

Preserving Sharing in the Partial Evaluation of Lazy Functional Programs^{*}

Sebastian Fischer¹, Josep Silva², Salvador Tamarit², and Germán Vidal²

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 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 functions.

1 Introduction

Partial evaluation [7] is an automatic technique for the specialization of programs. This technique has already been developed for a variety of programming languages, like C [4], Curry [12], Prolog [9], Scheme [13], etc.

In this work, we focus on a problem associated to the partial evaluation of *lazy* functional languages. In these languages (e.g., Haskell [10]), 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:³

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

where function `sumList` sums the elements of a list, `incList` increments the elements of a list by a given number, and `add` performs the addition of two natural numbers. Now, consider a partial evaluation of the following function call: `sumList(incList(\boxed{e} , Z : Z : []))`, where \boxed{e} is any arbitrary expression:

$$\text{sumList}(\text{incList}(\boxed{e}, Z : Z : [])) \Rightarrow \text{sumList}(\text{add}(\boxed{e}, Z) : \text{incList}(\boxed{e}, Z : []))$$

^{*} 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.

³ We use `[]` and `“:”` as constructors of lists and `Z` and `S` to build natural numbers.

Note that, although the expression \boxed{e} appears twice, it will only be evaluated once in current lazy programming languages since the two occurrences of variable n in the second rule of function `incList` are shared. If we build a residual rule—a *resultant*—associated to the above partial evaluation, we would get the rule

$$\text{new_function}(\dots) = \text{sumList}(\text{add}(\boxed{e}, Z) : \text{incList}(\boxed{e}, (Z : [])))$$

Now, however, if we evaluate `new_function` using the above rule, the expression \boxed{e} will be evaluated twice since both occurrences are not shared anymore, which is unacceptable from a performance point of view.

Current partial evaluation schemes for lazy functional (logic) languages have mostly ignored this point.⁴ Usually, partial evaluators include a restriction so that the unfolding of functions whose right-hand side is not *linear* (i.e., whose right-hand side contains multiple occurrences of the same variable) is forbidden.

In this work, we present an alternative to such trivial, overly restrictive treatment of sharing during partial evaluation. In particular, we would like to produce a residual rule of the following form:

$$\text{new_function}(\dots) = \text{let } w = \boxed{e} \\ \text{in } \text{sumList}(\text{add}(w, Z) : \text{incList}(w, Z : []))$$

In this way, the two occurrences of the fresh variable w would be shared and \boxed{e} would not be evaluated twice. In principle, one could define a post-unfolding phase where, given a partial evaluation $e_0 \Rightarrow e_1 \Rightarrow \dots \Rightarrow e_n$, every occurrence of a common subexpression e in e_n would be replaced by a fresh variable w and a new `let` of the form `let w = e in ...` would be added. However, if the semantics used for partial evaluation does not model sharing, the identification of common subexpressions would be rather difficult because their degree of evaluation need not be the same.

In contrast, here we present a novel method which is based on a lazy semantics [1] that models variable sharing by means of an updatable heap, which is appropriately extended in order to perform symbolic computations. Then, we also define how residual rules should be extracted from these symbolic 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, \dots, 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

⁴ We note that this is a critical issue that has been considered in the context of *inlining* (see, e.g., [11]), which could be seen like a rather simple form of partial evaluation.

$P ::= D_1 \dots D_m$	(program)	<i>Domains</i>
$D ::= f(x_1, \dots, x_n) = e$	(function definition)	
$e ::= x$	(variable)	$P_1, P_2, \dots \in Prog$ (Programs)
$c(x_1, \dots, x_n)$	(constructor call)	$x, y, z, \dots \in Var$ (Variables)
$f(x_1, \dots, x_n)$	(function call)	$a, b, c, \dots \in \mathcal{C}$ (Constructors)
$let \{\overline{x}_k \equiv \overline{e}_k\} in e$	(let binding)	$f, g, h, \dots \in \mathcal{F}$ (Functions)
$case x of \{\overline{p}_k \rightarrow \overline{e}_k\}$	(case expression)	$p_1, p_2, p_3, \dots \in Pat$ (Patterns)
$p ::= c(x_1, \dots, x_n)$	(pattern)	

Fig. 1. Syntax for normalized flat programs

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 $case x of \{c_1(\overline{x}_{n_1}) \rightarrow e_1; \dots; c_k(\overline{x}_{n_k}) \rightarrow e_k\}$, where x is a variable, c_1, \dots, c_k are different constructors, and e_1, \dots, e_k are expressions. The *pattern variables* \overline{x}_{n_i} are introduced locally and bind the corresponding variables of e_i .

Observe that, according to Fig. 1, the arguments of function and constructor calls are variables. As in [8], 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., [8, 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* [6] (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 which preserves 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); and iii) 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 [8]). It follows these naming conventions:

$$\Gamma, \Delta, \Theta \in Heap = Var \rightarrow Exp \quad v \in Value ::= x \mid c(\overline{x}_n)$$

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

Fig. 2. Small-Step Semantics for (Sharing-Based) Lazy Functional Programs

A *heap* is a partial mapping from variables to expressions (the *empty heap* is denoted by $[\]$). The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap with $\Gamma[x] = e$, i.e., we use this notation either as a condition on a heap Γ or as a modification of Γ . 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 $\langle \Gamma, e, S \rangle$, where Γ 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 since the next time the value of variable x is demanded, the value v will be returned thus avoiding the repeated evaluation of e .
- Rule **fun** implements a simple function unfolding. We assume that the considered program P is a global parameter of the calculus and that the variables of the rule are fresh in every application of rule **fun**.
- In order to reduce a **let** construct, rule **let** adds the bindings to the heap and proceed 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.
- Rule **case** initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $\{\overline{p_k} \rightarrow e_k\}$ 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 .

3.2 The Partial Evaluation Semantics

To perform partial computations in a functional context, missing data is usually denoted by free variables. Therefore, the semantics of Fig. 2 is not appropriate to perform computations at partial evaluation time. Here, we follow the approach of [3] and introduce a *residualizing* version of the standard semantics as follows.

First, logical variables are used to represent missing information. In a heap Γ , a logical variable x is represented by a circular binding $x \mapsto x$ such that $\Gamma[x] = x$. Now, they are also considered *values* in rule `val`.

In the new semantics, we assume that the rules of the semantics are applied until no more rule is applicable. The termination of partial computations is ensured by using function *annotations*.⁵ Basically, annotated functions—we use an underscore to annotate function calls and case expressions—should not be unfolded in order to have a finite computation.⁶ Underlined function calls and case expressions are also treated as values in rule `val`.

Because of the introduction of the new “values” (logical 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 following new rules, which are shown in Fig. 3:

- First, rule `fun_stop` applies when the argument of a case expression evaluates to an annotated function call $\underline{f}(\overline{x_n})$. The current stack, $x : \{\overline{p_k} \rightarrow \overline{e_k}\} : S$, means that the original case expression had the form *case* x *of* $\{\overline{p_k} \rightarrow \overline{e_k}\}$, so that x was eventually reduced to $\underline{f}(\overline{x_n})$. In this case, we annotate the original case expression, update the binding for x , and return the annotated case expression. Intuitively, once an annotated function call suspends the computation, we should reconstruct the context in the heap and terminate the computation.
- 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 logical variable (i.e., to some missing data). Here, rather than suspending the computation, we return the annotated case expression, which can then be reduced by rules `case_of_case` and `residualize`, depending on the current stack.

⁵ Note that we consider an *offline* scheme for partial evaluation for simplicity; indeed, our main contributions (the partial evaluation semantics and the extraction of residual rules) could also be used within an online scheme.

⁶ We do not deal with termination issues in this paper but refer the interested reader to, e.g., [12, 5].

fun_stop

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

case_stop

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

guess

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

case_of_case

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

residualize

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

Fig. 3. Partial Evaluation Rules

- Rule **case_of_case** (originally introduced in the context of deforestation [14]) is used to reduce a case whose argument is another case with a logical variable as argument. This rule moves the outer case to the branches of the inner case. Intuitively, it is used to lift case expressions with a logical variable, i.e., non-deterministic choices, to the topmost position so that rule **residualize** applies. Essentially, rule **residualize** *residualizes* the case expression (i.e., it is already considered part of the residual code) and continue with the evaluation of the different branches. Note that bindings of the form $x \mapsto p'_i$, $i = 1, \dots, k$, 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 logical variables, i.e., as circular bindings of the form $\overline{x_{ni}} \mapsto \overline{x_{ni}}$.

Note that the new rules are basically required in order to deal with missing information and annotated function calls. The preservation of sharing through the specialization process is achieved thanks to the use of a standard semantics that models sharing.

Observe that, if we apply the rules of the partial evaluation semantics as much as possible, every state $\langle \Gamma, e, S \rangle$ in the derived expression (if the computation does not fail) would have an empty stack. This 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 finally obtained.

	$\langle [], \text{let } \{x = x, w = \underline{\text{double}}(x)\} \text{ in } \text{double}(w), [] \rangle$	$[]$
\Rightarrow_{let}	$\langle [x \mapsto x, \text{double}(w), w \mapsto \underline{\text{double}}(x)], \dots \rangle$	$[]$
\Rightarrow_{fun}	$\langle [x \mapsto x, \text{add}(w, w), w \mapsto \underline{\text{double}}(x)], \dots \rangle$	$[]$
\Rightarrow_{fun}	$\langle [x \mapsto x, \text{case } w \text{ of } w \mapsto \underline{\text{double}}(x), \{Z \rightarrow w; S(u) \rightarrow \text{let } \{v = \text{add}(u, w)\} \text{ in } S(v)\} \dots \rangle$	$[]$
$\Rightarrow_{\text{case}}$	$\langle [x \mapsto x, w, w \mapsto \underline{\text{double}}(x)], \dots \rangle$	$\{\{\dots\}\}$
\Rightarrow_{var}	$\langle [x \mapsto x, \underline{\text{double}}(x), w \mapsto \underline{\text{double}}(x)], \dots \rangle$	$[w, \{\dots\}]$
$\Rightarrow_{\text{fun_stop}}$	$\langle [x \mapsto x, \text{case } w \text{ of } w \mapsto \underline{\text{double}}(x), \{Z \rightarrow w; S(u) \rightarrow \text{let } \{v = \text{add}(u, w)\} \text{ in } S(v)\} \dots \rangle$	$[]$

Fig. 4. Derivation with the partial evaluation semantics

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

Example 1. Consider the following simple program:

```

double(x) = add(x, x)
add(n, m) = case n of {Z → m; S(u) → let {v = add(u, m)} in S(v)}

```

Given the initial state $\langle [], \text{let } \{x = x, w = \underline{\text{double}}(x)\} \text{ in } \text{double}(w), [] \rangle$, we have the computation shown in Fig. 4. 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 Fig. 2 and 3.

Definition 1 (resultant). *Let P be an annotated program and e be an expression. Let $\langle [], e, [] \rangle \Rightarrow^* e'$ be a computation with the rules of Fig. 2 and 3 such that e' is irreducible. The associated resultant is given by the following rule:*

$$f(\overline{x_n}) = \llbracket \text{del}(e') \rrbracket$$

where f is a fresh function symbol,⁷ $\overline{x_n}$ are the logical variables of e , function del removes the annotations (if any), and the function $\llbracket \cdot \rrbracket$ is defined as follows:

$$\llbracket e \rrbracket = \begin{cases} \text{case } x \text{ of } \{\overline{p_k} \rightarrow \llbracket e_k \rrbracket\} & \text{if } e = \text{case } x \text{ of } \{\overline{p_k} \rightarrow e_k\} \\ \text{let } \overline{F} \text{ in } e' & \text{if } e = \langle \Gamma, e', [] \rangle \end{cases}$$

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 would be applicable.

Here, $\bar{\Gamma}$ represents the set of bindings stored in Γ except those for \bar{x}_n (which are now the parameters of the new function) as well as those which depend on \bar{x}_n .

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

Example 2. Consider the computation of Example 1 shown in Fig. 4. The associated resultant is as follows:

$$f(x) = \llbracket \langle [x \mapsto x, \quad \text{case } w \text{ of} \\ w \mapsto \text{double}(x)], \{Z \rightarrow w; S(u) \rightarrow \text{let } \{v = \text{add}(u, w)\} \text{ in } S(v)\}, [] \rangle \rrbracket$$

which is reduced to

$$f(x) = \text{let } \{w \mapsto \text{double}(x)\} \text{ in} \\ \text{case } w \text{ of } \{Z \rightarrow w; S(u) \rightarrow \text{let } \{v = \text{add}(u, w)\} \text{ in } S(v)\}$$

Observe that sharing is preserved despite the unfolding of a function which is not right-linear (i.e., `double`). Note also that inlining the let expression (i.e., replacing all occurrences of `w` by `double(x)`) would destroy this property since `double` would be evaluated twice, once as an argument of the case expression and another one 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. Basically, this means that, given a computation with the partial evaluation semantics, $e \Rightarrow^* e'$, it represents every possible computation with the standard semantics (i.e., the only difference is that non-deterministic branching is encoded by means of residualized case expressions). A similar proof (though for the simpler LNT semantics without sharing) can be found in [3].

Regarding the extraction of resultants from computations with the partial evaluation semantics, its correctness can easily be proved by exploiting previous results and the clear operational equivalence between a configuration of the form $\langle \Gamma, e, [] \rangle$ and an expression like *let \bar{T} in e* .

4 Discussion

Despite the extensive literature on partial evaluation, we are not aware of any approach to the specialization of lazy functional (logic) 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 promising approach by first extending a standard semantics (where sharing is modeled by using an updatable heap) and, then, defining a method to properly extract the associated residual rules. Our

new 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.

An implementation of the new partial evaluation scheme has been undertaken by extending a previous offline partial evaluator [12].

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. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
5. G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'06)*, pages 60–76. Springer LNCS 4407, 2007.
6. H.P. Barendregt. *The Lambda Calculus—Its Syntax and Semantics*. Elsevier, 1984.
7. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
8. 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.
9. 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.
10. S. Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
11. S.L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002.
12. 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.
13. P. Thiemann. The Program Generator Generator PGG. Available from the URL: <http://www.informatik.uni-freiburg.de/proglang/software/pgg/>.
14. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.