ART Memo 2
Comparison of Dependency-Based Debuggers for Lazy Functional Languages

Olaf Chitil

June 13, 2000

Contents

1 Methods

The following methods/systems for debugging lazy functional programs are all based on a trace structure which is not just a sequence of expressions refering to the actual computation steps.

Redex Trails [SR97b], [SR97a].

Freja [Nil98], [NS97].

Buddha [Pop98], [NB96].

Possibly interesting, but not discussed here:

- [HO85], [OH88]. Trace similar to EDT but too much is evaluated.
- [TR86], [Toy87]. Trace contains last redex for each value.
- [Kam90]. Trace similar to EDT, but simplified language and implementation.
- [HM93]. Transformed program constructs call path.
- Hood (`http://www.cse.ogi.edu/~andy/hood/`). Doesn't seem to produce a single trace structure. It is a kind of lazy man's `printf`, that is, the function `trace` without changing evaluation order.

2 Logical Structure of Trace

Redex Trails A redex trail of a value is a sequence of redexes which shows the computation history of the value. That is, the first redex is the one that created the value, the next redex is the redex that created the previous redex, etc. A trace consists of the redex trails of all values that were created by a computation, that is, the values of the result and all intermediate values. Furthermore, the trace contains redex trails for expressions that were created but have never been evaluated during the computation.

Because a redex consists of values and sometimes other redexes, the trace structure is complex: The trace is a graph with redexes and values as nodes. A Value may only be the source of an arrow and a redex nearly only the target of (possibly several) arrows. A redex usually contains subnodes which may be starting points of arrows but never targets.

Freja The trace structure is the evaluation dependency tree (EDT).

Buddha The trace structure is the evaluation dependency tree (EDT).

EDT An EDT is a tree where a node is a pair of a redex and a value. It is the derivation/proof tree for a call-by-value reduction with miraculous stops where expressions are not needed for the result. The derivation tree only contains function application, not other context rules.

2.1 Does Trace Structure Reflect Program Structure?

Redex Trails Yes. At least the outer part (application in case of a redex) of a sibling of a redex occurs textually in the definition body of the function of the redex.

EDT Yes. The same relationship holds for the redex of a sibling with respect to the redex of its parent.

2.2 Does Trace Contain Mostly Values or Arbitrary Expressions?

Redex Trails A redex contains function names/λ-expressions, values and sometimes unevaluated expressions (see below).

EDT A redex contains function names/λ-expressions, values and a special symbol for unevaluated expressions.

2.3 How is an Expression (Argument) Handled That is Never Evaluated?

Redex Trails It is not evaluated for trace creation to not to change the termination behaviour. It is shown as unevaluated redex. It may be visually represented by a special symbol. It has a redex trail like all other expressions of the trace.

EDT It is not evaluated for trace creation to not to change the termination behaviour. In the trace it is just represented by a special symbol.

2.4 Does Trace Contain Source Code References?

Redex Trails Yes. Every expression is linked to the definition body which created it and additionally every function name is linked to its definition.

Freja Every node is linked to the definition of the function of its redex.

Buddha Every node contains the definition body of the function of its redex.

2.5 Is Sharing Visible?

Redex Trails Yes. The trace is shared just as the arguments in the original computation are. This sharing is visible to the user to indicate which parts of a trace have already been viewed and to locate circular dependencies.

Furthermore, sharing is indirectly visible through how far arguments are evaluated.

EDT Sharing is only indirectly visible through how far arguments are evaluated.

2.6 How is Trace Defined/Explained?

Redex Trails By example and outline of the transformation that enables the creation of the trace.

Freja By example, by informal definition, by an extended continuation-based semantics of the language and the description of the implementation.

Buddha By example, by informal definition and outline of the transformation that enables the creation of the trace. Rather informally discussed correctness of the trace with respect to the original program behaviour.

3 Supported Language Features

3.1 Free Variables in Local Definitions

Redex Trails Are handled like other variables. A redex node does not contain the values of its free variables.

Freja Every EDT node contains the values of its free variables, so that the node represents an equation whose validity can be judged.

Buddha Currently uses λ-lifting to remove local definitions and thus changes program structure. In the future λ-lifting shall be hidden from the user, so that EDT nodes contain the values of free variables as additional information.

3.2 Partial Applications

An application of a function to arguments is a redex if the number of arguments is equal to the arity of the function, which is given through its definition.

Redex Trails Redexes are represented correctly in the trace. To distinguish non-saturated or over-saturated applications from redexes the implementation uses an impure pointer comparison of trails.

Freja Redexes are represented correctly in the trace. To distinguish non-saturated or over-saturated applications from redexes the implementation stores the arity with every function name.

Buddha If a definition body is a partial application, then η-expansion of the definition is performed to remove the partial application. So the arity used for redexes in the EDT may be different from the arity of the function definition.

Partial applications that appear as arguments are handled specially so that they can be shown as argument values. How to do this in general has not yet been solved.

3.3 CAFs

```
ones = 1 : ones

f True = head ones
f False = ones !! 2

main = print (f True)
```

Redex Trails A CAF is a special redex that has an empty redex trail and no link to the source code position where it is used. The description of the transformation does not handle CAFs; it should look as follows:

$$\mathcal{F}[\![f = e]\!] \Rightarrow f = \mathcal{E}[\![e]\!]_{(\texttt{Nm Root } \mathcal{N}[\![f]\!])}$$

Pseudo-CAFs, for example of type `Num n => n`, do not have redex trails either although they are usually not shared.

It would be possible for a CAF to have a redex trail. However, this seems to imply loss of sharing of the CAF.

$$\mathcal{F}[\![f = e]\!] \Rightarrow f\ \texttt{t} = \mathcal{E}[\![e]\!]_{(\texttt{Nm t } \mathcal{N}[\![f]\!])}$$

Freja Every CAF has its own EDT. So a program has a forest of EDTs which are ordered according to dependency (as far as possible).

Buddha Not described. It seems that no EDT is produced for a CAF and its evaluation history is lost.

3.4 Guards

```
f x | x < 0 = -1
f x | x > 0 && x < 10 = 1
    | x > 10 = 2
f x = 0

main = print (f 0)
```

Redex Trails A Function definition with guards leads to a special, extended redex. The redex has an additional list of truth values for the values of all the guards that were needed to reduce the redex. This is not discussed in any paper.

EDT There is no special node for guards.

Freja Guards are not discussed.

Buddha The EDTs of all guards and redexes that were evaluated to evaluate a guard are direct children of the redex whose definition contains the guards. This is only so for a single equation of a function definition. If all guards fail computation continues with a subsequent equation, then all EDTs of the guards are lost (probably easier to implement; hence Buddha forbids overlapping left hand sides).

3.5 List Comprehensions

Redex Trails Not yet dealt with. Shown in terms of the functions that internally implement them.

Freja A list comprehension is a redex. A simple, intuitive semantics of list comprehensions, which reduces a comprehension to a similar one, is used for the construction of the EDT. Not yet implemented.

Buddha Not yet dealt with.

3.6 I/O

```
putLine "" = return ()
putLine (c:cs) = do
  putChar c
  putLine cs

main = do
  xs <- getLine
  putLine xs
```

Redex Trails Handles IO for standard input and output. Leads to some weird redexes as for example `IO □ >> IO (\)`. Only exactly those parts of output have a trace / are selectable that were created by a single output action. It is not possible to select several at the same time. Not described in any paper.

Freja No IO. The function `main` of type `String` is reduced and its normal form printed.

Buddha No IO. Older paper discusses how stream based IO can be handled. It does not suggest to hide any IO details from the user.

4 Design of the Implementation

4.1 How is Trace Created?

Redex Trails A transformer transforms the program into another one which creates a trace besides the original output. The transformation is uniform for expressions of all types. Running the compiled transformed program creates the trace.

Freja By the object code of the compiled program. The low-level implementation promises better performance.

Buddha A transformer transforms the program into another one which creates a trace besides the original output. The transformation is not uniform. Only expressions of functional type are transformed, others are left unchanged. Hence type information is required.

4.2 Parts of Debugger

Redex Trails The system consists of a source-to-source program transformer, which is currently integrated into the nhc compiler, and a browser. When the compiled transformed program stops it connects to the separately running browser. So when the program stops a part written in C takes over to transfer trace information to the browser.

Freja A new compiler produces code that creates the trace. A quite separate navigator/debugger is linked to the object code. When the program stops its usual task the debugger automatically takes over.

Buddha The system consists of a source-to-source program transformer and a Haskell library which implements the debugger. When the program stops its usual task the debugger automatically takes over.

4.3 How is Trace Represented?

Redex Trails As Haskell data structure. All data type definitions of the original program are transformed. The data structures of the program and the trace structure are interwoven.

Freja As C data structure with links to the data structures of the original program. The normal heap nodes also contain additional information.

Buddha As Haskell data structure, separate from the data structures of the original program.

4.4 How is Printable Form Obtained?

Redex Trails A trace node contains a string and source references for that purpose.

Freja An EDT node contains a string and source references for that purpose.

Buddha The primitive function `dirt` transforms any value into printable form. Partial applications as arguments carry additional information in a string.

4.5 How is Assured That No Additional Evaluation Occurs?

Redex Trails The C part which connects to the browser can distinguish unevaluated from evaluated heap objects.

Freja The interface to the debugger permits distinguishing unevaluated from evaluated heap objects.

Buddha `dirt` transforms a value into printable form without evaluating it further.

5 How is Trace Used for Locating Bugs?

Redex Trails By following redex trails backwards from erroneous values. No general rule locating bugs.

EDT By algorithmic debugging, just answering yes/no questions.

5.1 Possible to Investigate Specific Function Definitions?

Redex Trails By making uninteresting functions trusted, only redexes for others are shown. However, its not possible to make functions trusted or untrusted in the browser for focusing. Not possible to find occurrences of specific function within a trace.

Freja An EDT may be created that starts with the specific function definition, skipping all computation needed to reach the function. Functions can be made trusted or untrusted in the navigator.

Buddha No.

6 Kinds of Errors for which Debugger is suitable

6.1 Termination with Wrong Result

Redex Trails Yes. Debugging consists of following the trace of a wrong result.

Freja Yes. Answer to first question will be “no”.

Buddha Yes. Answer to first question will be “no”.

6.2 Termination with an Error?

Redex Trails Yes. If an error occurs, then process connects to browser. Error is starting point of the trace..

Freja Yes. If an error occurs, then debugger takes over. An error has value $\perp$.

Buddha No. Non-erroneous termination is required.

6.3 Non-Termination (How Interrupt in Consistent State?)

Redex Trails Yes. The user can interrupt a non-terminating program. Then the process connects to the browser. The redex which was under evaluation at the time of interruption is the starting point of the trace.

Papers do not discuss how interruption is handled.

Freja Yes. The user can interrupt a non-terminating program. Then the debugger takes over. The result value of all redexes which were under evaluation at the time of interruption is $\perp$.

Test for interruption every time new heap space is claimed (problem if non-terminating program does not require heap).

Buddha No. Non-erroneous termination is required.

6.4 Black Hole

Redex Trails Yes. Similar to non-termination except that the program stops by itself.

Freja Yes. Similar to non-termination except that the program stops by itself.

Buddha No. Non-erroneous termination is required.

7 Practical Side of Debugger

7.1 How is Space Problem Attacked?

Redex Trails First, a function can be trusted so that no redex created by its definition body for a trusted function appears in the trace. Second, all redex trails can be limited to a given maximal size. At garbage collection time all redexes that are longer are pruned.

Freja First, a function can be trusted so that no EDT node appears for calls to it (for recursive functions the first call appears in the EDT). Second, the EDT may only be produced up to a given depth. When the cut-off part of the EDT is needed during debugging, then the program is reexecuted to produce another part of the EDT.

Buddha Mentions trusting of functions and production of upper part of EDT with reexecution. Authors believe that lazy evaluation of EDT will save space.

7.2 Incremental Archiving Feasible?

Redex Trails Colin says so.

Freja Very difficult, because EDT grows at so many edges.

Buddha . Not Discussed.

7.3 How is Run-Time Problem Attacked?

Redex Trails Programs run 6-36 times slower (without consideration of garbage collection). For improvement key auxiliary functions may be moved into the run-time system.

Freja Low level implementation was chosen to gain good run-time results. In the worst case a program ran 17 times slower, including garbage collection time, which by far took the major part.

Buddha Time is not discussed.

7.4 State of Implementation

Redex Trails Part of nhc. Interactive graphical browser. Covers most of Haskell 98.

Freja Executable for Solaris available. Lacks especially most of class system, `do`-notation, monadic IO and standard libraries.

Buddha Does not seems to be available. Supports the subset of Haskell that resembles Miranda.

8 Others

Buddha Argues that its debugging method is sound and complete.

References

[HM93] Jonathan E. Hazan and Richard G. Morgan. The location of errors in functional programs. In Peter Fritzon, editor, *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93*, LNCS 749, pages 135–152. Springer, 3–5 May 1993.

[HO85] C. V. Hall and J. T. O'Donnell. Debugging in a side effect free programming environment. *Proc. ACM Symposium on Language Issues in Programming Environments, SIGPLAN Notices*, 20(7):60–8, July 1985.

[Kam90] Samuel Kamin. A debugging environment for functional programming in centaur. Technical report, INRIA Sophia-Antipolis, Route des Lucioles, 06560 Valbonne, France, July 1990.

[NB96] Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. In *Proc. 19th Australasian Computer Science Conference*, January 1996.

[Nil98] H. Nilsson. *Declarative debugging for lazy functional languages.* PhD thesis, Linköping, Sweden, 1998.

[NS97] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering: An International Journal*, 4(2):121–150, April 1997.

[OH88] John T O'Donnell and Cordelia V Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1:113–145, 1988.

[Pop98] Bernard Pope. Buddha: a declarative debugger for haskell. Technical report, Dept. of Computer Science, University of Melbourne, Australia, June 1998. Honours Thesis.

[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 Intl. Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS Vol. 1467, September 1997.

[SR97b] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS Vol. 1292, September 1997.

[Toy87] Ian Toyn. Exploratory environments for functional programming. Report YCST 87/02, Department of Computer Science, University of York, York, UK, 1987. PhD Thesis.

[TR86] Ian Toyn and Colin Runciman. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing*, 4 A02(4):339–63, 1986.