Offline Narrowing-Driven Specialization in Practice*

G. Arroyo, J.G. Ramos, S. Tamarit, G. Vidal
DSIC, Technical University of Valencia
Camino de Vera s/n, 46022, Valencia, Spain
{garroyo, guadalupe, stamarit, gvidal}@dsic.upv.es

Abstract

Offline narrowing-driven partial evaluation is a recent technique for the specialization of functional and functional logic programs. In this work, we describe novel control strategies that can be used to design a powerful narrowing-driven partial evaluator.

1 Introduction

Partial evaluation [12] is a well-known technique for the specialization of programs. Basically, given a program and some restriction on its use (e.g., part of its input data), a partial evaluator returns a new—hopefully more efficient—*residual* program which is specialized for the given restriction.

Partial evaluation methods can be broadly classified into *online* or *offline* methods. Online partial evaluators perform a single, monolithic process which combines symbolic evaluation, propagation of partial values, and some dynamic analysis to ensure the termination of the process. Offline partial evaluators, on the other hand, have two clearly separated stages. The goal of the first stage, often known as *binding-time analysis* (BTA), is the inclusion of some program annotations to guide the proper specialization process. For this purpose, a BTA usually includes some static analysis to propagate known (abstract) values through the entire program as well as a termination analysis. Then, the second stage takes the annotated program and the actual values of the known inputs and produces the associated residual program mainly by following the program annotations.

In principle, offline specialization is less powerful—i.e., residual programs may run slower—but also more efficient—i.e., partial evaluators often run faster. Therefore, offline specialization scales up better to large applications (e.g., they can be used for "compiling" a program by partially evaluating an interpreter w.r.t. a given source program).

Functional logic languages amalgamate the main features of functional and logic languages [9]. In this context, partial evaluation is often driven by the standard operational semantics of these languages, i.e., *narrowing* [19]. The so called *narrowing-driven* approach [2] to partial evaluation, originally introduced as an online method, has recently been adapted to the offline style in [18, 4]. These works are mainly focused on the termination of the process, introducing a syntactic characterization of programs [18] and a (quasi-)termination[1] analysis [4] that can be used to annotate the source program. However, control issues have not been considered yet.

In this work, we present an annotation procedure that is based on the quasi-termination analysis of [4]. Then, we introduce an offline specialization algorithm that distinguishes two different levels. The *global* level ensures that the number of different specialized functions is kept finite. The *local* level takes a function

*This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2005-09207-C03-02 and *Acción Integrada* HA2006-0008.

[1] A computation quasi-terminates when it only contains a finite number of different terms modulo variable renaming.

call and builds a finite (possibly partial) evaluation of this call. The resulting method is purely offline and, thus, very efficient.

Finally, we also discuss a hybrid algorithm that includes some simple tests during partial evaluation so that the quality of the residual programs migth be improved.

2 The Language

In this section, we present the syntax of *flat* programs [10], a convenient standard representation for functional logic programs which makes explicit the pattern matching strategy by case expressions. This flat representation constitutes the kernel of modern functional logic languages like Curry [7] or Toy [16]. Similar representations are considered in [10, 11, 17]. Unlike them, we consider two kinds of case expressions in order to represent *flexible/rigid* evaluation annotations of source programs as expressions. Since inductively sequential programs [3] (with evaluation annotations) can be automatically translated into the flat representation, our approach covers recent proposals for multi-paradigm functional logic programming. The syntax for programs in the flat representation is as follows:

$$\begin{array}{lcl} \mathcal{R} & ::= & \mathcal{D}_1 \ldots \mathcal{D}_m \\ \mathcal{D} & ::= & f(\overline{x_n}) = e \\ e & ::= & x \\ & | & c(\overline{e_n}) \\ & | & f(\overline{e_n}) \\ & | & \mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} \\ & | & \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} \\ p & ::= & c(\overline{x_n}) \end{array}$$

Here, we write $\overline{o_n}$ for the sequence of objects $o_1, \ldots, o_n$. Thus, a program $\mathcal{R}$ consists of a sequence of function definitions D such that the left-hand side is linear and has only variable arguments, i.e., pattern matching is compiled into case expressions. The right-hand side of each function definition is an expression e composed by variables ($\mathcal{X}$), constructors ($\mathcal{C}$), function calls ($\mathcal{F}$), and case expressions for pattern matching. Variables are denoted by $x, y, z \ldots$, constructors by $a, b, c \ldots$, and defined functions by $f, g, h \ldots$ The general form of a case expression is:

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

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 case structures). The variables $\overline{x_{n_i}}$ are local variables which occur only in the corresponding expression e_i. The difference between *case* and *fcase* shows up when the argument e is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression and proceeds with the appropriate branch (which corresponds to narrowing). Functions defined only by *fcase* or *case* expressions are called *flexible* or *rigid*, respectively.

An expression is *operation-rooted* if it is rooted by a defined function symbol. It is *constructor-rooted* if the root symbol is a constructor symbol.

For instance, the (flexible) function "`app`" to concatenate two lists can be written in the flat representation by the following single rule:

$$\begin{array}{l} \texttt{app}\ (\texttt{x}, \texttt{y}) = \texttt{fcase x of}\ \{ \\ \qquad\qquad [\,] \to \texttt{y}; \\ \qquad (\texttt{z} : \texttt{zs}) \to \texttt{z} : \texttt{app}\ (\texttt{zs}, \texttt{y})\ \} \end{array}$$

The operational semantics of flat programs is based on the LNT (Lazy Narrowing with definitional Trees) calculus [10]. In Section 4.2 we present a slight extension of this semantics for performing computations at partial evaluation time.

3 Quasi-Termination Analysis and Program Annotation

We first adapt the quasi-termination analysis of [4], originally introduced for term rewriting systems, to the flat language. Then, we present an annotation procedure that is based on this quasi-termination analysis.

3.1 Quasi-Termination Analysis

A transitive and antisymmetric binary relation $\succ$ is an *order* and a transitive and reflexive binary relation $\succsim$ is a *quasi-order*. A binary relation $\succ$ is *well founded* iff there exist no infinite decreasing sequence $t_0 \succ t_1 \succ t_2 \succ \ldots$ An order $\succ$ is *closed under substitutions* (or *stable*) if $s \succ t$ implies $\sigma(s) \succ \sigma(t)$ for all terms s, t and substitution σ.

The quasi-termination analysis is based on the notion of *size-change* graph [13], which is used to trace size changes of function arguments when going from one call to another. In order to adapt the original notion of size-change graph to flat programs, we first need the following auxiliary definition.

Definition 1 (call pairs) *Given a function definition $f(\overline{x_k}) = e$, we let $pairs(f(\overline{x_k}), e)$ be the associated set of* call pairs, *which is inductively defined as shown in Fig. 1.*[2]

Following [20], our size-change graphs are parameterized by a reduction pair $(\succsim, \succ)$, where $\succsim$ is a quasi-order, $\succ$ is a well-founded order, both $\succsim$ and $\succ$ are closed under substitutions and compatible (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary). Furthermore, we also require that $s\ R\ t$ implies $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ for all $R \in \{\succsim, \succ\}$ and terms s and t.

Definition 2 (size-change graph) *Let $(\succsim, \succ)$ be a reduction pair. For every definition $f(\overline{x_n}) = e$ of a flat program $\mathcal{R}$ and every call pair $(f(\overline{s_n}), g(\overline{t_m})) \in pairs(f(\overline{x_n}), e)$, we have a size-change graph as follows:*

- *The graph has n output nodes marked with $\{1_f, \ldots, n_f\}$ and m input nodes marked with $\{1_g, \ldots, m_g\}$.*
- *If $s_i \succ t_j$, then there is a directed edge marked with $\succ$ from i_f to j_g. Otherwise, if $s_i \succsim t_j$, then there is an edge marked with $\succsim$ from i_f to j_g.*

A size-change graph is thus a bipartite labelled graph $G = (V, W, E)$ where $V = \{1_f, \ldots, n_f\}$ and $W = \{1_g, \ldots, m_g\}$ are the labels of the output and input nodes, respectively, and we have edges $E \subseteq V \times W \times \{\succsim, \succ\}$.

In order to analyze the termination of a program, it suffices to focus on its loops. For this purpose, we now compute the transitive closure of the size-change relations as follows:

Definition 3 (multigraph, concatenation) *Every size-change graph of $\mathcal{R}$ is a multigraph of $\mathcal{R}$. If $G = (\{1_f, \ldots, n_f\}, \{1_g, \ldots, m_g\}, E_1)$ and $H = (\{1_g, \ldots, m_g\}, \{1_h, \ldots, p_h\}, E_2)$ are multigraphs of $\mathcal{R}$ w.r.t. the same reduction pair $(\succsim, \succ)$, then the concatenation $G \cdot H = (\{1_f, \ldots, n_f\}, \{1_h, \ldots, p_h\}, E)$ is also a multigraph of $\mathcal{R}$. For $1 \leq i \leq n$ and $1 \leq k \leq p$, E contains an edge from i_f to k_h iff E_1 contains an edge from i_f to some j_g and E_2 contains an edge from j_g to k_h. Furthermore, if some of the edges are labelled with "$\succ$", then the edge in E is labelled with "$\succ$" as well. Otherwise, it is labelled with "$\succsim$".*

A multigraph G is idempotent if $G = G \cdot G$ (which implies that its input and output nodes are both labelled with $\{1_f, \ldots, n_f\}$ for some f). In the following, we will only focus on the idempotent multigraphs of a program, since they represent its (potential) loops.

Example 4 *Consider the deforestation example shown in Fig. 2 together with its associated call pairs. This example is a slight modification of the* `applast` *benchmark in the DPPD (Dozens of Problems for Partial Deduction [14]) library to better illustrate the notion of size-change graph.*

Let $(\succsim, \succ)$ be a reduction pair such that $s \succsim t$ if s and t are equal up to variable renaming and $s \succ t$ if s is a strict subterm of t modulo variable renaming. Then, we have five size-change graphs associated to the five call pairs which are depicted in Fig. 3. Finally, the three idempotent multigraphs are shown in Fig. 4.

In the following, we are interested in a notion of termination which is called *PE-termination* [4] (a particular case of quasi-termination).

[2] Here, we assume that case expressions only occur in outermost positions. This is a reasonable assumption since the flat programs obtained by translating source programs always fulfill it [10].

$$pairs(l,e) = \begin{cases} \bigcup_{i=1}^{k} pairs(\{x \mapsto p_i\}(l), e_i) & \text{if } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\}, \\ \{(l,r) \mid r \text{ is an operation-rooted subterm of } e\ \} & \text{otherwise} \end{cases}$$

Figure 1: Auxiliary function *pairs*

$$\begin{array}{lcll} d_1 & \equiv & \texttt{applast(xs,x)} & = & \texttt{last(append(xs,[x]))} \\ d_2 & \equiv & \texttt{last(xs)} & = & \texttt{fcase xs of \{ (y : ys) → last'(ys,y) \}} \\ d_3 & \equiv & \texttt{last'(ys,y)} & = & \texttt{fcase ys of \{ [ ] → [y]; (w : ws) → last(w : ws) \}} \\ d_4 & \equiv & \texttt{append(xs,ys)} & = & \texttt{fcase xs of \{ [ ] → ys; (w : ws) → w : append(ws,ys) \}} \end{array}$$

$$\begin{array}{lcl} pairs(d_1) & = & \{(\texttt{applast(xs,x)}, \texttt{append(xs,[x])}),\ (\texttt{applast(xs,x)}, \texttt{last(append(xs,[x]))})\} \\ pairs(d_2) & = & \{(\texttt{last(y : ys)}, \texttt{last'(ys,y)})\} \\ pairs(d_3) & = & \{(\texttt{last'(w : ws,y)}, \texttt{last(w : ws)})\} \\ pairs(d_4) & = & \{(\texttt{append(w : ws,ys)}, \texttt{append(ws,ys)})\} \end{array}$$

Figure 2: Deforestation example `applast` and its call pairs

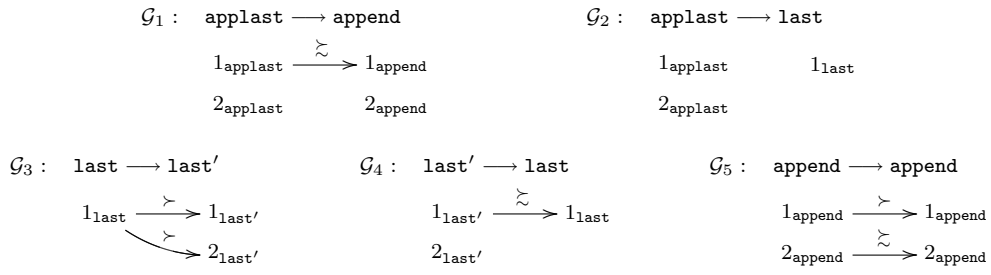

Figure 3: Size-change graphs for `applast`

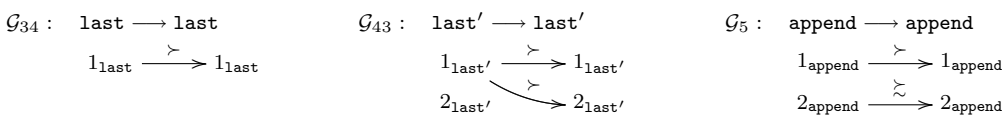

Figure 4: Idempotent multigraphs for `applast`

Definition 5 (PE-termination) *A computation is PE-terminating if only a finite number of nonvariant function calls are unfolded. A flat program is PE-terminating if every possible computation is PE-terminating.*

Basically, a PE-terminating computation is a (possibly infinite) computation in which only a finite number of function calls (modulo variable renaming) is unfolded.

Now, we consider that the output of a simple (monovariant) binding-time analysis (BTA) is available (see, e.g., [12]). Informally speaking, given a program and the information on which parameters of the initial function call are static and which are dynamic, a BTA maps each function to a list of static/dynamic values. Here, we consider that a static parameter is definitely known at specialization time (hence it is ground), while a dynamic parameter is possibly unknown at specialization time.

In the following, we will also require the component $\succsim$ of a reduction pair $(\succsim, \succ)$ to be *bounded*, i.e., the set $\{s \mid t \succsim s\}$ must contain a finite number of nonvariant terms for any term t. Some closely related notions are that of *rigidity* and *instantiated enough*, both defined w.r.t. a so called *norm*. These notions are used in many termination analyses for logic programs (e.g., [5, 15]).

The following theorem is a straightforward extension of a similar result in [4] for term rewrite systems.

Theorem 6 *Let $\mathcal{R}$ be a flat program and let $(\succsim, \succ)$ be a reduction pair. $\mathcal{R}$ is PE-terminating w.r.t. any linear term if every idempotent multigraph associated to a function f/n contains either*

(i) at least one edge $i_f \xrightarrow{\succ} i_f$ for some $i \in \{1, \ldots, n\}$ such that i_f is static, or

(ii) an edge $i_f \xrightarrow{R} i_f$, $R \in \{\succsim, \succ\}$, for all $i = 1, \ldots, n$, such that $\succsim$ is bounded.

Also, we require $\mathcal{R}$ to be right-linear w.r.t. the dynamic variables, i.e., no repeated occurrence of the same dynamic variable may occur in a right-hand side.

Boundedness of "$\succsim$" in the second case (ii) above is necessary to ensure that no infinite sequences of nonvariant function calls with arguments of the same "size" according to $\succsim$ are allowed. Right-linearity of dynamic variables is required to avoid the propagation of terms between the parameters of the same function. The reader is referred to [4] for further details.

3.2 Program Annotation

In this section, we present an annotation procedure that is based on the quasi-termination analysis presented in the previous section.

The following definition introduces our annotation procedure. Here, we consider that the program $\mathcal{R}$, the reduction pair $(\succsim, \succ)$,[3] the idempotent multigraphs of $\mathcal{R}$, and the output of a BTA, are global parameters.

Definition 7 (program annotation) *The program is annotated by replacing every rule $f(\overline{x_n}) = e$ in $\mathcal{R}$ by a new rule $f(\overline{x_n}) = ann^l(ann^g(ann^u(e)))$.*[4]

Function ann^u is used to add unfolding *annotations as shown in Fig. 5: a function f is annotated as f^u if it can be safely unfolded and f^m otherwise (where m stands for* memo*).*[5]

Function ann^g is used to add generalization *annotations (their use will be explained in the next section). Its definition is shown in Fig. 6, where $\mathcal{F}^{an}$ denotes the domain of annotated functions.*

Finally, function ann^l is used to linearize expressions as shown in Fig. 7.

Example 8 *Consider again the program* `applast` *shown in Fig. 2. Let us consider its specialization w.r.t. a known value of the first argument of* `applast`*. In this case, a BTA would return the following division:*

$$\{\ \texttt{applast} \mapsto (\texttt{S},\texttt{D}),\quad \texttt{append} \mapsto (\texttt{S},\texttt{D}),\quad \texttt{last} \mapsto (\texttt{D}),\quad \texttt{last}' \mapsto (\texttt{D},\texttt{D})\ \}$$

[3] There are many techniques to search for such reduction pairs automatically (LPO, polynomial interpretations, etc., see, e.g., [6]).

[4] We use three independent functions for clarity. The implementation only requires a single pass to add all annotations.

[5] As mentioned before, we only consider variables as arguments of case expressions.

$$ann^u(e) = \begin{cases} x & \text{if } e \equiv x \in \mathcal{X} \\ c(\overline{ann^u(e_n)}) & \text{if } e \equiv c(\overline{e_n}),\ c \in \mathcal{C} \\ (f)case\ x\ of\ \{\overline{p_k \to ann^u(e_k)}\} & \text{if } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\} \\ f^u(\overline{ann^u(e_n)}) & \text{if } e \equiv f(\overline{e_n}),\ f \in \mathcal{F}, \text{ and every idempotent multigraph associated to } f/n \text{ contains at least one edge } i_f \xrightarrow{\succ} i_f \text{ for some } i \in \{1,\ldots,n\} \text{ such that } i_f \text{ is static} \\ f^m(\overline{ann^u(e_n)}) & \text{otherwise, where } e \equiv f(\overline{e_n}) \end{cases}$$

Figure 5: Annotation function ann^u

Input: a program $\mathcal{R}$ and a set of calls T
Output: a set of calls 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 variable renaming)
Return: $S := T_i$

Figure 8: Generic procedure for NPE

where `S` *denotes that an argument is static (ground) and* `D` *that it is dynamic. Here, our annotation procedure returns*

```
applast(xs,x)=last^m(append^u(xs,[x]))
last(xs)      =fcase xs of
                 { (y : ys) → last'^m(ys,y) }
last'(ys,y)   =fcase ys of
                 { [] → [y];
                   (w : ws) → last^m(w : ws)  }
append(xs,ys)=fcase xs of
                 { [] → ys;
                   (w : ws) → w : append^u(ws,ys) }
```

4 Control Issues

In this section, we first recall the generic procedure for narrowing-driven partial evaluation and, then, present our novel strategies for the global and local control levels.

The generic procedure is shown in Figure 8. Similarly to Gallagher's procedure for the partial evaluation of logic programs [8], our algorithm clearly distinguishes two different levels:

Local level. Given a set of operation-rooted terms (i.e., function calls), the local level applies an unfolding operator *unfold* so that a set of residual rules is returned (see Section 4.2). The unfolding operator should ensure that the unfolding process is finite, i.e., that no partial computation runs forever.

Global level. This level should ensure that the number of different specialized functions is kept finite. For this purpose, an abstraction operator *abstract* is used. The abstraction operator takes a finite set of operation-rooted terms T_i and then properly adds the set of operation-rooted subterms in the right-hand sides of the unfolded calls, which is denoted by $\mathcal{R}'_{calls}$. The new set T_{i+1} may need further evaluation and, thus, the process is iteratively repeated while new terms are introduced.

Observe that this procedure does not return a partially evaluated program but a finite set of operation-rooted terms. The residual program, however, can easily be built by applying the unfolding operator to the returned set of terms and, then, renaming the rules by using a standard post-unfolding phase (see Section 4.2).

4.1 Global Control

Our abstraction operator is based on the following property.

Consider a (possibly infinite) computation in a program annotated according to Definition 7 and let $t_1, t_2, t_3, \ldots$ be any sequence of operation-rooted terms in this computation. Let $abs(t)$ be a function that replaces every annotated subterm $gen(t')$ in t (if any) by a fresh variable. Then, the sequence $abs(t_1), abs(t_2), abs(t_3), \ldots$ is quasi-terminating.

$$ann^g(e) = \begin{cases} x & \text{if } e \equiv x \in \mathcal{X} \\ c(\overline{ann^g(e_n)}) & \text{if } e \equiv c(\overline{e_n}),\ c \in \mathcal{C} \\ (f)case\ x\ of\ \{\overline{p_k \to ann^g(e_k)}\} & \text{if } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\} \\ f^{an}(\overline{e'_n}) & \text{if } e \equiv f^{an}(\overline{e_n}),\ f^{an} \in \mathcal{F}^{an}, \text{ and} \\ & \quad e'_i = \begin{cases} ann^g(e_i) & \text{if every idempotent multigraph} \\ & \text{associated to } f/n \text{ contains an} \\ & \text{edge } i_f \xrightarrow{R} i_f,\ R \in \{\succsim, \succ\} \\ e'_i = gen(ann^g(e_i)) & \text{otherwise} \end{cases} \end{cases}$$

Figure 6: Annotation function ann^g

$$ann^l(e) = \begin{cases} (f)case\ x\ of\ \{\overline{p_k \to ann^l(e_k)}\} & \text{if } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\} \\ linear(e) & \text{otherwise} \end{cases}$$

where function *linear* annotates every occurrence of a dynamic variable not yet annotated but one (e.g., the leftmost one)

Figure 7: Annotation function ann^l and auxiliary function *linear*

Therefore, our abstraction operator is based on replacing annotated subterms by fresh variables. In the following, we denote by $t \in \overline{T}$ the fact that there is a term $t' \in T$ such that t and t' are equal modulo variable renaming.

Definition 9 (abstraction operator) *Let T_1, T_2 be finite sets of terms. Then, $abstract(T_1, T_2)$ is defined as follows:*

$$abstract(T_1, T_2) = \begin{cases} T_1 & \text{if } T_2 = \{\,\} \\ abstract(T_1, T'_2) & \text{if } T_2 = \{t\} \cup T'_2 \\ & \text{and } gen(t) \in \overline{T}_1 \\ abstract(T_1 \cup \{t'\}, T'_2) & \text{if } T_2 = \{t\} \cup T'_2 \\ & \text{and } t' = gen(t) \notin \overline{T}_1 \end{cases}$$

4.2 Local Control

Now, we introduce our unfolding operator. It is driven by the unfolding annotations, so that functions of the form f^u should be unfolded while functions f^m should not. Clearly, every computation in which only functions f^u are unfolded should be finite.

Computations are performed with a slight extension of the RLNT calculus [1] as shown in Fig. 9. First, note that the symbols "$[\![$" and "$]\!]$" in an expression like $[\![e]\!]$ are purely syntactical (i.e., they do not denote "the value of e"). Indeed, they are only used to mark subexpressions where the inference rules may be applied. Let us briefly explain the rules of the calculus:

The first three rules deal with function calls. If the function is annotated with u, then rule Unfold performs a function unfolding. If it is annotated with m, rule Memo suspends the evaluation of the call. Finally, rule Gen is simply used to ignore the generalization annotations. Observe that the evaluated expression never contains annotations u nor m, since they are not needed in the global level.

The last four rules deal with case expressions. Rule Select is used to select matching branch of a case expression when its argument is a constructor-rooted term. Rule Guess applies when the argument is a free variable; here, 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). Rule Eval is used to evaluate case expressions with a function call or another case expression in the argument position. Here, $root(e)$ denotes the outermost symbol of e. Finally, rule Case-of-Case moves the outer case inside the branches of the inner one and, thus, the evaluation of the branches

Unfold

$$[\![f^u(\overline{e_n})]\!] \;\Rightarrow\; [\![\sigma(e')]\!] \quad \text{if } f(\overline{x_n}) = e' \in \mathcal{R} \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Memo

$$[\![f^m(\overline{e_n})]\!] \;\Rightarrow\; f(\overline{e_n})$$

Gen

$$[\![gen(e)]\!] \;\Rightarrow\; gen([\![e]\!])$$

Select

$$[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}]\!] \;\Rightarrow\; [\![\sigma(e'_i)]\!] \quad \text{if } p_i = c(\overline{x_n}) \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\},\ i \in \{1,\ldots,k\}$$

Guess

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

Eval

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

Case-of-Case

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

Figure 9: The offline RLNT calculus

can now proceed (similar rules can be found in the Glasgow Haskell Compiler as well as in Wadler's deforestation [21]).

Observe that RLNT computations with an annotated program are always finite as an easy consequence of Theorem 6.

Our unfolding operator can now be defined as follows:

Definition 10 (unfolding operator) *Given a flat program $\mathcal{R}$ and a set of terms T, we let $unfold(T, \mathcal{R}) =$*

$$\{f(\overline{e_n}) = [\![\sigma(e)]\!] \mid f(\overline{e_n}) \in T,\ f(\overline{x_n}) = e \in \mathcal{R},\ \textit{and } \sigma = \{\overline{x_n \mapsto e_n}\}\ \}$$

Observe that the unfolding operator does not return a legal flat program. This is not relevant during the specialization process. Once the iterative process finishes, one can add a standard post-processing of renaming that replaces every left-hand side of the form $f(\overline{e_n})$ by $f(\overline{x_m})$ where $\overline{x_m}$ are the different variables of $\overline{e_n}$ in the same order in which they occur and, then, renames accordingly the expressions in the right-hand sides.

Example 11 *Consider again the annotated program of Ex. 8. The partial evaluation w.r.t. the initial set of calls $T_0 = \{$*`applast`$([1], \mathtt{x})\}$ *produces the following sequence of calls (according to the algorithm in Fig. 8):*

$$\begin{array}{l} T_1 = T_0 \cup \{\mathtt{last}(\mathtt{append}([1], [\mathtt{x}]))\} \\ T_2 = T_1 \cup \{\mathtt{last'}(\mathtt{append}([\,], [\mathtt{x}]), 1)\} \\ T_3 = T_2 \cup \{\mathtt{last}([\mathtt{x}])\} \\ T_4 = T_3 \cup \{\mathtt{last'}([\,], \mathtt{x})\} \end{array}$$

and the algorithm stops since T_5 is equal to T_4 modulo variable renaming.

Using our prototype implementation of the partial evaluator, we get the following associated resultants (we write `app` *for* `append`*):*

$$\begin{array}{rcl} \mathtt{applast}([1], \mathtt{x}) & = & \mathtt{last}(\mathtt{app}([1], [\mathtt{x}])) \\ \mathtt{last}(\mathtt{app}([1], [\mathtt{x}])) & = & \mathtt{last'}(\mathtt{app}([\,], [\mathtt{x}]), 1) \\ \mathtt{last'}(\mathtt{app}([\,], [\mathtt{x}]), 1) & = & \mathtt{last}([\mathtt{x}]) \\ \mathtt{last}([\mathtt{x}]) & = & \mathtt{last'}([\,], \mathtt{x}) \\ \mathtt{last'}([\,], \mathtt{x}) & = & [\mathtt{x}] \end{array}$$

so that, after a simple process of renaming and simplification, the final result is simply the following rule (the optimal specialization):

$$\mathtt{applast}_1(\mathtt{x}) \;=\; [\mathtt{x}]$$

4.3 Refining the Local Control

Finally, we present a simple refinement of the unfolding operator presented in the previous section.

Unfold

$[\![f(\overline{e_n})]\!]^T \;\Rightarrow\; [\![\sigma(e')]\!]^{T \cup \{gen(f(\overline{e_n}))\}}$ if $gen(f(\overline{e_n})) \notin \overline{T}$, $f(\overline{x_n}) = e' \in \mathcal{R}$ and $\sigma = \{\overline{x_n \mapsto e_n}\}$

Memo

$[\![f(\overline{e_n})]\!]^T \;\Rightarrow\; f(\overline{[\![e_n]\!]^T})$ if $gen(f(\overline{e_n})) \in \overline{T}$

Figure 10: The hybrid RLNT calculus

The basic idea is as follows: we consider that the annotation process does not include u and m annotations so that the local level applies a termination test which is similar to that in the global level. For this purpose, the offline RLNT calculus is modified as follows:

During the evaluation, we have expressions of the form $[\![e]\!]^T$ where T records the calls already evaluated. Now, the initial expression has the form $[\![e]\!]^{\{\}}$.

The first two rules of the calculus are redefined as shown in Fig. 10. Basically, we unfold those function calls that, after replacing annotated subterms by fresh variables, are equal up to variable renaming to some previously unfolded call. In this case, the generalized call is added to the current set of memorized calls. Otherwise, the call is not unfolded and we proceed by evaluating its arguments.

The remaining rules just propagate the current set of memorized calls.

The termination of the new local strategy is still an easy consequence of Theorem 6. The main difference with the previous local strategy is that, now, it is not a pure offline strategy since we perform some (simple online) tests in the local level, thus we call it *hybrid*.

Table 1 shows the results from an experimental evaluation of both strategies. In general, both strategies achieve similar improvements and are equally efficient.

Example 12 *Consider again Example 8 but taking into accout the refined local control. Then, we have the following computation*

$$\begin{array}{l}\texttt{applast}([\texttt{1}],\texttt{x})^{\{\}} \\ \quad \Rightarrow \texttt{last}(\texttt{app}([\texttt{1}],[\texttt{x}]))^{\{\} \cup \{\texttt{applast}([\texttt{1}],\texttt{x})\}} \\ \quad \Rightarrow \ldots \Rightarrow\ [\texttt{x}]\end{array}$$

so that the renamed resultant is:

$$\texttt{applast}_1(\texttt{x}) \;=\; [\texttt{x}]$$

i.e., the call was completely unfolded.

5 Conclusions

In this work, we have introduced appropriate control strategies to design an offline narrowing-driven partial evaluator. An implementation of a partial evaluator that follows the ideas presented so far has been undertaken. Our preliminary results are encouraging.

References

[1] E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters*, 85(1):19–25, 2003.

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

[3] 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.

[4] G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Proc. of LOPSTR'06*, pages 60–76. Springer LNCS 4407, 2007.

[5] M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[6] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

[7] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.

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

Table 1: Benchmark results

benchmark	codesize (bytes)	original runtime	Hybrid spec. time	Hybrid runtime spec.	Hybrid speedup	Offline spec. time	Offline runtime spec.	Offline speedup
ackermann	759	1881	900	562	3,35	880	566	3,32
allones	683	1468	180	1448	1,01	200	1497	0,98
applast	663	1175	200	1161	1,01	210	1097	1,07
flip	903	794	220	802	0,99	280	789	1,01
gauss	2919	231	660	230	1,00	780	235	0,98
interSB	1768	168	1690	165	1,02	2280	176	0,95
kmp3B ∗ A	30603	96	19170	78	1,23	20020	52	1,85
power	816	524	1040	567	0,92	1230	544	0,96
revAccum	943	1	3190	0.1	10.00	1190	0.1	10
Average	**4451**	**704**	**3028**	**557**	**2**	**3008**	**551**	**2**

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

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

[11] T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of FLOPS 2001*, pages 216–232. Springer LNCS 2024, 2001.

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

[13] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.

[14] M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks. Available at URL: `www.ecs.soton.ac.uk/~mal/systems/dppd.html`.

[15] N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. of Int'l Conf. on Logic Programming (ICLP'97)*, pages 63–77. The MIT Press, 1997.

[16] 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.

[17] 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.

[18] J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of ICFP'05*, pages 228–239. ACM Press, 2005.

[19] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[20] R. Thiemann and J. Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

[21] P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.