizes our findings in the development of partial evaluation tools for Curry, a modern multi-paradigm declarative language which combines features from functional, logic and concurrent programming. From a practical point of view, the most promising approach appears to be a recent partial evaluation framework which translates source programs into a maximally simplified representation. We support this statement by extending the underlying method in order to design a practical partial evaluation tool for the language Curry. The process is fully automatic and can be incorporated into a Curry compiler as a source-to-source transformation on intermediate programs. An implementation of the partial evaluator has been undertaken. Experimental results confirm that our partial evaluator pays off in practice.

1 Introduction

Curry [13, 15] is a modern multi-paradigm declarative language which integrates features from functional, logic and concurrent programming. The most important features of the language include 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*. Furthermore, Curry is a complete programming language which has been used to implement distributed applications (e.g., Internet servers [14], dynamic web pages [17]) or graphical user interfaces [16]. Several efficient implementations of the language already exist (see, e.g., [7, 18, 25]), although there is still room for further improvements. Existing compilers for pure functional languages have been successfully improved by semantics-based program transformation techniques. For instance, the Glasgow Haskell Compiler includes a number of source-to-source program transformations which are able

* This work has been partially supported by CICYT TIC 98-0445-C03-01, by Acción Integrada hispano-alemana HA1997-0073, and by the DFG under grant Ha 2457/1-2.

to optimize the quality of code in many different aspects [27]. Encouraged by these successful experiences, we develop an automatic program transformation technique to improve the efficiency of Curry functional logic programs.

For instance, consider functions defined by higher-order combinators such as `map`, `foldr`, etc. Although such functions can be simply defined in a concise way, some overhead is introduced at runtime which can be eliminated by program transformation techniques. As an example, consider the following function `foo`:

$$\texttt{foo xs} \;=\; \texttt{foldr}\ (+)\ 0\ (\texttt{map}\ (+1)\ \texttt{xs})$$

to add `1` to the elements of a given list `xs` and then compute their total sum. From the programmer point of view, this definition is perfectly right, but there exist more efficient definitions for `foo`, like the following one:

$$\begin{array}{lcl} \texttt{foo}\ [] & = & 0 \\ \texttt{foo}\ (\texttt{x} : \texttt{xs}) & = & (\texttt{x} + 1) + (\texttt{foo xs}) \end{array}$$

In contrast to the original definition, it is a first-order function and, over existing functional logic compilers, it can be executed more efficiently (since it is completely "deforested" [30]). Therefore, we are concerned with program transformations which, given a program, output a *residual* program from which the overhead has been removed at compile time. Partial evaluation (PE) is an automatic technique for program optimization which preserves program semantics. Optimization is achieved by specializing programs w.r.t. parts of their input (hence also called *program specialization*). We note that several PE techniques are able to perform deforestation automatically and, thus, they can be useful to optimize functions like the above one. Informally, a partial evaluator is a mapping which takes a program P and a function call C and derives a more efficient, specialized program P_C which gives the same answers for C (and any of its instances) as P does.

PE techniques have been intensively studied in the context of a wide variety of declarative programming paradigms, specially in both the functional and logic programming communities (see, e.g., [9, 11, 20, 23] and references herein). Recently, a unified framework for the PE of languages which integrate features from functional and logic programming has been introduced in [4]. The original framework is defined for languages whose operational semantics is based solely on narrowing, although it has been extended to deal with residuation in [2]. The Indy partial evaluator [1] is a prototype implementation based on the above framework. The system is written in Prolog and accepts unconditional term rewriting systems as programs. The narrowing-driven approach to PE has the same potential for specialization as *positive supercompilation* [29] of functional programs and *conjunctive partial deduction* [10] of logic programs (it has been experimentally tested in [2, 4]).

Unfortunately, the use of Indy within a realistic functional logic language (e.g., Curry [15], Escher [22] or Toy [24]) becomes impractical since there are many facilities of these languages (e.g., higher-order functions, constraints, I/O, built-in's, etc.) which are not covered neither by Indy nor by the underlying PE framework. For instance, Indy cannot be used to optimize the above function `foo` due to occurrences of the built-in function + and the higher-order functions `map` and `foldr`. Furthermore, the PE framework of [4] suffers from some limitations, e.g., within a lazy (call-by-name) semantics, terms in head normal form (i.e., rooted by a constructor symbol) cannot be evaluated during the PE process. This can drastically reduce the optimization power of the method in many cases. To overcome this problem, [3] introduces a novel approach for the PE of functional logic languages. The new scheme considers a maximally simplified representation into which programs written in a higher-level language (i.e., inductively sequential programs [5] with evaluation annotations) can be automatically translated. The restriction to not evaluate terms in head normal form is avoided by defining a non-standard semantics which is well-suited to perform computations at PE time.

The aim of this work is to show how—in contrast to [4] and INDY—the framework of [3] can be successfully applied in practice. To this end, we first enrich the intermediate representation considered in [3] in order to cover all the facilities of the language Curry. The resulting representation is essentially equivalent to the standard intermediate language FlatCurry [18], which has been proposed to provide a common interface for connecting different tools working on Curry programs, e.g., back ends for various compilers [7]. Then, the non-standard semantics of [3] is carefully extended in order to cover the additional language features. This extension is far from trivial, since the underlying calculus does not compute bindings but represents them by "residual" case expressions. 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. Therefore, we are constrained to define a specific treatment for these important features. Finally, in order to make the resulting framework practically applicable, we define appropriate control strategies which take into account the particularities of the considered language and (non-standard) semantics. The resulting method is able to transform realistic Curry programs in contrast to other existing partial evaluators. For instance, the "higher-order" definition of `foo` can be automatically transformed into the more efficient version (see Sect. 5).

The structure of this paper is as follows. Section 2 recalls the basic notions and techniques associated to the PE of functional logic programs. Section 3 extends the previous approach in order to cover all the facilities provided by the language Curry. A description of the control issues involved in the PE process is presented in Sect. 4. Some experiments with the partial evaluator are described in Sect. 5 before we conclude in Sect. 6.

2 The Basic Approach

For the sake of completeness, in this section we briefly recall the approach presented in [3] for the PE of functional logic programs. Informally speaking, the process is based on two steps: firstly, the source program is translated into a maximally simplified representation (Sect. 2.1); then, function calls are partially evaluated using a non-standard semantics, the RLNT calculus, which is specially well-suited for performing computations at PE time (Sect. 2.2). To be precise, for each finite (possibly partial) computation of the form $e_1 \Rightarrow^+ e_2$ performed with the RLNT calculus, we generate a residual rule—a *resultant*—of the form: $e_1 = e_2$. Additionally, a post-processing of renaming is often required to recover the same class of programs.

2.1 The Flat Representation

Following [13], we consider inductively sequential rewrite systems [5] (with evaluation annotations) as programs and an operational semantics which integrates (needed) narrowing and residuation. In order to simplify the underlying semantics, a *flat* representation for programs is introduced. This representation is based on the formulation of [19] to express pattern-matching by case expressions. As it will become apparent in Sect. 3, it corresponds to a subset of the FlatCurry syntax [18], a standard intermediate representation for Curry programs.

$$
\begin{array}{lll}
\mathcal{R} ::= D_1 \ldots D_m & e ::= x & \text{(variable)} \\
D ::= f(x_1, \ldots, x_n) = e & \quad \mid \ c(e_1, \ldots, e_n) & \text{(constructor)} \\
 & \quad \mid \ f(e_1, \ldots, e_n) & \text{(function call)} \\
p \ ::= c(x_1, \ldots, x_n) & \quad \mid \ \mathit{case}\ e_0\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(rigid case)} \\
 & \quad \mid \ \mathit{fcase}\ e_0\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(flexible case)}
\end{array}
$$

A 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 variables as parameters. The right-hand side is an expression e composed by variables, constructors, function calls, and case expressions for pattern-matching. The form of a case expression is:[1]

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

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 $(f)\mathit{case}$'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* nondeterministically binds this variable to a pattern in a branch of the case expression (which corresponds to narrowing). Functions defined only by *fcase* (resp. *case*) expressions are called *flexible* (resp. *rigid*). Thus, flexible functions act as generators (like predicates in logic programming) and rigid functions act as consumers.

For example, consider the rules defining the (rigid) function "$\leqslant$":[2]

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

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

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

[2] Although we consider in this work a first-order representation for programs, we use a curried notation in concrete examples (as usual in functional languages).

$$\begin{array}{lll} \texttt{x} \leqslant \texttt{y} = & \texttt{case x of } \{\texttt{0} & \to \texttt{True}; \\ & \quad (\texttt{Succ x}_1) & \to \texttt{case y of } \{\texttt{0} \to \texttt{False}; \\ & & \qquad (\texttt{Succ y}_1) \to \texttt{x}_1 \leqslant \texttt{y}_1\} \ \} \end{array}$$

An automatic transformation from inductively sequential programs [5] to programs using case expressions is introduced in [19].

2.2 The Residualizing Semantics

The operational semantics of flat programs becomes simpler, since *definitional trees* [5] (used to guide the needed narrowing strategy [6]) have been compiled in the program by means of case expressions. The LNT calculus [19] (Lazy Narrowing with definitional Trees) is an operational semantics for inductively sequential programs expressed in terms of case expressions, which has been proved equivalent to needed narrowing. This calculus has been also extended to cover programs containing evaluation annotations in [3]; namely, flexible (resp. rigid) functions are translated by using only *fcase* (resp. *case*) expressions. In the following, we refer to the LNT calculus to mean the LNT calculus of [3].

In [3], it was shown that, by using the standard semantics during PE, one would have the same problems of previous approaches. In particular, one of the main problems comes from the *backpropagation* of variable bindings to the left-hand sides of residual rules (see Example 2 of [3]). Therefore, they propose a *residualizing* version of the LNT calculus which avoids this restriction. In this calculus, variable bindings are encoded by case expressions (and are considered "residual" code). The inference rules of the residualizing calculus, RLNT (Residualizing LNT), can be seen in Fig. 1. In the following, we consider a (*many-sorted*) *signature* partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of defined *functions* or *operations*.

Let us recall the six inference rules defining the one-step relation $\Rightarrow$.[3]

(1) HNF. The HNF (Head Normal Form) rules are used to evaluate terms in head normal form. If the expression is a variable or a constructor constant, the square brackets are removed and the evaluation process stops. Otherwise, the evaluation proceeds with the arguments.

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

(3) Case Select. This rule selects the appropriate branch of a case expression and continues with the evaluation of this branch.

(4) Case Guess. The treatment of case expressions with variable arguments distinguishes it from the LNT calculus. In the standard semantics, these expressions are evaluated by means of the following rules:

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

[3] 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.

HNF

$$\llbracket t \rrbracket \;\Rightarrow\; t \quad \text{if } t \in \mathcal{V} \text{ or } t = c() \text{ with } c/0 \in \mathcal{C}$$

$$\llbracket c(t_1, \ldots, t_n) \rrbracket \;\Rightarrow\; c(\llbracket t_1 \rrbracket, \ldots, \llbracket t_n \rrbracket)$$

Case-of-Case

$$\llbracket (f)case\ ((f)case\ t\ of\ \{\overline{p_k \to t_k}\})\ of\ \{\overline{p'_j \to t'_j}\} \rrbracket \;\Rightarrow\; \llbracket (f)case\ t\ of\ \{\overline{p_k \to (f)case\ t_k\ of\ \{\overline{p'_j \to t'_j}\}}\} \rrbracket$$

Case Function

$$\llbracket (f)case\ g(\overline{t_n})\ of\ \{\overline{p_k \to t'_k}\} \rrbracket \;\Rightarrow\; \llbracket (f)case\ \sigma(r)\ of\ \{\overline{p_k \to t'_k}\} \rrbracket$$
if $g(\overline{x_n}) = r \in \mathcal{R}$ is a rule with fresh variables and $\sigma = \{\overline{x_n \mapsto t_n}\}$

Case Select

$$\llbracket (f)case\ c(\overline{t_n})\ of\ \{\overline{p_k \to t'_k}\} \rrbracket \;\Rightarrow\; \llbracket \sigma(t'_i) \rrbracket \quad \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C},\ \sigma = \{\overline{x_n \mapsto t_n}\}$$

Case Guess

$$\llbracket (f)case\ x\ of\ \{\overline{p_k \to t_k}\} \rrbracket \;\Rightarrow\; (f)case\ x\ of\ \{\overline{p_k \to \llbracket \sigma_k(t_k) \rrbracket}\}$$
if $\sigma_i = \{x \mapsto p_i\}$, $i = 1, \ldots, k$

Function Eval

$$\llbracket g(\overline{t_n}) \rrbracket \;\Rightarrow\; \llbracket \sigma(r) \rrbracket \quad \text{if } g(\overline{x_n}) = r \in \mathcal{R} \text{ is a rule with fresh variables and } \sigma = \{\overline{x_n \mapsto t_n}\}$$

Fig. 1. RLNT Calculus

However, in this case, one would inherit the limitations of previous approaches. Therefore, it has been modified in order not to backpropagate the bindings of variables. In particular, the new Case Guess rule "residualizes" the case structure and continues with the evaluation of the different branches (by applying the corresponding substitution in order to propagate bindings forward in the computation). It 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. Also, the calculus becomes deterministic, i.e., there is no don't know nondeterminism involved in the computations. This means that only one derivation can be issued from a given expression (thus, there is no need to introduce a notion of RLNT "tree").

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

$$\llbracket \texttt{case (case x of \{ 0} \to \texttt{True} \quad \texttt{(Succ y)} \to \texttt{False\})\ \ of \{True} \to \texttt{C x\}} \rrbracket$$

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

[30]). By using the Case-of-Case rule, the above expression can be reduced to:

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

(which can be further simplified with the Case Guess and Case Select rules). 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.

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

The correctness of the PE scheme for flat programs based on the RLNT calculus can be found in [3].

3 Extending the Basic Framework

The aim of this section is to extend the basic approach in order to cover the facilities of a realistic multi-paradigm language: Curry [15]. To this end, we first enrich the flat representation of Sect. 2.1 with some additional features which constitute the most useful facilities of the language. Then, we correspondingly extend the rules of the RLNT calculus to properly deal with these new features.

3.1 An Intermediate Representation for Curry Programs

Our extended flat representation essentially coincides with the standard intermediate representation, FlatCurry [18], used during the compilation of Curry programs. It contains all the necessary information about a Curry program with all "syntactic sugar" compiled out and type-checking and lambda-lifting performed. In the extended representation, we allow the following expressions:

$$\begin{array}{lll}
e ::= & x & \text{(variable)} \\
| & c(e_1,\ldots,e_n) & \text{(constructor)} \\
| & f(e_1,\ldots,e_n) & \text{(function call)} \\
| & (f)case\ e_0\ of\ \{p_1 \to e_1;\ldots;p_n \to e_n\} & \text{(case expression)} \\
| & external(e) & \text{(external function call)} \\
| & partcall(f,e_1,\ldots,e_k) & \text{(partial application)} \\
| & apply(e_1,e_2) & \text{(application)} \\
| & constr(\{x_1,\ldots,x_n\},e) & \text{(constraint)} \\
| & or(e_1,e_2) & \text{(disjunction)} \\
| & guarded(\{x_1,\ldots,x_n\},e_1,e_2) & \text{(guarded expression)}
\end{array}$$

The right-hand side of each function definition is now an expression e composed by variables, constructors, function calls, case expressions, and additional features like: non user-defined ("external") functions, higher-order features like partial application and an application of a functional expression to an argument, constraints (like equational constraints e_1 =:= e_2, possibly containing existentially quantified variables), disjunctions (to represent functions with overlapping left-hand sides), and guarded expressions (to represent conditional rules, i.e., the first expression is always a constraint and the list of variables are the local variables which are visible in the constraint and the right-hand side). A detailed description of these features and their intended semantics can be found in [15].

3.2 Extending the RLNT Calculus

In principle, one could extend the RLNT calculus in order to deal with all the facilities of FlatCurry in a simple way. The naive idea is to treat all the additional features of the language as constructor symbols at PE time. This means that they are never partially evaluated but their original definitions are returned by the PE process. However, in realistic Curry programs, the presence of these additional features is perfectly common, hence it is an unacceptable restriction just to residualize them. Our experimental tests have shown that no specialization is obtained in most cases if we follow this simple approach.

On the other hand, extending the RLNT calculus of Sect. 2.2 with the standard semantics for the additional features of FlatCurry 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 case expressions from argument positions to the top level while propagating the corresponding bindings to the remaining arguments. For example, the expression[5]

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

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:

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

We notice that this transformation cannot be applied over arbitrary expressions since the intended (lazy) semantics is only preserved when the given function is *strict* in the position of the case expression. Nevertheless, typical FlatCurry programs contain many elements where the evaluation order is fixed. For instance, the condition of a guard is strict, since it must be reduced to `True` (or "`success`") before applying a conditional rule, the arguments of most external functions are also strict, because they must be reduced to ground constructor terms before executing the external call, etc.

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

[5] Following [15], "`success`" denotes a constraint which is always solvable.

Furthermore, there are a number of situations in which an expression cannot be evaluated until all (or some) of its arguments have some particular form. For example, a call of the form $apply(e_1, e_2)$ can be only reduced if the first argument e_1 is of the form $partcall(\ldots)$. In these cases, we will try to evaluate the arguments of the function to achieve the required form. For the sake of a simpler presentation, we introduce the auxiliary function *try_eval*. Given a function call $f(\overline{e_n})$ and a set of natural numbers I which represents the set of strict arguments of function f, we define *try_eval* as follows:

$$try\_eval(f(\overline{e_n}), I) \ = \begin{cases} [\![(f)case\ x\ of\ \{\overline{p_k \to f(e_1, \ldots, e_{i-1}, e'_k, e_{i+1}, \ldots, e_n)}\}]\!] \\ \quad\quad \text{if } e_i = (f)case\ x\ of\ \{\overline{p_k \to e'_k}\} \text{ for some } i \in I \\ [\![f(e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n)]\!] \\ \quad\quad \text{if } \exists i \in \{1, \ldots, n\}.\ [\![e_i]\!] \Rightarrow e''_i,\ e'_i = del_{sq}(e''_i),\ e_i \neq e'_i \\ f(\overline{e_n}) \text{ otherwise} \end{cases}$$

Here 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 function above. First, *try_eval* tries to float a case expression in the i-th argument (with $i \in I$) out of this argument. If this is not possible, it tries to evaluate some argument and, if this does not lead to a progress, the expression is just residualized. Since this definition of *try_eval* is ambiguous, we additionally require that the different cases are tried in their textual order and the arguments are evaluated from left to right.

Non User-Defined Functions. FlatCurry programs often contain functions which are not defined in Curry but implemented in another language (*external* functions, like arithmetic operators, basic input/output facilities, etc). Such functions are executed only if all arguments are evaluated to ground constructor terms.[6] The same restriction seems reasonable when computing the PE of an external function. This implies that all arguments of external functions are assumed to be strict and, thus, the call to *try_eval* is performed with the complete set of argument positions:

$$[\![external(f(\overline{e_n}))]\!] \Rightarrow \begin{cases} ext\_call(f(\overline{e_n})) & \text{if } e_1, \ldots, e_n \text{ are ground constructor} \\ [\![external(e')]\!] & \text{if } try\_eval(f(\overline{e_n}), \{1, \ldots, n\}) = [\![e']\!] \\ external(f(\overline{e_n})) & \text{otherwise} \end{cases}$$

where $ext\_call(e)$ evaluates e according to its predefined semantics. Basically, the partial evaluator first tries to execute the external function and, if this is not possible because all arguments are not ground constructor terms, then it tries to evaluate its arguments. Furthermore, we need to add the rule:

$$[\![external((f)case\ x\ of\ \{\overline{p_k \to e_k}\})]\!] \ \Rightarrow\ [\![(f)case\ x\ of\ \{\overline{p_k \to external(e_k)}\}]\!]$$

to move a case expression obtained by *try_eval* outside the external call (in order to allow further evaluation of the branches).

[6] 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.

The only exception to the above rule are I/O actions, for which Curry follows the monadic approach to I/O. These functions act on the current "state of the outside world". They are residualized since this state is not known at PE time.

Constraints. The treatment for constraints heavily depends on the associated constraint solver. In the following, we only consider *equational* constraints. An elementary constraint is an equation e_1 =:= e_2 between two expressions which is solvable if both sides are reducible to unifiable constructor terms. This notion of equality, the so-called *strict* equality, is incorporated in our calculus by

$$[\![e_1 \;=:=\; e_2]\!] \;\Rightarrow\; \begin{cases} case_\sigma(\texttt{success}) & \text{if } \sigma = mgu(e_1, e_2) \text{ and } e_1, e_2 \\ & \quad\text{are constructor terms} \\ try\_eval(e_1 \;=:=\; e_2, \{1,2\}) & \text{otherwise} \end{cases}$$

Note that we call to *try_eval* with the set of positions $\{1, 2\}$ since function "=:=" is strict in its two arguments. Here, we use $case_\sigma(\texttt{success})$ as a shorthand for denoting the encoding of σ by nested (flexible) case expressions with `success` at the final branch. For example, the expression $[\![\texttt{C x 2} =:= \texttt{C 1 y}]\!]$, whose mgu is $\{\texttt{x} \mapsto \texttt{1}, \texttt{y} \mapsto \texttt{2}\}$ is evaluated to: $\texttt{fcase x of } \{\texttt{1} \to \texttt{fcase y of } \{\texttt{2} \to \texttt{success}\}\}$. This simple treatment of constraints is not sufficient in practical programs since they are often used in concurrent conjunctions, written as c_1 & ... & c_n ("&" is a built-in operator which evaluates its arguments concurrently). In this case, constraints may instantiate variables and the corresponding bindings should be propagated to the remaining conjuncts. The problematic point is that we cannot move arbitrary case expressions to the top level, but only flexible case expressions (otherwise, we could change the floundering behavior of the program). Consider, for instance, the following simple functions:

$$\begin{array}{l} \texttt{f x} = \texttt{case x of } \{\texttt{1} \to \texttt{success}\} \\ \texttt{g x} = \texttt{fcase x of } \{\texttt{1} \to \texttt{success}\} \end{array}$$

where `f` is rigid and `g` is flexible. Given the expression $[\![\texttt{f x} \;\&\; \texttt{g x}]\!]$, if we allow to float out arbitrary case expressions, we could perform the following evaluation:

$$\begin{array}{rl} [\![\texttt{f x} \;\&\; \texttt{g x}]\!] & \Rightarrow [\![\texttt{case x of } \{\texttt{1} \to \texttt{success}\} \;\&\; \texttt{g x}]\!] \\ & \Rightarrow [\![\texttt{case x of } \{\texttt{1} \to \texttt{success} \;\&\; \texttt{g x}\}]\!] \\ & \Rightarrow \texttt{case x of } \{\texttt{1} \to [\![\texttt{success} \;\&\; \texttt{g 1}]\!]\} \end{array}$$

which ends up in $\texttt{case x of } \{\texttt{1} \to \texttt{success}\}$. Note that this residual expression suspends if variable `x` is not instantiated, whereas the original expression could be reduced by evaluating first function `g` and then function `f`. Therefore, we handle concurrent conjunctions as follows:

$$[\![c_1 \;\&\; \ldots \;\&\; c_n]\!] \;\Rightarrow$$

$$\begin{cases} \texttt{success} & \text{if } c_i = \texttt{success} \text{ for all } i \in \{1, \ldots, n\} \\ [\![fcase\ x\ of\ \{\overline{p_k \to (c_1 \;\&\ldots\;\&\; c_{i-1} \;\&\; e'_k \;\&\; c_{i+1} \;\&\; \ldots \;\&\; c_n)}\}]\!] & \\ & \text{if } c_i = fcase\ x\ of\ \{\overline{p_k \to e'_k}\} \text{ for some } i \in \{1, \ldots, n\} \\ [\![c_1 \;\&\ldots\;\&\; c_{i-1} \;\&\; c'_i \;\&\; c_{i+1} \;\&\; \ldots \;\&\; c_n]\!] & \\ & \text{if } \exists i \in \{1, \ldots, n\}.\ [\![c_i]\!] \Rightarrow c''_i,\ c'_i = del_{sq}(c''_i), c_i \neq c'_i \\ c_1 \;\&\; \ldots \;\&\; c_n & \text{otherwise} \end{cases}$$

Note that, in contrast to external functions, only flexible case expressions are moved to the top level. Equational constraints can also contain local existentially quantified variables. In this case they take the form $constr(vars, c)$, where $vars$ are the existentially quantified variables in the constraint c. We treat these constraints as follows:

$$[\![constr(vars, c)]\!] \Rightarrow \begin{cases} \texttt{success} & \text{if } c = \texttt{success} \\ try\_eval(constr(vars, c), \{2\}) & \text{otherwise} \end{cases}$$

Note that the above rule moves all bindings to the top level, even those for the local variables in $vars$. In practice, case expressions denoting bindings for the variables in $vars$ are removed since they are local, but we keep the above formulation for simplicity.

Guarded Expressions. In Curry, functions can be defined by conditional rules of the form

$$f\ e_1 \ldots e_n \mid c \ = \ e$$

where c is a constraint (rules with multiple guards are also allowed but considered as syntactic sugar for denoting a sequence of rules). Conditional rules are represented in FlatCurry by the *guarded* construct. At PE 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 this is the unique strict argument):

$$[\![guarded(vars, gc, e)]\!] \Rightarrow \begin{cases} [\![e]\!] & \text{if } gc = \texttt{success} \\ try\_eval(guarded(vars, gc, e), \{2\}) & \text{otherwise} \end{cases}$$

As in the case of constraints, the application of try_eval can unnecessarily move some bindings (i.e., those for the variables in $vars$) outside the guarded expression. A precise treatment can be easily defined, but we preserve the above presentation for the sake of readability.

Higher-Order Functions. The higher-order features of functional programming are implemented in Curry by providing a (first-order) definition of the application function (*apply*). Since Curry excludes higher-order unification, the operational semantics of Curry covers the usual higher-order features of functional languages by adding the following axiom [15]:

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

if f has arity $n > m$. Thus, an application is evaluated by simply adding the argument to a partial call. In FlatCurry, 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:

$$[\![apply(e_1, e_2)]\!] \Rightarrow \begin{cases} [\![f(\overline{c_k}, e_2)]\!] & \text{if } e_1 = partcall(f, \overline{c_k}),\ k+1 = ar(f) \\ partcall(f, \overline{c_k}, e_2) & \text{if } e_1 = partcall(f, \overline{c_k}),\ k+1 < ar(f) \\ try\_eval(apply(e_1, e_2), \{1\}) & \text{otherwise} \end{cases}$$

where $ar(f)$ denotes 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 after adding the new argument, 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. Note that *try_eval* is called with the set $\{1\}$ in order to avoid the propagation of bindings from the evaluation of non-strict arguments (i.e., from the second argument of *apply*).

Overlapping Left-Hand Sides. Overlapping left-hand sides in Curry programs produce a disjunction where the different alternatives have to be considered. Similarly, we treat *or* expressions in FlatCurry as follows:

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

4 The Partial Evaluator in Practice

In this section, we describe the structure of a simple *on-line* partial evaluator in the style of [11] which follows the ideas presented so far. Essentially, the partial evaluator proceeds as follows:

Unfolding phase. Firstly, given a program and a set of function calls, we compute a finite (possibly incomplete) RLNT derivation for each call of the set according to an *unfolding rule* $\mathcal{U}$. Roughly speaking, the unfolding rule determines how to stop RLNT derivations in order to avoid infinite computations. Formally, given a program $\mathcal{R}$ and a set of function calls $T = \{t_1, \ldots, t_n\}$, $\mathcal{U}$ is a (total) function such that, whenever $\mathcal{U}(T, \mathcal{R}) = S$, then $S = \{s_1, \ldots, s_n\}$ and there exist finite RLNT derivations of the form $[\![t_i]\!] \Rightarrow^+ s_i$ in $\mathcal{R}$, with $i = 1, \ldots, n$.

Abstraction phase. Since some of the derived expressions $S = \{s_1, \ldots, s_n\}$ may contain function calls which are not covered by the already (partially) evaluated calls T, this process is iteratively repeated for any term of S which is not *closed* w.r.t. the set T. Informally, a term s is closed w.r.t. a set of terms T (or, simply, T-closed) if the maximal operation-rooted subterms of s are instances of some terms in T and the terms in the matching substitution are recursively T-closed (see [4] for a precise definition). In order to avoid repeating this process infinitely, an *abstraction operator* is commonly used. In particular, we consider a mapping *abstract* which takes two sets of terms T and S (which represent the set of terms already evaluated and the set of terms to be added to this set, respectively) and returns a *safe* approximation $abstract(T, S)$ of $T \cup S$. Here, by "safe" we mean that each term in $T \cup S$ is closed w.r.t. the result of $abstract(T, S)$ (i.e., no function call is lost during the abstraction process).

Following the structure of many on-line partial evaluators (see, e.g., [11]), we sketch a PE algorithm which is parametric w.r.t. an unfolding rule $\mathcal{U}$ and an abstraction operator *abstract*:

> **Input**: a program $\mathcal{R}$ and a set of terms T / **Output**: a set of terms S
> **Initialization**: $i := 0;\ T_0 := T$
> **Repeat** $S := \mathcal{U}(T_i, \mathcal{R});\ T_{i+1} := abstract(T_i, S);\ i := i + 1$
> **Until** $T_i = T_{i-1}$ (modulo renaming)
> **Return** $S := T_i$

The above PE algorithm involves two control issues: the so-called *local* control, which concerns the definition of an unfolding rule $\mathcal{U}$ to compute finite partial evaluations, and the *global* control, which consists of defining a safe abstraction operator *abstract* to ensure the termination of the iterative process.

Local Control. In the 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 at PE time. Since RLNT computations do not produce bindings, the restriction to not evaluate terms in head normal form of previous partial evaluators is avoided.

In order to ensure the finiteness of RLNT derivations, there exist a number of well-known techniques in the literature, e.g., depth-bounds, loop-checks, well-founded (or well-quasi) orderings (see, e.g., [8, 21, 28]). For instance, an unfolding rule based on the use of the homeomorphic embedding ordering was used in the INDY partial evaluator. Informally, expression e_1 *embeds* expression e_2 if e_2 can be obtained from e_1 by deleting some operators. For example, $Succ\ (\underline{Succ}\ ((\underline{u} + w)\underline{\times}(\underline{u+}(Succ\ \underline{v}))))$ embeds $Succ\ (u \times (u + v))$. However, in the presence of an infinite signature (e.g., natural numbers in Curry), this unfolding rule can lead to non-terminating computations. For example, consider the following Curry program which generates a list of natural numbers within two given limits:

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

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.

Therefore, in our partial evaluator we have chosen a safe (and "cheap") unfolding rule: only the unfolding of one function call is allowed (the positive supercompiler of [20] employs a similar strategy). The main advantage of this approach is that expressions can be "folded back" (i.e., can be proved closed) w.r.t. any partially evaluated call. In practice, this generates optimal recursive functions in many cases. As a counterpart, many (unnecessary) intermediate 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 by a post-unfolding phase (similarly to [20]). Our experiments with the one-step unfolding rule and the post-unfolding phase indicate that this leads to optimal (and concise) residual functions in many cases.

Global Control. As for global control, an abstraction operator usually relies on a concrete ordering over terms in order to keep the sequence of partially evaluated terms finite. As discussed above, a well-quasi ordering like the homeomorphic embedding ordering cannot be used since we consider an infinite signature. Therefore, we implement an abstraction operator which uses a *well-founded* order to ensure termination and generalizes those calls which do not satisfy this ordering by using the *msg* (*most specific generalization*). Abstraction operators based on this relation are defined in, e.g., [26].

The main novelty of our abstraction operator w.r.t. previous operators is that it is *guided* by the RLNT calculus. The key idea is to take into account the position of the square brackets of the calculus in expressions; namely, subterms within square brackets should be added to the set of partially evaluated terms (if possible, otherwise generalized) since further evaluation is still required, while subterms which are not within square brackets should be definitively residualized (i.e., ignored by the abstraction operator, except for operation-rooted terms). The combination of this strategy with the above unfolding rule gives rise to efficient residual programs in many cases, while still guaranteeing termination.

5 Experimental Results

This section describes some experiments with an implementation of a partial evaluator for Curry programs which follows the guidelines presented in previous sections. Our PE tool is implemented in Curry itself and it is publicly available at `http://www.dsic.upv.es/users/elp/soft.html`. The implemented partial evaluator first translates the subject program into a FlatCurry program. To do this, we use the module `Flat` for meta-programming in Curry (included in the current distribution of PAKCS [18]). This module contains datatype definitions to treat FlatCurry programs as data objects (using a kind of *ground representation*). In particular, the I/O action `readFlatCurry` reads a Curry program, translates it to the intermediate language FlatCurry, and finally returns a data structure representing the input program.

The effectiveness of the narrowing-driven approach to PE is out of question (see, e.g., [2–4] for typical PE benchmarks). Unfortunately, these ad-hoc programs do not happen very frequently in real Curry applications. This motivated us to benchmark more realistic examples where the most important features of Curry programs appear. 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). These practical features were not treated in previous approaches to PE of 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 _ []       = []                    foldr _ z []      = z
map f (x : xs) = f x : map f xs        foldr f z (h : t) = f h (foldr f z t)
```

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 `6`. However, instead of such constant expressions, realistic programs contain calls to higher-order functions which are partially instantiated. For instance, the expression `foldr (+) 0 xs` is specialized

Benchmark	original		specialized		speedup
	time	heap	time	heap	
`foldr (+) 0 xs`	210	2219168	60	619180	3.5
`foldr (+) 0 (map (+1) xs)`	400	4059180	100	859180	4.0
`foldr (+) 0 (map square xs)`	480	4970716	170	1770704	2.8
`foldr (++) [] xs` (*concat*)	290	2560228	110	560228	2.6
`filter (>100) (map (*3) xs)`	750	6639908	430	3120172	1.7
`map (iterate (+1) 2) xs`	1730	17120280	600	6720228	2.9

Table 1. Benchmark results

by our partial evaluator to `f xs` where `f` is a first-order function defined by:

$$\begin{array}{lcl} \mathtt{f}\ [] & = & \mathtt{0} \\ \mathtt{f}\ (\mathtt{x} : \mathtt{xs}) & = & \mathtt{x} + \mathtt{f\ xs} \end{array}$$

Calls to this residual function run 3.5 times faster (in the Curry→Prolog compiler [7] of PAKCS [18]) than calls to the original definitions; also, memory usage has been reduced significatively (see Table 1, first row). Similarly, the expression `foldr` (+) 0 (`map` (+1) `xs`) has been successfully specialized to the efficient version of Sect. 1. Note that our partial evaluator neither requires function definitions in a specific format (like "foldr/build" in short cut deforestation [12]) nor it is restricted to "higher-order macros" (as in [30]), but can handle arbitrary higher-order functions. For instance, the higher-order function

$$\mathtt{iterate\ f\ n}\ =\ \mathtt{if\ n == 0\ \ then\ f\ \ else\ iterate}\ (\mathtt{f.f})\ (\mathtt{n} - 1)$$

which modifies its higher-order argument in each recursive call (`f.f` denotes function composition) can be successfully handled by our partial evaluator. Table 1 shows the results of specializing some calls to higher-order and built-in functions with our partial evaluator. For each benchmark, we show the execution time and heap usage for the original and specialized calls and the speedups achieved. The input list `xs` contains 20,000 elements in each call. Times are expressed in milliseconds and measured on a 700 MHz Linux-PC (Pentium III, 256 KB cache). The programs were executed with the Curry→Prolog compiler [7] of PAKCS.

Another important feature of Curry is the use of (concurrent) constraints. Consider, for instance, the following function `arith`:

$$\begin{array}{l} \mathtt{digit\ 0}\ \ \ = \mathtt{success} \\ \ldots \\ \mathtt{digit\ 9}\ \ \ = \mathtt{success} \\ \mathtt{arith\ x\ y} = \mathtt{x} + \mathtt{x} =:= \mathtt{y}\ \&\ \mathtt{x} * \mathtt{x} =:= \mathtt{y}\ \&\ \mathtt{digit\ x} \end{array}$$

Calls to "`arith`" might be completely evaluated at PE time. Actually, our partial evaluator returns the residual function $\mathtt{arith}'$ for the call `arith x y` :

$$\begin{array}{l} \mathtt{arith}'\ 0\ 0 = \mathtt{success} \\ \mathtt{arith}'\ 2\ 4 = \mathtt{success} \end{array}$$

In this section, we have shown the specialization of calls to some small functions. However, this does not mean that only the specialization of small programs is feasible in our PE tool. Actually, in the specialization of larger programs the programmer would include some annotations indicating the function calls to be specialized (which usually only involve calls to small functions). In this case, the same program would be returned by the system except for the annotated calls, which are replaced by new calls to the specialized functions, together with the definitions of such specialized functions. The PE tool is not fully integrated into PAKCS yet and, thus, transformed programs cannot be directly executed in Curry (since they are in FlatCurry format). Our aim is to incorporate the partial evaluator in PAKCS as a source-to-source transformation on FlatCurry programs. The process would be automatic and transparent to the user.

6 Conclusions

This paper describes a successful experience in the development of a program transformation technique for a realistic multi-paradigm declarative language. Our method builds on the theoretical basis of [3] for the PE of functional logic languages. We have shown how this framework can be successfully applied in practice by extending the underlying method in order to cover the facilities of the language Curry. A partial evaluator has been implemented which is able to generate efficient (and reasonably small) specialized programs.

Future work in partial evaluation of multi-paradigm functional logic languages should still address several issues. For a more effective deployment, *off-line* partial evaluators perform a binding-time analysis which annotates each function call in the source program as either reducible or subject to residualize. Although our PE scheme is essentially *on-line*, we think that it could be also improved with the information gathered by a pre-processing consisting of a binding-time analysis. In particular, this will allow us to use the standard semantics of the language over those expressions which can be fully evaluated in a finite number of steps, thus improving the effectiveness of the PE process.

References

1. 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`.
2. E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. of LPAR'99*, pages 376–395. Springer LNAI 1705, 1999.
3. E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of LPAR'2000*, pages 381–398. Springer LNAI 1955, 2000.
4. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Trans. on Programming Lang. and Systems*, 20(4):768–844, 1998.
5. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.
6. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

7. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of FroCoS'2000*, pages 171–185. Springer LNCS 1794, 2000.
8. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.
9. C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of POPL'93*, pages 493–501. ACM, New York, 1993.
10. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
11. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
12. A.J. Gill, J. Launchbury, and S.L. Peyton Jones. A Short Cut to Deforestation. In *Proc. of the FPCA'93*, pages 223–232, New York, NY, USA, 1993. ACM Press.
13. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of POPL'97*, pages 80–93. ACM, New York, 1997.
14. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of PPDP'99*, pages 376–395. Springer LNCS 1702, 1999.
15. M. Hanus. Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~curry/`, 2000.
16. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *Proc. of PADL'00*, pages 47–62. Springer LNCS 1753, 2000.
17. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of PADL'01*, Springer LNCS (to appear), 2001.
18. M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.3: The Portland Aachen Kiel Curry System User Manual. University of Kiel, Germany, 2000.
19. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
20. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
21. M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In G. Levi, editor, *Proc. of SAS'98*, pages 230–245. Springer LNCS 1503, 1998.
22. J.W. Lloyd. Combining Functional and Logic Programming Languages. In *Proc. of the International Logic Programming Symposium*, pages 43–57, 1994.
23. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
24. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
25. W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of WFLP'99*, pages 171–181, 1999.
26. B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.
27. S.L. Peyton-Jones. Compiling Haskell by Program Transformation: a Report from the Trenches. In *Proc. of ESOP'96*, pages 18–44. Springer LNCS 1058, 1996.
28. M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. MIT Press, 1995.
29. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
30. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.