Lazy Call-By-Value Evaluation *

Bernd Braßel Sebastian Fischer
Michael Hanus Frank Huch
Institute of Computer Science, CAU Kiel, Germany.
{bbr,sebf,mh,fhu}@informatik.uni-kiel.de

Germán Vidal
DSIC, Technical University of Valencia, Spain
gvidal@dsic.upv.es

Abstract

Designing debugging tools for lazy functional programming languages is a complex task which is often solved by expensive tracing of lazy computations. We present a new approach in which the information collected as a trace is reduced considerably (kilobytes instead of megabytes). The idea is to collect a kind of step information for a call-by-value interpreter, which can then efficiently reconstruct the computation for debugging/viewing tools, like declarative debugging. We show the correctness of the approach, discuss a proof-of-concept implementation with a declarative debugger as back end and present some benchmarks comparing our new approach with the Haskell debugger Hat.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Theory

Keywords Laziness, debugging techniques.

1. Introduction

The demand-driven nature of lazy evaluation is one of the most appealing features of modern functional languages like Haskell (Peyton Jones 2003). Unfortunately, it is also one of the most complex features one should face in order to design a debugging tool for these languages. In particular, printing the step-by-step trace of a lazy computation is generally useless from a programmer's point of view, mainly because arguments of function calls are often shown unevaluated and because the order of evaluation is counterintuitive.

There are several approaches that improve this situation by hiding the details of lazy evaluation to the programmer. The main such approaches are: Freja (Nilsson and Sparud 1997) and Buddha (Pope and Naish 2003), which are based on the declarative debugging technique from logic programming (Shapiro 1983), Hat (Sparud and Runciman 1997b), which enables the exploration of a computation backwards starting at the program output or error message, and Hood (Gill 2000), which allows the programmer to observe the data structures at given program points.

Many of these approaches are based on recording a tree or graph structure representing the whole computation, like the Evaluation Dependence Tree (EDT) for declarative debugging or the *redex trail* in Hat. For finding bugs, this recorded structure is represented in a user-friendly (usually innermost-style) way to the programmer in a separate viewing phase. Unfortunately, this structure dramatically grows for larger computations and can contain several mega-, even gigabytes of information.

In this paper, we introduce an alternative approach to debugging lazy functional programs. Instead of storing a complete redex trail or EDT, we memorize only the information necessary to guide a call-by-value interpreter to produce the same results. To avoid unnecessary reductions, similarly to the lazy semantics, the call-by-value interpreter is controlled by a list of step numbers determining which redexes should not be evaluated. If every redex is evaluated even by a lazy strategy, the list of step numbers reduces to a single number – the total number of reduction steps in the complete computation – which demonstrates the compactness of our representation. Furthermore, we are able to prove the correctness of our approach, in contrast to the existing approaches in which the compression of stored information is only motivated as an implementation issue. We illustrate our approach with a small example.

EXAMPLE 1.1. *Consider the following simple (but erroneous) Haskell program (where the concrete code for* `fib`*, which computes the corresponding Fibonacci number, is omitted):*

```
data Nat = Zero | S Nat

take Zero   _      = []
take (S x) (y:ys) = y : take x ys

length []     = Zero
length (_:xs) = length xs

fibs x = fib x : fibs (S x)

main = length (take (S (S Zero)) (fibs Zero))
```

The lazy evaluation of `main` *does not evaluate the individual list elements. This behavior is represented by the step list* `[2,1,0,14]`*. This list is interpreted by a call-by-value interpreter to perform (in innermost order) two steps, then discard the next innermost redex (i.e., replace it by some value representing an unevaluated thunk), perform one step, discard the next redex and the following one, and finally perform 14 reduction steps. The three discarded redexes correspond to the partial evaluation of the expression* `(fibs Zero)` *to* `(_:_:_)` *(where _ denotes a discarded redex).*

This example demonstrates the compactness of our representation that usually only requires a fairly limited amount of memory (kilobytes instead of megabytes). The step list is used in the subsequent tracing/debugging session to control the call-by-value interpreter

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02, and by the DFG under grant Ha 2457/1-2.

ICFP'07 October 1–3, 2007, Freiburg, Germany.

$$\textsf{Lam}\quad \Gamma : \lambda y.e \Downarrow_\epsilon \Gamma : \lambda y.e \qquad \textsf{Con}\quad \Gamma : C\,\overline{x_n} \Downarrow_\epsilon \Gamma : C\,\overline{x_n} \qquad \textsf{Var}\quad \frac{\Gamma : e \Downarrow_E \Delta : z}{\Gamma[x \mapsto e] : x \Downarrow_E \Delta[x \mapsto z] : z}$$

$$\textsf{App}\quad \frac{\Gamma : x_1 \Downarrow_{E_1} \Delta : \lambda y.e \qquad \Delta : e^l[x_2/y] \Downarrow_{E_2} \Theta : z}{\Gamma : x_1 \mathbin{@^r} x_2 \Downarrow_{E_1 \cdot r \mapsto [|e^l|] \cdot E_2} \Theta : z} \quad \text{where } e^l = l(e)$$

$$\textsf{Let}\quad \frac{\Gamma[y \mapsto e_1[y/x]] : e_2^l[y/x] \Downarrow_E \Delta : z}{\Gamma : \textsf{let}^r\ x = e_1\ \textsf{in}\ e_2 \Downarrow_{r \mapsto [|e_2^l|] \cdot E} \Delta : z} \quad \begin{array}{l}\text{where } y \text{ is a fresh variable}\\ \text{and } e_2^l = l(e_2)\end{array}$$

$$\textsf{Case}\quad \frac{\Gamma : x \Downarrow_{E_1} \Delta : C_i\,\overline{x_{k_i}} \qquad \Delta : {e_i}^l[\overline{x_{k_i}/y_{k_i}}] \Downarrow_{E_2} \Theta : z}{\Gamma : \textsf{case}^r\ x\ \textsf{of}\ \{\overline{C_n\ \overline{y_{k_n}} \mapsto e_n}\} \Downarrow_{E_1 \cdot r \mapsto [|{e_i}^l|] \cdot E_2} \Theta : z} \quad \text{where } {e_i}^l = l(e_i)$$

Figure 2. Instrumented lazy semantics

$$\begin{array}{lcll} x & \in & \mathit{Var} & \\ z & \in & \mathit{Value} & ::= \quad \lambda x.e \mid C\,\overline{x_n} \\ e & \in & \mathit{Exp} & ::= \quad \lambda x.e \\ & & & \mid \quad C\,\overline{x_n} \\ & & & \mid \quad x_1 \mathbin{@} x_2 \\ & & & \mid \quad x \\ & & & \mid \quad \textsf{let}\ x = e_1\ \textsf{in}\ e_2 \\ & & & \mid \quad \textsf{case}\ x\ \textsf{of}\ \{\overline{C_n\ \overline{x_{k_n}} \mapsto e_n}\} \end{array}$$

Figure 1. Syntax of normalized expressions

that shows the original evaluation in a more comprehensible order. In a nutshell, we trade time for space in our approach.

This paper is organized as follows. The next section introduces an instrumented version of Launchbury's natural semantics for lazy evaluation so that it produces a simple trace of the computation. Then, Section 3 presents a *lazy* call-by-value semantics that is driven by (a compressed form) of the trace produced by the instrumented lazy semantics. A prototype implementation of a debugging tool for Haskell that follows the ideas presented in this paper is described in Section 4. Section 5 discusses some related work before we conclude in Section 6. The proofs of some technical results are omitted but can be found in a technical report (Braßel et al. 2007).

2. Instrumented Lazy Semantics

In this section, we consider the natural semantics of Launchbury (1993) for lazy evaluation. In this semantics, laziness is modeled in two steps. First, the input expression is normalized such that all arguments of applications and case expressions are variables. This can easily be achieved by introducing additional let bindings. Then, a semantics for normalized expressions is given, where one can easily define the semantics of sharing, i.e., laziness.

We do not go into details of normalization and directly assume normalized lambda expressions extended with (recursive) lets and constructors, as shown in Figure 1, for the syntax of our expressions. Here and in the following the notation $\overline{o_n}$ is used to denote a sequence of *objects* of the form $o_1, \ldots, o_n$. In addition to the normalization of Launchbury (1993), we assume that the first argument of an application is a variable and we restrict to lets which define only one variable. With these restrictions, it will be easier to relate the lazy and the call-by-value evaluation order. The first requirement can easily be achieved by introducing additional lets for the first argument of @. The second can be obtained by constructing a tuple containing all mutually recursive definitions within a let and selecting the corresponding arguments, cf. (Braßel et al. 2007). In the examples, we use lets with multiple bindings. Since these examples do not contain mutually recursive definitions, these lets can simply be interpreted as syntactic sugar for nested lets.

The lazy semantics is shown in Figure 2 (ignore the indices labeling the arrows and the superscripts l, r for the moment). Our rules obey the following naming conventions:

$$\Gamma, \Delta, \Theta, \Omega \in \mathit{Heap} = \mathit{Var} \to \mathit{Exp} \qquad z \in \mathit{Value}$$

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. In the examples, we will also use the notation $\Gamma \setminus x$ for a heap with $\Gamma \setminus x[y] = \Gamma[y]$ if $y \neq x$ and $\Gamma[x]$ undefined.

We use judgments of the form "$\Gamma : e \Downarrow \Delta : z$" which are interpreted as "the expression e in the context of the heap Γ evaluates to the value z with the (possibly modified) heap Δ", according to the rules of Figure 2. We briefly explain the more complex rules of our semantics:

(App) This rule allows us to evaluate function applications of the form $x_1 @ x_2$.[1] For this purpose, x_1 is first evaluated to a lambda abstraction so that a β-reduction can be performed.

(Var) This rule is used to look up the bindings of variables in the heap. When the variable x is bound to an expression e, this e is evaluated to a value z so that the binding $x \mapsto z$ replaces the original binding $x \mapsto e$ in the heap. This is essential to achieve the effect of sharing, since subsequent attempts to evaluate x will use the value z instead of repeating the evaluation of e. Note that e is evaluated in a heap which does not include the binding $x \mapsto e$. This is done to detect black holes, i.e., nonterminating computations because the evaluation of x requires again the evaluation of the same x.

(Let) In order to reduce a let expression, we add the bindings to the heap and proceed with the evaluation of the let expression. Note that we rename the variables introduced by the let construct with fresh names in order to avoid variable name clashes.

(Case) This rule is used to evaluate a case expression. For this purpose, the case argument should first be reduced to a constructor call. Then, the evaluation continues by selecting the matching branch of the case expression and by applying the corresponding matching substitution.

The proof of a judgment is a proof tree using the rules of Figure 2.

[1] The reason to write the application symbol "@" explicitly will become apparent later when we present the instrumentation of the semantics.

Labeling of expressions (function l):

$$\begin{aligned} l(\lambda x.e) &= \lambda x.e \\ l(C\ \overline{x_n}) &= C\ \overline{x_n} \\ l(x) &= x \\ l(x_1 \mathbin{@} x_2) &= x_1 \mathbin{@}^r x_2 \\ l(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) &= \mathsf{let}^r\ x = l(e_1)\ \mathsf{in}\ e_2 \\ l(\mathsf{case}\ x\ \mathsf{of}\ \{\overline{C_n\ \overline{y_{k_n}} \mapsto e_n}\}) &= \mathsf{case}^r\ x\ \mathsf{of}\ \{\overline{C_n\ \overline{y_{k_n}} \mapsto e_n}\} \end{aligned}$$

where r is always a fresh reference

Extraction of references (function $[|\ |]$):

$$\begin{aligned} [|\lambda x.e|] &= \epsilon \\ [|C\ \overline{x_n}|] &= \epsilon \\ [|x|] &= \epsilon \\ [|x_1 \mathbin{@}^r x_2|] &= r \\ [|\mathsf{let}^r\ x = e_1\ \mathsf{in}\ e_2|] &= [|e_1|] \cdot r \\ [|\mathsf{case}^r\ x\ \mathsf{of}\ \{\overline{C_n\ \overline{y_{k_n}} \mapsto e_n}\}|] &= r \end{aligned}$$

Figure 3. Labeling and extraction of references

2.1 Collecting Events

The basic idea of collecting events representing a call-by-value reduction is to keep track of a trace of the evaluated redexes in the correct left-to-right innermost order by means of an instrumented version of the lazy semantics.

Here, the evaluation of an expression e starts with an empty heap and the *labeled* expression $l(e)$ where the labeling function l is shown in Figure 3. The labeling adds a *reference* from some domain Ref with $\bot \in Ref$ to every reducible symbol of a given expression – i.e., applications, lets and cases. References need to be identifiable and the distinguished reference $\bot$ is never considered fresh. In the examples we use the natural numbers augmented with $\bot$. However, we do not need any order or arithmetic operations on references. As shown in Figure 3, we only label the *topmost evaluable symbol* in every expression. Thus, nothing is labeled in a value or a variable. Also case branches are not labeled until they are demanded by the computation.

During evaluation, the instrumented semantics will dynamically generate new references for each newly introduced expression. These references are then used to extract the history of the evaluation in form of a *sequence of events*. These events will later be consumed to compute a *step list*, cf. Example 1.1 and Definition 3.9.

We collect the references by means of an extraction function $[|\ |]$ defined in Figure 3. Since our aim is a call-by-value evaluation, this function takes a labeled expression and extracts a sequence of references following a *left-to-right innermost* order. As we will see, collecting references in this order will enable us to replay the lazy evaluation employing a sort of *call-by-value* strategy.

References are collected in the form of *events*. Each event maps a reference r_0 to a (possibly empty) sequence of references. Informally, we collect an event $r_0 \mapsto r_1 \cdot \ldots \cdot r_n$ whenever the expression whose topmost reference is r_0 reduces to the expression labeled with the references $r_1 \cdot \ldots \cdot r_n$. In the call-by-value evaluation, the reduction corresponding to r_0 will be performed before the reduction corresponding to r_1, which will be performed before the reduction corresponding to r_2, and so on. Because of the nature of events, there cannot be cycles nor more than one reference mapping to the same reference.

DEFINITION 2.1 (Sequence, $^*, |\ |, \cdot, \epsilon, \{\!\{\ \}\!\}$). *Let M be a set. Then M^* is the set of finite sequences over M with the concatenation operator " $\cdot$ ". I.e., $M^* := \{a_1 \cdot \ldots \cdot a_n \mid \{a_1, \ldots, a_n\} \subseteq M\}$. Furthermore, we denote the empty sequence by ϵ and use u, v, w to denote sequences. For a given sequence $v := a_1 \cdot \ldots \cdot a_n$ we denote by $\{\!\{v\}\!\}$ the set $\{a_1, \ldots, a_n\}$ and by $|v|$ the length n of v.*

DEFINITION 2.2 (Set of Events, Sequence of Events). *Let $M : 2^{Ref \setminus \{\bot\} \mapsto Ref^*}$ where $\bot \in Ref$ be a finite partial mapping of references different from $\bot$ to sequences of references. We write $r_0 \mapsto r_1 \cdot \ldots \cdot r_n \in M$ to denote that in M, r_0 maps to $r_1 \cdot \ldots \cdot r_n$. Moreover, we define $\hat{M}$ as the smallest set satisfying*

$$\hat{M}(r) = (\bigcup_{i=1}^{n} \hat{M}(r_i)) \cup \{\overline{r_n}\}$$

for all $r \mapsto r_1 \cdot \ldots \cdot r_n \in M$. We say that M is a set of events *iff*

- *there is no reference r with $r \in \hat{M}(r)$ and*
- *$r_1 \mapsto u \cdot r \cdot v, r_2 \mapsto u' \cdot r \cdot v' \in M$ implies $r_1 \cdot u \cdot v = r_2 \cdot u' \cdot v'$ or $(r = \bot \wedge u \cdot v \cdot u' \cdot v' = \epsilon)$.*

Furthermore, we denote by $ref(M)$ the set of all references in M, formally $ref(M) := \cup_{r \mapsto r_1 \ldots r_n \in M} \{r, r_1, \ldots, r_n\}$.

A sequence of reference mappings E is a sequence of events *iff $\{\!\{E\}\!\}$ is a set of events.*

To understand sets of events it is often useful to view them as forests, i.e., sets of *trees* because their definition requires that there is no undirected cycle in the structure of events. $\hat{M}(r)$ denotes all references reachable from r by following the structure of events, similar to a transitive closure.

Note that not every reference occurring in a set of events E is mapped to a sequence. Informally, events are generated only for those references which label expressions that were reduced during the lazy evaluation. The special reference $\bot$ will later be used to generate an event $r \mapsto \bot$ for a reference which was not reduced. In our framework this is only possible in the (non-deterministic) call-by-value semantics that we will introduce in the next section. In the context of lazy evaluation such an event can only be produced by a so-called "finalizer", a side-effect triggered by the garbage collector. We will not consider garbage collection in this paper.

The instrumented semantics for computing sequences of events is presented in Figure 2. Whenever the evaluation of a new expression is started in the rules performing relevant steps (rules App, Let, and Case), we introduce a new labeling for this new expression. For instance, in rule App, we label the body of the function (e) with fresh references (e^l). This reduction has to be performed in the call-by-value evaluation as well, which will possibly introduce further innermost redexes represented by the references in $[|e^l|]$.

Following Launchbury (1993), we write judgments sequentially as follows: if $\Gamma : e \Downarrow_E \Delta : z$, we write

$$\begin{array}{l} \Gamma : e \\ \left[\begin{array}{l} \text{a sub-proof} \\ \text{another sub-proof} \end{array}\right. \\ \Delta : z \end{array}$$

where the elements of the proof tree are depicted to the right of the corresponding step. As an additional abbreviation, we omit copying the result $\Delta : z$, if it stays unchanged, like in the rules App, Let, and Case. Furthermore, instead of writing the whole sequence of events in each proof step we only annotate the new events added to the sequence to the right hand-side of the proof step. The following example illustrates the instrumented semantics using this notation.

EXAMPLE 2.3. *Consider the data type for natural numbers introduced in Example 1.1 and the expression*

$$\begin{array}{lll} main & \equiv & \mathsf{let}\ \{\ const = \mathsf{let}\ \{w = \lambda a.\lambda b.\lambda c.b\}\ \mathsf{in}\ w \mathbin{@} w; \\ & & \quad\ z = Zero; \\ & & \quad\ id = \lambda x.x; \\ & & \quad\ s = id \mathbin{@} z; \\ & & \quad\ f = const \mathbin{@} z \\ & & \}\ \mathsf{in}\ f \mathbin{@} s \end{array}$$

$[\,] : \mathsf{let}^0\ \{const = \mathsf{let}^1\ \{w = \lambda a.\lambda b.\lambda c.b\}\ \mathsf{in}\ w\ @\ w;\ z = Zero; id = \lambda x.x; s = id\ @\ z; f = const\ @\ z\}\ \mathsf{in}\ \ f\ @\ s$ $\quad 0 \mapsto 2$

$\lceil\ \Gamma_1 \equiv [const \mapsto \mathsf{let}^1\ \{w = \lambda a.\lambda b.\lambda c.b\}\ \mathsf{in}\ w\ @\ w] : \mathsf{let}^2\{z = Zero;\ id = \lambda x.x;\ s = id\ @\ z;\ f = const\ @\ z\}\ \mathsf{in}\ f\ @\ s$ $\quad 2 \mapsto 3$

$\lceil\ \Gamma_2 \equiv \Gamma_1[z \mapsto Zero] : \mathsf{let}^3\{id = \lambda x.x;\ s = id\ @\ z;\ f = const\ @\ z\}\ \mathsf{in}\ f\ @\ s$ $\quad 3 \mapsto 5 \cdot 4$

$\lceil\ \Gamma_3 \equiv \Gamma_2[id \mapsto \lambda x.x] : \mathsf{let}^4\{s = id\ @^5\ z; f = const\ @\ z\}\ \mathsf{in}\ f\ @\ s$ $\quad 4 \mapsto 7 \cdot 6$

$\lceil\ \Gamma_4 \equiv \Gamma_3[s \mapsto id\ @^5\ z] : \mathsf{let}^6\{f = const\ @^7\ z\}\ \mathsf{in}\ f\ @\ s$ $\quad 6 \mapsto 8$

$\lceil\ \Gamma_5 \equiv \Gamma_4[f \mapsto const\ @^7\ z] : f\ @^8\ s$

$\lceil\ \Gamma_5 :\ f$

$\lceil\ \Gamma_4 :\ const\ @^7\ z$

$\lceil\ \Gamma_4 :\ const$

$\lceil\ \Gamma_6 \equiv \Gamma_4 \setminus const :\ \mathsf{let}^1\ \{w = \lambda a.\lambda b.\lambda c.b\}\ \mathsf{in}\ w\ @\ w$ $\quad 1 \mapsto 9$

$\lceil\ \Gamma_7 \equiv \Gamma_6[w \mapsto \lambda a.\lambda b.\lambda c.b] :\ w\ @^9\ w$

$\lceil\ \Gamma_7 :\ w$

$\lceil\ \Gamma_6 :\ \lambda a.\lambda b.\lambda c.b$

$\lfloor\ \Gamma_7 : \lambda a.\lambda b.\lambda c.b$ $\quad 9 \mapsto \epsilon$

$\lfloor\ \Gamma_7 :\ \lambda b.\lambda c.b$

$\lfloor\ \Gamma_8 \equiv \Gamma_7[const \mapsto \lambda b.\lambda c.b] :\ \lambda b.\lambda c.b$

$\lceil\ \Gamma_8 :\ \lambda c.z$ $\quad 7 \mapsto \epsilon$

$\lfloor\ \Gamma_9 \equiv \Gamma_8[f \mapsto \lambda c.z] :\ \lambda c.z$

$\lceil\ \Gamma_9 :\ z$

$\lceil\ \Gamma_9 \setminus z :\ Zero$ $\quad 8 \mapsto \epsilon$

$\lfloor\ \Gamma_9 : Zero$

$\Gamma_9 : Zero$

Figure 4. Example of instrumented lazy evaluation

The complete lazy derivation of evaluating $l(main)$ is shown in Figure 4. The example derivation includes the generated sequence of events on the right hand side of the figure. Considering that the references in the initial expression are 0 and 1 (remember that no further references are created since the outer let *is an abbreviation for* let $const\ =\ \ldots$ in let $z\ =\ \ldots$*), the events in this sequence can be interpreted as follows: every evaluable expression has been reduced but the one labeled with reference 5.*

It is easy to see that the sequences produced by the semantics are indeed sequences of events in the sense of Definition 2.2.

LEMMA 2.4. *Let $\Gamma : e \Downarrow_E \Delta : z$ be a lazy derivation. Then E is a sequence of events.*

Proof. Cf. (Braßel et al. 2007). □

Now, we prove a basic property of the instrumented semantics: the computed events are preserved – up to renaming of references – if closures are evaluated immediately (in rule Let), as long as their evaluation would have been eventually required in the lazy computation. This property is stated in two steps.

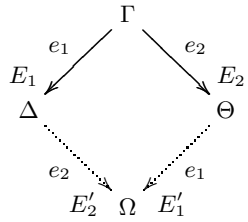
In the following, we say that two sets of events are equal up to renaming of references, denoted $M_1 \mathrel{\widehat{=}} M_2$, if they have exactly the same *shape* and only differ in the names of references of their nodes. Formally, $M_1 \mathrel{\widehat{=}} M_2$ if there exists a bijective mapping $\sigma : ref(M_1) \mapsto ref(M_2)$ such that $r \mapsto r_1 \ldots r_n \in M_1$ iff $\sigma(r) \mapsto \sigma(r_1) \ldots \sigma(r_n) \in M_2$. We also write $E_1 \mathrel{\widehat{=}} E_2$ for two *sequences* of events iff $\{\!|E_1|\!\} \mathrel{\widehat{=}} \{\!|E_2|\!\}$.

Our first result is the following.[2]

THEOREM 2.5.

If $\Gamma : e_1 \Downarrow_{E_1} \Delta : z_1$ *and* $\Gamma : e_2 \Downarrow_{E_2} \Theta : z_2$
then $\Theta : e_1 \Downarrow_{E_1'} \Omega : z_1$ *and* $\Delta : e_2 \Downarrow_{E_2'} \Omega : z_2$
and $E_2 \cdot E_1' \mathrel{\widehat{=}} E_1 \cdot E_2'$

Basically this result states that, given a heap Γ and two expressions e_1 and e_2, they can be evaluated in any order, producing the same values, heap, and events. Graphically:

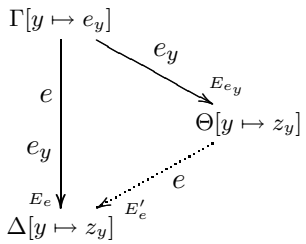


Our second result states a key property for our development:

COROLLARY 2.6.

If $\Gamma[y \mapsto e_y] : e \Downarrow_{E_e} \Delta[y \mapsto z_y] : z,\ e_y \not\equiv z_y$
and $\Gamma : e_y \Downarrow_{E_{e_y}} \Theta : z_y'$
then $z_y \equiv z_y'$ *and* $\Theta[y \mapsto z_y] : e \Downarrow_{E_e'} \Delta[y \mapsto z_y] : z$
and $E_e \mathrel{\widehat{=}} E_{e_y} \cdot E_e'$

Informally speaking it states that, given a let expression of the form let $y = e_y$ in e, we can evaluate first the closure e_y and then e in the resulting heap with the updated binding for y, and we still produce the same value, heap and events as when the standard lazy evaluation order is followed, provided that the evaluation of e_y was demanded during the evaluation of e. The following diagram depicts this property:



Proof. Applying rule Var, we have:

$$\frac{\Gamma : e_y \Downarrow_{E_{e_y}} \Theta : z_y'}{\Gamma[y \mapsto e_y] : y \Downarrow_{E_{e_y}} \Theta[y \mapsto z_y'] : z_y'}$$

[2] In the following result – and in Corollary 2.6 – we omit the details about renaming variables in the heap, which are addressed in the full version of this paper (Braßel et al. 2007).

$$\textsf{Discard} \qquad \Gamma : e \Downarrow_{(r_1 \mapsto \bot)\cdot\ldots\cdot(r_n \mapsto \bot)} \Gamma : \_ \qquad \text{where } e \notin \mathit{Value} \cup \mathit{Var} \text{ and } r_1 \cdot \ldots \cdot r_n = [|e|]$$

$$\textsf{Let} \qquad \frac{\Gamma : e_1[y/x] \Downarrow_{E_1} \Delta : z \quad \Delta[y \mapsto z] : e_2^l[y/x] \Downarrow_{E_2} \Theta : z'}{\Gamma : \text{let}^r\ x = e_1 \text{ in } e_2 \Downarrow_{E_1 \cdot (r \mapsto [|e_2^l|]) \cdot E_2} \Theta : z'} \qquad \text{where } y \text{ fresh and } e_2^l = l(e_2)$$

Figure 5. Non-deterministic Call-by-Value Rules

Since e and y are reducible with the same heap $\Gamma\,[y \mapsto e_y]$, we can apply Theorem 2.5 which yields the existence of two dual derivations. One of them is very simple:

$$\frac{\Delta : z_y \Downarrow_\epsilon \Delta : z_y}{\Delta[y \mapsto z_y] : y \Downarrow_\epsilon \Delta[y \mapsto z_y] : z'_y}$$

This already allows us to deduce that $z_y \equiv z'_y$ and simplifies the conclusion of the other derivation as intended:

$$\Theta[y \mapsto z_y] : e \Downarrow_{E'_e} \Delta[y \mapsto z_y] : z$$

Finally, we have $E_e \cdot \epsilon \mathrel{\widehat{=}} E_{e_y} \cdot E'_e$. □

Corollary 2.6 implies that, for any lazy computation, we can construct an equivalent computation that follows the call-by-value evaluation order with the exception that expressions are only evaluated as much as needed in the original lazy computation (i.e., it is basically a reordering of the lazy computation).

In the next section, we will introduce a call-by-value semantics that is driven by the events of an associated lazy computation.

3. Lazy Call-By-Value Evaluation

In this section, we introduce a *lazy* call-by-value semantics. It is call-by-value because the arguments of a function are evaluated before the function is called. It is still lazy because every argument is only evaluated as much as needed in the corresponding lazy evaluation. Our first step towards this goal is to present a semantics which discards unevaluated expressions non-deterministically.

3.1 Non-deterministic Call-By-Value Evaluation

Assume that we replace rule Let from Figure 2 with the rules of Figure 5. A derivation obtained by this new set of rules will be called a *non-deterministic call-by-value derivation* in the following. This is because the new version of rule Let implements a call-by-value semantics where closures are evaluated as soon as they are introduced. On the other hand, rule Discard allows us to non-deterministically discard the evaluation of a closure when its value is not needed in the considered computation. In this case, we introduce a fresh value denoted by _, i.e., _ is a constructor which does not appear in the initial expression.

The rule Discard is only applicable to reducible expressions whose topmost symbol is a let, a case or an application @, i.e., it is only applicable where the call-by-value reduction would perform superfluous steps from the perspective of the lazy computation.

EXAMPLE 3.1. *Consider again the program of Example 2.3. The unique non-deterministic call-by-value computation that produces an equivalent sequence of events is shown in Figure 6. In this case, the associated sequence is:*

$$(1 \mapsto 2) \cdot (2 \mapsto \epsilon) \cdot (0 \mapsto 3) \cdot (3 \mapsto 4) \cdot (4 \mapsto 6 \cdot 5) \cdot (6 \mapsto \bot) \cdot (5 \mapsto 8 \cdot 7) \cdot (8 \mapsto \epsilon) \cdot (7 \mapsto 9) \cdot (9 \mapsto \epsilon)$$

Apart from event $(6 \mapsto \bot)$*, it is equivalent to the sequence of Example 2.3.*

The following result states the equivalence between the original lazy semantics and the non-deterministic call-by-value version. In the following, given a heap Γ, we denote by Γ^- any heap that can be obtained from Γ by replacing unevaluated closures $x \mapsto e$ by $x \mapsto \_$. Likewise we denote by $E^\bot$ a sequence of events that can be obtained from E by replacing all events $r \mapsto \bot$ by ϵ.

THEOREM 3.2.
If $\Gamma : e \Downarrow_E \Delta : z$ *is a lazy derivation*
then there is one and only one non-deterministic call-by-value derivation $\Gamma : e \Downarrow_{E'} \Delta^- : z$ *such that* $E \mathrel{\widehat{=}} E'^\bot$.

Proof. By repeatedly applying Corollary 2.6 to the initial derivation $\Gamma : e \Downarrow_E \Delta : z$, we know that there exists a derivation $\Gamma : e \Downarrow_{E_2} \Delta : z$ such that $E \mathrel{\widehat{=}} E_2$ and for every let expression $\textsf{let}\ x = e\ \textsf{in}\ e'$, the closure e was evaluated before e' if needed. From this derivation we can construct a non-deterministic call-by-value derivation $\Gamma : e \Downarrow_{E'} \Theta : z$. We only need to apply rule Discard for every closure which is not evaluated in Δ. Because neither rule Var nor rule Discard produce any events adding to $E'^\bot$, we have $E_2 = E'^\bot$. Also, by definition of Discard, Δ and Θ only differ in the value _ which yields $\Delta^- = \Theta$.

Now, we should prove that there is no other non-deterministic call-by-value derivation $\Gamma : e \Downarrow_{E''} \Delta^- : z$ such that $E \mathrel{\widehat{=}} E''$.

The only source of non-determinism introduced by the rules of Figure 5 is the overlapping of rule Discard with the rules Let, App and Case. All three rules produce an event adding to $E'^\bot$ whereas rule Discard does not. As the labeling function l always introduces fresh labels, every reference occurring in Δ only occurs once. As the rule Discard eliminates a *labeled expression* from the heap, the remaining derivation cannot generate an event for the discarded references. This means that applying any of the three rules instead of Discard necessarily produces a different sequence of events. □

A direct consequence of this theorem is that we can use the events produced by a lazy evaluation and construct, using this information, the *unique* corresponding call-by-value derivation. The order in which the events were produced does not matter for this theorem. However, this order will become important for the considerations in the next sub sections.

Concluding the section, we make a simple but useful observation about the heaps occurring in non-deterministic call-by-value derivations,

DEFINITION 3.3 (Call-by-Value Heap). *A* call-by-value heap *is a partial mapping from variables to* values *rather than expressions. The notation for heaps carries over to call-by-value heaps.*

PROPOSITION 3.4. *Let* $[] : e \Downarrow_E \Delta : z$ *be a non-deterministic call-by-value derivation. Then each heap occurring in that derivation is a call-by-value heap.*

Proof. The only rules manipulating the heap are the rule Let of Figure 5 and rule Var of Figure 2. Both rules introduce only bindings to values. □

3.2 From Events to Step List

In the introduction an example of using a *step list* was given. This section describes how to obtain a step list from a sequence of events and how to use this step list to drive a lazy call-by-value semantics.

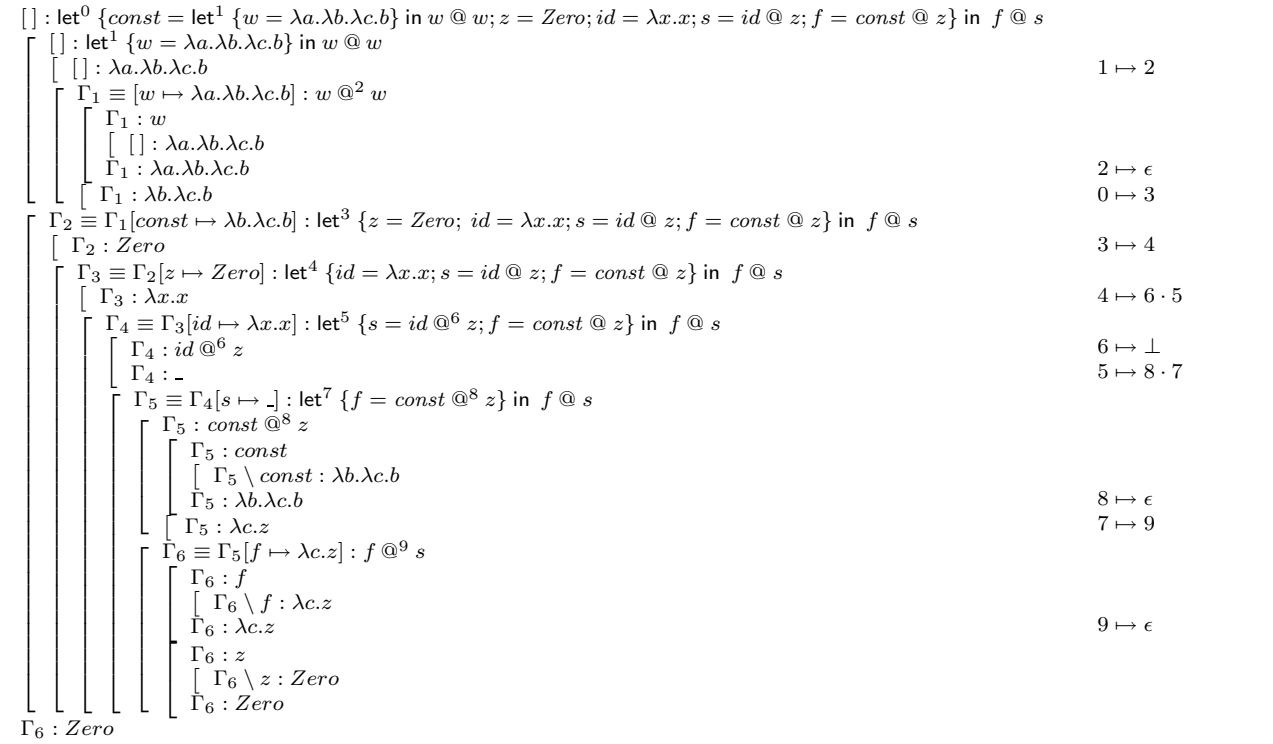


Figure 6. Example of non-deterministic call-by-value computation

The main idea is that, by considering only a given *set* of events, one can see in which sequence references would be mapped by the lazy call-by-value semantics and where in this order references are discarded. A sequence of reduced references means that ordinary rules, i.e., not Discard, were applied. Simply counting the reduced references between the discarded ones tells us how many ordinary rules can be applied in the call-by-value derivation.

According to this main idea, we first define an ordering on references for a given set of events. The additional condition on the set of events E is equivalent to requiring that the mappings in E form a tree (rather than a forest). The defined ordering is then a standard prefix ordering.

DEFINITION 3.5 (Root, Rooted, Reference Ordering $<_M$). *Let M be a set of events containing a reference r such that $ref(M) = \{r\} \cup \hat{M}(r)$. Then we call M* rooted *and r the* root *of M and define the* reference ordering $<_M$ *as:*

$$p <_M q \;:\Leftrightarrow\; (q \in \hat{M}(p) \vee (\exists s, u, s_p, v, s_q, w : s \mapsto u \cdot s_p \cdot v \cdot s_q \cdot w \in M \wedge p \in \hat{M}(s_p) \cup \{s_p\} \wedge\; q \in \hat{M}(s_q) \cup \{s_q\}))$$

LEMMA 3.6. *For a rooted set of events M, ordering $<_M$ is linear.*

Proof. $<_M$ is irreflexive and antisymmetric because M contains no cycles. $<_M$ is transitive as $\hat{M}$ is a transitive closure. $<_M$ is total because there exists a reference $r \in M$ such that $ref(M) = \{r\} \cup \hat{M}(r)$, yielding that any two references p, q must be comparable by choosing r for the existential proposition. □

We will show that this "prefix order" on the references in (special) sets of events is exactly the order in which events are generated in the non-deterministic call-by-value derivation, i.e., if the generated sequence is of the form $u \cdot (r \mapsto \ldots) \cdot v \cdot (q \mapsto \ldots) \cdot w$ then $r <_{\{\!\{E}\!\}} q$. Before showing this, we need to extend the generated events by a root, according to Definition 3.5. This is done by bringing together the references labeling the initial expression.

LEMMA 3.7. *Let $\Gamma : e \Downarrow_E \Delta : z$ and $r \notin ref(\{\!\{E\}\!\})$ a fresh reference. Then for $M' := \{\!\{(r \mapsto [|e|]) \cdot E\}\!\}$ holds $ref(M') = \{r\} \cup \hat{M'}(r)$.*

Proof. Since all references $r_1 \cdot \ldots \cdot r_n$ created dynamically during the derivation $\Gamma : e \Downarrow_E \Delta : z$ are immediately mapped to other references by an event $r \mapsto r_1 \cdot \ldots \cdot r_n$ upon creation, we have $ref(\{\!\{E\}\!\}) = \bigcup_{q \in [|e|]} \hat{\{\!\{E\}\!\}}(q) \cup \{q\}$. Thus, adding the event $r \mapsto [|e|]$ yields the desired property. □

Lemma 3.7 essentially means that we can use the ordering $<_{\{\!\{E\}\!\}}$ on the events generated by our instrumented semantics. This enables us to formulate our theorem that the non-deterministic call-by-value derivation generates the events in exactly that order.

THEOREM 3.8. *Let $\Gamma : e \Downarrow_E \Delta : z$ be a non-deterministic call-by-value derivation, $\diamond$ a reference with $\diamond \notin ref(\{\!\{E\}\!\})$ and $M' = \{\!\{(\diamond \mapsto [|e|]) \cdot E\}\!\}$. Then $r <_{M'} q$ holds for all u, r, u', v, q, v', w with $E = u \cdot (r \mapsto u') \cdot v \cdot (q \mapsto v') \cdot w$.*

Proof. By induction on the structure of the derivation:
The base cases, rules Lam and Val, are immediate as $[|e|] = E = \epsilon$.
The base case for rule Discard holds, since the generated sequence $(r_1 \mapsto \bot) \cdot \ldots \cdot (r_n \mapsto \bot)$ is constructed in correspondence to $[|e|] = r_1 \cdot \ldots \cdot r_n$.
Inductive cases:
Var: The claim directly stems from the induction hypothesis, since the sequence of events is unchanged in this rule.
App: By Proposition 3.4, the event sequence of the left antecedence for applications of this rule is always ϵ. By the induction hypothesis, the claim holds for the evaluation of the body $\Delta : e^l[x_2/y] \Downarrow_{E_2} \Theta : z$. Since this expression is freshly labeled, adding the event

$$\textsf{Discard} \quad 0 : ns,\ \Gamma : e \ \downarrow\ \Gamma : \_,\ ns \qquad \text{where } e \notin \mathit{Value} \cup \mathit{Var} \text{ and } e \neq \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

$$\textsf{Lam} \quad ns,\ \Gamma : \lambda y.e \ \downarrow\ \Gamma : \lambda y.e,\ ns$$

$$\textsf{Con} \quad ns,\ \Gamma : C\ \overline{x_n} \ \downarrow\ \Gamma : C\ \overline{x_n},\ ns$$

$$\textsf{App} \quad \frac{n : ns,\ \Gamma[x_1 \mapsto \lambda y.e] : e[x_2/y] \ \downarrow\ \Theta : z,\ ms}{n+1 : ns,\ \Gamma[x_1 \mapsto \lambda y.e] : x_1\ @\ x_2 \ \downarrow\ \Theta : z,\ ms}$$

$$\textsf{Var} \quad ns, \Gamma : x \ \downarrow\ \Gamma : \Gamma[x], ns$$

$$\textsf{Let1} \quad \frac{ns,\ \Gamma : e_1[y/x] \downarrow \Delta : z, m+1 : ms \qquad m : ms,\ \Delta[y \mapsto z] : e_2[y/x] \downarrow \Theta : z',\ ks}{ns,\ \Gamma : \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \ \downarrow\ \Theta : z',\ ks} \quad \text{where } y \text{ fresh}$$

$$\textsf{Let2} \quad \frac{ns,\ \Gamma : e_1[y/x] \downarrow \Delta : z, 0 : ms}{ns,\ \Gamma : \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \ \downarrow\ \Delta[y \mapsto z] : \_,\ ms} \quad \text{where } y \text{ fresh}$$

$$\textsf{Case} \quad \frac{n : ns,\ \Gamma[x \mapsto C_i\ \overline{x_{k_i}}] : e_i[\overline{x_{k_i}/y_{k_i}}] \ \downarrow\ \Delta : z,\ ms}{n+1 : ns,\ \Gamma[x \mapsto C_i\ \overline{x_{k_i}}] : \mathsf{case}\ x\ \mathsf{of}\ \{\overline{C_n\ \overline{y_{k_n}} \mapsto e_n}\} \ \downarrow\ \Delta : z,\ ms}$$

Figure 7. Lazy call-by-value semantics

$r \mapsto [|e^l[x_2/y]|]$ means that $r <_{M'} r'$ for all $r' \in \mathit{ref}(\{\!|E_2|\!\})$. Since r is reduced in this application. i.e., before the body $e^l[x_1/y]$, this is the desired property.
Case: As Proposition 3.4 also holds for case expressions, this case is analogous to the previous case App.
Let: An application of this rule is of the following form:

$$\frac{\Gamma\,[y \mapsto e_1[y/x]] : y \Downarrow_{E_1} \Delta : z \ \ \Delta[y \mapsto z] : e_2^l[y/x] \Downarrow_{E_2} \Theta : z'}{\Gamma : \mathsf{let}^r\ x = e_1\ \mathsf{in}\ e_2 \ \Downarrow_{E_1 \cdot (r \mapsto [|e_2^l|]) \cdot E_2} \Theta : z'}$$

where y fresh and $e_2^l = l(e_2)$. Analogous to the previous cases, adding the event $r \mapsto [|e_2^l|]$ yields the desired property with respect to the evaluation of the body e_2^l. Moreover, by definition of function $[|\ |]$ the event $\diamond \mapsto [|\mathsf{let}^r\ x = e_1\ \mathsf{in}\ e_2|]$ is equal to $\diamond \mapsto [|e_1|] \cdot r$ which means that $r' <_{M'} r$ for all $r' \in \mathit{ref}(\{\!|E_1|\!\})$. As the expression e_1 is indeed evaluated before reducing $\mathsf{let}^r\ x = e_1\ \mathsf{in}\ e_2$, this concludes the proof. □

Theorem 3.8 directly justifies how to define the step list. Following the linear sequence provided by $<_{M'}$, we count the length of reference sequences which do not contain a discarded reference, i.e., one that is mapped to $\bot$.

DEFINITION 3.9 (Step List). *Let E be a sequence of events. Then the* step list $st(E)$ *is defined as*

$$\begin{aligned} st(v \cdot (r \mapsto \bot) \cdot w) &= |v| : st(w) \ \textit{if}\ v = v^\bot \\ st(v) &= |v| : [\,] \qquad \textit{if}\ v = v^\bot \end{aligned}$$

EXAMPLE 3.10. *Consider the events from Example 3.1. With*

$$E_1 = (1 \mapsto 2) \cdot (2 \mapsto \epsilon) \cdot (0 \mapsto 3) \cdot (3 \mapsto 4) \cdot (4 \mapsto 6 \cdot 5)$$
$$E_2 = (5 \mapsto 8 \cdot 7) \cdot (8 \mapsto \epsilon) \cdot (7 \mapsto 9) \cdot (9 \mapsto \epsilon)$$

we produce the following step list:

$$st(E) = st(E_1 \cdot (6 \mapsto \bot) \cdot E_2) = 5 : st(E_2) = [5, 4]$$

The step list can be interpreted as follows: apply the rules of the call-by-value semantics so that the first five pre-redexes should be evaluated, the next one should be discarded, and the remaining ones (four) should be evaluated.

3.3 Step-driven Call-by-Value Evaluation

A simple observation will be useful in the following.

PROPOSITION 3.11. *Let r be a reference, $v \neq \bot$ a sequence of references and E a sequence of events. Then there exist a natural number n and a list of numbers ns such that $st(E) = n : ns$ and $st((r \mapsto v) \cdot E) = n + 1 : ns$.*

Proof. Immediate consequence of Definition 3.9. □

We now introduce our *lazy* call-by-value semantics that is driven by the computed step list. The rules of the semantics are shown in Figure 7. The judgments are extended with step lists denoted as ns. We briefly explain the most relevant rules:

- Rule Discard can only be applied when the step list begins with 0. This means that a number in the list has been consumed and, thus, no innermost reduction should be performed.
- In the rules App, Var and Case observe that we assume that the variables are already bound in the current heap to a value. This is justified by Proposition 3.4.
- Rule Let is split into rules Let1 and Let2. This is motivated by the fact that an expression may contain nested let bindings labeled with different references. Then, if the outermost let is to be discarded, one should first also discard the inner lets, and this is precisely the reason to introduce the new rule Let2. The choice between the two rules is determined by the outgoing step list of the first premise.

There is one main difference between the non-deterministic call-by-value semantics and the lazy call-by-value semantics. The non-deterministic discard rule can replace a complex expression, i.e., one with more than one label, by _, whereas the lazy call-by-value semantics needs to discard every labeled sub-expression separately. In the following lemma we show that we can construct a lazy call-by-value derivation consisting of applications of Let2 and Discard to discard complex expressions.

LEMMA 3.12 (Discarding complex expressions). *Let $n > 0$ be a natural number, e an expression with $|[|l(e)|]| = n$, Γ a call-by-value heap and ns an arbitrary sequence of natural numbers. Then*

there exists a lazy call-by-value derivation

$$\underbrace{0 : \ldots : 0}_{n} : ns, \Gamma : e \downarrow \Delta : \_, ns \text{ such that } \Gamma \subseteq \Delta.$$

Proof. Cf. (Braßel et al. 2007) □

Now we can prove the correspondence between the non-deterministic and the lazy call-by-value semantics. To compare heaps between the different derivations we denote by $\Gamma^{\overline{l}}$ the heap Γ without labels.

THEOREM 3.13. *Let e be an expression, $e^l := l(e)$, Γ a call-by-value heap and $\Gamma : e^l \Downarrow_E \Delta : z$ a non-deterministic call-by-value derivation. Then for all sequences of events E' there exists a step-driven derivation $st(E \cdot E'), \Gamma^{\overline{l}} : e \downarrow \Delta' : z, st(E')$ with $\Delta^{\overline{l}} \subseteq \Delta'$.*

Proof. We inductively construct the step-driven derivation from the non-deterministic derivation.
Base cases:
Lam: The derivation is $\Gamma : \lambda x.e' \Downarrow_\epsilon \Gamma : \lambda x.e'$.
Since for all E', $st(\epsilon \cdot E') = st(E')$, we can construct the derivation $st(E'), \Gamma^{\overline{l}} : \lambda x.e' \downarrow \Gamma^{\overline{l}} : \lambda x.e', st(E')$.
Con: This case is analogous to the previous case.
Discard: An application of the rule Discard looks as follows:

$$\Gamma : e^l \Downarrow_{(r_1 \mapsto \bot) \cdot \ldots \cdot (r_n \mapsto \bot)} \Gamma : \_$$

where $r_1 \cdot \ldots \cdot r_n = [|e^l|]$. Now we have

$$st((r_1 \mapsto \bot) \cdot \ldots \cdot (r_n \mapsto \bot) \cdot E') = \underbrace{0 : \ldots : 0}_{n} : st(E')$$

Therefore, an application of Lemma 3.12 yields the existence of

$$\underbrace{0 : \ldots : 0}_{n} : st(E'), \Gamma^{\overline{l}} : e \downarrow \Delta' : \_, st(E') \text{ where } \Gamma^{\overline{l}} \subseteq \Delta'.$$

Inductive cases:
Var: Both variants of this rule obviously neither change the set of events nor the step lists. Due to Proposition 3.4 the heap also stays unchanged. Therefore, we can omit evaluating the binding of x in Γ and the claim directly stems from the inductive hypothesis.
App: By Proposition 3.4 the derivation has the form

$$\frac{\dfrac{\Gamma : \lambda x.e' \Downarrow_\epsilon \Gamma : \lambda x.e'}{\Gamma' : x_1 \Downarrow_\epsilon \Gamma' : \lambda x.e'} \qquad \Gamma' : e^l \Downarrow_E \Delta : z}{\Gamma' : x_1 \mathbin{@} x_2 \Downarrow_{(r \mapsto [|e^l|]) \cdot E} \Delta : z}$$

where $\Gamma' = \Gamma[x_1 \mapsto \lambda x.e']$ and $e^l = l(e'[x_2/x])$.
By induction hypothesis we can construct a derivation for all E':

$$st(E \cdot E'), \Gamma'^{\overline{l}} : e'[x_2/x] \downarrow \Delta' : z, st(E')$$

where $\Delta^{\overline{l}} \subseteq \Delta'$. By Proposition 3.11, $st((r \mapsto [|e^l|]) \cdot E \cdot E')$ is of the form $n + 1 : ns$. Hence we can construct the derivation

$$\frac{n : ns, \Gamma'^{\overline{l}} : e'[x_2/x] \downarrow \Delta' : z, st(E')}{n + 1 : ns, \Gamma'^{\overline{l}} : x_1 \mathbin{@} x_2 \downarrow \Delta' : z, st(E')}$$

Case: As Proposition 3.4 also holds for case expressions, this case is analogous to the previous case App.
Let: The non-deterministic derivation has the form

$$\frac{\Gamma : e_1^l[y/x] \Downarrow_{E_1} \Delta : z \quad \Delta[y \mapsto z] : e_2^l[y/x] \Downarrow_{E_2} \Theta : z'}{\Gamma : \text{let}^r\ x = e_1^l \text{ in } e_2 \ \Downarrow_{E_1 \cdot (r \mapsto [|e_2^l|]) \cdot E_2} \ \Theta : z'}$$

where y is a fresh variable and $e_2^l = l(e_2)$.
We have $[|e_1^l[y/x]|] \cdot r = [|e_1^l|] \cdot r = [|\text{let}^r\ x = e_1^l \text{ in } e_2|]$. Furthermore, by induction hypothesis we can construct the derivations for all E', E''

$$st(E_1 \cdot E''), \Gamma^{\overline{l}} : e_1[y/x] \downarrow \Delta' : z, st(E'') \text{ and}$$
$$st(E_2 \cdot E'), \Delta'[y \mapsto z] : e_2[y/x] \downarrow \Theta' : z, st(E')$$

where $\Delta^{\overline{l}} \subseteq \Delta'$ and $\Theta^{\overline{l}} \subseteq \Theta'$. We choose $E'' = (r \mapsto [|e_2^l|]) \cdot E_2 \cdot E'$ and conclude that by Proposition 3.11 $st(E'')$ is of the form $m + 1 : ms$. Therefore, we can construct the derivation

$$\frac{st(E_1 \cdot E''), \Gamma^{\overline{l}} : e_1[y/x] \downarrow \Delta' : z, m + 1 : ms \quad m : ms, \Delta'[y \mapsto z] : e_2[y/x] \downarrow \Theta' : z, st(E')}{st(E_1 \cdot E''), \Gamma : \text{let } x = e_1 \text{ in } e_2 \downarrow \Theta' : z, st(E')}$$

This completes the proof. □

EXAMPLE 3.14. *Consider again the expression of Example 2.3 and the step list computed in Example 3.10:* $[5, 4]$. *The associated lazy call-by-value computation according to the rules of Figure 7 is shown in Figure 8.*

3.4 Consuming Events in Arbitrary Top-Down Order

The algorithm to compute the step list given in Definition 3.9 consumes events generated by a call-by-value derivation. We already know that the same *set* of events is also computed by the lazy derivation. Furthermore, we know how to order these events to obtain the call-by-value sequence, cf. Lemma 3.6. A naive way to implement the approach would therefore collect all events E generated by the lazy semantics then order them according to $<_{\{\!\{E\}\!\}}$ and finally compute the step list. However, we can do better and process each event as soon as its exact structure is known but before we know its exact position in the complete sequence. This last step substantially increases the efficiency with which the step list is generated in our approach. This algorithm is also straight forward to implement as described in the next section.

In this section, we first give an alternative definition of how to compute the step list that is less dependent on the order in which the events are generated. After that we show that the sequence of events generated by the lazy semantics meets the requirements of this alternative definition.

The events generated by both our semantics have an important property. Each reference occurring during the evaluation of an expression e was either contained in e or is *introduced* on the right hand side of an event *before* being *reduced* on the left hand side of an event. We denote this property of "introduction before usage" as "top-down" because in the view of events as trees it means that parents always come before all of their children. Formally, the top-down property is defined as follows.

DEFINITION 3.15 (Top-Down Sequence). *Let E be a sequence of events. We call E* top-down *iff for all sequences u, v, w and all references r holds If $E = u \cdot (r \mapsto v) \cdot w$ and $u \neq \epsilon$ then there exist a reference q and sequences u_1, u_2, v_1, v_2 such that $u = u_1 \cdot (q \mapsto v_1 \cdot r \cdot v_2) \cdot u_2$.*

The challenge now is to compute the unique call-by-value step list from a different sequence of events and in the presence of missing information of the form $r \mapsto \bot$. The algorithm that meets this challenge is defined as follows.

DEFINITION 3.16. *Let $E = (r \mapsto v) \cdot w$ be a top-down sequence of events and $\diamond \notin ref\,\{\!\{E\}\!\}$ a fresh reference. Then, the associated step list is computed from*

$$as(E) := list(count(E^\bot, (0, r) \cdot (0, \diamond))),$$

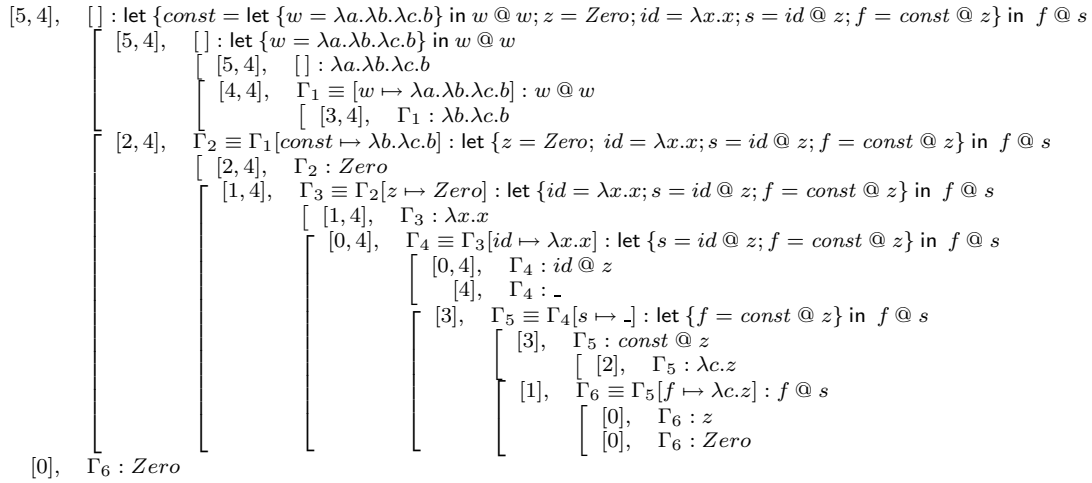


Figure 8. Example of lazy call-by-value computation with the step list [5,4]

where the functions $count$ and $list$ are defined as follows:

$$\begin{aligned}
&count((r \mapsto \epsilon) \cdot E, v \cdot (n, r) \cdot (o, s) \cdot w) &&= \\
&\qquad count(E, v \cdot (n + 1 + o, s) \cdot w) \\
&count((r \mapsto r_0 \cdot r_1 \cdot \ldots \cdot r_n) \cdot E, v \cdot (n, r) \cdot w) = \\
&\qquad count(E, v \cdot (n + 1, r_0) \cdot (0, r_1) \cdot \ldots \cdot (0, r_n) \cdot w) \\
&count(\epsilon, w) &&= w \\
&list((n, r) \cdot w) &&= n : list(w) \\
&list(\epsilon) &&= []
\end{aligned}$$

LEMMA 3.17 (Soundness of $count$). *Let $E = (r \mapsto v) \cdot w$ be a top-down sequence of events and E' the sequence where the events in E are sorted according to $<_{\{\!\{E}\!\}}$. Then $st(E') = as(E)$.*

Proof. First, observe that E' is indeed unique as $<_{\{\!\{E}\!\}}$ is a linear ordering by Lemma 3.6. Furthermore, since E is a top-down sequence and since $count$ processes this sequence left to right, each r_i in a given event $r \mapsto r_1 \cdot \ldots \cdot r_n$ cannot yet have been processed. As neither the root r nor the fresh $\diamond$ appear on the right hand side of an event, this property also holds for the initial argument.

We extend $<_{\{\!\{E}\!\}}$ such that $\diamond$ is the largest reference and observe that in the initial argument the references in the second sequence are ordered w.r.t. $<_{\{\!\{E}\!\}}$. This ordering is maintained by the algorithm. In the first rule of $count$ this is obvious as no new references are introduced to the second sequence. In the second rule the introduced references are greater than those in v as r is greater than the references in v and smaller than each r_i. The introduced references r_i are also smaller than the references in w because those are not contained in $\hat{E}(r)$ because E is a top-down sequence.

Furthermore, each number n in $(m, q) \cdot (n, r)$ within the second sequence exactly counts all references s which have been eliminated and for which $q <_{\{\!\{E}\!\}} s <_{\{\!\{E}\!\}} r$ holds. This is immediate, since the second sequence is ordered and the elimination of a reference x is added to the counter directly behind x in that sequence.

Finally, by definition of $E^{\perp}$, $count$ eliminates and counts all references except those r for which E contained an event $r \mapsto \perp$. All in all, $count$ computes the number of events between $r \mapsto \perp$ events, as $st()$ does. □

The only thing left to show is that the sequence of events generated by the lazy semantics is indeed top-down.

LEMMA 3.18. *Let $[] : e \Downarrow_E \Delta : z$ be a lazy derivation and r be a reference not occuring in $ref(\{\!\{E}\!\})$. Then $(r \mapsto [|e|]) \cdot E$ is a sequence of events which is top down.*

Proof. Cf. (Braßel et al. 2007). □

We are ready to prove our main theorem.

THEOREM 3.19. *Let e be an expression $e^l := l(e)$ and $[] : e^l \Downarrow_E \Delta : z$ be a lazy derivation, and $1 + m : ms = as(E)$. Then there exists a lazy call-by-value derivation $m : ms, [] : e \downarrow \Delta' : z, [0]$.*

Proof. By Theorem 3.2 there exists a non-deterministic call by value derivation $[] : e^l \Downarrow_{E'} \Delta^- : z$ with $E \mathrel{\widehat{=}} E'$. Since $st(\epsilon) = [0]$, by Theorem 3.13 there exists a lazy call by value derivation $st(E'), [] : e \downarrow \Delta' : z, [0]$. Let r be a reference not occuring in $ref(\{\!\{E}\!\})$. Then by Lemma 3.18, $E'' := (r \mapsto [|e|]) \cdot E'$ is top-down and therefore Lemma 3.17 yields $st(E'') = as(E'')$. Finally, by Proposition 3.11 $st(E'')$ is of the form $1 + m : ms$ with $m : ms = st(E')$. □

4. Implementation

We have implemented a debugging tool for a kernel lazy functional language (basically, Haskell without the IO monad)[3]. The tool consists of two main components:

1. A program transformation (cf. Section 4.1) that associates to each source program an instrumented program that behaves as the source program but additionally stores the step list from the sequence of events during its execution.
2. A combined tracer/declarative debugger that executes the source program in a call-by-value manner w.r.t. the step list. To ensure an efficient execution of the program during debugging time, the source program is compiled into a strict language where the functions have additional arguments to pass the step list.

4.1 Trace Generation

In this section, we describe the generation of the step list representing the information about laziness. We call this generation *tracing* because the step list is produced by side effects during the execution of a program. In this sense, the step list is a significantly condensed

[3] At the moment, we also do not provide classes. However, in contrast to the IO monad, supporting classes will not demand a conceptual extension.

version of the information retrieved by debugging approaches like the ART of Hat (Sparud and Runciman 1997b) or the EDT of Freja (Nilsson and Sparud 1997) and Buddha (Pope and Naish 2003). Tracing a given program is implemented by transforming the program into an instrumented version. Executing this new version will generate a file containing the list of steps.

One of the main goals of the presented approach is the efficient generation of the trace. To achieve this, we will not generate the whole sequence of events produced by the instrumented lazy semantics. Rather, we collapse the recorded data as much as possible during the execution of the program. The structure we keep at run time is at all times only the *list of numbers* which is built according to the operation $count$ (Definition 3.16) and, therefore, at any time reduced as much as possible. We represent the step list as a doubly-linked list where pointers are modeled as `IORefs` in order to allow efficient destructive updates of the list structure. The data types for list nodes and references to them are defined as follows:

```
type Ref     = IORef Node
data Node    = Empty | Node Ref StepCnt Ref
type StepCnt = Int
```

`Empty` is a special constructor used to mark the beginning and the end of a linked list.

We provide an interface that reflects the structure of the events that are released during the computation. In order to modify the original program as little as possible, we use a projection function with side effects to release events. The function

```
event :: Ref -> [Ref] -> a -> a
```

can be used to release an event of the form $r \mapsto r_1 \cdot r_2 \ldots \cdot r_n$ before the evaluation of some expression e using the call

`event` r `[`r_1`,`r_2`,...,`r_n`]` e

Corresponding to the definition of the operation $count$ (Definition 3.16), we can consume each event directly when it is released and do not have to consider a specific order in which they are generated. The first two rules of Definition 3.16 can be directly translated into Haskell code. The first defining rule of the function `event` corresponds to the event $r \mapsto \epsilon$ so it removes r from the linked list and adds its increased step count to its successor.

```
event r [] x = unsafePerformIO (do
  Node v n w <- readIORef r
  updSucc v w
  Node _ o s <- readIORef w
  writeIORef w (Node v (n+1+o) s)
  return x)
```

We employ the impure function `unsafePerformIO` to modify the list structure as a side effect. The second rule releases an event $r \mapsto s \cdot \overline{r_n}$ with a non-empty list of references:

```
event r (s:rs) x = unsafePerformIO (do
  Node v n w <- readIORef r
  updSucc v s
  link v (n+1) (s:rs) w
  return x)
```

Here, we employ the action

```
link :: Ref -> StepCnt -> [Ref] -> Ref -> IO Ref
```

to replace the reference given as first argument to `event` by the given list of references. `link` is only defined for non-empty lists and the given step count is added to the head of this list.

```
link v n [r] w = do
  writeIORef r (Node v n w)
  updPred w r
link v n (r:s:rs) w = do
  writeIORef r (Node v n s)
  link r 0 (s:rs) w
```

The auxiliary functions `updPred` and `updSucc` update the predecessor or the successor of a reference in a linked list respectively:

```
updPred, updSucc :: Ref -> Ref -> IO ()
updPred r p = do
  Node _ n s <- readIORef r
  writeIORef r (Node p n s)

updSucc r s = do
  Node p n _ <- readIORef r
  writeIORef r (Node p n s)
```

To compute a list of step numbers from a linked list represented by pointers, we define an action `list`, which is the direct translation of $list$ in Definition 3.16:

```
list :: Ref -> IO [Int]
list r = do node <- readIORef r
            case node of
              Empty -> return []
              Node _ n s -> do ns <- list s
                               return (n:ns)
```

The function `list` is called after the execution of the transformed program and the resulting list of step numbers is then written into a file. During the execution, the linked list represented by references is held in main memory. This list is small compared to traces generated by other debugging approaches, because it is compressed on the fly whenever a subcomputation is executed completely. The computation is started with a reference that points to distinguished *start* and *end* nodes (cf. Theorem 3.19). The constant `initialRef` returns the never collapsed *start* reference. Its successor is later supplied to the instrumented program:

```
initialRef :: Ref
initialRef = unsafePerformIO (do
  [start,hd,end,empty]
    <- sequenceM (replicate 4 (newIORef Empty))
  writeIORef start (Node empty 0 hd)
  writeIORef hd (Node start 0 end)
  writeIORef end (Node hd 0 empty)
  return start)
```

Our definition of the function `event` can be improved. Whenever a reference is replaced by a non-empty list of references, the original reference is deleted from the list by updating the successor of its predecessor. We can modify our approach to reuse references whenever such an event occurs: the original reference can be reused for the first reference in the list of replacements. As a consequence, the predecessor of this reference can be left unchanged and we generate less fresh references during the execution of the instrumented program. In our implementation, we use three specialized functions instead of the single function `event` to release events:

```
collapse :: Ref -> a -> a
onlyInc  :: Ref -> a -> a
extend   :: Ref -> [Ref] -> a -> a
```

These functions behave as follows:

- `collapse` deletes the given reference from the linked list and adds its incremented step count to its successor. It releases events of the form $r \mapsto \epsilon$.
- `onlyInc` only increments the step count stored in the given reference. It releases events of the form $r \mapsto r'$ in order to reuse the `IORef` of r for r'.

- `extend` increments the step count in the first reference and inserts the list of references after this reference. It implements events $r \mapsto \overline{r_n}$ where $n > 1$ and reuses the `IORef` of r for r_1.

Note that the call `onlyInc r` needs to modify only a single `IORef`, while `event r [r']` accesses four.

The specific function to be applied is statically known during the program transformation. Thus, there is no additional overhead for dispatching to one of these functions at run time. The reuse of references helps to save both memory and run time because less references need to be created and less references are accessed in order to process the events.

4.2 Program Debugging

In this section, we show a concrete example for a debugging session by considering the program of Example 1.1. When we run our debugging tool with this example, the program is transformed so that the step list `[2,1,0,14]` is generated during the evaluation of the initial expression `main`. As already discussed in Example 1.1, this step list points out that there are three expressions that should be discarded during the evaluation of `main` due to the partial evaluation of the expression `(fibs Zero)` to `(_:_:_)`.

The tracer/debugger uses the step list to control the call-by-value evaluation. It starts with the result of the initial expression:

```
main --> Zero
```

The result is computed during debugging time according to our lazy call-by-value evaluation. Although we trade here computation time during the debugging session against memory for storing the trace, the evaluation time required during debugging is acceptable due to the efficient innermost execution and the time required by the user to decide the next interaction.

After printing an expression and its value, the user has the option to proceed like a (call-by-value) tracer, i.e., a typical procedural debugger, or like a declarative debugger. For the latter, he has the options c (stating that the result is $c$orrect w.r.t. his intended meaning) or w (indicating a $w$rong result). For tracing, the user can move to the next reduction step or $s$kip the entire computation. Tracing is useful when the user is undecided about a computation w.r.t. the intended semantics. A debugging session is finished when the bug has been located, i.e., when there is an application of a rule where the evaluations of all functions occurring in the right-hand side are correct. If the end of a computation is reached without locating a bug (e.g., when the user has skipped over some potentially wrong subcomputation), the session is restarted from the beginning.

In our example, the entire computation is buggy so that we type w. Since the expression `(fibs Zero)` is innermost in the right-hand side of the rule for `main`, its (partial) evaluation is shown next:

```
fibs Zero --> _:_:_
```

Here we type s in order to skip over this computation (which has not enough information for a definitive decision at this point). Thus, we see the next subexpression in innermost order:

```
take (S (S Zero)) (_:_:_) --> [_,_]
```

Although this looks fine, we type s to see the next subexpression:

```
length [_,_] --> Zero
```

Since this is definitely a wrong length, we type w and obtain:

```
length [_] --> Zero
```

We type w again and obtain

```
length [] --> Zero
```

which is intended so that we type c. Immediately, we get the report that the bug is in the rule that reduces the redex `length [_]`. Looking at the source code for this rule, we can now see that an application of the constructor `S` is missing in the right-hand side.

Similar to other debugging tools, our tool is also equipped with a source code viewer that always shows the current rule applied to the outermost function of the expression under consideration. The algorithm for bug location is similar to that in other declarative debugging tools (Shapiro 1983). However, note that the call-by-value view is important to obtain the questions and trace order in a comprehensible manner.

Although the example was simple due to lack of space, the debugging tool also covers higher-order features (as already shown in the semantics) and primitive functions like arithmetic operations. Currently, the tool does not handle I/O actions. However, this is not a principle limitation: I/O actions can be treated by storing their result in the step list that is then used during the debugging session (instead of executing them again). The details about their representation are non-trivial (e.g., the effect of `getChar` and `putChar` should be visualized in some console window) so that we leave this part of the implementation for future work.

4.3 Practical Experience

In this section we compare our debugger with the Haskell Tracer Hat (Wallace et al. 2001). The time that is needed to generate our trace is comparable to the time needed to generate Hat's ART. Obviously, our trace file is much smaller than the ART. The call-by-value evaluation steps are performed reasonably fast.

We achieve this performance with a modest implementation effort. We neither use highly optimized external functions[4] to generate the trace, nor do we need sophisticated random access file operations in order to navigate our trace file. Although our trace generation is already acceptably efficient, we plan to optimize it using Haskell's foreign function interface to access a C implementation. First experiments show that such an implementation will be about five times faster than the current implementation.

As we only prototypically implemented our approach, we can not make significant comparisons with Hat. We expect that the main performance overhead of our approach is the re-computation in call-by-value order. We choose an example with a lot of computation steps in order to compare the time spent in re-computing the results with the time necessary to look them up in a trace file.

EXAMPLE 4.1. *The following example program would print the 16th prime number if we would not have introduced a bug:*

```
main = print (primes !! 15)

primes :: [Int]
primes = sieve [2..]

sieve :: [Int] -> [Int]
sieve (x:xs) =
  x : sieve (filter ((==0) . (`mod` x)) xs)
```

If we run this program, it prints the surprisingly big number 65536 which is definitely not a prime number. We use Hat and our debugger in order to debug the evaluation of the main expression `(primes !! 15)`. We measure the time that is used for generating the traces and also compare their size. Our benchmarks were run on an Apple™ PowerBook with an 1.33 GHz PowerPC G4 processor and 768 MB DDR SDRAM main memory. Hat's ART is generated within 7 seconds and is 4.7 MB big. Our trace is generated within about 5 seconds and is about 100 bytes big. So, our trace generation is as efficient as the generation of Hat's ART although it is completely implemented in Haskell. Hat's trace file is about 50000 times bigger than ours which can be printed on a few lines:

[4] We only use unboxed `Int`s to represent the step count.

```
[327673,0,1179615,589791,294879,147423,
 73695,36831,18399,9183,4575,2271,1119,
 543,255,111,39,2,0,0,184]
```

We can see that up to a million steps can be performed in innermost order and only 20 suspensions are needed during the computation.

We used the declarative debugger hat-detect and our step debugger and compared their response times. Unfortunately, debugging the program in Example 4.1 was not feasible in both debuggers, so we replaced the number 15 by 10 for this test.

Hat-detect poses the following question almost immediately:

```
primes = 2:4:8:16:32:64:128:256:512:1024:2048:_
```

We can see that we compute powers of 2 instead of primes. Stating that this is wrong, it takes more than 10 sec until the next question appears. Subsequent questions are generated within a few seconds. Our tool shows every computation step in less than a second. The time to show each step depends on how long it takes to compute the arguments and the result of the displayed call. Recomputing these values seems to be as feasible as looking them up in a redex trail.

5. Related Work

As already mentioned in the introduction, many of the approaches to debug lazy functional programs are based on recording the computation in a *redex trail* (Sparud and Runciman 1997b) or an EDT (Nilsson and Sparud 1997). The size of these structures crucially grows with the length of the computation to be debugged. As a solution to this problem, some different approaches were proposed.

Sparud and Runciman (1997a) present one of the first approaches to reduce the size of redex trails. It is based on not recording *trusted* computations (e.g., the evaluation of Prelude functions) and on pruning trails. Unfortunately, considering trusted functions does not reduce memory consumption as expected, since trusted functions are applied to expressions for which debug information has to be recorded, cf. Pope and Naish (2003). Pruning trails was not considered further since the resulting trail is incomplete and the buggy computation can be cut from the recorded information.

In the further development of Hat (Wallace et al. 2001), the problem of recording large *redex trails* was not really tackled. All information is written to a large file which results in a slowdown when generating this file and analyzing it in viewer components.

The piecemeal tracing approach of Nilsson (1999, 2001) and Pope and Naish (2003) was defined for declarative debugging by means of an *Evaluation Dependence Tree* (EDT) used in Freja (Nilsson and Sparud 1997) or Buddha (Pope 2005). In this approach, only a piece of the entire EDT is initially generated, and new pieces are computed only if they are needed by re-executing the entire program. Hence, the saving of space is purchased by additional run-time during debugging. In contrast, our approach is directly space efficient and only stores a minimal amount of information at execution time. Furthermore, their approach is basically oriented to declarative debugging in contrast to step lists.

Pope and Naish (2003) propose an optimization of the piecemeal EDT construction in which it is not necessary to restart the whole computation to compute new pieces of the EDT. Instead computations are stored which allow the generation of the missing parts of the EDT. However, there exists no evaluation on how much memory is needed for storing these computations yet (Pope and Naish 2003). Furthermore, the implementation highly depends on the internal structure of the Glasgow Haskell Compiler (ghc) and the underlying C heap in which structures are stored to prevent them from garbage collection. This makes it non-portable to other Haskell systems. The whole approach is motivated from the implementation perspective, without any correctness proofs.

6. Conclusions

We have presented a novel approach to debug lazy functional programs by re-executing them in the context of call-by-value evaluation. To avoid unnecessary computation steps, we have designed an instrumented lazy semantics that produces the information necessary to drive the call-by-value evaluator so that it discards those expressions whose evaluation is not needed. Furthermore, we have shown that this information can be obtained by program transformation. We have formally defined the resulting lazy call-by-value semantics and have proved its correctness. Our first experiments with a prototypical implementation of a debugger are encouraging.

The approach is a completely new technique of relating lazy and call-by-value computations. Although we developed it in the context of debugging it should be applicable to arbitrary analyzes of the run-time behavior of lazy functional language. As future work, we expect many fruitful applications of this technique and want to improve the implementation as well as the available back ends.

References

B. Braßel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy Call-By-Value Evaluation. Technical report, CAU Kiel, 2007.

A. Gill. Debugging Haskell by Observing Intermediate Data Structures. In *Proc. of the 4th Haskell Workshop*. Technical report of the University of Nottingham, 2000.

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.

H. Nilsson. Tracing Piece by Piece: Affordable Debugging for Lazy Functional Languages. In *Proc. of the 1999 Int'l Conf. on Functional Programming (ICFP'99)*, pages 36–47. ACM Press, 1999.

H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2): 121–150, 1997.

S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

B. Pope. Declarative Debugging with Buddha. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer Verlag, September 2005. ISBN 3-540-28540-7.

B. Pope and Lee Naish. Practical aspects of declarative debugging in Haskell-98. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240, 2003. ISBN:1-58113-705-2.

P.M. Sansom and S.L. Peyton Jones. Formally Based Profi ling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, 1997.

E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

J. Sparud and C. Runciman. Complete and Partial Redex Trails of Functional Computations. In *Proc. of the 9th Int'l Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS 1467, 1997a.

J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997b.

M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23, 2001.