# Rectus - Locally Eager Haskell Debugger

Oleg Mürk
oleg.myrk@gmail.com

Lennart Kolmodin
kolmodin@dtek.chalmers.se

## ABSTRACT

Lazy programming languages have a very different execution order of expressions which makes debugging such programs a very different experience from debugging eager or imperative programs. We describe a simple idea of how to apply conventional eager language debugging concepts to lazy programming language Haskell. We give a detailed description of how such debugger could be implemented and describe our prototype of this system. Further, we elaborate on how such debugger could be implemented in an efficient way and describe some Haskell runtime extensions that would help this process.

## 1. INTRODUCTION

Haskell is a lazy language and its execution order differs a lot from how eager programming languages execute. As a result it is harder to apply conventional debugging techniques based on tracing statements in the order of their evaluation. The resulting execution order is rather unintelligible and, even worse, the resulting call stack gives information not about where a function was applied, but where the corresponding closure was forced. As a result conventional debugging techniques when applied without modifications to Haskell give incomprehensible information to the programmers whose brains have been irreversibly mutilated by eager if not imperative languages.

For this reason existing Haskell debuggers such as HAT [3], Buddha [5], or HOOD [1] collect exhaustive information about program execution and then provide tools to browse it post-mortem, i.e. after the program has finished. Although useful, such debuggers have their limitations. They don't support concurrency, don't allow adapting program inputs during the debugging session, and work badly with long-running processes because information about whole program execution must be collected. By being more heavy-weight and different from conventional debugging techniques they create rather high adoption barrier. As a result when a programmer is faced with a problem of finding a bug in his program he has the following options: think a little bit more and find the bug without running a program, add some tracing print-outs, evaluate some functions in eval loop, or spend considerable time learning and configuring the tool. As a result the latter almost never happens. This view is supported by highly non-scientific poll among badly-biased selection of Chalmers students none of whom have used any Haskell debugger.

## 2. THE IDEA

Our idea is very simple—although Haskell programs cannot be computed eagerly, because of infinite data structures, finite number of function applications can be computed eagerly without any differences in the result. Moreover, any closures or even arbitrary expressions can be computed out-of-(lazy)-order without any harm for program result. Of course, errors arising from forcing closures (e.g. arguments) out-of-order must be ignored until the closure is really needed. There are some exceptions, though.

If one forces a closure that is an infinite computation that otherwise would not have been forced the program will diverge. The solution is to allow escaping an infinite computation by throwing an error which is semantically equivalent (i.e. $\perp$). Another problem is side-effects. Luckily, `IO` monad can only be executed in unique (sequential) order. Of course, if executing some closure causes execution of `unsafePerformIO` (e.g. `m = putStrLn "side-effect"`) then obviously execution semantics changes—some things are just meant to be lazy after all. However, we don't see it as a serious problem because the programmer has to think in terms of operational semantics anyway.

Further, when evaluating function arguments eagerly a ”normal” call stack is created, which also includes tail calls, although this too can be made only finitely many times. Some form of call stacks arises even when executing in lazy mode. As a complementing tool we can keep (some suffix of) a trail of all events of closures being forced and later returning a value.

Based on this foundation we'd like to create a debugging engine that would mimic conventional debugging as much as possible. First, we need some simple navigation rules that would allow deciding when to apply eager evaluation mode and when to keep closures on the call stack. Now we can allow programmer to set break points when closures are forced (or when they return) and inspect local syntactic bindings and the call stack and the trail. At the break points the programmer could force any new closures or even evaluate whole new expressions. This could be done by providing an eval-loop environment that gives access to local bindings etc. It is important to provide good tools for filtering out relevant events because Haskell programs are usually built with higher-level abstractions whose execution details might be irrelevant for particular debugging session. For instance, when debugging monadic program a programmer may want to ignore how monadic bind is implemented.

The approach that we have described can be implemented as follows:

- Transform debugged program by wrapping an interceptor around each function application.
- The interceptor having access to function application closure but also to the arguments can force eager evaluation using `seq`. Interceptor also implements break point logic and provides an eval loop environment.
- Eval loop environment would execute expressions in IO monad. All debugger commands would be a domain specific language (DSL) embedded into IO monad.

We now proceed to describe these steps in more detail.

## 2.1 Transformation

We propose to instrument a program by wrapping the following syntactic constructions into an interceptor:

- function applications,
- data constructor applications,
- case expressions,
- introducing new bindings into the syntactical scope.

An interceptor gets access to the following information that we call collectively a *frame*:

- the closure itself (we also call it just expression when its not ambiguous),
- the arguments of function or data constructor applications, or argument of case expression,
- local bindings in the current syntactic scope,
- symbol information including the location in the source file, module name, and top-level function name.

For all values accessible in a frame it is essential to be able at run-time to:

- force them,
- inspect their values.

The latter is actually highly non-trivial because of Haskell's type system. We will come back to this issue later.

## 2.2 Interceptor

An interceptor maintains for each Haskell thread the following information:

- current call stack depth ($D$),
- current call stack,
- current call trail (some suffix of it),
- a stack of strategies, explained below,
- a list of break points.

A strategy is a structure with the following values:

- eager limit ($ED$),
- stack limit ($SL$),
- break level ($BL$),

For convenience, these values are actually represented as relative values to the value of stack depth $D$ when strategy was pushed on the stack.

Break point is a function that gets access to all structures maintained by the interceptor (stack, ...), current frame, and current event (e.g. `EnteredFrame`) and returns `IO (Maybe Bool)`. Returning `Nothing` means abstaining from decision, `Just True` means voting for breaking and `Just False` voting against. `Just False` has veto right, i.e. if at least one break point returns `Just False` the program will go on executing. Each break point has an associated level $L$. Only those break points that have level greater or equal to current break level $L \geq BL$ are tested. This allows easily adjusting the level of details that the programmer sees. At break level 0 there is normally one break point that always returns `Just True` which allows stepping through all events.

When program execution is braked at some frame, the debugger asks commands from a user in a loop, compiles and runs them. Commands get the same information as break points and are also run in `IO` monad. Each command can complete with the following results:

- A string value to be shown to the user. The user will have to enter a new command after that.
- `IO` exception after which the user will have to enter a new command too.
- `Continue`—meaning that interceptor execution should continue as normal.
- `Escape`—meaning that interceptor should leave current frame and let evaluate current expression in lazy mode without putting anything on the call stack.
- `Crash msg`—replace current expression with `error msg`. Useful for breaking out of infinite computations.

When interceptor enters a frame (i.e. when the closure is forced) it executes as follows:

1. Increment stack depth $D$, push frame on the stack.
2. Check all break points for event `EnteredFrame`. If break points vote to break, send event information to the user and execute eval loop. If command returns `Escape`, then go to the last step. If command returns `Crash msg` go to last step, but instead of returning the wrapped expression return `error msg`. If command returns `Continue`, then continue.
3. For each frame argument: check if current depth is less or equal than current eager limit $D \leq EL$. If true, force the argument using `seq` function and when it returns check break points for event `ArgReturned i err`, where `i` is argument's index and `err` tells whether the argument failed with an error. After that execute eval loop if needed and act according to command's return value.
4. Check if current depth is less or equal to current stack limit $D \leq SL$. If false, go to the last step without forcing the expression. It will be forced immediately anyway, but the frame will not be recorded on the call stack. Otherwise force the expression with `seq` and when it returns check break points for event `ExprReturned err`. After that execute eval loop if needed and act according to command's return value.
5. Decrement stack depth $D$, pop current frame from the call stack and return the wrapped expression.

Note that each time frame is entered/returns and argument/expression are forced/return, corresponding event is recorded in stack trail.

Interceptor drives execution order by applying `seq` to the closures. For instance, when in eager evaluation mode it first forces frame arguments before forcing the expression itself. `seq` forces a closure only into weak head normal form, however if doing `seq` on an argument reaches a new frame, this new frame can force its arguments again and this gives eager evaluation mode until stack depth reaches eager limit $ED$. Notably, if some closure representing a data structure has been evaluated eagerly until some depth, we cannot apply a new eager evaluation with greater depth because `seq` would just reach a data constructor without executing any interceptors. Of course, we can always find unevaluated nodes of the data structure and force them.

## 2.3 Domain Specific Language

As we have already mentioned, all debugger commands would be implemented as values in `IO` monad to be compiled and executed in eval loop, whereas the following the following information is available to the executed commands through a `let` clause:

- `info` - information about the current event,
- `frame` - current frame,
- `args` - arguments of the frame,
- `expr` - the wrapped expression,
- `bindings` - local bindings; all bindings are also accessible by their real names,

We propose the following set of elementary statements:

- Accessing the values of bindings, arguments, and the expression itself: `force` and `cast`. The latter is needed to read the value and will be explained later.
- Driving execution: `continue`, `escape`, `crash msg`—corresponding to the command return values described above. Further, statement `return str` allows returning values from a command and `ask` is equivalent to `return ""`. These statements should be the last ones in a command.
- Altering strategy: `currentStrategy` allows to see and alter the current strategy and `withStrategy` allows to execute arbitrary `IO` statement within modified copy of current strategy pushed on the stack.
- Breakpoints: `addBreakPoint` and `removeBreakPoint` with obvious semantics.
- Information: `getStack` and `getTrail` that allow reading current call stack and trail.

Of course, for usability, one could (and should) build a much more sophisticated DSL (if not GUI) using these commands as elementary blocks.

## 2.4 The Big Picture

With such debugging engine one could for instance do the following things, which is actually much more than one can do in conventional eager and/or imperative language debuggers:

- Set some break points with level $L = 1$.
- Set break level $BL = 1$ to avoid tracing all events.
- Execute lazily but with stack limit $SL = 1000$.
- On break points look at the trail and "lazy" stack.
- Force/show some local binding, frame argument, the closure itself, or arbitrary expression comprising them within a new strategy with eager limit $EL = 1000$.
- On break points look at "eager" call stack.
- After seeing return value of some closure evaluate expression equivalent to it with break level $BL = 0$ to trace how it was computed.
- When tracing, step over some computations without tracing and break points by forcing them within strategy with $BL = 2$.

## 3. PROTOTYPE

We have implemented a prototype (three man-weeks) of previously described debugging engine using Glasgow Haskell Compiler. Source-to-source transformation is made with `haskell-src-exts`. We don't instrument currently any library code and consequently all library functions are executed as atomic blocks and their execution cannot be traced. Unfortunately, `haskell-src-exts` does not give precise enough information about source locations of elements of abstract syntax tree (AST) and consequently is not suitable for generating symbol information needed for the frames, however it can be easily corrected to do that. In the meantime we just pretty-print expressions and use this as symbol information.

The biggest problem that we have faced is representing arbitrary values (bindings, frame arguments, the expression itself) as a data type that can be kept say in `Map`. The solution that we have chosen is to store them in

```
data AnyValue = forall a. AnyValue a
```

By using `unsafeCoerce` the user can access any value, but alas if he forces values to the wrong type he usually gets a segfault. Alternatively, we could also wrap values in

```
data AnyTypeable =
       forall a. (Typeable a) => AnyTypeable a
```

that allows doing safe casting of values. As not all values are `Typeable` we need somehow to decide which values can be wrapped into `AnyTypeable` and which not. This requires in general inter-modular analysis of programs if, for instance, a polymorphic function in one module is applied to a non-`Typeable` value in another module. However, even expressions like `AnyTypeable 1` give ambiguous type variable error and require giving explicit type annotation `1::Int`. Similar problems arise with `AnyValue show` that require adding a type annotation, e.g. `show::Int->String`. Altogether our current implementation of code instrumentation transformation is partial (only function/data constructor applications are transformed) and doesn't work in many border-line cases outlined above.

The interceptor is a polymorphic function

```
interceptor :: Frame -> a -> a
interceptor frame expr = unsafePerformIO $ do
  ...
  return expr
```

where the second argument is the expression wrapped and eventually it returns this expression. The function itself is implemented with `unsafePerformIO` and interceptor logic works in `IO` monad. The interceptor is completely working, but is quite inefficient. Altogether this probably slows down the program 100x (no real measurements). For compiling user commands we used `hsPlugins` [4]. It seems to fit quite well, although it is a bit slow. However we found an annoying problem with implementing a singleton pattern needed for interceptors to access the global mutable state:

```
globalState = unsafePerformIO (newMVar ...)
```

Although it works perfectly with statically compiled programs, it creates a new instance of `globalState` with dynamically loadable modules even when inlining is turned off. A related problem is that Haskell doesn't have thread-local variables needed to represent interceptor's state per thread (call stack, etc), but this can be emulated with a singleton map from thread IDs to thread info. As a result, when user commands try to access thread state, a new debugger instance is started. To solve this problem, user commands get explicit reference to the thread state. Unfortunately, this also means that commands cannot execute expressions containing instrumented functions, but they can still reference all values explicitly passed to them (bindings, arguments, the closure itself) and, of course, they can use all non-instrumented functions from standard libraries.

Finally, we have developed a simple command-line based user interface to interact with the debugger. In Appendix A we show how to compile the debugger, instrument and compile a program, run it, and then debug it using the debugger client. In Appendix B we give an example of debugging session.

## 4. CONCLUSIONS AND FURTHER DIRECTIONS

A real implementation of a debugger would have to be written quite differently. First, we find that the right solution would be to embed code instrumentation transformation into Haskell compiler or use GHC API. The biggest challenge is to provide safe access to all possible values arising during the execution of the program. We think that the best option is for Haskell runtime to have a Java-like reflection API that would allow to navigate program data graphs and check if a closure was forced already.

Currently, executing interceptor at each frame incurs quite large run-time overhead. Although our implementation is completely working, a much more efficient solution would be to embed its logic into Haskell runtime, which is of course quite much labour. Another useful addition to Haskell runtime would be a possibility to escape from a computation without rising an error so that later the closures would have to be recomputed again. This would be useful for escaping from computations that are not necessary if a user wishes to do that.

Finally, it is essential for a debugger to have an advanced graphical user interface allowing to navigate source code and all the information provided by the debugger engine. Happily, there are good examples from the imperative world.

Although we have tried the debugger on simple programs we find that this debugger concept should be tested on real programs that use specific features of Haskell such as monads and rely heavily on lazy evaluation (e.g. tying the knot, Fudgets [2], etc). Some probable additions needed to conveniently debug such programs would be:

- Marking specific parts of program to be always executed lazily/eagerly.

- Specialized support for monads, e.g. allowing to see state in state monads.

## 5. REFERENCES

[1] The Haskell Object Observation Debugger. Available at `http://www.haskell.org/hood/`.

[2] M. Carlsson and T. Hallgren. Fudgets: a graphical user interface in a lazy functional language. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 321–330, New York, NY, USA, 1993. ACM Press.

[3] O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for Tracing. In *Lecture Notes in Computer Science*, volume Volume 2670, pages 167–181, Jan 2003.

[4] A. Pang, D. Stewart, S. Seefried, and M. M. T. Chakravarty. Plugging haskell in. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 10–21, New York, NY, USA, 2004. ACM Press.

[5] B. Pope. Buddha: A declarative debugger for Haskell, 1998.

## APPENDIX

## A. RUNNING THE PROTOTYPE

Requirements:

- Glasgow Haskell Compiler 6.4+,
- POSIX (excludes GHC on MinGW, but not Cygwin).

Directions:

1. Download and unpack `rectus.tar.gz`.
2. Go to directory `rectus/src/` and run

   ```
   bash build.sh
   ```

   This builds Rectus.
3. Go to directory `rectus/work/programs/` and run

   ```
   bash instrument.sh objects/bugs/Main.hs \
    ../server/objects/bugs/ \
    ../client/objects/bugs/
   ```

   This instruments one program file. Instrumented program files goes to directory `../server/objects/bugs/` and pretty-printed version of input file goes to directory `../client/objects/bugs/`. The latter is useful for setting breakpoints by matching on pretty-printed representation of frame's Haskell code.
4. Go to directory `rectus/work/server` and run

   ```
   bash conf.sh
   ```

   This prepares instrumented program to be debugged. This includes copying needed Rectus server files, compiling them and generating a configuration file `rectus_server.conf`.
5. Run the debugged program

   ```
   ./objects/bugs/a.out
   ```

   The server will block waiting for the client to connect:

   ```
   <...>.Engine: Creating new thread 'ThreadId 1'
   <...>.Core: initializing core
   <...>.Core: waiting for connection...
   ```
6. Open a new terminal in directory `rectus/work/client` and run

   ```
   bash client.sh localhost 1979
   ```

   This connects the client to the server.
7. Currently, server prints a lot of debug output including all occurring events and Haskell source code of executed commands.

Generally, the work of Rectus client consists of:

1. Receiving an event from a server.
2. Asking for user command to execute and sending it to the server.
3. Showing command result after the command completes. Note that generally multiple new events can happen between sending a command and receiving command results.
4. If there is no events from the server and Ctrl-C is pressed, ask user which thread to break and send corresponding command to the server.

When entering user commands the following syntax is used:

- Zero or more lines starting with `import` clause.
- Zero or more lines of the command that will be wrapped into a monadic `do` syntax. No indentation is needed.
- The command ends with first empty line.
- Empty command is equivalent to `rctContinue`.

## B. SAMPLE DEBUG SESSION

We intend to debug the program on Figure 1. It can be also found in `rectus/work/programs/objects/bugs/commands.txt`. Type the following commands into the client and observe a complete debugging session:

1. Evaluate eagerly and keep the stack:

   ```
   rctSetEagerLimit rctThread 1000
   rctSetStackLimit rctThread 1000
   rctAsk
   ```
2. Continue to the next event (run twice):

   ```
   rctContinue
   ```
3. Print the first argument:

   ```
   rctReturn $ show (
      (rctUnsafeCast (head rctArgs))::[Int]
     )
   ```
4. Continue to the next event (run twice):

   ```
   rctContinue
   ```
5. Step over the argument (run twice):

```
main = putStrLn (show $
  msort ([10, 3, 7, 1, 8] :: [Int])
  )

msort :: (Ord a) => [a] -> [a]
msort = merging . map unit
  where unit x = [x]

merging :: (Ord a) => [[a]] -> [a]
merging [xs] = xs
merging xss = merging (pairwise xss)

pairwise :: (Ord a) => [[a]] -> [[a]]
pairwise (xs : ys : xss) =
  xs ‘merge‘ ys : pairwise xss
pairwise xss = xss

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x : xs) (y : ys)
  | x <= y = x : (xs ‘merge‘ ys)
  | otherwise = y : (xs ‘merge‘ ys)
```

**Figure 1: The program to debug.**

```
rctWithStrategy rctThread (do
      -- Turn off tracing
      rctSetBreakLevel rctThread 1
      -- Step over the argument
      rctForce (head rctArgs)
      )
rctEscape
```

6. Print value of local binding xss:

```
rctReturn $ show (
    (rctUnsafeCast (xss))::[[Int]]
  )
```

7. Continue to the next event (run trice):

```
rctContinue
```

8. Print value of local binding ys:

```
rctReturn $ show (
    (rctUnsafeCast (ys)::[Int])
  )
```

9. Add break point:

```
rctAddBreakPoint
  rctThread
  "myBreakPoint"
  1
  (rctSimpleBreakPoint
     ("Main", "merge (xs) (ys)")
     RctEventEnterFrame
  )
rctAsk
```

10. Go on without tracing:

```
rctSetBreakLevel rctThread 1
rctContinue
```

11. Print the call stack:

```
stack <- rctGetStack rctThread
list <- rctStackRead 0 1000 stack
let
  format i = rctSymbolUID
    (rctFrameSymbol (rctInvocationFrame i))
  symbols = Prelude.map format list
rctReturn $ unlines symbols
```

12. Print value of local bindings xs and ys:

```
rctReturn $ show (
    ((rctUnsafeCast (xs))::[Int]),
    (rctUnsafeCast (ys))::[Int]
  )
```

13. Step over the computation:

```
rctWithStrategy rctThread (do
      -- Turn off tracing
      rctSetBreakLevel rctThread 2
      -- Step over the argument
      rctForce (head rctArgs)
      -- Step over the expression
      rctForce (rctExpr)
      )
rctAsk
```

14. Print value of expression result:

```
rctReturn $ show (
   (rctUnsafeCast (xs))::[Int],
   (rctUnsafeCast (ys))::[Int],
   (rctUnsafeCast (rctExpr))::[Int]
 )
```

15. Work some more:

```
rctContinue
```

16. Print the call stack:

```
stack <- rctGetStack rctThread
list <- rctStackRead 0 1000 stack
let
  format i = rctSymbolUID
    (rctFrameSymbol (rctInvocationFrame i))
  symbols = Prelude.map format list
rctReturn $ unlines symbols
```

17. Remove breakpoint:

```
rctRemoveBreakPoint rctThread "myBreakPoint"
rctAsk
```

18. Sleep 15 seconds and then run until the end:

```
import Control.Concurrent
threadDelay 15000
rctContinue
```

19. Within next 15 seconds press Ctrl-C and enter thread ID to break ”ThreadId 1”.

20. Run until the end:

```
rctContinue
```