Preserving Sharing in the Partial Evaluation of Lazy Functional Programs[⋆]

Sebastian Fischer[1], Josep Silva[2], Salvador Tamarit[2], and Germán Vidal[2]

[1] University of Kiel, Olshausenstr. 40, D-24098 Kiel, Germany.
sebf@informatik.uni-kiel.de
[2] Technical University of Valencia, Camino de Vera S/N, E-46022 Valencia, Spain.
{jsilva,stamarit,gvidal}@dsic.upv.es

Abstract. The goal of partial evaluation is the specialization of programs w.r.t. part of their input data. Although this technique is already well-known in the context of functional languages, current approaches are either overly restrictive or destroy sharing through the specialization process, which is unacceptable from a performance point of view. In this work, we present the basis of a new partial evaluation scheme for first-order lazy functional programs that preserves sharing through the specialization process and still allows the unfolding of arbitrary function calls.

1 Introduction

Partial evaluation [12] is an automatic technique for the specialization of programs. Currently, one can find partial evaluation techniques for a variety of programming languages, like C [5], Curry [17], Prolog [14], Scheme [8], etc.

In this work, we focus on solving a problem associated with the partial evaluation of *lazy* functional languages. In these languages (e.g., Haskell [15]), it is essential to *share* program variables in order to avoid losing efficiency due to the repeated evaluation of the same expression. Consider, e.g., the following program excerpt (we use [ ] and ":" as constructors of lists):

$$\begin{array}{lll} \mathtt{sumList}([\,]) & = \mathtt{0} & \mathtt{incList}(\mathtt{n},[\,]) \quad = [\,] \\ \mathtt{sumList}(\mathtt{x}:\mathtt{xs}) & = \mathtt{x} + \mathtt{sumList}(\mathtt{xs}) & \mathtt{incList}(\mathtt{n},\mathtt{x}:\mathtt{xs}) = (\mathtt{x}+\mathtt{n}) : \mathtt{incList}(\mathtt{n},\mathtt{xs}) \end{array}$$

where function `sumList` sums the elements of a list and `incList` increments the elements of a list by a given number. Let us now consider different alternatives for the partial evaluation of the following expression:

$$\mathtt{sumList}(\mathtt{incList}(\boxed{e},[\mathtt{a},\mathtt{b}]))$$

where $\boxed{e}$ is a closed expression (i.e., without free variables) whose evaluation is expensive, $\mathtt{a},\mathtt{b}$ are natural numbers, and $[\mathtt{a},\mathtt{b}]$ is a shorthand for $\mathtt{a}:\mathtt{b}:[\,]$.

⋆ 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.

First Try: Downgrading Program Efficiency. A naive partial evaluator may reduce the previous expression as follows:

$$\begin{array}{l}\texttt{sumList}(\texttt{incList}(\boxed{e}\,,[\texttt{a},\texttt{b}]))\\ \qquad\qquad\qquad\qquad \Rightarrow\ \texttt{sumList}(\texttt{a}+\boxed{e}\,):\texttt{incList}(\boxed{e}\,,[\texttt{b}]))\\ \qquad\qquad\qquad\qquad \Rightarrow\ \texttt{sumList}(\texttt{a}+(\boxed{e1}\,+42)):\texttt{incList}(\boxed{e}\,,[\texttt{b}]))\end{array}$$

where we assume that $\boxed{e}$ is unfolded to $\boxed{e1}\,+42$ in one reduction step. Now, we would build the following residual rule—called *resultant*—associated to the above partial computation:

$$\texttt{new\_function}\ =\ \texttt{sumList}(\texttt{a}+(\boxed{e1}\,+42)):\texttt{incList}(\boxed{e}\,,[\texttt{b}]))$$

Then, if we evaluate `new_function` using the above rule, the original expression $\boxed{e}$ will be eventually evaluated twice since $\boxed{e}$ and $\boxed{e1}$ are not shared anymore. Actually, since their degree of evaluation is different (i.e., $\boxed{e1}$ comes from a one-step reduction of $\boxed{e}$), the identification of common subexpressions by means of some post-processing analysis is not generally feasible.

Clearly, the duplicate evaluation of $\boxed{e}$ is unacceptable from a performance point of view. Note that, in the original program, the expression $\boxed{e}$ is only evaluated once since the two occurrences of the variable `n` in the second rule of function `incList` are shared.

Second Try: Conservative Partial Evaluation. In order to avoid downgrading performance, partial evaluators of lazy languages usually include a restriction so that the unfolding of functions which are not *right-linear* (i.e., whose right-hand side contains multiple occurrences of the same variable) is forbidden.

In this case, the partial evaluation of $\texttt{sumList}(\texttt{incList}(\boxed{e}\,,[\texttt{a},\texttt{b}]))$ would mainly return the original program unchanged since the function `incList` is not right-linear and, thus, cannot be unfolded. Unfortunately, this strategy is often overly restrictive since it may happen that $\boxed{e}$ can be completely evaluated at partial evaluation time, thus allowing the subsequent reduction of `sumList`.

Our Proposal: Sharing-Based Partial Evaluation. Current partial evaluation techniques for lazy functional (logic) languages have mostly ignored the issue of sharing,[3] generally implementing the conservative approach.

In this work, we present an alternative to such trivial, overly restrictive treatment of sharing during partial evaluation. Basically, we allow the unfolding of arbitrary function calls but still ensure that sharing is never destroyed. For this purpose, our partial evaluation scheme is based on a lazy semantics that models sharing by means of an updatable heap. For instance, given the above expression, we could produce a partial computation of the form

$$\begin{array}{l}[\,]\ \&\ \texttt{sumList}(\texttt{incList}(\boxed{e}\,,[\texttt{a},\texttt{b}]))\\ \quad\Rightarrow\ [\texttt{n}\mapsto\boxed{e},\texttt{x}\mapsto\texttt{a},\texttt{xs}\mapsto[\texttt{b}]]\ \&\ \texttt{sumList}(\texttt{x}+\texttt{n}:\texttt{incList}(\texttt{n},\texttt{xs}))\\ \quad\Rightarrow\ [\texttt{n}\mapsto\boxed{e1}+42,\texttt{x}\mapsto\texttt{a},\texttt{xs}\mapsto[\texttt{b}]]\ \&\ \texttt{sumList}(\texttt{x}+\texttt{n}:\texttt{incList}(\texttt{n},\texttt{xs}))\end{array}$$

[3] We note that this is a critical issue that has been considered in the context of *inlining* (see, e.g., [16]), which could be seen like a rather simple form of partial evaluation.

$P ::= D_1 \ldots D_m$	(program)	*Domains*		
$D ::= f(x_1,\ldots,x_n) = e$	(function definition)			
$e ::= x$	(variable)	$P_1, P_2, \ldots \in$	*Prog*	(Programs)
$\mid\ c(x_1,\ldots,x_n)$	(constructor call)	$x, y, z, \ldots \in$	*Var*	(Variables)
$\mid\ f(x_1,\ldots,x_n)$	(function call)	$a, b, c, \ldots \in$	$\mathcal{C}$	(Constructors)
$\mid\ let\ \{\overline{x_k = e_k}\}\ in\ e$	(let binding)	$f, g, h, \ldots \in$	$\mathcal{F}$	(Functions)
$\mid\ case\ x\ of\ \{\overline{p_k \to e_k}\}$	(case expression)	$p_1, p_2, p_3, \ldots \in$	*Pat*	(Patterns)
$p ::= c(x_1,\ldots,x_n)$	(pattern)			

Fig. 1. Syntax for normalized flat programs

where *states* are informally denoted by a pair *heap* & *expression* and [ ] denotes the empty heap. Observe that there is a single binding for all occurrences of variable n and, thus, duplicated computations are not possible. Here, we would produce the following associated residual rule:

$$\texttt{new\_function}\ =\ \texttt{let n} = \boxed{e1} + 42\ \texttt{in sumList}(\texttt{a} + \texttt{n} : \texttt{incList}(\texttt{n}, [\texttt{b}]))$$

where the bindings for variables x and xs are inlined since they only occur once in the expression, and the binding for n that appears twice is kept in a let expression so that sharing is preserved.

In summary, our new approach is based on the definition of an unfolding strategy that extends a lazy semantics [1] (which models variable sharing by means of an updatable heap) in order to perform symbolic computations, i.e., in order to deal with free variables in expressions denoting missing information at partial evaluation time. Then, we introduce how residual rules should be extracted from these partial computations. For simplicity, we will not introduce the details of a complete partial evaluation scheme (but it would be similar to that of [2] by replacing the underlying partial evaluation semantics and the construction of residual rules from partial computations, i.e., control issues would remain basically unaltered).

2 Preliminaries

We consider in this work a simple, first-order lazy functional language. The syntax is shown in Fig. 1, where we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$. A program consists of a sequence of function definitions such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression composed by variables, data constructors, function calls, let bindings (where the local variables $\overline{x_k}$ are only visible in $\overline{e_k}$ and e), and case expressions of the form $case\ x\ of\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$, where x is a variable, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are introduced locally and bind the corresponding variables of e_i. We say that an expression is *closed* if it contains no occurrences of free variables (i.e., variables which are not bound by let bindings).

Observe that, according to Fig. 1, the arguments of function and constructor calls are variables. As in [13], this is essential to express sharing without the use of graph structures. This is not a serious restriction since source programs can be *normalized* so that they follow the syntax of Fig. 1 (see, e.g., [13, 1]).

Laziness of computations will show up in the description of the behavior of function calls and case expressions. In a function call, parameters are not evaluated but directly passed to the body of the function. In a case expression, the outermost symbol of the case argument is required. Therefore, the case argument should be evaluated to *head normal form* [7] (i.e., a variable or an expression with a constructor at the outermost position).

3 Partial Evaluation of Lazy Functional Programs

The main ingredients of our new proposal for preserving sharing through the specialization process are the following: i) partial computations are performed with a lazy semantics that models sharing by means of an updatable heap (cf. Sect. 3.1); ii) this semantics is then extended in order to perform symbolic computations during partial evaluation (cf. Sect. 3.2); iii) finally, we introduce a method to extract residual rules from partial computations (cf. Sect. 3.3).

3.1 The Standard Semantics

First, we present a lazy evaluation semantics for our first-order functional programs that models sharing. The rules of the small-step semantics are shown in Fig. 2 (they are a simplification of the calculus in [1], which in turn originates from an adaptation of Launchbury's natural semantics [13]). It follows these naming conventions:

$$\mathit{\Gamma}, \mathit{\Delta}, \mathit{\Theta} \in \mathit{Heap} \;=\; \mathit{Var} \to \mathit{Exp} \qquad\qquad v \in \mathit{Value} \;::=\; x \mid c(\overline{x_n})$$

A *heap* is a partial mapping from variables to expressions (the *empty heap* is denoted by $[\,]$). The value associated to variable x in heap $\mathit{\Gamma}$ is denoted by $\mathit{\Gamma}[x]$. $\mathit{\Gamma}[x \mapsto e]$ denotes a heap with $\mathit{\Gamma}[x] = e$, i.e., we use this notation either as a condition on a heap $\mathit{\Gamma}$ or as a modification of $\mathit{\Gamma}$. A *value* is a constructor-rooted term (i.e., a term whose outermost function symbol is a constructor symbol).

A *state* of the small-step semantics is a triple $\langle \mathit{\Gamma}, e, S\rangle$, where $\mathit{\Gamma}$ is the current heap, e is the expression to be evaluated (often called the *control* of the small-step semantics), and S is the stack which represents the current context. We briefly describe the transition rules:

In rule var, the evaluation of a variable x that is bound to an expression e proceeds by evaluating e and adding to the stack the reference to variable x. If a value v is eventually computed and there is a variable x on top of the stack, rule val updates the heap with $x \mapsto v$. This rule achieves the effect of sharing

var
$$\langle \Gamma[x \mapsto e], x, S\rangle \Rightarrow \langle \Gamma[x \mapsto e], e, x : S\rangle$$
val
$$\langle \Gamma, v, x : S\rangle \Rightarrow \langle \Gamma[x \mapsto v], v, S\rangle \qquad \text{where } v \text{ is a value}$$
fun
$$\langle \Gamma, f(\overline{x_n}), S\rangle \Rightarrow \langle \Gamma, \rho(e), S\rangle \qquad \text{where } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}$$
let
$$\langle \Gamma, let\ \{\overline{x_k = e_k}\}\ in\ e, S\rangle \Rightarrow \langle \Gamma[\overline{y_k \mapsto \rho(e_k)}], \rho(e), S\rangle \qquad \begin{array}{l}\text{where } \rho = \{\overline{x_k \mapsto y_k}\}\\ \text{and } \overline{y_n} \text{ are fresh variables}\end{array}$$
case
$$\langle \Gamma, case\ e\ of\ \{\overline{p_k \to e_k}\}, S\rangle \Rightarrow \langle \Gamma, e, \{\overline{p_k \to e_k}\} : S\rangle$$
select
$$\langle \Gamma, c(\overline{x_n}), \{\overline{p_k \to e_k}\} : S\rangle \Rightarrow \langle \Gamma, \rho(e_i), S\rangle \qquad \begin{array}{l}\text{where } p_i = c(\overline{y_n}) \text{ and } \rho = \{\overline{y_n \mapsto x_n}\},\\ \text{with } i \in \{1, \ldots, k\}\end{array}$$

Fig. 2. Small-step semantics for (sharing-based) lazy functional programs

since the next time the value of variable x is demanded, the value v will be immediately returned thus avoiding the repeated evaluation of e.

Rule fun implements a simple function unfolding. Here, $\rho : Var \to Var$ denotes a variable *substitution*. We assume that the considered program P is a global parameter of the calculus.

In order to reduce a let construct, rule let adds the bindings to the heap and proceeds with the evaluation of the main argument of *let*. Note that the variables introduced by the let construct are renamed with fresh names in order to avoid variable name clashes. For this purpose, we use *variable renamings*, a particular case of substitutions which are bijections on the domain of variables *Var*.

Rule case initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $\{\overline{p_k \to e_k}\}$ on top of the stack. If a value is eventually reached, then rule select is used to select the appropriate branch and continue with the evaluation of this branch.

In order to evaluate an expression e, we construct an *initial state* of the form $\langle [\,], e, [\,]\rangle$ and apply the rules of Fig. 2. We denote by $\Rightarrow^*$ the reflexive and transitive closure of $\Rightarrow$. A derivation $\langle [\,], e, [\,]\rangle \Rightarrow^* \langle \Gamma, e', S\rangle$ is *complete* if e' is a value and S is the empty stack.[4]

Example 1. Consider the following simple functions:

$$\begin{array}{ll}\mathtt{double(x)} & = \mathtt{add(x,x)}\\ \mathtt{add(n,m)} & = \mathtt{case\ n\ of\ \{Z \to m;\ S(u) \to let\ \{v = add(u,m)\}\ in\ S(v)\}}\end{array}$$

where natural numbers are built from Z and S. In order to evaluate the expression double(add(Z, Z)), we proceed as follows. First, we normalize it, i.e.,

$$exp \equiv \mathtt{let\ \{x_1 = Z,\ x_2 = Z\}\ in\ (let\ \{y = add(x_1,x_2)\}\ in\ double(y))}$$

[4] We ignore failing derivations (e.g., a case expression with no matching branch) in this work in order to keep the presentation simple.

$$
\begin{aligned}
\langle [\,], exp, [\,]\rangle &\Rightarrow_{\mathsf{let}} \langle \Gamma_1 \equiv [\mathtt{w}_1 \mapsto \mathtt{Z}, \mathtt{w}_2 \mapsto \mathtt{Z}],\ \mathtt{let}\ \{\mathtt{v} = \mathtt{add}(\mathtt{w}_1, \mathtt{w}_2)\}\ \mathtt{in}\ \mathtt{double}(\mathtt{v}), [\,]\rangle \\
&\Rightarrow_{\mathsf{let}} \langle \Gamma_2 \equiv \Gamma_1[\mathtt{v} \mapsto \mathtt{add}(\mathtt{w}_1, \mathtt{w}_2)],\ \mathtt{double}(\mathtt{v}), [\,]\rangle \\
&\Rightarrow_{\mathsf{fun}} \langle \Gamma_2, \mathtt{add}(\mathtt{v}, \mathtt{v}), [\,]\rangle \\
&\Rightarrow_{\mathsf{fun}} \langle \Gamma_2, \mathtt{case}\ \mathtt{v}\ \mathtt{of}\ \{\mathtt{Z} \to \mathtt{v};\ \mathtt{S}(\mathtt{u}) \to \mathtt{let}\ \{\mathtt{v} = \mathtt{add}(\mathtt{u}, \mathtt{v})\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\}, [\,]\rangle \\
&\Rightarrow_{\mathsf{case}} \langle \Gamma_2,\ \mathtt{v}, S_1 \equiv [\{\mathtt{Z} \to \mathtt{v};\ \mathtt{S}(\mathtt{u}) \to \mathtt{let}\ \{\mathtt{v} = \mathtt{add}(\mathtt{u}, \mathtt{v})\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\}]\rangle \\
&\Rightarrow_{\mathsf{var}} \langle \Gamma_2, \mathtt{add}(\mathtt{w}_1, \mathtt{w}_2), S_2 \equiv \mathtt{v} : S_1\rangle \\
&\Rightarrow_{\mathsf{fun}} \langle \Gamma_2, \mathtt{case}\ \mathtt{w}_1\ \mathtt{of}\ \{\mathtt{Z} \to \mathtt{w}_2;\ \mathtt{S}(\mathtt{u}) \to \mathtt{let}\ \{\mathtt{v} = \mathtt{add}(\mathtt{u}, \mathtt{w}_2)\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\}, S_2\rangle \\
&\Rightarrow_{\mathsf{case}} \langle \Gamma_2,\ \mathtt{w}_1,\ S_3 \equiv \{\mathtt{Z} \to \mathtt{w}_2;\ \mathtt{S}(\mathtt{u}) \to \mathtt{let}\ \{\mathtt{v} = \mathtt{add}(\mathtt{u}, \mathtt{w}_2)\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\} : S_2\rangle \\
&\Rightarrow_{\mathsf{var}} \langle \Gamma_2,\ \mathtt{Z},\ \mathtt{w}_1 : S_3\rangle \\
&\Rightarrow_{\mathsf{val}} \langle \Gamma_2,\ \mathtt{Z},\ S_3\rangle \\
&\Rightarrow_{\mathsf{select}} \langle \Gamma_2,\ \mathtt{w}_2,\ S_2\rangle \\
&\Rightarrow_{\mathsf{var}} \langle \Gamma_2,\ \mathtt{Z},\ \mathtt{w}_2 : S_2\rangle \\
&\Rightarrow_{\mathsf{val}} \langle \Gamma_2,\ \mathtt{Z},\ S_2\rangle \\
&\Rightarrow_{\mathsf{val}} \langle \Gamma_3 \equiv [\mathtt{w}_1 \mapsto \mathtt{Z}, \mathtt{w}_2 \mapsto \mathtt{Z}, \mathtt{v} \mapsto \mathtt{Z}],\ \mathtt{Z},\ S_1\rangle \\
&\Rightarrow_{\mathsf{select}} \langle \Gamma_3,\ \mathtt{v},\ [\,]\rangle \\
&\Rightarrow_{\mathsf{var}} \langle \Gamma_3,\ \mathtt{Z},\ [\mathtt{v}]\rangle \\
&\Rightarrow_{\mathsf{val}} \langle \Gamma_3,\ \mathtt{Z},\ [\,]\rangle
\end{aligned}
$$

Fig. 3. Complete derivation for $\mathtt{double}(\mathtt{add}(\mathtt{Z}, \mathtt{Z}))$

Then, we construct the initial state $\langle [], exp, [] \rangle$ and apply the rules of the standard semantics. The complete derivation is shown in Fig. 3 (where variables $\mathtt{x}_1, \mathtt{x}_2, \mathtt{y}$ are renamed as $\mathtt{w}_1, \mathtt{w}_2, \mathtt{v}$, respectively).

Observe that the expression $\mathtt{add}(\mathtt{Z}, \mathtt{Z})$ is only evaluated once: look at the 6th state in the derivation, where its evaluation is first demanded (since variable $\mathtt{v}$ is bound to this expression in the heap), and at the 16th state, where it is demanded again and the computed value is just returned from the heap.

3.2 The Partial Evaluation Semantics

While expressions to be evaluated at run time should be *closed* (i.e., without free variables), expressions to be partially evaluated are usually *incomplete* so that missing information is denoted by means of free variables. The standard semantics of Fig. 2 is not appropriate to perform computations at partial evaluation time since there is no rule for evaluating variables that are not bound in the associated heap.

In this work, we follow the approach of [3] and introduce a *residualizing* version of the standard semantics.[5] Essentially, the resulting partial evaluation semantics has the following features:

– A free variable x is represented in a heap Γ by a circular binding $x \mapsto x$ such that $\Gamma[x] = x$. Furthermore, such free variables are now considered as *values* in rule val.

[5] Note, however, that [3] does not consider a sharing-based standard semantics and, thus, the residualizing extensions are rather different.

fun_stop

$$\langle \Gamma, \underline{f}(\overline{x_n}), x : \{\overline{p_k \to e_k}\} : S\rangle \;\Rightarrow\; \langle \Gamma[x \mapsto \underline{f}(\overline{x_n})], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, S\rangle$$

case_stop

$$\langle \Gamma, \underline{case}\ y\ of\ \{\overline{p'_q \to e'_q}\}, x : \{\overline{p_k \to e_k}\} : S\rangle$$
$$\Rightarrow\; \langle \Gamma[x \mapsto \underline{case}\ y\ of\ \{\overline{p'_q \to e'_q}\}], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, S\rangle$$

guess

$$\langle \Gamma[x \mapsto x], x, \{\overline{p_k \to e_k}\} : S\rangle \;\Rightarrow\; \langle \Gamma[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, S\rangle$$

case_of_case

$$\langle \Gamma[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p'_m \to e'_m}\}, \{\overline{p_k \to e_k}\} : S\rangle$$
$$\Rightarrow\; \langle \Gamma[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p'_m \to case\ e'_m\ of\ \{\overline{p_k \to e_k}\}}\}, S\rangle$$

residualize

$$\langle \Gamma[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, [\,]\rangle \;\Rightarrow\; case\ x\ of\ \{\overline{p'_k \to \langle \Gamma[x \mapsto p'_k, \overline{y_{nk} \mapsto y_{nk}}], e'_k, [\,]\rangle}\}$$
where $p_i = c(\overline{x_{ni}})$, $\rho_i = \{\overline{x_{ni} \mapsto y_{ni}}\}$, $\overline{y_{ni}}$ are fresh,
with $p'_i = \rho_i(p_i)$, and $e'_i = \rho_i(e_i)$, for all $i = 1, \ldots, k$

Fig. 4. Partial evaluation rules

– Sharing is preserved thanks to the use of an unfolding strategy based on a (residualizing) semantics that models sharing, together with an appropriate procedure for extracting residual rules from partial computations (see Sect. 3.3). This is orthogonal to control issues and, thus, our approach can be integrated in both online or offline partial evaluation schemes (see, e.g., [10] for a gentle introduction to online and offline partial evaluation). For simplicity, though, we consider in the following an *offline* scheme for partial evaluation and assume that the program contains some function *annotations* that can be used to ensure the termination of partial computations.
  To be precise, we denote annotated function calls by underlining the function name and annotated case expressions by underlining the word *case*. Basically, annotated function calls or case expressions *should not be reduced* in order to have a finite computation.[6]
  Underlined function calls and case expressions are no longer evaluable and, thus, they are also treated as values in rule val.

Because of the introduction of the new "values" (free variables and annotated functions and cases), rule select does not suffice anymore to evaluate a case expression whose argument reduces to a value. Therefore, we introduce the rules shown in Fig. 4, which we now briefly describe.

Rule fun_stop applies when the argument of a case expression evaluates to an annotated function call $\underline{f}(\overline{x_n})$. Here, the form of the current stack is $x : \{\overline{p_k \to e_k}\} : S$, which means that the original case expression had the form $case\ x\ of\ \{\overline{p_k \to e_k}\}$ and x was eventually reduced to $\underline{f}(\overline{x_n})$. In this case, we an-

[6] We do not deal with termination issues and the computation of program annotations in this paper but refer the interested reader to, e.g., [9, 11, 12, 17, 6] (within the functional and functional logic paradigms).

notate the original case expression (since it is not reducible because $\underline{f}(\overline{x_n})$ is not reducible), update the binding for x, and return the annotated case expression. Intuitively speaking, once an annotated function call suspends the computation, we should go *backwards* and reconstruct the case expression whose branches were stored in the stack: $\underline{case}\ \underline{f}(\overline{x_n})\ of\ \{\overline{p_k \to e_k}\}$.

Rule case_stop proceeds in a similar way, the only difference being that the computed value is now an annotated case expression.

Rule guess applies when the argument of a case expression reduces to a free variable. Similarly to the previous rules, an annotated case expression is returned. Observe, however, that it does not mean that the computation is suspended; rather, the annotated case expression can still be further evaluated by rules case_of_case and residualize (see below).

Rule case_of_case, originally introduced in the context of deforestation [18], is used to reduce a case expression whose argument is another case with a free variable as argument. It moves the outer case to the branches of the inner case, e.g., it transforms an expression like

$$\texttt{case (case x of } \{\texttt{p}_1 \mapsto \texttt{e}_1; \texttt{p}_2 \mapsto \texttt{e}_2\}\texttt{) of } \{\texttt{q}_1 \mapsto \texttt{t}_1; \texttt{q}_2 \mapsto \texttt{t}_2\}$$

into $\texttt{case x of } \{\ \texttt{p}_1 \mapsto \texttt{case e}_1 \texttt{ of } \{\texttt{q}_1 \mapsto \texttt{t}_1; \texttt{q}_2 \mapsto \texttt{t}_2\};$
$\qquad\qquad \texttt{p}_2 \mapsto \texttt{case e}_2 \texttt{ of } \{\texttt{q}_1 \mapsto \texttt{t}_1; \texttt{q}_2 \mapsto \texttt{t}_2\}\ \}$.

It is often the case that the transformed expression has more opportunities for further reduction (look at the inner cases, where possibly known arguments $\texttt{e}_1$ and $\texttt{e}_2$ may allow the application of rule select). Basically, we use this rule to *lift* case expressions with a free variable to the topmost position so that rule residualize applies.

Finally, rule residualize *residualizes* a case expression (i.e., it is already considered part of the residual code) but allows us to continue evaluating the states in the branches of the residualized case expression. Observe that, because of this rule, the type of the semantics is no longer $State \to State$, where $State$ is the domain of possible states, but $State^{Exp} \to State^{Exp}$, where $State^{Exp}$ is defined as follows: $State^{Exp}\ ::=\ State \mid case\ x\ of\ \{\overline{p_k \to State^{Exp}}\}$. Note that bindings of the form $x \mapsto p'_i$, $i = 1,\ldots,k$, are applied to the different branches so that information is propagated forward in the computation. As in rule let, we rename the variables of the case patterns to avoid variable name clashes, so that p'_i and e'_i denote the renaming of p_i and e_i, respectively. Moreover, since the pattern variables of p'_i are not bound in e'_i, we add them to the heap as free variables, i.e., as circular bindings of the form $\overline{y_{ni} \mapsto y_{ni}}$.

Now, our *partial evaluation semantics* includes the rules of Fig. 2 (standard component) and Fig. 4 (residualizing component). We note that rule val overlaps with rules fun_stop and case_stop since annotated expressions are considered values. This overlapping is not intended and can easily be avoided by adding an additional side condition for rule val:

(val redefined)

$\langle \Gamma, v, x : S\rangle \Rightarrow \langle \Gamma[x \mapsto v], v, S\rangle$ if rules fun_stop & case_stop are not applicable
where v is a value

Also, we note that an additional condition should be added in rule var in order to avoid undesired loops due to the evaluation of free variables:

(var redefined)

$$\langle \Gamma[x \mapsto e], x, S\rangle \Rightarrow \langle \Gamma[x \mapsto e], e, x : S\rangle \quad \text{where } e \neq x$$

In our partial evaluation semantics, we should always construct *complete* computations, i.e., we should apply the rules of the partial evaluation semantics as much as possible. Note that it does not mean that every function is unfolded, since one can still stop the unfolding process by means of annotations (so that termination is guaranteed). Then, we have the following trivial property:

Lemma 1. *Let $s_0, s_n \in State^{Exp}$ be states such that there exists a complete derivation $s_0 \Rightarrow^* s_n$ using the rules of the partial evaluation semantics (Figures 2 and 4). Then, every state $s \in State$ occurring in s_n has an empty stack.*

This result is an easy consequence of the fact that every function and case expression is either reduced, annotated or residualized, so that an empty stack is eventually obtained.

Another trivial but important property relates the standard and the partial evaluation semantics as follows:

Lemma 2. *Let $\mathcal{P}$ be a program without annotations and $s_0 = \langle [\,], e, [\,]\rangle$ be an initial state where e is closed. Then, $s_0 \Rightarrow^* s_n$ holds in the standard semantics iff $s_0 \Rightarrow^* s_n$ holds in the partial evaluation semantics.*

Intuitively speaking, the above lemma says that, as long as no annotated function call occurs, both calculi have exactly the same behavior.

The following simple example illustrates the way our partial evaluation semantics deals with sharing in a partial computation.

Example 2. Consider again functions `double` and `add` from Example 1 and the initial expression `double(double(x))`. By normalizing this expression, we build the following initial state:

$$\langle [\,],\ \texttt{let}\ \{\texttt{x} = \texttt{x}, \texttt{w} = \underline{\texttt{double}}(\texttt{x})\}\ \texttt{in double(w)},\ [\,]\rangle$$

The associated complete computation with the partial evaluation semantics is shown in Fig. 5 (variables `x` and `w` are renamed as `n` and `m`, respectively). Note that, thanks to the use of the partial evaluation semantics, we can evaluate the considered expression as much as needed but we still keep track of shared expressions in the associated heap.

3.3 Extracting Residual Rules

Now, we consider how residual rules are extracted from the computations performed with the semantics of Figures 2 and 4.

$$\begin{array}{llll}
 & \langle [\,], & \mathtt{let}\ \{\mathtt{x}=\mathtt{x},\mathtt{w}=\underline{\mathtt{double}}(\mathtt{x})\}\ \mathtt{in}\ \mathtt{double}(\mathtt{w}), & [\,]\rangle \\
\Rightarrow_{\mathsf{let}} & \langle [\mathtt{n}\mapsto\mathtt{n}, & \mathtt{double}(\mathtt{m}), & [\,]\rangle \\
 & \ \mathtt{m}\mapsto\underline{\mathtt{double}}(\mathtt{n})], & & \\
\Rightarrow_{\mathsf{fun}} & \langle [\mathtt{n}\mapsto\mathtt{n}, & \mathtt{add}(\mathtt{m},\mathtt{m}), & [\,]\rangle \\
 & \ \mathtt{m}\mapsto\underline{\mathtt{double}}(\mathtt{n})], & & \\
\Rightarrow_{\mathsf{fun}} & \langle [\mathtt{n}\mapsto\mathtt{n}, & \mathtt{case}\ \mathtt{m}\ \mathtt{of} & [\,]\rangle \\
 & \ \mathtt{m}\mapsto\underline{\mathtt{double}}(\mathtt{n})], & \{\mathtt{Z}\to\mathtt{m};\ \mathtt{S}(\mathtt{u})\to\mathtt{let}\ \{\mathtt{v}=\mathtt{add}(\mathtt{u},\mathtt{m})\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\} & \\
\Rightarrow_{\mathsf{case}} & \langle [\mathtt{n}\mapsto\mathtt{n}, & \mathtt{m}, & [\{\ldots\}]\rangle \\
 & \ \mathtt{m}\mapsto\underline{\mathtt{double}}(\mathtt{n})], & & \\
\Rightarrow_{\mathsf{var}} & \langle [\mathtt{n}\mapsto\mathtt{n}, & \underline{\mathtt{double}}(\mathtt{n}), & [\mathtt{m},\{\ldots\}]\rangle \\
 & \ \mathtt{m}\mapsto\underline{\mathtt{double}}(\mathtt{n})], & & \\
\Rightarrow_{\mathsf{fun\_stop}} & \langle [\mathtt{n}\mapsto\mathtt{n}, & \underline{\mathtt{case}}\ \mathtt{m}\ \mathtt{of} & [\,]\rangle \\
 & \ \mathtt{m}\mapsto\underline{\mathtt{double}}(\mathtt{n})], & \{\mathtt{Z}\to\mathtt{m};\ \mathtt{S}(\mathtt{u})\to\mathtt{let}\ \{\mathtt{v}=\mathtt{add}(\mathtt{u},\mathtt{m})\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\} &
\end{array}$$

Fig. 5. Derivation with the partial evaluation semantics

Definition 1 (resultant). *Let P be an annotated program and e be an expression. Let $\langle [], e, [] \rangle \Rightarrow^* e'$ be a complete derivation with the rules of Figures 2 and 4 (i.e., e' is irreducible). The associated resultant is given by the following rule:*

$$f(\overline{x_n}) \;=\; [\![del(e')]\!]$$

where f is a fresh function symbol,[7] *$\overline{x_n}$ are the free variables of e (appropriately renamed as in the considered computation), function del removes the annotations (if any), and function $[\![\ ]\!]$ is defined as follows:*

$$[\![e]\!] \;=\; \begin{cases} case\ x\ of\ \{\overline{p_k \to [\![e_k]\!]}\} & \text{if } e = case\ x\ of\ \{\overline{p_k \to e_k}\} \\ let\ \overline{\Gamma}\ in\ e' & \text{if } e = \langle \Gamma, e', [\,] \rangle \end{cases}$$

Here, $\overline{\Gamma}$ represents the set of bindings stored in Γ except those for $\overline{x_n}$ (which are now the parameters of the new function).

Let us illustrate the extraction of a residual rule with an example.

Example 3. Consider the computation of Example 2 shown in Fig. 5. The associated resultant is as follows:

$$\begin{array}{lll}
\mathtt{f}(\mathtt{n}) \;=\; [\![\langle\ [\mathtt{n}\mapsto\mathtt{n}, & \mathtt{case}\ \mathtt{m}\ \mathtt{of} & \\
\qquad\qquad\ \ \mathtt{m}\mapsto\mathtt{double}(\mathtt{n})], & \{\mathtt{Z}\to\mathtt{m};\ \mathtt{S}(\mathtt{u})\to\mathtt{let}\ \{\mathtt{v}=\mathtt{add}(\mathtt{u},\mathtt{m})\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\}, [\,]\rangle]\!]
\end{array}$$

which is reduced to

$$\begin{array}{ll}
\mathtt{f}(\mathtt{n}) & = \mathtt{let}\ \{\mathtt{m}\mapsto\mathtt{double}(\mathtt{n})\}\ \mathtt{in} \\
 & \quad \mathtt{case}\ \mathtt{m}\ \mathtt{of}\ \{\mathtt{Z}\to\mathtt{m};\ \mathtt{S}(\mathtt{u})\to\mathtt{let}\ \{\mathtt{v}=\mathtt{add}(\mathtt{u},\mathtt{m})\}\ \mathtt{in}\ \mathtt{S}(\mathtt{v})\}
\end{array}$$

[7] Consequently, some calls in the right-hand side should also be renamed. We do not deal with renaming of function calls in this paper; nevertheless, standard techniques for functional (logic) languages like those in [4] would be applicable.

Observe that sharing is preserved despite the unfolding of a function which is not right-linear (i.e., the outer call to function `double`). Note also that inlining the let expression (i.e., replacing all occurrences of `m` by `double(n)`) would destroy this property since `double` would be evaluated twice, once as an argument of the case expression and once when selecting the corresponding case branch.

3.4 Correctness

The correctness of our approach to the partial evaluation of first-order lazy functional programs relies on two results. On the one hand, one should prove that the partial evaluation semantics is somehow equivalent to the standard one. Regarding the extraction of resultants from computations with the partial evaluation semantics, its correctness can easily be proved by exploiting the clear operational equivalence between a state of the form $\langle \Gamma, e, [\,]\rangle$ and an expression like $let\ \overline{\Gamma}\ in\ e$ (i.e., we have that $\langle [\,], let\ \overline{\Gamma}\ in\ e, [\,]\rangle$ reduces to $\langle \Gamma, e, [\,]\rangle$ in one reduction step by applying rule let).

Let us first consider the equivalence between the standard and the partial evaluation semantics for closed expressions. In the following, we say that two states $\langle \Gamma, e, S\rangle$ and $\langle \Gamma', e', S'\rangle$ are *equivalent under annotations*, in symbols $\langle \Gamma, e, S\rangle \approx \langle \Gamma', e', S'\rangle$, iff Γ and Γ' become equal when removing bindings with annotated expressions, $e = e'$, and $S = S'$. By abuse, we say that a derivation is complete when no more rules are applicable, even if this is due to an annotated function call (which is irreducible in the standard semantics since it does not deal with annotations).

Theorem 1. *Let $\mathcal{P}$ be an annotated program and s be an initial state. If $s \Rightarrow^* s'$ is a complete derivation in $\mathcal{P}$ with the standard semantics then, for any derivation $s \Rightarrow^* s' \Rightarrow^* \langle \Gamma, e, [\,]\rangle$ in $\mathcal{P}$ with the partial evaluation semantics, we have $\langle \Gamma, e, [\,]\rangle \Rightarrow^* s''$ with the standard semantics and $s' \approx s''$.*

Intuitively, the above result can be depicted graphically as follows (SS and PES stand for Standard Semantics and Partial Evaluation Semantics, respectively):

$$\begin{tikzcd}
 & \langle [\,], e_0, [\,]\rangle \arrow[dl, "\mathsf{SS}"', "*" near end] \arrow[dr, "\mathsf{PES}", "*"' near end] & \\
\langle \Gamma_s, e_s, S_s\rangle & & \langle \Gamma, e, [\,]\rangle \arrow[ll, "\mathsf{SS}", "*"' near end]
\end{tikzcd}$$

Proof. Let $s \Rightarrow^* s'$ in $\mathcal{P}$ with the standard semantics, where $s' = \langle \Gamma_s, e_s, S_s\rangle$. Now, we distinguish two possibilities. If e_s is a value (and, thus, $S_s = [\,]$) then the proof is trivial by Lemma 2, with $\langle \Gamma, e, [\,]\rangle = \langle \Gamma_s, e_s, S_s\rangle$.

Otherwise, $e_s \equiv \underline{f}(\overline{x_n})$ for some function symbol f. By Lemma 2, we have $s \Rightarrow^* s'$ with the partial evaluation semantics. Trivially, since e was closed, only rules $\mathsf{fun\_stop}$ and $\mathsf{case\_stop}$ from the partial evaluation semantics can be applied to s'. Let $s' \Rightarrow^* \langle \Gamma, e, [\,]\rangle$ be a derivation with the partial evaluation semantics where rules $\mathsf{fun\_stop}$ and $\mathsf{case\_stop}$ are applied as much as possible. Then, we can also construct a sort of inverse computation using rules case and var from the standard semantics; namely, every application of rule fun_stop or case_stop can be undone by applying rules case and var in this order. Hence, we have $\langle \Gamma, e, [\,]\rangle \Rightarrow^* \langle \Gamma'', e'', S''\rangle \equiv s''$ by applying rules case and var as much as possible. Finally, it is clear that $s' \approx s''$ since Γ'' adds only bindings with annotated expressions to Γ_s, $e'' = e_s$, and $S'' = S_s$.

Now, we focus on expressions which are not closed. Since this is orthogonal to program annotations, we now consider programs without annotations.

In the following, we introduce the following reduction rules over the expressions produced by the partial evaluation semantics:

$$\begin{array}{rl} \mathit{case}\ c(\overline{v_n})\ \mathit{of}\ \{\overline{p_k \to e_k}\} \hookrightarrow \sigma_i(e_i) & \text{if } p_i = c(\overline{y_n}) \text{ and } \sigma_i = \{\overline{y_n \mapsto v_n}\} \\ s \hookrightarrow s' & \text{if } s \Rightarrow s' \text{ with the standard semantics} \end{array}$$

These rules are used to evaluate expressions from $State^{Exp}$ (as returned by rule residualize). Our next result is then stated as follows:

Theorem 2. *Let $\mathcal{P}$ be a program and e be a (not necessarily closed) expression. Let σ be a substitution mapping the free variables of e to values. If there exists a complete derivation $\langle [\,], \sigma(e), [\,]\rangle \Rightarrow^* \langle \Gamma, v, [\,]\rangle$ with the standard semantics, then for all derivations $\langle [\,], e, [\,]\rangle \Rightarrow^* s$ with the partial evaluation semantics we have $\sigma(s) \hookrightarrow^* s'$ and $s' \approx \langle \Gamma, v, [\,]\rangle$.*

Intuitively speaking, this result ensures that computations with the partial evaluation semantics and some incomplete expression including free variables appropriately capture every possible computation with the standard semantics and a closed instance of the incomplete expression.

Proof. For simplicity, we consider that e contains a single free variable x and that $\sigma = \{x \mapsto c\}$ maps x to a constructor constant c. Assume that the derivation with the standard semantics has the form

$$\begin{array}{rll} \langle [\,], \sigma(e), [\,]\rangle \Rightarrow^* & \langle \Gamma_x[x \mapsto c], x, [\{\overline{p_k \to e_k}\}]\rangle & \\ \Rightarrow_{\mathsf{var}} & \langle \Gamma_x[x \mapsto c], c, x : \{\overline{p_k \to e_k}\}\rangle & \\ \Rightarrow_{\mathsf{val}} & \langle \Gamma_x[x \mapsto c], c, [\{\overline{p_k \to e_k}\}]\rangle & \\ \Rightarrow_{\mathsf{select}} & \langle \Gamma_x[x \mapsto c], e_i, [\,]\rangle & (\text{with } p_i = c,\ i \in \{1, \ldots, k\}) \\ \Rightarrow^* & \langle \Gamma, v, [\,]\rangle & \end{array}$$

Observe that we considered a stack with the branches of a single case expression. A generalization to consider nested case expressions is not difficult and only require some additional applications of rule case_of_case.

Trivially, we have $\langle [], e, []\rangle \Rightarrow^* \langle \Gamma_x, x, [\{\overline{p_k \to e_k}\}]\rangle$ with the standard semantics. Therefore, $\langle [\,], e, [\,]\rangle \Rightarrow^* \langle \Gamma_x, x, [\{\overline{p_k \to e_k}\}]\rangle$ is also a derivation with the partial evaluation semantics by Lemma 2.

We now consider two possibilities for the partial evaluation semantics. If the derivation is stopped before applying rule guess, the claim follows trivially by the definition of $\hookrightarrow$. Otherwise, we have a derivation of the form

$$
\begin{array}{lll}
\langle [\,], e, [\,] \rangle & \Rightarrow^* & \langle \mathit{\Gamma}_x[x \mapsto x], x, [\{\overline{p_k \to e_k}\}] \rangle \\
& \Rightarrow_{\mathsf{guess}} & \langle \mathit{\Gamma}_x[x \mapsto x], \underline{case}\ x\ of\ \{\overline{p_k \to e_k}\}, [\,] \rangle \\
& \Rightarrow_{\mathsf{residualize}} & case\ x\ of\ \{\overline{p_k \to \langle \mathit{\Gamma}_x[x \mapsto \rho_k(p_k), \overline{y_{nk} \mapsto y_{nk}}], \rho_k(e_k), [\,] \rangle}\} \\
& \Rightarrow^* & \ldots
\end{array}
$$

Now, the claim follows since

$$
case\ \sigma(x)\ of\ \{\overline{p_k \to \langle \mathit{\Gamma}_x[x \mapsto \rho_k(p_k), \overline{y_{nk} \mapsto y_{nk}}], \rho_k(e_k), [\,] \rangle}\} \hookrightarrow \langle \mathit{\Gamma}_x[x \mapsto c], e_i, [] \rangle
$$

and the fact that there are no more free variables (and, thus, computations in the standard and the partial evaluation semantics coincide from this point on).

4 Partial Evaluation in Practice

We have already developed an offline partial evaluator for functional and functional logic programs following the basic technique of [17] (later improved with a stronger termination analysis in [6]). The implementation is publicly available from `http://www.dsic.upv.es/~gvidal/german/offpeval/`.

Now, we have added the new unfolding strategy presented so far (i.e., the rules of Figures 2 and 4), together with the procedure for the extraction of resultants of Sect. 3.3. In order to check the usefulness of the new approach, we have considered three different unfolding strategies:

(aggressive) This strategy does not take into account the linearity of functions, i.e., a function call is annotated (classified as "not unfoldable") only if there is a risk of nontermination (according to the already implemented termination analysis [6]). Furthermore, unfolding is performed with a semantics that does not model sharing.

(conservative) This strategy annotates a function call if either there is a risk of nontermination or the associated function definition is not right-linear. Again, unfolding is performed with a semantics that does not model sharing.

(sharing-based) This is the new strategy described in this paper, where function calls are annotated only if there is a risk of nontermination but a sharing-based unfolding is used.

The first two strategies could easily be adopted by the old partial evaluator, but the third one required the implementation of the sharing-based partial evaluation semantics.

We have tested the implemented system on a number of examples, and the sharing-based strategy generally produces residual programs which are as good as the best of the other two strategies.

Let us illustrate this point with some examples. Consider the program (in Haskell-like notation) shown in Fig. 6. The annotations are given by the termination analysis of our partial evaluator when considering the expression

`dapp (incList (S`^{100}` Z) x)`

```
append [] x = x                                     dapp x = append x x
append (x : xs) ys = x : append xs ys

incList n [] = []                                   add Z m = m
incList n (x : xs) = (add n x) : (incList n xs)     add (S n) m = S (add n m)
```

Fig. 6. Double-append program

to be partially evaluated, where ($\mathtt{S}^{100}$ Z) is a shorthand for the natural number S (S (... Z)) with 100 nested applications of S.

Now, the three strategies mentioned above proceed as follows:

(aggressive) Here, we get the following residual program:

```
new [] = []
new (y : ys) = (S^100 y) : append (incList100 ys) (incList100 (y : ys))
incList100 [] = []
incList100 (x : xs) = (S^100 x) : (incList100 xs)
```

together with the original definition of append. The following function renamings were considered:

```
dapp (incList (S^100 Z) x)  ↦  new x
incList (S^100 Z) x         ↦  incList100 x
```

Observe that function new has repeated calls to function incList100, which will cause a slower execution of the residual program.

(conservative) This strategy basically returns the original program unchanged because the call to dapp is also annotated in order to avoid the unfolding of a function which is not right-linear. In this case, no slowdown is produced in the residual program, but its run time is essentially the same as that of the original program.

(sharing-based) By using our new partial evaluation semantics, we get the following residual program:

```
new [] = []
new (y : ys) = let w = incList100 (y : ys) in append w w
incList100 [] = []
incList100 (x : xs) = (S^100 x) : (incList100 xs)
```

together with the original definition of append. Here, we use the same renamings of the aggressive strategy.

In this case, the performance of the residual program is comparable to the outcome of the conservative approach, i.e., we avoid producing a slower residual program but no significant improvement is achieved.

Now, consider the following expression to be partially evaluated:

```
dapp (decList x [Z, Z, Z])
```

where function `decList` is defined as follows:

```
decList n [ ] = [ ]
decList n (x : xs) = (minus n x) : (decList n xs)

minus n Z = n
minus (S n) (S m) = minus n m
```

The difference with the previous example is that the inner call to `decList` can be fully unfolded. Now, the three strategies mentioned above proceed as follows:

(aggressive) It returns a residual program of the form

```
new x = [x, x, x, x, x, x]
```

where (`dapp (decList x [Z, Z, Z])`) is renamed as (`new x`).

(conservative) This strategy basically returns the original program unchanged because the call to `dapp` is not unfolded.

(sharing-based) We get the same residual program as in the aggressive case. No let expression is necessary in the residual rule since the argument of `dapp` is fully evaluated and thus repeated values are not problematic (note that the residual function `new` can only be called with a *value*, see Theorem 2).

To summarize, our preliminary experiments are encouraging and show that the new sharing-based approach could be able to get the best of previous approaches.

5 Discussion

Despite the extensive literature on partial evaluation, we are not aware of any approach to the specialization of lazy functional languages where sharing is preserved through the specialization process in a non-trivial way. For instance, [2, 3] presents a partial evaluation scheme for a lazy language but sharing is not preserved since the underlying semantics does not model variable sharing.

In this paper, we have presented a novel approach by first extending a standard lazy semantics (where sharing is modeled by using an updatable heap) and, then, defining a method to properly extract the associated residual rules. Our approach is not overly restrictive since every function can be unfolded (even if it is not right-linear) and still preserves sharing, thus avoiding the introduction of redundant computations in the residual program.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2007 for many useful comments and suggestions.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
3. 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.
4. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.
5. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
6. 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.
7. H.P. Barendregt. *The Lambda Calculus—Its Syntax and Semantics*. Elsevier, 1984.
8. A. Bondorf. *Similix 5.0 Manual*, 1993.
9. A. Bondorf and J. Jørgensen. Efficient Analyses for Realistic Off-Line Partial Evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.
10. C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.
11. A.J. Glenstrup and N.D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM TOPLAS*, 27(6):1147–1215, 2005.
12. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
13. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
14. M. Leuschel, D. Elphick, M. Varea, S. Craig, and M. Fontaine. The Ecce and Logen Partial Evaluators and Their Web Interfaces. In *Proc. of PEPM'06*, pages 88–94. IBM Press, 2006.
15. S. Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
16. S.L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002.
17. J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'05)*, pages 228–239. ACM Press, 2005.
18. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.