Towards a Safe Partial Evaluation of Lazy Functional Logic Programs

Sebastian Fischer[1]

Department of Computer Science, University of Kiel
Olshausenstr. 40, D-24098, Kiel, Germany

Josep Silva[2] Salvador Tamarit[2] Germán Vidal[2]

DSIC, Technical University of Valencia
Camino de Vera S/N, E-46022 Valencia, Spain

Abstract

Partial Evaluation is a well-known technique for specializing programs w.r.t. a given restriction of their input data. Although partial evaluation has been widely investigated in the context of functional and functional logic languages like Haskell or Curry, current schemes are either overly restrictive or destroy sharing through the specialization process, which may produce incorrect specializations when non-deterministic functions are considered. In this work, we present a new partial evaluation scheme for lazy functional logic programs that preserves sharing through the specialization process and still allows the unfolding of arbitrary functions. Furthermore, our approach ensures that sharing is also preserved across non-deterministic computations.

Keywords: Partial evaluation, laziness

1 Introduction

Partial evaluation is a well-known technique for the specialization of programs w.r.t. part of their input data. Consider a program P whose input data are split into *static* (i.e., known), say $\mathbf{s}$, and *dynamic* (i.e., unknown) data, say $\mathbf{d}$. Then, the partial evaluation of P w.r.t. the static data $\mathbf{s}$ returns a *residual* program $P_{\mathbf{s}}$ specialized for $\mathbf{s}$ which is equivalent to the original program w.r.t. the dynamic data, i.e., $P(\mathbf{s}, \mathbf{d}) = P_{\mathbf{s}}(\mathbf{d})$ for all $\mathbf{d}$.

Partial evaluators fall in two main categories, *online* and *offline*, according to the time when termination issues are addressed. Online partial evaluators include a single, monolithic stage which performs partial computations, termination tests, generation of residual rules, etc. Offline partial evaluators, on the other hand, include

[1] Email: sebf@informatik.uni-kiel.de
[2] Email: {jsilva,stamarit,gvidal}@dsic.upv.es

two separate stages. The first stage—called Binding-Time Analysis (BTA)—mainly propagates the static data through the entire program and often includes a static termination analysis. The outcome of the BTA is a copy of the original program with a number of *annotations* to guide the specialization process (i.e., whether a function call should be unfolded or not, whether a function argument should be generalized or not, etc). The second stage is usually an extension of an interpreter for the considered programming language: it takes an annotated program and evaluates it as much as possible according to the annotations.

An offline partial evaluation scheme for first-order functional and functional logic programs has recently been proposed in [12]. Unfortunately, this scheme does not include a special treatment for sharing, which means that either the unfolding strategy should be rather restrictive or it might destroy sharing through the specialization process, which may produce incorrect specializations when non-deterministic functions are considered. Let us illustrate this problem with a simple example. Consider the following program (natural numbers are built from `Z` and `S`):

```
main       = double e
double x   = add x x
add Z y    = y
add (S x) y = S (add x y)
...
```

where `e` is an expression whose evaluation is expensive. Here, if we allow the unfolding of function `double`, we could get the following specialized program:

```
main       = add e e
add Z y    = y
add (S x) y = S (add x y)
...
```

where the evaluation of `e` is now duplicated, which is unacceptable. Observe that this duplicated evaluation does not occur in the original program since the two occurrences of variable `x` in the right-hand side of function `double` are *shared* in current lazy functional languages like Haskell or Curry.

Furthermore, if the evaluation of `e` is non-deterministic, then the specialized program would not be sound. Consider, for instance, that `e` $\equiv$ `coin` with

```
coin = Z
coin = S Z
```

Then, `main` can be non-deterministically reduced to `Z` and `S (S Z)` in the original program, while it could also be reduced to `S Z` in the specialized one.

Clearly, one can easily avoid this situation by forbidding the unfolding of functions whose right-hand side is not *linear* (i.e., whose right-hand side contains multiple occurrences of the same variable). However, this turns out to be rather restrictive

$$
\begin{array}{rcll}
P & ::= & D_1 \ldots D_m & \text{(program)} \\
D & ::= & f(x_1, \ldots, x_n) = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & | & c(x_1, \ldots, x_n) & \text{(constructor call)} \\
 & | & f(x_1, \ldots, x_n) & \text{(function call)} \\
 & | & \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e & \text{(let binding)} \\
 & | & x_1\ \mathit{or}\ x_2 & \text{(disjunction)} \\
 & | & \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\} & \text{(case expression)} \\
p & ::= & c(x_1, \ldots, x_n) & \text{(pattern)}
\end{array}
$$

Domains

$$
\begin{array}{rcll}
P_1, P_2, \ldots & \in & \mathit{Prog} & \text{(Programs)} \\
x, y, z, \ldots & \in & \mathit{Var} & \text{(Variables)} \\
a, b, c, \ldots & \in & \mathcal{C} & \text{(Constructors)} \\
f, g, h, \ldots & \in & \mathcal{F} & \text{(Functions)} \\
p_1, p_2, p_3, \ldots & \in & \mathit{Pat} & \text{(Patterns)}
\end{array}
$$

Fig. 1. Syntax for normalized flat programs

in practice. To the best of our knowledge, there is no approach to partial evaluation that includes a non-trivial treatment of sharing during the specialization process.

In this work, we present a first step towards the development of a partial evaluation scheme for lazy functional logic programs that is not overly restrictive and still ensures that sharing is not destroyed. For this purpose, we extend a natural semantics for lazy functional logic languages so that it becomes appropriate to perform *partial* computations. Then, we describe how residual programs can be extracted from the partial computations performed with the extended semantics.

2 Preliminaries

In this work we consider *flat* programs [9], a convenient standard representation of 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 [8] or Toy [11].

Furthermore, we assume that flat programs are *normalized*, i.e., *let* constructs are used to ensure that the arguments of function and constructor calls are always variables (not necessarily pairwise different). As in [10], this is essential to express sharing without the use of graph structures. A normalization algorithm can be found in [1]. Basically, normalization introduces one new let construct for each non-variable argument, e.g., $f(e)$ is transformed into "$\mathit{let}\ \{x = e\}\ \mathit{in}\ f(x)$".

The syntax for normalized flat programs is shown in Figure 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), disjunctions (e.g., to represent non-deterministic operations), and case expressions of the form $\mathit{case}\ x\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$, 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 the subexpression e_i.

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* [5] (i.e., a variable or an expression with a construc-

tor at the outermost position). Consequently, the operational semantics that we consider will describe the evaluation of expressions only to head normal form. This is not a restriction since the evaluation to normal form or the solving of equations can be reduced to head normal form computations (see, e.g., [9]).

Following a recent proposal in [4,7], we assume in this work that logical variables are modeled by means of non-deterministic *value generators*. For instance, in order to write `add (S Z) x`, where `x` is a logical variable, we write `add (S Z) genNat` instead, where the auxiliary function `genNat` is defined as follows:

```
genNat = Z or (S genNat)
```

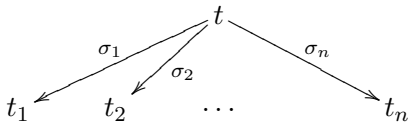
This allows one to consider a simpler semantics by ignoring logical variables. In particular, in this work we consider the semantics of [6] which extends the operational semantics of [1] in two ways: 1) logical variables are represented by value generators and 2) it preserves sharing even across non-deterministic computations.

3 Partial Evaluation of Lazy Functional Logic Programs

In this section, we introduce the core of a new scheme for the partial evaluation of lazy functional logic programs. Basically, at the heart of an offline partial evaluator, we usually find

- a non-standard semantics, which is used to perform symbolic computations (according to the program annotations) and
- a method to extract residual rules from partial computations.

For instance, within the original *narrowing-driven* partial evaluation framework [3], one proceeds as follows: given a term-rewriting system $\mathcal{R}$ and a term t, a *finite (possibly incomplete) narrowing tree* for t in $\mathcal{R}$ is built:



Then, for every (possibly incomplete) narrowing derivation $t \leadsto_{\sigma_i} t_i$, we produce an associated residual rule of the following form: $\sigma_i(t) = t_i$.

This simple approach, however, is not applicable to the (normalized) flat language. Here, the standard semantics (the LNT calculus [9]) is not appropriate and, thus, a *residualizing* version, the RLNT calculus, has been introduced in [2]. The RLNT semantics does not compute bindings but encodes them by means of case expressions (which are considered "residual" code). For instance, given a case expression of the form

$$case\ x\ of\ \{\overline{p_k \to e_k}\}$$

where x is a logical variable, the RLNT semantics performs the following step:

$$case\ x\ of\ \{\overline{p_k \to \sigma_k(e_k)}\}$$

$$\textsf{(Lookup)}\quad \frac{\Gamma\backslash\{(x,\Gamma[x])\} : \Gamma[x] \;\Downarrow\; \Delta : v}{\Gamma : x \;\Downarrow\; \Delta[x \mapsto v] : v}$$

$$\textsf{(Val)}\quad \Gamma : v \;\Downarrow\; \Gamma : v$$

$$\textsf{(Fun)}\quad \frac{\Gamma : \sigma(e) \;\Downarrow\; \Delta : v}{\Gamma : f(\overline{x_n}) \;\Downarrow\; \Delta : v} \quad \text{where } f(\overline{y_n}) = e \in P \text{ and } \sigma = \{\overline{y_n \mapsto x_n}\}$$

$$\textsf{(Let)}\quad \frac{\Gamma[\overline{y_k \longmapsto \sigma(e_k)}] : \sigma(e) \;\Downarrow\; \Delta : v}{\Gamma : \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e \;\Downarrow\; \Delta : v} \quad \text{where } \sigma = \{\overline{x_k \mapsto y_k}\} \text{ and } \overline{y_k} \text{ are fresh variables}$$

$$\textsf{(Select)}\quad \frac{\Gamma : e \;\Downarrow\; \Delta : c(\overline{y_n}) \qquad \Delta : \sigma(e_i) \;\Downarrow\; \Theta : v}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v} \quad \text{where } p_i = c(\overline{x_n}) \text{ and } \sigma = \{\overline{x_n \mapsto y_n}\}$$

$$\textsf{(OR)}\quad \Gamma : x_1\ \mathit{or}\ x_2 \;\Downarrow\; \Gamma : \mathit{OR}\ r\ [x_1, x_2] \quad \text{where } r \text{ is fresh}$$

$$\textsf{(Lift)}\quad \frac{\Gamma : e \;\Downarrow\; \Delta : \mathit{OR}\ r\ [\overline{x_n}]}{\Gamma : \mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Delta[y_n \mapsto \mathit{case}\ x_n\ \mathit{of}\ \{\overline{p_k \to e_k}\}] : \mathit{OR}\ r\ [\overline{y_n}]} \quad \text{where } \overline{y_n} \text{ are fresh}$$

Fig. 2. Core semantics

with $\sigma_i = \{x \mapsto p_i\}$, $i = 1, \ldots, k$, and continues with the evaluation of the different branches $\sigma_1(e_1), \ldots, \sigma_k(e_k)$.

In our approach, however, a more complex extension is required since—in contrast to the LNT/RLNT calculi—the considered semantics [6] copes with sharing (even across non-deterministic computations) and deals with *configurations* of the form $\Gamma : e$, where Γ is a *heap*—an updatable set of bindings, which is essential to model sharing—and e is an expression.

Therefore, in the following section, we introduce an extension of this semantics which is appropriate for performing partial computations. Then, in Section 3.2, we describe the extraction of residual rules from partial computations.

3.1 Partial Evaluation Semantics

First, we introduce the core of the considered semantics [6], whose rules are shown in Fig. 2. Here, a *heap*, denoted by Γ, Δ, or Θ, is a partial mapping from variables to expressions (the *empty heap* is denoted by $[\,]$). We let $\mathcal{D}om(\Gamma)$ denote the set $\{x \mid x \mapsto e \in \Gamma\}$. 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* v is a constructor-rooted term $c(\overline{e_n})$ (i.e., a term whose outermost function symbol is a constructor symbol).

We use judgments like "$\Gamma : e \;\Downarrow\; \Delta : v$" which are interpreted as "in the context of heap Γ, the expression e evaluates to value v producing a new heap Δ." Let us briefly explain the rules of the semantics:

- In order to evaluate a variable x in the context of a heap Γ, rule Lookup first reduces $\Gamma[x]$ to a value v in the context of the heap $\Gamma\backslash\{(x, \Gamma[x])\}$; note that the binding for x is removed from Γ in order to detect *black holes* (a self-dependent infinite loop). Then, Γ is updated with the computed value for x. Therefore, if the value of x is required again in the computation, the value v is immediately returned so that the effects of sharing are achieved.

$$\text{(Hnf-Val)}\quad \frac{\Gamma : e \;\Downarrow\; \Delta : c(\overline{x_n})}{\Gamma : hnf\ \rho\ e \;\Downarrow\; \Delta : c(\overline{x_n})} \qquad \text{where } c \neq OR$$

$$\text{(Hnf-Choose)}\quad \frac{\Gamma : e \;\Downarrow\; \Delta : OR\ r\ [\overline{x_n}] \qquad \Delta : hnf\ \rho\ x_{\rho(r)} \;\Downarrow\; \Theta : v}{\Gamma : hnf\ \rho\ e \;\Downarrow\; \Theta : v} \qquad \text{if } r \in \mathrm{Dom}(\rho)$$

$$\text{(Hnf-Or)}\quad \frac{\Gamma : hnf\ \rho\ x \;\Downarrow\; \Delta_1 : OR\ r\ [\overline{x_n}] \qquad \overline{\Delta_n : hnf\ (\rho \cup \{(r,i)\})\ x_n \;\Downarrow\; \Delta_{n+1} : v_n}}{\Gamma : hnf\ \rho\ x \;\Downarrow\; \Delta_{n+1} : OR\ r\ [\overline{x_n}]} \qquad \text{if } r \notin \mathrm{Dom}(\rho)$$

Fig. 3. HNF rules

- Rule Val is used to evaluate a value and its definition is self-explanatory.
- Rule Fun is used to perform a function unfolding; here, we assume that the considered program P is a global parameter of the calculus.
- Rule Let evaluates a let expression by adding its bindings to the heap and then proceeding with the evaluation of its expression. Note that we introduce fresh names in order to avoid variable name clashes.
- Rule Select is used to evaluate a case expression whose argument reduces to a constructor-rooted term. In this case, we select the matching branch and, then, proceed with the evaluation of the expression in this branch.
- Rules OR and Lift are the main novelty of this calculus w.r.t. the original semantics of [1]. Here, non-deterministic choices of the form *or* $x_1\ x_2$ are not evaluated but replaced by $OR\ r\ [x_1, x_2]$, where OR is a fresh constructor symbol and r is a fresh *reference* (e.g., a natural number) to identify this particular occurrence of OR (see below).

  The occurrences of OR in an expression (if any) are intended to denote the *non-deterministic branches* of the considered evaluation,[3] e.g., an expression of the form $OR\ r_1\ [OR\ r_2\ [x_1\ x_2],\ x_3]$ denotes a search tree with three branches x_1, x_2 and x_3. References in OR expressions are useful to record previous choices. For instance, given an expression like $OR\ r_1\ [OR\ r_2\ [x_1\ x_2],\ x_3]$, if $r_1 = r_2$, then it should actually be $OR\ r_1\ [x_1,\ x_3]$, i.e., we replace $OR\ r_2\ [x_1\ x_2]$ by x_1 because we are inside the first branch of another OR expression with the same reference.

  Since OR constructors are used to represent the structure of the non-deterministic search tree, we want them to be the topmost symbols in an evaluated expression. Here, rule Lift is used to push computations inside the branches of an OR expression (i.e., to lift the OR constructor one level up).

  Thanks to these rules, sharing is not only preserved along a deterministic computation, but also across different non-deterministic branches.

In principle, given a normalized expression e, we could evaluate e using the initial configuration $[\,] : e$ and then applying the rules of the semantics. However, in its current formulation, only the topmost constructor of an expression is computed. While this is fine when the computation is deterministic, it is not acceptable when it is non-deterministic since every evaluated expression would be headed by the constructor OR and, thus, its branches would not be evaluated at all.

In order to overcome this limitation, we now introduce a set of rules that are

[3] Therefore, they are only used internally but should not be part of the computed values.

used to evaluate an expression as much as needed to have a head normal form in every branch.[4] The new rules are shown in Figure 3.

Now, the computation for e in P will start with an initial configuration of the form "$[] : \ hnf\ \{\}\ e$", where $\{\}$ is a mapping which is used to keep track of the non-deterministic branches found so far in the computation (thus it is initially empty). The new rules are the following:

- Rule Hnf-Val is used to compute the head normal form of a deterministic expression. Here, the result is a value and, thus, ρ is simply ignored.
- Rule Hnf-Choose applies when the considered expression evaluates to an OR expression *with the same reference of a previously encountered OR*. Therefore, we delete the OR constructor, choose the same branch as before, and continue with its evaluation.
- Finally, rule Hnf-Or is used when the expression evaluates to an OR expression *with a reference not found so far*. This means that the expression being evaluated is actually non-deterministic and, thus, the computed result should include different values. These values are obtained by applying function *hnf* to each branch of the OR expression with $\rho \cup \{(r, i)\}$ to record the fact that the i-th branch has been chosen.

Compared to [6], we lose the possibility to employ different search strategies and implicitly evaluate non-deterministic branches of the computation in depth-first order. We can do this because we assume that the use of a binding-time analysis ensures the termination of computations by placing appropriate annotations in the source program (cf. Section 1).

The appendix shows an example which illustrates the rules of the new semantics.

3.2 Extraction of Residual Rules

In this section, we describe the extraction of a residual rule associated to a computation using the rules of the semantics shown in Figures 2 and 3.

In principle, one could think that the following naive approach would be appropriate: given a derivation of the form

$$[] : \ let\ \{\ x = main\ \}\ in\ hnf\ \{\}\ x \Downarrow \Gamma : v$$

the associated residual rule would be

$$f(\overline{x_n}) \ = \ let\ \overline{\Gamma}\ in\ v$$

where f is a fresh function symbol, $\overline{x_n}$ are the "logical variables" in the evaluation of *main* (i.e., those variables that were bound to value generators), and $\overline{\Gamma}$ denotes the set of bindings of Γ in equational form. Unfortunately, these rules would not be appropriate for a number of reasons:

- the computed value v may contain *illegal* symbols (e.g., occurrences of OR) and

[4] This is done in [6] by means of a primitive function `searchTree` which builds the non-deterministic search tree for a given expression in a lazy manner.

- v may also contain calls to value generators that should rather be parameters of the function (i.e., variables from $\overline{x_n}$).

In order to overcome these drawbacks, now we introduce a number of transformations that eventually produce a right-hand side with the appropriate form.

First, we need the following preliminary definition.

Definition 3.1 (reachable variables) *Given a heap Γ and two variables $x, y \in \mathcal{D}om(\Gamma)$, we say that y is reachable from x if $y \in reach(\Gamma, x)$, where function reach is defined as follows:*

$$reach(\Gamma, x) = \{x\} \cup \bigcup_{y \in Var(\Gamma[x])} reach(\Gamma, y)$$

Intuitively, a variable x is reachable from another variable y if y is bound to an expression that contains x, or to an expression that contains a variable w that is bound to an expression that contains x, and so on.

Now, we can already define the extraction of residual rules. Given a derivation of the form

$$[\,] : \; let \; \{ \; x = main \; \} \; in \; hnf \; \{\} \; x \Downarrow \Gamma : v$$

we first construct a residual rule as explained before:

$$f(\overline{x_n}) \;=\; let \; \overline{\Gamma} \; in \; v$$

We consider, for instance, the computation shown in the appendix. In this case, the initial residual rule is as follows: [5]

```
f x6 = let { x6 = OR 1 x10 x11,  x8 = S x9,  x9 = Z,  x10 = Z,  x11 = S x18
           x18 = genNat,  x19 = add x18 x6,  x20 = add x19 x8 }
       in OR 1 (S x9) (S x20)
```

Then, we apply the following transformations to the right-hand side of the rule:

(i) First, we move the outermost bindings to every *leaf* of the computed expression. Moreover, the bindings should be modified according to the branch where they appear, e.g., a binding `x` $\mapsto$ `OR 1 y z` should be replaced by `x` $\mapsto$ `y` within the first branch of an `OR` labeled with reference 1 and by `x` $\mapsto$ `z` within the second branch of the same `OR`.

In our running example, the residual rule is transformed as follows:

```
f x6 = OR 1 (let { x9 = Z } in S x9)
            (let { x6 = x11,  x8 = S x9,  x9 = Z,  x11 = S x18,
                   x18 = genNat,  x19 = add x18 x6,  x20 = add x19 x8 }
             in  S x20)
```

[5] We only show in this section the relevant bindings. The complete example can be found in the appendix.

(ii) The next step consists in transforming those `OR` expressions that come from the evaluation of a parameter in the left-hand side by a case expression.

For instance, in our example, we have a single parameter `x6` in the left-hand side of the residual rule. In the final heap of the computation, we had the following binding:

$$\texttt{x6} \mapsto \texttt{OR 1 x10 x11}$$

Therefore, every expression of the form `OR 1 e1 e2` in the right-hand side of the residual rule should be replaced by

$$\texttt{case x6 of } \{\texttt{p}_1 \mapsto \texttt{e1};\ \texttt{p}_2 \mapsto \texttt{e}_2\}$$

where $\texttt{p}_1$ and $\texttt{p}_2$ are obtained by dereferencing `x10` and `x11`.

In this way, we obtain the following rule:

```
f x6 = case x6 of
       {    Z ↦ let { x9 = Z } in S x9;
        S x18 ↦ let { x8 = S x9,  x9 = Z,  x19 = add x18 x6,
                      x20 = add x19 x8 } in S x20 }
```

(iii) In the next step, we remove those bindings that are not reachable (using function *reach*) from the variables in the different branches.

We do not show the application of this step here since non-reachable variables have been removed from the beginning to improve readability. The interested reader is referred to the appendix.

(iv) Now, we should move outwards those bindings that are common to several branches, so that sharing is enforced.

In our example, we can only move outwards the binding `x9 = Z` which appears in both branches of the residual rule:

```
f x6 = let { x9 = Z } in
       case x6 of { Z ↦ S x9;
                    S x18 ↦ let { x8 = S x9,  x19 = add x18 x6,
                                  x20 = add x19 x8 } in S x20 }
```

(v) Finally, we should replace every remaining occurrence of $OR\ n\ x_1\ x_2$ by a standard *or* expression of the form $or\ x_1\ x_2$ (thus losing the connections given by references).

In our example, no remaining `OR` appears and, thus, no further transformation is required.

4 Conclusions and Future Work

This work has introduced a new partial evaluation scheme which is particularly well suited for specializing *lazy* functional logic programs. In particular, it 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.

As future work, we first plan to formalize the extraction of residual rules and prove its correctness. Then, we will extend the semantics of Figures 2 and 3 in order to cope with the additional features of the language, e.g., higher-order functions, constraints, etc. Finally, we undertake the development of a partial evaluator for a significant subset of a lazy language like Haskell or Curry following the ideas presented so far.

References

[1] Albert, E., M. Hanus, F. Huch, J. Oliver and G. Vidal, *Operational Semantics for Declarative Multi-Paradigm Languages*, Journal of Symbolic Computation **40** (2005), pp. 795–829.

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

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

[4] Antoy, S. and M. Hanus, *Overlapping Rules and Logic Variables in Functional Logic Programs*, in: *Proc. of the 22snd Int'l Conf. on Logic Programming (ICLP'06)* (2006), pp. 87–101.

[5] Barendregt, H., "The Lambda Calculus—Its Syntax and Semantics," Elsevier, 1984.

[6] Braßel, B. and F. Huch, *On a Tighter Integration of Functional and Logic Programming* (2007), personal communication.

[7] de Dios-Castro, J. and F. López-Fraguas, *Elimination of Extra-Variables in Functional Logic Programs*, in: *Proc. of the 6th Spanish Conf. on Programming and Languages (PROLE'06)* (2007).

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

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

[10] Launchbury, J., *A Natural Semantics for Lazy Evaluation*, in: *Proc. of the ACM Symp. on Principles of Programming Languages (POPL'93)* (1993), pp. 144–154.

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

[12] Ramos, J., J. Silva and G. Vidal, *Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems*, in: *Proc. of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)* (2005), pp. 228–239.

A An Example

We illustrate the rules of the semantics in Figures 2 and 3 with an example. Consider the following flat program:

```
data Nat = Z | S Nat

add x y = case x of { Z -> y;
                      S n -> let { m = add n y } in S m }

double x = add x x

genNat = let { n = Z, m = let { x = genNat } in S x } in or n m

main = let { n = genNat, x = double n, y = S m, m = Z } in add x y
```

and the initial configuration:

$$[\,] : \texttt{let } \{\, \texttt{x} = \texttt{main} \,\} \texttt{ in hnf } \{\} \texttt{ x}$$

The evaluation produced by the semantics is shown in Figures A.1 and A.2, together with the final heap of the computation.

The initial residual rule associated to the computation shown in Figures A.1 and A.2 can be found in Figure A.3.

Then, we apply the following transformations to the right-hand side of the rule:

(i) First, we move the outermost bindings to every *leaf* of the computed expression. Moreover, the bindings should be modified according to the branch where they appear, e.g., a binding $\texttt{x} \mapsto \texttt{OR 1 y z}$ should be replaced by $\texttt{x} \mapsto \texttt{y}$ within the first branch of an `OR` labeled with reference 1 and by $\texttt{x} \mapsto \texttt{z}$ within the second branch of the same `OR`.

In our running example, the residual rule is transformed as follows:

```
f x6 = OR 1 (let {  x5 = x14,  x6 = x10,  x7 = x12,
                    x8 = S x9,  x9 = Z,  x10 = Z,
                    x11 = S x18,  x12 = x10,  x13 = S x19,
                    x14 = x16,  x15 = S x20,  x16 = S x9,
                    x17 = case x11 of {Z ↦ x8;
                                       S x3 ↦ let {x4 = add x3 x8} in S x4},
                    x18 = genNat,  x19 = add x18 x6,
                    x20 = add x19 x8 } in S x9)
            (let { x5 = x15,  x6 = x11,  x7 = x13,
                    x8 = S x9,  x9 = Z,  x10 = Z,
                    x11 = S x18,  x12 = x11,  x13 = S x19,
                    x14 = x17,  x15 = S x20,  x16 = Sx9,
                    x17 = case x11 of {Z ↦ x8;
                                       S x3 ↦ let {x4 = add x3 x8} in S x4},
                    x18 = genNat,  x19 = add x18 x6,
                    x20 = add x19 x8 } in S x20)
```

(ii) The next step consists in transforming those `OR` expressions that come from the evaluation of a parameter in the left-hand side by a case expression.

For instance, in our example, we have a single parameter `x6` in the left-hand side of the residual rule. In the final heap of the computation, we have the following binding:

$$\texttt{x6} \mapsto \texttt{OR 1 x10 x11}$$

$[\,] : \texttt{let } \{\texttt{x0} = \texttt{main}\} \texttt{ in hnf x0}$

$\Rightarrow_{\mathsf{Let}}$ $\Delta_2 \equiv \Delta_1[\texttt{x5} \mapsto \texttt{main}] : \texttt{hnf x5}$

$\Rightarrow_{\mathsf{Hnf-Or}}$ $\Delta_2 : \texttt{x5}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_3 \equiv 2\backslash[\texttt{x5} \mapsto \texttt{main}] : \texttt{main}$

$\Rightarrow_{\mathsf{Fun}}$ $\Delta_3 : \texttt{let x1} = \texttt{genNat}, \texttt{x2} = \texttt{double x1}, \texttt{x3} = \texttt{S x4}, \texttt{x4} = \texttt{Z in add x2 x3}$

$\Rightarrow_{\mathsf{Let}}$ $\Delta_4 \equiv \Delta_3[\texttt{x6} \mapsto \texttt{genNat}, \texttt{x7} \mapsto \texttt{double x6}, \texttt{x8} \mapsto \texttt{S x9}, \texttt{x9} \mapsto \texttt{Z}] : \texttt{add x7 x8}$

$\Rightarrow_{\mathsf{Fun}}$ $\Delta_4 : \texttt{case x7 of } \{\texttt{Z} \mapsto \texttt{x8};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x8}\} \texttt{ in S x4}\}$

$\Rightarrow_{\mathsf{Lift}}$ $\Delta_4 : \texttt{x7}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_5 \equiv \Delta_4\backslash[\texttt{x7} \mapsto \texttt{double x6}] : \texttt{double x6}$

$\Rightarrow_{\mathsf{Fun}}$ $\Delta_5 : \texttt{add x6 x6}$

$\Rightarrow_{\mathsf{Fun}}$ $\Delta_5 : \texttt{case x6 of } \{\texttt{Z} \mapsto \texttt{x6};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x6}\} \texttt{ in S x4}\}$

$\Rightarrow_{\mathsf{Lift}}$ $\Delta_5 : \texttt{x6}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_6 \equiv \Delta_5\backslash[\texttt{x6} \mapsto \texttt{genNat}] : \texttt{genNat}$

$\Rightarrow_{\mathsf{Fun}}$ $\Delta_6 : \texttt{let } \{\texttt{x2} = \texttt{Z}, \texttt{x3} = \texttt{let } \{\texttt{x1} = \texttt{genNat}\} \texttt{ in S x1}\} \texttt{ in or x2 x3}$

$\Rightarrow_{\mathsf{Let}}$ $\Delta_7 \equiv \Delta_6[\texttt{x10} \mapsto \texttt{Z}, \texttt{x11} \mapsto \texttt{let } \{\texttt{x1} = \texttt{genNat}\} \texttt{ in S x1}] : \texttt{or x10 x11}$

$\Rightarrow_{\mathsf{Or}}$ $\Delta_7 : \texttt{OR 1 x10 x11}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_8 \equiv \Delta_7[\texttt{x6} \mapsto \texttt{OR 1 x10 x11}] : \texttt{OR 1 x10 x11}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_9 \equiv \Delta_8[\texttt{x12} \mapsto \texttt{case x10 of } \{\texttt{Z} \mapsto \texttt{x6};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x6}\} \texttt{ in S x4}\}] : \texttt{OR 1 x12 x13}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{10} \equiv \Delta_9[\texttt{x7} \mapsto \texttt{OR 1 x12 x13}] : \texttt{OR 1 x12 x13}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{11} \equiv \Delta_{10}[\texttt{x14} \mapsto \texttt{case x12 of } \{\texttt{Z} \mapsto \texttt{x8};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x8}\} \texttt{ in S x4}\},$
$\texttt{x15} \mapsto \texttt{case x13 of } \{\texttt{Z} \mapsto \texttt{x8};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x8}\} \texttt{ in S x4}\}] : \texttt{OR 1 x14 x15}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{12} \equiv \Delta_{11}[\texttt{x5} \mapsto \texttt{OR 1 x14 x15}] : \texttt{OR 1 x14 x15}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{12} : \texttt{hnf } \{(1,1)\} \texttt{ x14}$

$\Rightarrow_{\mathsf{Hnf-Choose}}$ $\Delta_{12} : \texttt{x14}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{13} \equiv \Delta_{12}\backslash[\texttt{x14} \mapsto \texttt{case x12 of } \{\texttt{Z} \mapsto \texttt{x8};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x8}\} \texttt{ in S x4}\}] :$
$\texttt{case x12 of } \{\texttt{Z} \mapsto \texttt{x8};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x8}\} \texttt{ in S x4}\}$

$\Rightarrow_{\mathsf{Lift}}$ $\Delta_{13} : \texttt{x12}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{14} \equiv \Delta_{13}\backslash[\texttt{x12} \mapsto \texttt{case x10 of } \{\texttt{Z} \mapsto \texttt{x6};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x6}\} \texttt{ in S x4}\}] :$
$\texttt{case x10 of } \{\texttt{Z} \mapsto \texttt{x6};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x6}\} \texttt{ in S x4}\}$

$\Rightarrow_{\mathsf{Select}}$ $\Delta_{14} : \texttt{x10}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{15} \equiv \Delta_{14}\backslash[\texttt{x10} \mapsto \texttt{Z}] : \texttt{Z}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{16} \equiv \Delta_{15}\backslash[\texttt{x10} \mapsto \texttt{Z}] : \texttt{Z}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{16} : \texttt{x6}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{17} \equiv \Delta_{16}\backslash[\texttt{x6} \mapsto \texttt{OR 1 x10 x11}] : \texttt{OR 1 x10 x11}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{18} \equiv \Delta_{17}[\texttt{x6} \mapsto \texttt{OR 1 x10 x11}] : \texttt{OR 1 x10 x11}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{19} \equiv \Delta_{18}[\texttt{x12} \mapsto \texttt{OR 1 x10 x11}] : \texttt{OR 1 x10 x11}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{20} \equiv \Delta_{19}[\texttt{x16} \mapsto \texttt{case x10 of } \{\texttt{Z} \mapsto \texttt{x6};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x6}\} \texttt{ in S x4}\},$
$\texttt{x17} \mapsto \texttt{case x11 of } \{\texttt{Z} \mapsto \texttt{x8};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x8}\} \texttt{ in S x4}\}] : \texttt{OR 1 x16 x17}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{21} \equiv \Delta_{20}[\texttt{x14} \mapsto \texttt{OR 1 x16 x17}] : \texttt{OR 1 x16 x17}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{21} : \texttt{hnf } \{(1,1)\} \texttt{ x16}$

$\Rightarrow_{\mathsf{Hnf-Val}}$ $\Delta_{21} : \texttt{x16}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{22} \equiv \Delta_{21}\backslash[\texttt{x16} \mapsto \texttt{case x10 of } \{\texttt{Z} \mapsto \texttt{x6};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x6}\} \texttt{ in S x4}\}] :$
$\texttt{case x10 of } \{\texttt{Z} \mapsto \texttt{x6};\ \texttt{S x3} \mapsto \texttt{let } \{\texttt{x4} = \texttt{add x3 x6}\} \texttt{ in S x4}\}$

$\Rightarrow_{\mathsf{Select}}$ $\Delta_{22} : \texttt{x10}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{23} \equiv \Delta_{22}\backslash[\texttt{x10} \mapsto \texttt{Z}] : \texttt{Z}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{24} \equiv \Delta_{23}[\texttt{x10} \mapsto \texttt{Z}] : \texttt{Z}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{24} : \texttt{x8}$

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{25} \equiv \Delta_{24}\backslash[\texttt{x8} \mapsto \texttt{S x9}] : \texttt{S x9}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{26} \equiv \Delta_{25}[\texttt{x8} \mapsto \texttt{S x9}] : \texttt{S x9}$

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{27} \equiv \Delta_{26}[\texttt{x16} \mapsto \texttt{S x9}] : \texttt{S x9}$

Fig. A.1. Evaluation using the semantics of Figures 2 and 3

$\Rightarrow_{\mathsf{Val}}$ Δ_{27} : S x9

$\Rightarrow_{\mathsf{Val}}$ Δ_{27} : S x9

$\Rightarrow_{\mathsf{Val}}$ Δ_{27} : hnf {(1, 2)} x15

$\Rightarrow_{\mathsf{Hnf-Val}}$ Δ_{27} : x15

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{28} \equiv \Delta_{27}\backslash[\mathtt{x15} \mapsto \mathtt{case\ x13\ of\ \{Z \mapsto x8;\ S\ x3 \mapsto let\ \{x4 = add\ x3\ x8\}\ in\ S\ x4\}}] :$
case x13 of {Z ↦ x8; S x3 ↦ let {x4 = add x3 x8} in S x4}

$\Rightarrow_{\mathsf{Select}}$ Δ_{28} : x13

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{29} \equiv \Delta_{28}\backslash[\mathtt{x13} \mapsto \mathtt{case\ x11\ of\ \{Z \mapsto x8;\ S\ x3 \mapsto let\ \{x4 = add\ x3\ x8\}\ in\ S\ x4\}}] :$
case x11 of {Z ↦ x8; S x3 ↦ let {x4 = add x3 x8} in S x4}

$\Rightarrow_{\mathsf{Select}}$ Δ_{29} : x11

$\Rightarrow_{\mathsf{Lookup}}$ $\Delta_{30} \equiv \Delta_{29}\backslash[\mathtt{x11} \mapsto \mathtt{let\ x1 = genNat\ in\ S\ x1}] :$ let x1 = genNat in S x1

$\Rightarrow_{\mathsf{Let}}$ $\Delta_{31} \equiv \Delta_{30}[\mathtt{x18} \mapsto \mathtt{genNat}] :$ S x18

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{32} \equiv \Delta_{31}[\mathtt{x11} \mapsto \mathtt{S\ x18}] :$ S x18

$\Rightarrow_{\mathsf{Val}}$ Δ_{32} : let {x4 = add x18 x6} in S x4

$\Rightarrow_{\mathsf{Let}}$ $\Delta_{33} \equiv \Delta_{32}[\mathtt{x19} \mapsto \mathtt{add\ x18\ x6}] :$ S x19

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{34} \equiv \Delta_{33}[\mathtt{x13} \mapsto \mathtt{S\ x19}] :$ S x19

$\Rightarrow_{\mathsf{Val}}$ Δ_{34} : let {x4 = add x19 x8} in S x4

$\Rightarrow_{\mathsf{Let}}$ $\Delta_{35} \equiv \Delta_{34}[\mathtt{x20} \mapsto \mathtt{add\ x19\ x8}] :$ S x20

$\Rightarrow_{\mathsf{Val}}$ $\Delta_{36} \equiv \Delta_{35}[\mathtt{x15} \mapsto \mathtt{S\ x20}] :$ S x20

$\Rightarrow_{\mathsf{Val}}$ Δ_{36} : S x20

$\Rightarrow_{\mathsf{Val}}$ Δ_{36} : OR 1 (S x9) (S x20)

with

Δ_{36} = [x5 ↦ OR 1 x14 x15,
x6 ↦ OR 1 x10 x11,
x7 ↦ OR 1 x12 x13,
x8 ↦ S x9,
x9 ↦ Z,
x10 ↦ Z,
x11 ↦ S x18,
x12 ↦ OR 1 x10 x11,
x13 ↦ S x19,
x14 ↦ OR 1 x16 x17,
x15 ↦ S x20,
x16 ↦ S x9,
x17 ↦ case x11 of {Z ↦ x8; S x3 ↦ let {x4 = add x3 x8} in S x4},
x18 ↦ genNat,
x19 ↦ add x18 x6,
x20 ↦ add x19 x8]

Fig. A.2. Evaluation using the semantics of Figures 2 and 3 (continued)

Therefore, every expression of the form `OR 1 e1 e2` in the right-hand side of the residual rule should be replaced by

$$\mathtt{case\ x6\ of\ \{p_1 \mapsto e1;\ p_2 \mapsto e_2\}}$$

where $\mathtt{p}_1$ and $\mathtt{p}_2$ are obtained by dereferencing `x10` and `x11`.

In this way, we obtain the following rule:

```
f x6 = let {  x5 = OR 1 x14 x15,
              x6 = OR 1 x10 x11,  x7 = OR 1 x12 x13,
              x8 = S x9,  x9 = Z,  x10 = Z,
              x11 = S x18,  x12 = OR 1 x10 x11,
              x13 = S x19,  x14 = OR 1 x16 x17,
              x15 = S x20,  x16 = S x9,
              x17 = case x11 of {Z ↦ x8; S x3 ↦ let {x4 = add x3 x8} in S x4},
              x18 = genNat,  x19 = add x18 x6,
              x20 = add x19 x8 }
           in OR 1 (S x9) (S x20)
```

Fig. A.3. Initial residual rule for the computation of Figures A.1 and A.2

```
f x6 = case x6 of
       { Z ↦ let {   x5 = x14,  x6 = x10,  x7 = x12,
                     x8 = S x9,  x9 = Z,  x10 = Z,
                     x11 = S x18,  x12 = x10,  x13 = S x19,
                     x14 = x16,  x15 = S x20,  x16 = S x9,
                     x17 = case x11 of {Z ↦ x8;
                                        S x3 ↦ let {x4 = add x3 x8} in S x4},
                     x18 = genNat,  x19 = add x18 x6,
                     x20 = add x19 x8 } in S x9;
         S x18 ↦ let { x5 = x15,  x6 = x11,  x7 = x13,
                     x8 = S x9,  x9 = Z,  x10 = Z,
                     x11 = S x18,  x12 = x11,  x13 = S x19,
                     x14 = x17,  x15 = S x20,  x16 = S x9,
                     x17 = case x11 of {Z ↦ x8;
                                        S x3 ↦ let {x4 = add x3 x8} in S x4},
                     x18 = genNat,  x19 = add x18 x6,
                     x20 = add x19 x8 } in S x20 }
```

(iii) In the next step, we remove those bindings that are not reachable from the variables in the different branches.

For instance, in our example, the first branch is `S x9`. Here, we have

```
reach([ x5 ↦ x14,  x6 ↦ x10,  x7 ↦ x12,  x8 ↦ S x9,  x9 ↦ Z,  x10 = Z,
        x11 = S x18,  x12 = x10,  x13 = S x19,  x14 = x16,  x15 = S x20,
        x16 = S x9, x17 = case x11 of {...},  x18 = genNat,
        x19 = add x18 x6,  x20 = add x19 x8], x9) = {x9}
```

and, therefore, only the equation `x9 = Z` is needed.

Analogously, the set of reachable variables from the second branch, `S x20`, is the following:

```
reach([ x5 ↦ x15,  x6 ↦ x11,  x7 ↦ x13,  x8 ↦ S x9,  x9 ↦ Z,  x10 = Z,
        x11 = S x18,  x12 = x11,  x13 = S x19,  x14 = x17,  x15 = S x20,
        x16 = S x9, x17 = case x11 of {...},  x18 = genNat,
        x19 = add x18 x6,  x20 = add x19 x8], x20) = {x8, x9, x19, x20}
```

Observe that we exclude from the above list the variable *parameters* (i.e., `x6` and `x18`) since their bindings should not appear anymore. Therefore, the transformed residual rule is now as follows:

```
f x6 = case x6 of
       {Z ↦ let {     x9 = Z } in S x9;
        S x18 ↦ let { x8 = S x9,
                      x9 = Z,
                      x19 = add x18 x6,
                      x20 = add x19 x8 } in S x20 }
```

(iv) Now, we should move outwards those bindings that are common to several branches, so that sharing is enforced.

In our example, we can only move outwards the binding `x9 = Z` which appears in both branches of the residual rule:

```
f x6 = let { x9 = Z } in  case x6 of
                          {Z ↦ S x9;
                           S x18 ↦ let { x8 = S x9,
                                         x19 = add x18 x6,
                                         x20 = add x19 x8 } in S x20 }
```

(v) Finally, we should replace every remaining occurrence of $OR\ n\ x_1\ x_2$ by a standard *or* expression of the form $or\ x_1\ x_2$ (thus losing the connections given by references).

In our example, no remaining `OR` appears and, thus, no further transformation is required.