

Debugging Lazy Functional Programs by Asking the Oracle^{*}

Holger Siegel, Bernd Braßel

CAU Kiel usw

Abstract. The complexity of lazy evaluation forbids classic debugging techniques like a simple step-by-step representation of the buggy program run. Therefore, most sophisticated tools for searching bugs in lazy functional programs try to display the run as if the program’s underlying semantics was strict. In order to provide such a strict representation current approaches gather much information about the executed program. We utilized a new technique to drastically reduce the amount of gathered data and show how to use the reduced information to implement a debugging tool which supports declarative debugging as well as a strict step-by-step tracer.

1 Introduction

The task of designing tools to find bugs in lazy functional programs is demanding. On one hand, the sophisticated strategy employed by the underlying implementation enables the programmer to write code on a high level of abstraction. On the other hand the same sophisticated strategy makes it very hard to understand how a given program is executed step-by-step. Most successful approaches to debugging solve this problem by collecting enough data to represent the program’s execution as if the underlying strategy was strict, which is much easier to understand. Examples for such approaches are declarative debugging, cf. [4, 5], observations, cf. [3], and redex trailing, cf. [7]. The most comprehensive tool, HAT [2], comprises all three approaches among others.

In order to present information about the program in a simple way, the above approaches collect data during its execution. This is especially true, the more powerful the tool is, e.g., the HAT system collects megabytes of data in many cases. In [1], we developed an approach to collect considerably less data and still be able to provide the user with a strict view on the execution of his program. The basic idea is that the critical information to replay a lazy computation as if the underlying semantics was strict is when unneeded redexes are discarded. Therefore, we only count the number of strict steps between such discarding steps. We call the resulting list of numbers an *oracle*. Different debugging tools can then be realized as strict monadic versions of the original program correctly consuming the number of steps in an oracle. [1] contains a soundness proof for this technique, this paper presents the implementation of a debugging tool based on the oracle technique.

^{*} This work has been partially supported by DFG grant Ha 2457/5-2.

```

module Example where

import Prelude hiding (length)

length []      = 0
length (_:xs) = length xs

exp = length (take 2 (fiblist 0))

fiblist x = fib x : fibs (x+1)

fib :: Int -> Int
fib _ = error "this will not be evaluated"

```

Fig. 1. Example program

2 Example Sessions

So far, our debugging tool supports two modes. The first is an implementation of the well known declarative debugging method, described in Section 2.1. The second is a step-by-step tracer allowing us to follow a program’s execution as if the underlying semantics was strict, skipping uninteresting sub computations. In addition, the tool gives some support to search bugs in programs employing I/O. This approach to “virtual I/O” is presented along with the step-by-step mode in Section 2.2.

2.1 Declarative Debugging

Figure 1 shows a small example program containing an intentional error to demonstrate the declarative debugging mode. The function `fiblist` creates a potentially infinite list of delayed calls to function `fib`. Due to laziness, `fib` is never evaluated in our example, so we omit its definition. The infinite list is cut to the first three elements by a call to function `take`, which is defined in the usual way. On top level, function `length` is applied to count the elements of the resulting list. It is to be expected that the program returns the number 2.

```

> :l Example
Example> exp
0

```

We see that running the program reveals the result 0, which indicates that there must be a bug somewhere. Therefore, we switch on the debug mode and execute the program once again.

```

Example> :set +d
Example> exp

```

In the background, our example program (along with all of its imported modules) is transformed to a program `OracleExample`.

```
Example> exp
processing: OracleExample
up-to-date: OraclePrelude
```

As we see, our `OraclePrelude` is still up to date, and thus not generated again. The programs resulting from this transformation behave exactly like the original program. The only difference is that – as a side effect – it will produce an “oracle”. Before continuing with the debugging session we take a look at this oracle.

After evaluating `exp` in `OracleExample` has successfully terminated, the current directory contains a file called `Example.steps`:

```
$ cat Example.steps
[2,0,1,0,0,23]
```

These numbers compactly represent the laziness information. If every expression in the program was evaluated there would have been only a single number. This number indicates how many steps that evaluation would have taken. The fact that there are six numbers for this example tells us that five expressions have been discarded without evaluation. (Two calls to `fib`, two to `(+)` and one to `fiblist`). For more details about the oracle format, how it is produced and why it can be utilized to correctly execute the traced program strictly, cf. [1].

The user does not see anything of the oracle or the steps file. Directly after the steps file has been produced, the debugging tool proceeds by applying a second transformation on the modules.

```
up-to-date: StrictPrelude.hs
generating ./StrictExample.hs
```

The second transformation produces Haskell modules named `Strict*.hs`. These Haskell modules contain the definitions to execute the original program with an underlying strict semantics. The details of this transformation will be presented in Section 3 on page 7.

After the second transformation the actual debugging session is started.

```
----      ----      -----
( _ \ ( _ _ ) ( _ _ ) Believe
) _ < _ ) ( _ ) ( _ ) in
(____/() (____) () (____) () Oracles
-----type ? for help-----
```

```
exp
```

Initially, we only see a call to function `exp` which represents the whole program. By pressing `i` we turn on *inspect mode*. In inspection mode, the result of every sub computation is directly shown and can be “inspected” by the user, i.e., rated as correct or wrong. Inspection mode corresponds therefore to the declarative

debugging method. But as we will see in the next section, showing the results of sub computations can be turned on and off at will. (Of course, there is a help menu available, showing a list of all possible inputs.)

After pressing `i` the debugger evaluates the expression and displays the result.

```
exp ~> 0
```

We expected `main` to have value 2, but the program delivered value 0. Thus, we enter `w` (*wrong*) in order to tell the debugger that the result was wrong. The debugging tool stores this choice as explained in Section 3. As the value of `exp` depends on several function calls on the right hand side of its definition, the tool now displays the first of these calls in a left-most innermost order:

```
fiblist 0 ~> _ : (_ : _)
```

The line above shows that the expression `fiblist 0` has been evaluated to a list that has at least two elements. This might be correct, but we are not too sure, since this result depends strongly on the evaluation context. A “don’t know” in declarative debugging actually corresponds to the skipping of sub computations in the step-by-step mode, as described in the next section. We therefore press `s` (*skip*).

```
take 2 (_ : (_ : _)) ~> [_,_]
```

Actually, this looks quite good. By entering `c` (*correct*) we declare that this sub computation meets our expectation. Now the following calculation is displayed:

```
length [_,_] ~> 0
```

The function `length` is supposed to count the elements of a list. Since the argument is a two-element list, the result should be 2, but it is actually 0. By pressing `w` we therefore state that this calculation is erroneous. Now the debugger asks for the first sub computation leading to this result:

```
length [_] ~> 0
```

This is wrong, too, but for the sake of demonstration we delay our decision. By pressing the space bar (*step into*) we move to the sub expressions of `length [_]`. We now get to the final question:

```
length [] ~> 0
```

The length of an empty list `[]` is zero, so by pressing `c` (*correct*) we state that this evaluation step is correct. Now we have reached the end of the program execution, but a bug has not been isolated yet. We have narrowed down the error to the function call `length [_,_]`, but still there are unrated sub computations which might have contributed to the erroneous result. The tool asks if the user wants to restart the debugging session re-using previously given ratings:

```
end reached. press 'q' to abort or any other key to restart.
```

```

module IOExample where

import Prelude hiding (getLine)

getLine :: IO String
getLine = getChar >>= testEOL

testEOL :: Char -> IO String
testEOL c = if c=='\n' then return []
            else getLine >>= \ cs -> return (c:cs)

main = getLine >>= writeFile "userInput"

```

Fig. 2. I/O example

After pressing <SPACE>, the debugger restarts and asks for the remaining function calls. There is only one unrated call left within the erroneous sub computation:

```
length [] ~> 0
```

Now we provide the rating we previously skipped. After entering *w* (*wrong*) it is evident which definition contains the error:

```

found bug in rule:
  lhs = length []
  rhs = 0

```

2.2 Step-by-Step Debugging and Virtual I/O

A further interesting advantage of our approach to reexecute the program with a strict evaluation strategy is the possibility to include “virtual I/O”. During the execution of the original program, all externally defined I/O-actions with non-trivial results, i.e., other than `IO ()`, are stored in a special file. These values are retrieved during the debugging session. In addition, selected externally defined I/O-actions, e.g., `putChar`, are provided with a “virtual implementation”. To show what this means, we demonstrate how the `main` action of the program found in figure 2 is treated by our debugging tool. As described in the previous section, the program is executed to obtain the oracle in the file `IOExample.steps`. As this program contains user interaction, we also have to enter a line. We type `abc` for this demonstration. Meanwhile, along with the file containing the steps, a file `IOExample.ext` was written, containing only the sequence of values of `getChar` (and their size).

```

~/oracle> cat IOExample.ext
3,'a'3,'b'3,'c'4,'\n'

```

There is no need to identify the different calls to external functions, since I/O-actions will be executed in the strict version in exactly the same order as in the original program. This is of course essential for correctness. We now start the debugging tool, and look at single steps by typing <SPACE> twice. This is, what the tool displays:

```
main
getLine
getLine ~> getChar >>= testEOL
main ~> (getChar >>= testEOL) >>= writeFile "userInput"
initial action computed. press any key to execute it
```

In step-by-step mode, we only get to see results when a subcomputation is finished. The above lines mean that the evaluation of both, `getLine` and `main` is now complete. The results are partial calls of the bind operator (`>>=`) waiting for the world, so to speak. We press an arbitrary key to start the action followed by a <SPACE> to make on more single step and get:

```
getChar >>= testEOL
getChar
```

When we hit <SPACE> now, two things happen at once. First, the value 'a' is retrieved from the file and, second, the GUI called `B.I.O.tope` is started, which represents the virtual I/O environment. The `B.I.O.tope` is told that someone has typed an `a` on the console. This is the "virtual I/O-action" we connected with `getChar`. The window coming up is shown in Figure 3. Meanwhile on the

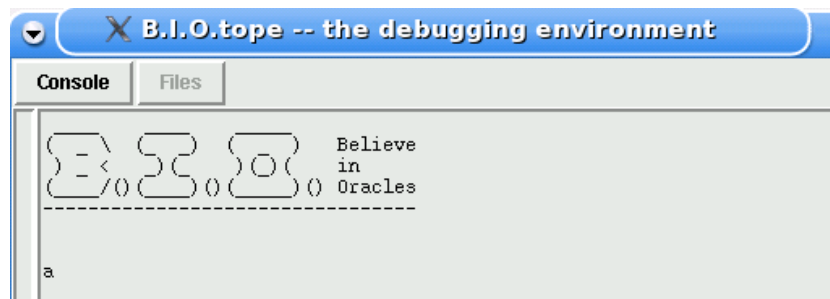


Fig. 3. The B.I.O.tope Virtual I/O Environment

console we see the call `testEOL 'a'`, which we skip by typing `s`. We directly see the result:

```
testEOL 'a' ~> (getChar >>= testEOL) >>= testEOL_lambda 'a'
(getChar >>= testEOL) >>= testEOL_lambda 'a'
```

Admittedly, the expression `testEOL_lambda 'a'` shows that the source code binding is improvable. Now we wonder, whether or not the current sub computation is interesting or not. We type `r` to have a look at the result and get:

```
(getChar >>= testEOL) >>= testEOL_lambda 'a' ~> IO "abc"
```

This is fine, so we decide to skip the computation by pressing `s`. Note, that as soon as a result is shown, we could also rate the sub computation, i.e., tell the tool that this result is correct or wrong. This information will then be considered if we restart the debugging session in inspection mode, cf. Section 2.1. It is also noteworthy that the virtual I/O commands are never issued twice although even, if we would have decided on going into the sub computation instead of skipping it.

The final action of our program is:

```
writeFile "userInput" "abc"
```

Executing this action brings another change to the `B.I.O.tope` as shown in Figure 4. There we can see the GUI has switched to the file dialog. It contains a list of files which have been read (R:) or written (W:) during the debugging session and clicking a file in this list makes the file contents visible as they are at the current point of the debugging session.

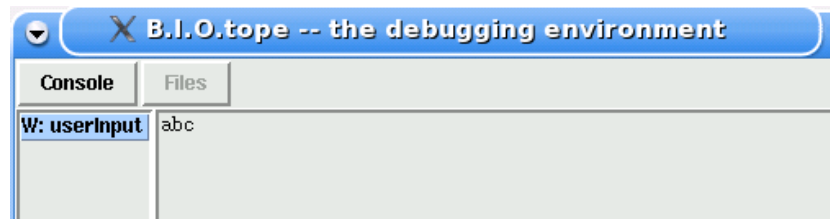


Fig. 4. Files in the B.I.O.tope Virtual I/O Environment

3 Implementation

In this section we present the runtime library `StrictSteps.hs` and its interaction with the programs that have been generated by the transformation introduced in Section 2.1. Reconsider the program in Figure 1. For this program the transformation creates a Haskell program called `StrictExample.hs`. Each generated module uses the functions that are exported by the library `StrictSteps.hs`. It also imports the transformed versions of its original imports. In this case the only such module is `StrictPrelude`. Finally, some functions from the original Haskell `Prelude` are needed. All in all, the module head of the generated module looks like this:

```

module StrictExample where

import StrictSteps
import Prelude (Maybe(..),(.),Eq(..),Show(..),Ordering(..),
                Either(..),String,Bool(..),Char(..),Float(..))
import qualified Prelude (IO,return,(>>=))
import StrictPrelude

```

3.1 Encoding of Oracles

Conceptually, an oracle is a list of logical values. This list is being consumed while an instrumented program is evaluated according to a leftmost innermost strategy. If the next entry of the oracle has value `True`, the next reduction step according to strict evaluation order will be evaluated. If the next entry has value `False`, then the next redex will be skipped and the result will be replaced by a placeholder value called `underscore`. In order to represent this oracle in a compact way it is encoded as a list of natural numbers. A number n stands for a list of n entries whose value is `True` followed by one entry of value `False`.

```

type Oracle = BoolStack
type BoolStack = [Int]

```

The function `popBoolStack` removes the first entry from an encoded oracle and returns the resulting oracle paired with the value of the removed entry. The function `pushBoolStack` appends an entry to the head of an oracle and returns the resulting oracle.

```

popBoolStack :: BoolStack -> (BoolStack, Bool)
pushBoolStack :: BoolStack -> Bool -> BoolStack

```

An empty oracle is constructed according to the following declaration:

```

emptyBoolStack :: BoolStack
emptyBoolStack = [0]

```

For example, the oracle produced in Section 2.1 to evaluate `exp` in Figure 1 was `[2,0,1,0,0,23]`. It stands for a list containing 31 entries: the first two entries have value `True`, then there are two `False` followed by one `True`, then three `False` and finally 23 `True`. Having value `True`, the first entry indicates that expression `exp` has to be evaluated. The next entry tells us that the next leftmost innermost call in Figure 1, i.e., `fiblist 0`, is also unfolded. The next two entries have value `False` and thereby indicate that the leftmost innermost calls `fib x` and `x+1` must not be evaluated but replaced by an `underscore` (also denoted by `_`) as a placeholder value. Replacing `x+1` by `_` the next call is `fibs _`. This call is then also evaluated whereas the next free expressions, `fib _`, `x+1` and another `fibs _` are discarded. The final 23 entries tell us that the remaining computation is totally strict (and 23 steps long).

3.2 The Representation of `underscore`

Expressions which are not evaluated are replaced by the special value `underscore`. Therefore, at least conceptually, every data type has to be augmented with a new element which represents that value. From a semantical point of view `underscore` resembles the undefined value \perp . If we were only interested in computing the same result with a strict semantics, we could actually represent `underscore` by a call to the Haskell function `undefined` as discarded expressions are guaranteed to be never needed in the evaluation. However, we want the debugging tool to print intermediate results and, therefore, we need to be able to tell undefined values from defined ones.

Another option is to add a new constructor to every data declaration. However, this would make the inclusion of external functions and data types much more complicated. In order to, e.g., call `(+)`, we would need to convert to and fro for each argument and the result. In addition, much of the behavior already provided would have to be duplicated, e.g., the way strings of characters are shown.

Therefore a trick is applied, which makes use of the fact that `underscore` is never evaluated outside of the display routines. We represent `underscore` by an exception:

```
underscore :: a
underscore = throw NonTermination
```

Since the oracle that guides the evaluation indicates which expressions are needed to run the program, it is guaranteed that this value will never be accessed while the program is being evaluated. It will be accessed when the debugger tries to print out a data value. As the whole evaluation is monadic, cf. the next subsection, the exception can be safely caught whenever values are printed.

3.3 Representing Values

The debugger must be able to display a textual representation of the arguments and results of the function calls being debugged. In order to provide more flexibility for the debugging tool, we represent expressions in a term structure rather than a simple string. This makes it possible to, e.g., restrict the depth in which expressions are shown and enables pretty printing. Therefore, the data type `Term` is introduced:

```
data Term = Term String [Term] | Underscore | Fail String
```

The constructor `Term` contains the name of the applied symbol and a term representation of its arguments. As discussed above, `Underscore` stands for those expressions, which were not evaluated. Finally, `Fail` represents exceptions that occurred during the execution along with an error message. The implementation of the corresponding mechanism is, however, not finished by now.

In order to retrieve term representations of a given expression, each data type has to be an instance of the class `ShowTerm`:

```
class ShowTerm a where
  showCons  :: a -> DebugMonad Term
```

These instances are automatically generated by the transformation. For example, the following instance declaration is generated for lists:

```
instance ShowTerm a => ShowTerm [a] where
  showCons [] = return (consTerm "[]" [])
  showCons (x1:x2) = do sx1 <- showTerm x1
                       sx2 <- showTerm x2
                       return (consTerm ":" [sx1,sx2])
```

The generated instances depend on the generic function `showStep`, which is responsible for catching the exception thrown by `underscore`:

```
showTerm :: ShowTerm a => a -> DebugMonad Term
showTerm x = liftIO (catch (x `seq` return Nothing) (return . Just)) >>=
  maybe (showCons x) ( e -> case e of
                          NonTermination -> return Underscore
                          ErrorCall s    -> return (Fail s))
```

3.4 The Debug Monad

The whole evaluation of the generated program takes place in a monad, the `DebugMonad`. This monad is a state monad, managing the following state:

```
data DebuggerState = DebuggerState { oracle      :: Oracle,
                                     displayMode :: IORef DMode,
                                     skipped      :: BoolStack,
                                     unrated     :: BoolStack }
```

`oracle` contains the part of the oracle that has not yet been consumed.

`displayMode` contains display options like the verbosity level and the depth up to which terms should be printed. In addition this field contains a flag indicating, which of the two debugging modes introduced in Section 2 is active.

`skipped`, `unrated` are encoded the same way as oracles. They indicate which functions are still unrated (`True`) or have been rated as correct (`False`). The list `skipped` holds the ratings of the functions that have already been displayed in the current program run, whereas `unrated` holds the ratings of the function calls that still have not been displayed in the current program run, but might have been rated in a former evaluation cycle. The purpose of `skipped` and `unrated` will be explained in greater detail in Section 3.5.

In addition to the `DebugState`, three *evaluation modes* are encoded in data type `StepMode`:

StepBackground The expression is evaluated without user interaction. For every sub expression previously given information about its correctness will be preserved by moving the corresponding entries from `unrated` to `skipped`.

StepInteractive Function calls are displayed and rated by the user. To display its result, an expression is evaluated in mode **StepBackground**. When the user presses `<SPACE>`, meaning that he wants to inspect a sub computation in greater detail, the calls of that sub computation are re-evaluated in mode **StepInteractive**.

StepCorrect Like in background mode, the expression is evaluated without user interaction. When an expression is rated as correct, all its subexpressions are considered correct, too. Thus, its subexpressions are evaluated with evaluation mode **StepCorrect**, and the information stored in **skipped** and **unrated** will remain unchanged.

The monad **DebugMonad a** is used by the debugger to manage its internal state while evaluating an expression that has result type **a**.

```
type DebugMonad a
  = StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

Values of type **BugReport** are used to report an erroneous program rule which is represented by two constructor terms, i.e., the call along with arguments and its result.

The result delivered by the debugger is lifted into monad **IO**, because the debugger has to interact with the user via **IO** actions while it evaluates the expression.

This monad is transformed by the monad transformer **ErrorT**, so that not only the result can be returned, but also the evaluation can be truncated reporting the location of an error (**Just bug**) or indicating that the user has aborted the debugging session (**Nothing**). As soon as a program error has been pinned down to a single program rule, the evaluation is truncated and that program rule is reported to the user.

Going one step further, this monad is transformed by the monad transformer **StateT** in order to let the debugger read and write its state while executing a program.

```
type Step a = StepMode -> DebugMonad a
```

Step a is the type of an evaluation step with result type **a**. Since the evaluation of an expression depends on its evaluation mode, an evaluation step consists of an expression of type **DebugMonad a** that is parameterized with an evaluation mode of type **StepMode**.

The function **eval** consumes one entry from the current oracle. Depending on the value of this entry, it either evaluates its argument and returns the result of this evaluation, or it refrains from evaluating its argument and returns **underscore** as a placeholder.

```
eval :: ShowTerm a => Step a -> Step a
eval a mode = do state <- get
                let (orc, needed) = popBoolStack (oracle state)
                    put (state {oracle = orc})
```

```

        if needed then a mode
        else return underscore

```

The following pair of functions is used to combine evaluation steps:

```

(>>>=) :: ShowTerm a => (Step a) -> (a -> Step b) -> Step b
a >>>= b = \ mode -> do a' <- eval a mode
                    b a' mode

```

```

return' :: ShowTerm a => a -> Step a
return' x = \ _ -> return x

```

At first, function (`>>>=`) calls `eval` with parameter `a` as its argument. Depending on the current entry of the oracle, it either evaluates this parameter and returns the result, or it returns a placeholder value `underscore`. Then the second argument is applied to the value returned by `eval`. Since this application is done by function (`>>=`) of type `DebugMonad`, managing the debugger state by `StateT` as well handling exceptions by `ErrorT` is done in background by the monad `DebugMonad`.

The function `return'` turns its argument into an evaluation step which returns this argument as a result when evaluated.

Assuming that all entries of the current oracle have value `True`, type `Step a`, together with functions (`>>>=`) and `return'`, form a monad.

For example, the types of the transformed versions of the functions in Figure 1 are:

```

length :: ShowTerm a => [a] -> Step Int
exp :: Step Int
fiblist :: Int -> Step [Int]
fib :: Int -> Step Int

```

In the original program `exp` is a value of type `Int`, but in the transformed program every evaluation takes place in the debug monad. The resulting function is an evaluation step which may return a value of type `Int` or a bug report. Similarly, the transformed version of `length` is still a function taking two arguments, but now it yields an evaluation step that has to be executed in the debug monad to retrieve its result.

The transformation also adds a function `main`, an action of type `IO ()`. This action contains the whole debugging session from loading the oracle from disk to debugging the program interactively. The details of this transformation will be explained in Section 3.6.

3.5 The Function `traceFunCall`

The function `traceFunCall` interfaces the instrumented program with the interactive debugger. Every top level declaration is augmented with a call to this function, which has the following type signature:

```
traceFunCall :: ShowTerm a => DebugMonad Term -> Step a -> Step a
```

The first argument contains an action to retrieve the term representation of the function call about to be evaluated, cf. Section 3.3. The second argument is the function body that has been lifted into the debugging monad as described in the previous section. The class context `ShowTerm a =>` makes sure that the result can be displayed to the user. There is a third argument hidden in the resulting type `Step a` as given in the type declaration above: the current evaluation mode of the debugger. Depending on the evaluation mode, `traceFunCall` shows one of the following behaviors:

Mode StepCorrect If a function call is rated as correct, all its sub computations are considered as correct, too, so that the components `skipped` and `unrated` of the global state do not contain entries for those calls. Therefore, `skipped` and `unrated` will not be changed while evaluation a sub expression with mode `Stepcorrect`.

Mode StepBackground The resulting value is calculated without user interaction. Since `unrated` might contain ratings for sub computations and those ratings shall be preserved, with mode `StepBackground` every call of `traceFunction` takes one rating from `unrated` and puts it onto `skipped`. Therefore at the end of a debugging cycle, the entries of `skipped` can be put back onto `unrated`, so that in the next debugging cycle those ratings will be reused and there is no need to re-state the correctness of the function calls that have already been rated.

There is one exception from this rule: If the entry that was taken from `unrated` is `True`, the current function call has been rated as correct in a former evaluation, and this function call is evaluated with mode `StepCorrect`.

Mode StepInteractive All interaction between the user and the debugger takes place with mode `StepInteractive`. In inspection mode, cf. Section 2.1, function `traceFunCall` first evaluates the function body with mode `StepBackground`. Then the function call is – preliminary – rated as correct by appending value `False` to the component `skipped` of the debugger state. After that the resulting value is displayed and the user is asked whether it is correct.

- Since for every function call that has been rated as as correct all its subfunction calls are considered as correct, too, their rating as well as the preliminary rating of the current function call are removed from `skipped` and from `unrated`. They are replaced by a single entry of value `True` that states the correctness of the whole subexpression.
- If the user has rated the resulting value as wrong, the debugging session will confine itself to searching the bug in the current expression. If it finds a bug in one of them, then in turn it restricts itself to searching the bug in that subexpression. If it finds no bug in any of its subexpressions, it is clear that the definition of the currently called function is erroneous, and the current function call will be returned as the result of the debugging session.

- If rating the current function call is skipped, the ratings that had already been present in `unrated` – and have been moved to `skipped` while evaluating the function call with mode `StepBackground` – are kept, so that they will be available in subsequent evaluation cycles.
- If the user moves to evaluating the subexpressions of the current function calls (*step into*), the body of the function currently called is evaluated with mode `StepInteractive`. The rating of the current expression, that has already been appended to `skipped`, will be kept, since it resembles the fact that the current function call is still `unrated`.

If the debugging tool is in step-by-step mode, cf. Section 2.2, `traceFunCall` will not calculate the result of the current function call until the user requests it. Instead it starts by displaying only the function call and giving the user an opportunity to move forward to rating its subexpressions without having to evaluate the whole function first.

We have now explained all the components introduced by the transformation and can now give show how, for example, function `fiblist` of Figure 1 is transformed:

```
fiblist :: Int -> Step [Int]
fiblist x1 = traceFunCall (do sx1 <- showTerm x1
                           return (Term "fiblist" [sx1]))
              (fib x1 >>>= \x4 ->
               x1 + 1 >>>= \x2 ->
               fiblist x2 >>>= \x3 ->
               return' (x4 : x3))
```

3.6 Starting the Debugger

The functions `traceWithStepfile` and `traceProgram` are used to start a debugging session. The function `traceProgram` has the following type signature:

```
traceProgram :: ShowTerm a => Step a -> DebuggerState -> IO ()
```

As parameters it receives both an expression of type `Step a`, which represents the program to be debugged, and the state containing an oracle, which will guide the evaluation of this program.

The program will be repeatedly executed, and the evaluation is stopped when the location of a bug is found, when every function call is rated as correct, or when the user cancels the debugging session.

The function `traceWithStepfile` has the following type signature:

```
traceWithStepfile :: ShowTerm a => String -> Step a -> IO ()
```

It loads an oracle from a file whose name is given in the first parameter, and then it debugs the program given in the second parameter by calling `traceProgram`. The name of the file from which the oracle is loaded consists of the string given in the first argument followed by a suffix `".steps"`.

The expression to be evaluated by the debugger is included in function `main :: IO ()`. In addition to the transformed expression, the resulting declaration of `main` also contains a call to function `traceProgram`. The first parameter, which indicates the name of the oracle file, is derived from the name of the program file being transformed, and the second parameter is the transformed expression of type `Step a`, where `a` is the type of the expression to be evaluated.

For example, the function `main` added to the program in Figure 1 in order to evaluate the expression `exp` is:

```
main :: IO ()
main = traceWithStepfile "Example"
      (return (traceFunCall "main" []))
      exp
```

Since `main` has type `IO ()`, the transformed program can now be compiled by the Haskell compiler `ghc`. Also one can start a debugging session by first loading the transformed program into a Haskell interpreter and then calling `main`.

4 Conclusion and Future Work

We have presented the usage and implementation of a debugging tool utilizing the oracle technique developed in [1]. Up to now, the debugger features a declarative debugging mode as well as a step-by-step mode corresponding to a leftmost innermost evaluation strategy. In addition, a virtual I/O environment gives the user the opportunity to see side effects issued by the program. Up to now, this environment features console output and file access. An extension to other often used I/O actions like `IORefs` and sockets is straight forward. The main limitation of the approach as it is developed by now is the lack of treating run-time errors. Improving this is clearly important for debugging purposes.

Other room for improvement is of course adding to the list of debugging features. Many useful techniques are easy to integrate into the framework like spy points, trusted functions and remembering questions already asked. We plan to include some of the features described in [6] as well as those provided by HAT [2], as far as they fit into the framework.

A thorough benchmarking comparison with HAT [2] is also future work. First impressions are that tracing computations is about five times faster than HAT due to the fact that our approach needs much less space. The size of programs that HAT and our approach can handle in declarative debugging mode seem to be roughly the same. The size of programs manageable in step-by-step mode is only limited in the size of programs that can be traced and we have not yet seen a program that could be successfully executed but not be traced due to memory limitations.

References

1. B. Braßel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, 2007. To be published.
2. O. Chitil, C. Runciman, and M. Wallace. Freja, hat and hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pages 176–193. Springer LNCS 2011, 2001.
3. Andy Gill. Debugging Haskell by observing intermediate datastructures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001.
4. H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150, 1997.
5. 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.
6. Josep Silva. A comparative study of algorithmic debugging strategies. In Germán Puebla, editor, *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2006.
7. 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, 1997.