Adding traces to a lazy monadic evaluator

C. Pareja, R. Peña, F. Rubio, and C. Segura

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid

1 Introduction

The debugging of lazy functional programs is a non yet satisfactorily solved problem. In recent years there have been several proposals for incorporating execution traces to lazy functional languages. In [CRW00], an extensive comparison of three of these systems can be found, namely Freja [NS97,Nil98], the Redex Trail System (RTS) [SR97a,SR97b] and Hood [Gil00]. They have been incorporated to different Haskell [JH99] compilers. Freja is a question-answer system that directs the programmer to the cause of an incorrect value. RTS allows the user to travel backwards from a value to the redex history leading to it. In Hood, the programmer first instruments the program marking the variables he wants to observe and then the system produces a printing of their final value. *Final value* is understood as the evaluation state of the variable at the point of observation.

Their implementation follows different strategies such as modifying the abstract machine or transforming the source program, but all of them have a property in common: The graph produced by the traced program is different from the original graph, i.e. the one without traces.

In this paper we propose a cleaner and more modular approach to the trace problem. We regard traces as observations of the program and at the same time we want to preserve the original graph. In this way, we establish a clean separation between the trace and the program being observed. Consequently, there may be variables in the trace referencing parts of the graph (i.e. pointers from the trace to the graph), but not the other way around. Moreover, we would like to experiment with different trace systems without modifying the normal evaluation. To this purpose, a monadic approach is followed: First a normal evaluator for a lazy language is defined; then it is trivially converted to a monadic one following well known patterns (e.g. see [Wad95]). A state transformer monad is used for incorporating traces. The trace is kept in the hidden state while the normal evaluation proceeds in the visible part. When the evaluation finishes, a *browser* can access the state and print or consult the trace (see Figure 1). We define three different monads, one for every of the above mentioned systems; Freja, RTS and Hood. Due to our restriction of not allowing references from the program to the trace, a few limitations arise with respect to these systems.

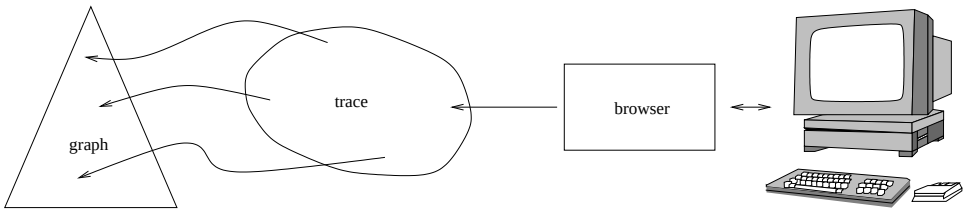

Fig. 1. General scheme of the debugging process. The user interacts with the browser, and the browser obtains the information from the trace. Notice that the trace points to the graph, but not the other way around.

The evaluator is written in Haskell and implements a simple version of Sestoft's lazy abstract machine [Ses97]. This has been chosen in order to keep the explanation manageable, but the basic elements such as the heap, the stack and the updating and sharing of nodes are already present in this machine. The same approach could be used for evaluators based on more complex machines.

2 A lazy evaluator based on Sestof's abstract machine

2.1 The language

It is an enriched λ-calculus with recursive **let**, (saturated) constructor applications and **case** expressions. To ensure sharing, the argument of an application is always a variable.

$$
\begin{array}{ll}
e \to x & \text{-- variable} \\
\mid\ \lambda x.e & \text{-- lambda abstraction} \\
\mid\ C\ \overline{x_i} & \text{-- constructor application} \\
\mid\ \textbf{let}\ \overline{x_i = e_i}\ \textbf{in}\ e & \text{-- recursive let} \\
\mid\ e\ x & \text{-- application} \\
\mid\ \textbf{case}\ e\ \textbf{of}\ \overline{C_k\ \overline{x_{ki}} \to e_k} & \text{-- algebraic case}
\end{array}
$$

2.2 A non-monadic evaluator

We present an interpreter, written in Haskell, for Sestof's machine *Mark I*. In this simple machine there is no environment binding the free variables, so β-reduction is accomplished just by substituting variables in the body of a λ-abstraction. A configuration in this machine consists of a heap, a control expression and a stack. The stack contains three different objects: arguments of pending applications, alternatives of pending pattern matchings, and update markers of pending updates. Weak-head normal forms are λ-abstractions and constructor applications. If a normal form is reached with an empty stack, the machine stops. Otherwise, the continuation is looked for in the stack.

$$\textbf{type}\ \mathit{Config} = (\mathit{Heap}, \mathit{Exp}, \mathit{Stack})$$

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Exp} \to (\mathit{Heap}, \mathit{Exp}) \\
\mathit{eval}\ e = \mathit{eval}'\ (\{\,\}, e, [\,])
\end{array}
$$

$$
\begin{array}{l}
\mathit{eval}' :: \mathit{Config} \to (\mathit{Heap}, \mathit{Exp}) \\
\mathit{eval}'\ (\Gamma, \lambda x.e, [\,]) = (\Gamma, \lambda x.e) \\
\mathit{eval}'\ (\Gamma, C\ \overline{y_i}, [\,]) = (\Gamma, C\ \overline{y_i}) \\
\mathit{eval}'\ c = (\mathit{eval}'\ .\ \mathit{step})\ c
\end{array}
$$

$$
\begin{array}{l}
\mathit{step} :: \mathit{Config} \to \mathit{Config} \\
\{\text{Rules for variables}\} \\
\mathit{step}\ (\Gamma \cup [y \mapsto \lambda x.e], y, s) = (\Gamma \cup [y \mapsto \lambda x.e], \lambda x.e, s) \\
\mathit{step}\ (\Gamma \cup [y \mapsto C\ \overline{y_i}], y, s) = (\Gamma \cup [y \mapsto C\ \overline{y_i}], C\ \overline{y_i}, s) \\
\mathit{step}\ (\Gamma \cup [y \mapsto e], y, s) \quad = (\Gamma, e, \#y : s) \quad \text{where } e \neq \lambda x.e' \wedge e \neq C\ \overline{y_i}
\end{array}
$$

$$
\begin{array}{l}
\{\text{Rule for } \textbf{let}\} \\
\mathit{step}\ (\Gamma, \textbf{let}\ \overline{x_i = e_i}\ \textbf{in}\ e, s) = (\Gamma \cup \overline{[y_i \mapsto e_i\ \overline{[y_j/x_j]}]}, e\ \overline{[y_j/x_j]}, s) \quad \text{where } \overline{y_i} \text{ are fresh variables}
\end{array}
$$

$$
\begin{array}{ll}
\{\text{Rules for application}\} & \\
\mathit{step}\ (\Gamma, e\ y, s) & = (\Gamma, e, y : s) \\
\mathit{step}\ (\Gamma, \lambda x.e, y : s) & = (\Gamma, e\ [y/x], s)
\end{array}
$$

$$
\begin{array}{ll}
\{\text{Rules for } \textbf{case}\} & \\
\mathit{step}\ (\Gamma, \textbf{case}\ e\ \textbf{of}\ \mathit{alts}, s) & = (\Gamma, e, \mathit{alts} : s) \\
\mathit{step}\ (\Gamma, C_k\ \overline{y_i}, \overline{C_j\ \overline{x_{ji}} \to e_j} : s) & = (\Gamma, e_k\ \overline{[y_i/x_{ki}]}, s)
\end{array}
$$

$$
\begin{array}{ll}
\{\text{Rules for updates}\} & \\
\mathit{step}\ (\Gamma, \lambda x.e, \#y : s) & = (\Gamma \cup [y \mapsto \lambda x.e], \lambda x.e, s) \\
\mathit{step}\ (\Gamma, C_k\ \overline{y_i}, \#y : s) & = (\Gamma \cup [y \mapsto C_k\ \overline{y_i}], C_k\ \overline{y_i}, s)
\end{array}
$$

Sestof proves some interesting properties for this machine. The most important for us is the fact that all free variables in the control expression or in the heap expressions are *pointers* (i.e. they are *not* program variables), and all bound variables (either let-bound, lambda-bound or bound in a case alternative) are *program variables*. Additionally, all pointers belong to $dom\ \Gamma$ (i.e. they are defined in the heap) and all program variables are different, provided that all bound variables are different in the initial expression. Pointers are denoted as y, and program variables are denoted as x, in the rules above. We will consistently use this convention along the paper.

2.3 A monadic evaluator

Our monadic evaluators make use of the *state transformer* monad (see e.g. [Bir98, Chapter 10]). The visible part of the monad returns the final heap and the final control expression. The hidden part —i.e. the state— stores the trace of the computation. By defining different states and the specific functions side-effecting the state we can construct different trace systems. The specific functions to be defined are *start*, which computes the initial state, and *change* which computes the state change at every transition of the machine.

```
data ST s a = ST (s → (s, a))
instance Monad (ST s) where ...

start :: Exp → State
run :: Exp → (Exp → State) → (State, Heap, Exp)
run e0 start = (sf, h, ef)
   where s0          = start e0
         ST f        = evalM e0
         (sf, (h, ef)) = f s0

evalM :: Exp → ST s (Heap, Exp)
evalM e = evalM' ({ }, e, [ ])

evalM' :: Config → ST s (Heap, Exp)
evalM' (Γ, λx.e, [ ]) = return (Γ, λx.e)
evalM' (Γ, C y_i, [ ]) = return (Γ, C y_i)
evalM' c = do c' ← stepM c
              evalM' c'

stepM :: Config → ST s Config
stepM c = ST (λs.let c' = step c
                     s' = change c c' s
                 in (s', c'))
```

The function *stepM* reflects the clean separation between the normal abstract machine transition represented by *step*, and the additional information we want to produce, that is, the building of a trace. Function *change* is responsible for carrying out such a trace construction. In the trivial case it is the identity function with respect to the state.

```
type State = ()
start e_o = ()
change :: Config → State → State
change c c' s = s
```

3 Specific functions for RTS traces

An RTS tracer computes a trace allowing the user to 'travel' backwards from the final expression to the original one by following the trail of redexes. Moreover, when an expression has several subexpressions, the user is offered the possibility of knowing the normal form of each subexpression and of following its redex trail in order to detect an erroneous reduction. With this in mind, we consider a RTS state to consist of a list of traces representing

the redex history of the current control expression. Each trace points to an expression in this redex history, so avoiding the garbage collector to consider them as garbage. If the expression is not a normal form, it will have subexpressions which can be independently reduced. In our language, the only ones having this property are applications $e\ y$, and expressions **case** e **of** $alts$. In both cases there is a single subexpression e. So, the traces for applications and **case** expressions record also the redex trail of its subexpression as a list of traces. The state has also a second component which can be seen as an emulation of Sestof's machine stack. Each time an argument or a set of alternatives is stored in machine's stack, the current expression and its redex trail are stored in state's stack. A new history is started for the subexpression. When this one reaches its normal form, the history of its parent expression is recovered from the state stack and re-started again. For our purposes, variables and **let** expressions are not traced. They are considered just as intermediate expressions leading, in the first case to the corresponding expression bound for the variable in the heap and, in the second case, to the main expression of the **let**.

$$\begin{aligned}&\textbf{data}\ RTSTrace\ a = Root\ a\\ &\qquad\qquad\quad \mid Node\ a\ [RTSTrace\ a]\\ &\qquad\qquad\quad \mid NF\ a\end{aligned}$$

$$\textbf{type}\ RTSState = ([RTSTrace\ Exp], [(Exp, [RTSTrace\ Exp])])$$

$$start\ e = ([Root\ e], [\,])$$

$$\begin{aligned}
&change\ (\Gamma, y, s)\ c'\ (ts, ets) &&= (ts, ets)\\
&change\ (\Gamma, \textbf{let}\ \overline{x_i = e_i}\ \textbf{in}\ e, s)\ c'\ (ts, ets) &&= (ts, ets)\\
&change\ (\Gamma, e_0@(e\ y), s)\ c'\ (ts, ets) &&= ([Root\ e], (e_o, ts) : ets)\\
&change\ (\Gamma, e_0@(\lambda x.e), y : s)\ c'\ (ts, (e_1, ts_1) : ets) &&= (Node\ e_1\ (NF\ e_0 : ts) : ts_1, ets)\\
&change\ (\Gamma, \lambda x.e, \#y : s)\ c'\ (ts, ets) &&= (ts, ets)\\
&change\ (\Gamma, e_0@(\textbf{case}\ e\ \textbf{of}\ alts), s)\ c'\ (ts, ets) &&= ([Root\ e], (e_o, ts) : ets)\\
&change\ (\Gamma, e_0@(C_k\ \overline{y_i}), alts : s)\ c'\ (ts, (e_1, ts_1) : ets) &&= (Node\ e_1\ (NF\ e_0 : ts) : ts_1, ets)\\
&change\ (\Gamma, C_k\ \overline{y_i}, \#y : s)\ c'\ (ts, ets) &&= (ts, ets)
\end{aligned}$$

3.1 The RTS browser

The RTS browser allows the user to travel backwards from the final expression to the original one along the trail of redexes and also to follow the redex trail of its subexpressions. When a subexpression trail is being followed, the user is allowed to come back to the trail of its parent expression.

The main work is done by function `menu`. Its first argument is a list of traces, that represents the current trail of redexes that have not been visited yet. Initially, it is the list of traces in the final state. The second argument is a list of lists of traces, initially empty. The head of this list keeps the traces list of the parent expression, the second element is the traces list of the grandfather expression, an so on. This function first shows the current redex, and then shows a menu with several options. This is done by the function `showMenu`. The options may be one or more of the following ones:

1. Show the previous redex in the current trail (the next one in the list).
2. Start the trail of the subexpression.
3. Come back to the parent expression trail.
4. Stop.

The offered options depend on the current trace `t` and on the lists of parents trails `tss1`. The stop option (number 4) is always offered. Option 3 is only offered if the list of parents is non-empty. The rest of options are as follows:

- If the current trace is $Root\ e$ and the list of parents is empty, then the end of the main trail has been reached and only option 4 is offered.
- If the current trace is $Node\ e\ ts$, the user is offered the possibility of going a step further in the current trail and also of jumping to the redex trail of its subexpression.
- If the current trace is $NF\ e$, then the user is offered the possibility of continuing with the current trail.

```
browser :: Heap -> Exp -> [RTSTrace Exp] -> IO ()
browser h ef ts =
   do
    putStr("The final expression is: " ++ showE h ef)
    menu h ts []

menu :: Heap -> [RTSTrace Exp] -> [[RTSTrace Exp]] -> IO()
menu h ts1@(t:ts2) tss1 =
  do
    let e        = expr t
        tse      = traceOfSubExpr t
        ts3:tss2 = tss1
    putStrLn("The current redex is: " ++ showE h e)
    i <- showMenu t tss1
    case i of
      1 -> menu ts2 tss1         -- show the previous redex
      2 -> menu tse (ts1:tss1)   -- go to normal form of subexpression
      3 -> menu ts3 tss2         -- come back to parent expression
      4 -> return ()             -- stop
```

Function `showE` prints an expression by using the heap to follow the definitions of the free variables. This is an important feature of real browsers because expressions may be shown with different precission degrees. For our purposes, we may assume a simple solution. For instance, that pointers are followed only one step. If a normal form is reached, then this is printed with its own free variables as single question marks `?`. This avoids entering in cycles. If an unevaluated expression is reached instead (i.e. a **case**, an application or a **let**), then a double question mark `??` is printed.

4 Specific functions for EDT traces

A Freja tracer must construct an *Evaluation Dependency Tree* (in what follows, EDT) reflecting how the main expression is reduced to normal form. If an expression consists of several subexpressions, the tree will have as children the EDT of each subexpression, plus an additional tree showing how the main expression is itself reduced. We call this last tree a *substitute* for the main expression. In our language, the only expressions having subexpressions are applications and **case**, and they have only one subexpression. As in RTS traces, variables and **let** expressions are not traced. So, an EDT trace consists of a *sequence* of substitutes for the main expression showing its evaluation path to normal form. This sequence is recorded in the state as a list of EDT traces. Also, for every intermediate expression, the trace of its only subexpression must be recorded. The natural way to do it is by using again a list of traces. In turn, each of these traces will have the same structure and so on recursively until all normal are reached. This leads us to a definition structurally identical to that of RTS traces:

$$\textbf{type } EDTTrace\ a = RTSTrace\ a$$
$$\textbf{type } EDTState\ \ = RTSState$$

The only difference with RTS traces is that now, every list of substitutes must begin with a *Root* node and end up with a *NF* one. That is, EDT traces are just the *reverse* of RTS traces. This should be clear from the fact that RTS traces show the evaluation path of every subexpression but *backwards* instead of *forwards*. It turns out that to define a tracer constructing EDT traces on the fly (i.e. as the program is running) is a very inefficient task because it implies to concatenate each new trace *at the end* of a list representing the forward history of the current expression. A more efficient approach is to construct first an RTS trace for the program and then to *reverse* the whole tree. The function to do that is surprisingly simple:

$$\begin{array}{ll} & \text{--reverse the trace} \\ EDTTrace\ e_0 = & (map\ rev\ (reverse\ (NF\ e_f : tsMain)), \Gamma, e_f) \\ & \text{--run the RTS tracer} \\ & \textbf{where}\ ((tsMain, [\,]), \Gamma, e_f) = run\ e_0\ startRTS \\ & \qquad rev\ (Node\ e\ ts) \quad = Node\ e\ (map\ rev\ (reverse\ ts)) \\ & \qquad rev\ t \qquad\qquad\ = t \end{array}$$

4.1 The EDT browser

The EDT browser follows a question-answer protocol to locate the error. It always shows at the terminal the current expression and its normal form (its *value*). Initially, the current expression is the original one. It assumes that the current expression wrongly reduces to its value and seeks to find why.

1. First, it shows the subexpressions of the current expression (in our language there is at most one) and asks whether the reductions to their corresponding values are correct or not.
2. If any of the subexpressions is wrong, this one becomes the current expression and the search proceeds in this subtree.
3. If all the subexpressions are correct, then it asks whether the first substitute reduces correctly.
   - If the answer is 'yes', then the reduction from the current expression to its first substitute is the wrong one.
   - Otherwise, the first substitute becomes the current expression and the search proceeds in the tree starting with it.

```
browser :: Heap -> [EDTTrace Exp] -> IO ()
browser h (Root e :t:ts) =  do
  let NF nf = last (t:ts)
  putStrLn ("Main expression: " showE h e ++ "=>" ++ showE h nf)
  a <- question
  case a of
     Y -> return ()           -- Initial expression is correct, stop
     N -> browser' h t ts nf  -- Initial expression not correct, start search
```

The invariant assertion of `browser' h t ts nf` is that the expression e in t wrongly reduces to the value nf, that $t \neq Root\ e$, and that ts is the trace of substitutes of e. Its definition is as follows:

```
browser' :: Heap -> EDTTrace Exp -> [EDTTrace Exp] -> Exp -> IO ()
browser' h (Node e ts1) ts2 nfe = do
   let Root se :t':ts' = ts1
       NF nfse         = last (t':ts')
       t'':ts''        = ts2
       sust            = expr t''
   putStrLn ("Subexpression: " ++ showE h se ++ "=>" ++ showE h nfse)
   a1 <- question
   case a1 of
     Y -> do       -- Subexpression is correct, investigate main expression
            putStrLn ("Main expression: " ++ showE h sust ++ "=>" ++ showE h nfe)
            a2 <- question
            case a2 of
                   -- Main expression also correct, error located
              Y -> putStr("Wrong reduction is: " ++ showE h e ++ "=>" ++ showE h sust)
                   -- Main expression not correct, investigate its successors
              N -> browser' h t'' ts'' nfe
                   -- Subexpression not correct, investigate its successors.
     N -> browser' h t' ts' nfse

browser' h (NF e) [] nf = putStr ("Error not located. Reconsider your answers")
```

The real Freja browser offers more possibilities to the programmer. For instance, the user can move backwards and forwards in the dialog, reconsidering any of the previous answers. We are sure that the information collected in our trace is enough to 'travel' to any redex produced during program execution, and that more sophisticated browsers could be programmed having the same trace structure as a basis.

5 Specific functions for Hood traces

In Hood the user can mark which program variables he/she wants to observe, and the system will show the result of evaluating the expressions associated to those variables. It only makes sense to observe let-bound variables. A let-bound variable x may have several different *incarnations* $\{y_j\}$ which are pointers. So, what the tracer must do is to watch *when* a let-bound variable x is renamed with an incarnation y, and to maintain a list of incarnations for x. When the program stops, the browser prints the expressions bound in the heap for these incarnations of x. So, as a first approach, Hood's state consists of a table *tpv* with program variables used as access keys, and lists of pointers as associated values.

```
type StateHood = Table VarP [VarH]
```

Initially, the variables selected by the user are introduced in this table, with an associated empty list of incarnations. Each time a variable x is renamed in a **let** rule, if x belongs to the observed variables its incarnation is added to the list for x. For this purpose, we assume a function

```
lastN :: Heap -> Int -> [VarH]
```

giving the names of the last n fresh variables which has been bound in the heap.

$$
\begin{array}{l}
start' :: [VarH] \to Exp \to StateHood \\
start'\ xs\ \_ = addPairs\ emptyTable\ (zip\ xs\ (repeat\ [\,]))\\
start = start'\ userSelectedVariables
\end{array}
$$

$$
\begin{array}{l}
change\ (\Gamma, \textbf{let}\ \{\overline{x_i = e_i}\}_{i\in\{1..n\}}\ \textbf{in}\ e, s)\ (\Gamma', \_, \_)\ tpv \\
\quad = foldl\ insert\ tpv\ [(x,y) \mid (x,y) \leftarrow zip\ xs\ ys,\ isObserved\ x\ tpv] \\
\quad \textbf{where}\ xs = [x_1, \ldots, x_n] \\
\qquad\quad ys = lastN\ n\ \Gamma' \\
change\ c\ c'\ tpv = tpv
\end{array}
$$

where *insert tpv* (x, y) is assumed to insert y in the list associated to x, and *isObserved x tpv* gives *True* if x belong to table *tpv*.

In HOOD, it is also possible to observe program variables bound to functions. If x is a program variable bound to a functional expression, Hood observes all applications of (the incarnations of) x to different arguments, and collects the pairs $(argument, result)$ of these applications. When the program stops, the browser prints the observations of every incarnation of x as a collection of pairs.

To emulate this feature, the first step is to introduce the incarnations y_j of x in the table *tpv* of program variables as before. Then, all the y_j are observed until (if ever) they are referenced after being updated by a lambda normal form (see first rule for variables in Section 2.2). Should this happen, the resulting lambda form in the control expression is observed until it is applied to an argument (see second application rule and flag m below). Then a pair (y, y') is stored in an auxiliary stack *ps* of pairs, being y the incarnation of the lambda and y' the pointer to the argument. Finally, when the body of the lambda is reduced to a normal form, the pair (y', nf) is stored in a table *thl* for lambda heap-variables, using y as access key.

```
type HoodState =
     (Table VarP [VarH],        -- relates program variables to incarnations
      Table VarH [(VarH, Exp)], -- relates lambda variables to pairs (arg,result)
      [Maybe (VarH, VarH)],     -- auxiliary stack for observing lambda-bodies
      Maybe VarH)               -- auxiliary flag to observe lambda-applications
```

Function *start'* and the rule given for **let** above, should be modified for this more complex state. In the first case, an empty *thl* table, an empty auxiliary stack and a *Nothing* flag should be returned. In the second case, *change* behaves as the identity function for the three new components.

$$
\begin{array}{ll}
change\ (\Gamma \cup [y \mapsto \lambda x.e], y, s)\ c'\ (tpv,\ thl,\ ps,\ m) & \\
\quad \mid y \in range\ tpv & = (tpv,\ insert\ thl\ (y, [\,]),\ ps,\ Just\ y) \\
\quad \mid \text{otherwise} & = (tpv,\ thl,\ ps,\ Nothing) \\
change\ (\Gamma, \lambda x.e, y' : s)\ c'\ (tpv,\ thl,\ ps,\ Just\ y) & = (tpv,\ thl,\ Just\ (y, y') : ps,\ Nothing) \\
change\ (\Gamma, \lambda x.e, y' : s)\ c'\ (tpv,\ thl,\ ps,\ Nothing) & = (tpv,\ thl,\ ps,\ Nothing) \\
change\ (\Gamma, e\ y, s)\ c'\ (tpv,\ thl,\ ps,\ m) & = (tpv,\ thl,\ Nothing : ps,\ Nothing) \\
change\ (\Gamma, \textbf{case}\ e\ \textbf{of}\ alts, s)\ c'\ (tpv,\ thl,\ ps,\ m) & = (tpv,\ thl,\ Nothing : ps,\ Nothing) \\
change\ (\Gamma, C_k\ \overline{y_i}, s)\ c'\ (tpv,\ thl,\ Just\ (y, y') : Nothing : ps,\ m) & = (tpv,\ insert\ thl\ (y,\ (y',\ C_k\ \overline{y_i})),\ ps,\ m) \\
change\ (\Gamma, C_k\ \overline{y_i}, s)\ c'\ (tpv,\ thl,\ Nothing : ps,\ m) & = (tpv,\ thl,\ ps,\ m) \\
change\ (\Gamma, \lambda x.e, s)\ c'\ (tpv,\ thl,\ Just\ (y, y') : Nothing : ps,\ m) & = (tpv,\ insert\ thl\ (y,\ (y', \lambda x.e)),\ ps,\ m) \\
change\ (\Gamma, \lambda x.e, s)\ c'\ (tpv,\ thl,\ Nothing : ps,\ m) & = (tpv,\ thl,\ ps,\ m)
\end{array}
$$

5.1 The Hood browser

Hood's browser is a very simple one. It just print, for each observed value, and for each incarnation of it, the values collected in the two tables. For non-funcional values, only table *tpv* is used. For functional ones, both *tpv* and *thl* tables are needed: the first one gives the different incarnations of the functional variable, each one representing a possibly different function; the second one gives the collected pairs $(arg, result)$ collected for the incarnation.

```
browser :: Heap -> Table VarP [VarH] -> Table VarH [(VarH, Exp)] -> IO ()
browser h tpv thl = mapM_ printPV (dom tp)
   where printPV x = mapM_ printHL (associatedValue x tpv)
         printHL y
            | y `notElem` dom thl = showV h y
            | otherwise           = mapM_ printPair (associatedValue y thl)
         printPair (y,e) = putStrLn (showV h y ++ "=>" showE h e)
```

where function `showV` is very similar to `showE`. It shows the expression bound in the heap to a pointer y.

6 Conclusions

The work presented here can be regarded in several different ways. The most interesting for us is to look at it as a *what if* paper. We imposed ourselves a severe restriction at the beginning: *not to modify either the original program or the abstract machine executing it*. Keeping this restriction, we have tried to emulate the main features of three up-to-date tracers for lazy languages. Our purpose was to investigate how far we could reach in this task, and also to know which could be the observations we would require both from the program and from the abstract machine in order to collect enough information to build the traces.

The conclusions in this respect are very optimistic: there is not need to modify anything in order to collect the required information. The tracers essentially need to have access to the machine configuration *previous* to each reduction step. Only in one case —the *lastN* function needed for Hood's tracer— an access to the configuration *after* the reduction step was needed. The reason was to know the most recent n pointers created in the heap. The features offered by the emulated tracers are of course more sophisticated that the ones presented here, but this has to do with the kind of browsers they provide rather than with the information collected.

A second conclusion is that correctness of our approach is guaranteed *by definition*. Our tracers are implemented by a function *change* which is *not* allowed to influence the machine configuration, i.e. the heap, the control expression or the control stack. In this way, a traced program will produce exactly the same reductions and results as the original one.

Another conclusion is that the collected data structures for RTS traces and for EDT traces are basically the same. This was already anticipated in [CRW00]. The difference between these two tracers resides in how this information is presented to the user, i.e. they differ in the browsers. In the conclusions of [CRW00], the authors explain that perhaps the EDT tracer is more appropriate for beginners because of its friendly question-answer interface, while RTS seems more adequate for expert programmers knowning about redexes and reductions. Our work offers a third possilitity of having a common trace collector and two browsers for the user to choose between.

Finally, the work presented here provides a framework and a complete set of algorithms which can be used to experiment with other alternative designs of trace systems before embarking in a fully-fledged implementation.

References

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2 edition, 1998.

[CRW00] O. Chitil, C. Runciman, and M. Wallace. Tracing and Debugging of Lazy Functional Programs — A Comparative Evaluation of Three Systems. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, number AIB-00-7 in Aachener Informatik Berichte, pages 47–62. RWTH Aachen, 2000.

[Gil00] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. In *Proceedings of the 4th Haskell Workshop*. Technical Report of the University of Nottingham, 2000.

[JH99] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. URL http://www.haskell.org, February 1999.

[Nil98] H. Nilsson. Declarative debugging for lazy functional languages. PhD Thesis, Linköpings Universitet, 1998.

[NS97] H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering: An International Journal*, April 1997.

[Ses97] P. Sestoft. Deriving a Lazy Abstract Machine. *Journal on Functional Programming*, 7(3):231–264, 1997.

[SR97a] J. Sparud and C. Runciman. Complete and Partial Redex Trails of Functional Computations. In C. Clack, K. Hammond, and T. Davie, editors, *Selected Papers from 9th International Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS 1467, 1997.

[SR97b] J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of the 9th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.

[Wad95] P. Wadler. Monads for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming. LNCS 925*. Springer-Verlag, 1995.