ART Memo 2 (Version 1)
Comparison of Dependency-Based Debuggers for Lazy Functional Languages

Olaf Chitil

June 23, 2000

Contents

1 Methods **2**

2 Logical Structure of Trace **2**
2.1 Does Trace Structure Reflect Program Structure? . . . 3
2.2 Does Trace Contain Mostly Values or Arbitrary Expressions? . . . 3
2.3 How is an Expression (Argument) Handled That is Never Evaluated? . . . 4
2.4 Does Trace Contain Source Code References? . . . 5
2.5 Is Sharing Visible? . . . 5
2.6 How is Trace Defined/Explained? . . . 5

3 Supported Language Features **5**
3.1 Free Variables in Local Definitions . . . 6
3.2 Partial Applications . . . 6
3.3 CAFs . . . 6
3.4 Guards . . . 8
3.5 List Comprehensions . . . 8
3.6 I/O . . . 9

4 Design of the Implementation **9**
4.1 How is Trace Created? . . . 9
4.2 Parts of Debugger . . . 10
4.3 How is Trace Represented? . . . 10
4.4 How is Printable Form Obtained? . . . 10
4.5 How is Assured That No Additional Evaluation Occurs? . . . 11

5 Intended/Possible Use of Trace **11**
5.1 How is Trace Used for Locating Bugs? . . . 11
5.2 Possible to Investigate Specific Function Definitions? . . . 11

6 Kinds of Errors for which Debugger is suitable **12**
6.1 Termination with Wrong Result . . . 12
6.2 Termination with an Error? . . . 12
6.3 Non-Termination (How Interrupt in Consistent State?) . . . 12
6.4 Black Hole . . . 13

7 Performance **13**
7.1 How is Space Problem Attacked? . . . 13
7.2 Incremental Archiving Feasible? . . . 13
7.3 How is Run-Time Problem Attacked? . . . 13

8 Practical Side of Debugger **14**
8.1 Portability to Different Haskell systems 14
8.2 Which influence have compiler optimisations on trace construction? 14
8.3 Availability 14

9 Specific Advantages and Disadvantages **15**

10 Others **15**

A Debugging with Freja in York **15**

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]. Basically same trace as Freja.

Stories [JB95],[BJ97]. Trace similar to Freja.

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.

Possibly interesting, but not discussed here:

- [HM93]. Transformed program constructs call path, a kind of strict stack trace.

Older work:

- [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.

A comparison of various debugging/tracing methods is made in

- Section 2.2 of [Wat96].
- Chapter 2 (also 7.5) of [Pen99].
- Chapter 11 of [Nil98] (shorter).

2 Logical Structure of Trace

Redex Trails A redex trail is a directed graph with an application at every node. An application is either a redex or a value (constant, constructor application, partial application). The redex trail shows the computation history of an application. A redex is the parent of an application, if it created that application. An application has exactly one parent (except for root redexes), but every of its subexpressions also has a parent which may be distinct. The redex trail of an application is the graph consisting of its parent and the parents of its subexpressions and recursively the redex trails of these parents and the subexpressions of these parents. Although the sequence of parents can be seen as the main redex trail of an application, the whole trail is a complex branching structure.

The trace of a program run is the redex trail of its result. This trace contains the redex trails of all intermediate values and also of redexes that were created but have never been evaluated during the computation.

Because an application consists of values and sometimes redexes, the graph structure of the trace is complex. A redex is the source of a single arrow and the target of (possibly several) arrows. A value or a subexpressions of an application is the source of a single arrow but never the target of an arrow. Note that a redex is a subexpression of an application only if this redex was never evaluated. Hence it cannot be the target of an arrow.

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.

Stories A story is a tree structure which describes the computational history of a value. A story is either a value (full normal form) or an application node with three children: the story of the function, the story of the argument and the story of the result.

The story of a value is the same value, if this value appears literally in the program text. The story of a value is an application node, if this application was reduced to compute the value. Let us define the value of a story to be just the story, if the story is a value, and the value of the result story, if the story is an application node. Then the value of the story of a value is this very value.

A story has a top-down structure like the EDT, but it differs in that a story is always attached to a value, not an application node. In this respect it resembles the redex trail.

HOOD The trace describes data structures created by the program. That is, the creation of data constructors (and primitive value) is observed. The trace is a set of observations. An observation states *where* in the computation term/graph (with respect to an occurrence of an `observe` combinator) reduction to *which* data constructor (or primitive value) took place.

Figure 1 shows the various trace structures for an example program.

2.1 Does Trace Structure Reflect Program Structure?

Redex Trails Yes. A parent redex is always the left hand side of an instance of a function definition. Its children are parts of the (partially evaluated) right hand side of this instance. At least the outer part (application in case of a redex) of a child occurs textually in the right hand side of the function definition.

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

Stories Yes. Similar to EDT, but whereas the children of an EDT node are unordered, a story orders them reflecting the subexpression structure of the right hand side of a definition.

HOOD Not really. Every observation belongs to an occurrence of the `observe` combinator, that was inserted in the program. However, the relationship between the observations belonging to a single occurrence is independent of the program structure (they are related according to the data structure that they form).

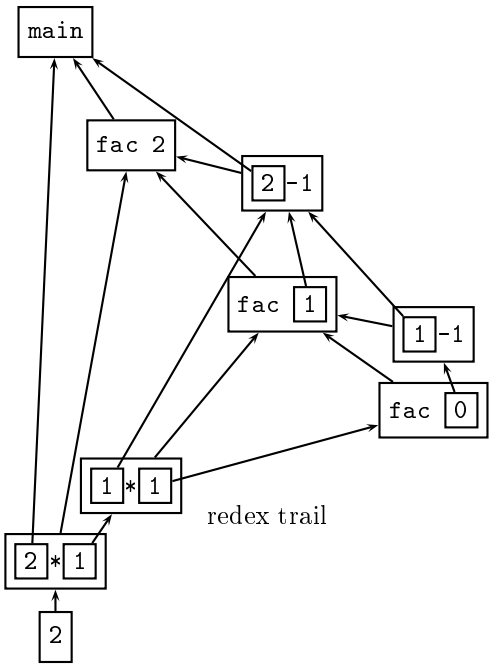
2.2 Does Trace Contain Mostly Values or Arbitrary Expressions?

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

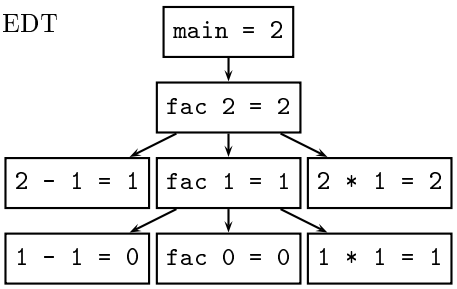
EDT A redex contains function names/λ-expressions, values and a special symbol for unevaluated expressions.

Stories Only values, that is, the function, argument and result of an application node are values. In case of non-termination the story may not be constructed completely.

```
fac 0 = 1
fac n = n * fac (n-1)

main = fac 2
```



EDT



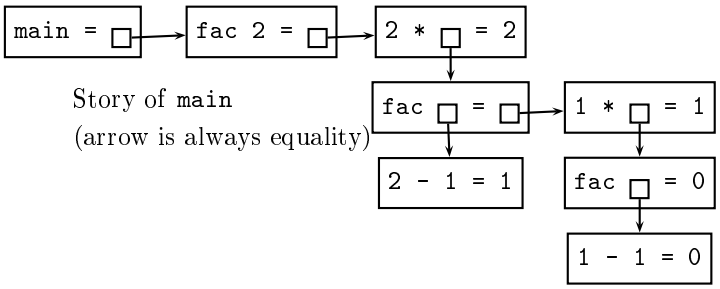


Figure 1: Traces for an example program

HOOD The trace contains only (partial) values.

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

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

EDT It is not evaluated for trace creation. So the termination behaviour does not change. In the trace the unevaluated expression is just represented by a special symbol.

Stories It is evaluted for trace creation! Creating the story only up to a given depth prevents non-termination, but not black holes.

HOOD It is not evaluated for trace creation. So the termination behaviour does not change. Unevaluated expressions correspond to missing observations in the trace.

2.4 Does Trace Contain Source Code References?

Redex Trails Yes. Every expression is linked to the part of a 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.

Stories Yes. Every node contains a source code reference (to the part of a definition body that created it?)

HOOD Observations refer to the name (string) that the user gives as first argument to the `observe` combinator.

2.5 Is Sharing Visible?

Redex Trails Yes. The trace is shared just as the arguments in the original computation are. This sharing is made visible to the user through a colour coding (identical, parent, siblings) to indicate which parts of a trace have already been viewed and to locate circular dependencies.

Furthermore, sharing also influences how far arguments are evaluated.

EDT Sharing is only indirectly visible through how far arguments are evaluated. The values of free variables of the right hand side of a local function definition are shown as part of a redex. However, this is not intended to point out that these values are shared.

Stories I believe not at all.

HOOD In so far that every evaluation of the `observe` combinator leads to a separate set of observations. Hence sharing of this evaluation pays a role.

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.

Stories By example, by informal explanation and by outline of the transformation that enables the creation of the trace.

HOOD Structure of trace only outlined in textual description.

3 Supported Language Features

HOOD Language features are irrelevant for the trace (except for IO, see later), because not reductions but data are traced.

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. This lack of information may hinder understanding of the traced computation (Problem that needs attention).

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.

Stories Not discussed.

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.

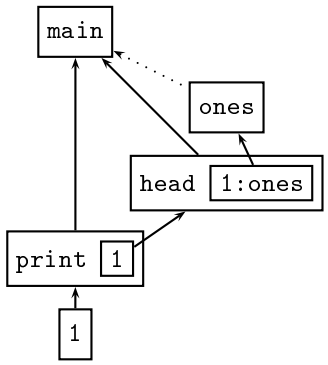
Buddha If a definition body is a partial application, then η-expansion of the definition is performed to remove the partial application. So the arity used for redexes in the EDT may be different from the arity of the function definition.

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.

Stories A redex is the application of a function to a single argument, that is, application to several arguments is broken down into several applications and represented like this in the story. The value of a partial application is always shown as a constant `Partial`.

3.3 CAFs

```
ones = 1 : ones

main = print (head ones)
```



Redex Trails A CAF is a special redex that has no parent, because its value may be required by several redexes, but it is only evaluated once.

Hence a CAF has an empty redex trail and no link to the source code position where it is used. In the preceding example `main` and `ones` are CAFs. The indicated parent arrow from `ones` to `main` does *not* exist.

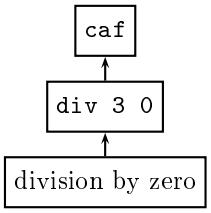
The description of the transformation does not handle CAFs; it should look as follows:

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

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

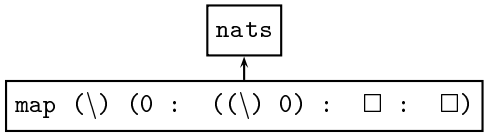
It could be useful for a CAF to have a parent, if evaluation of a CAF fails. Although the value of the CAF lead to the error, it may be important to know where the CAF was called, because this evaluation of the CAF may not have been intended. However, because the error (or in case of non-termination the redex under evaluation) is the starting point of the redex trail of the program, the redex that called the CAF is not part of the redex trail. The following example demonstrates the lack of information:

```
caf :: (Int,Int)
caf = (3 `div` 0, 3 `div` 0)

main = print (snd caf)
```



A more realistic example. The infinite computation is interrupted by the user:

```
nats :: [Int]
nats = 0 : map (+1) nats

main = print (last nats)
```



The redex trail would be more interesting, if the user had interrupted the program while a recursive call of `last nats` was evaluated.

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

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

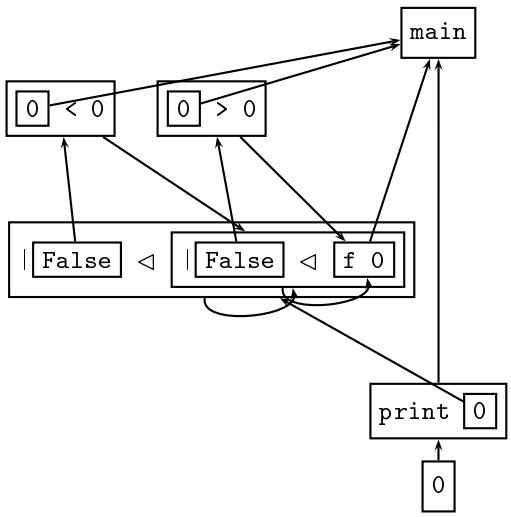
Freja Every CAF has its own EDT. So a program has a forest of EDTs which are ordered according to dependency (as far as possible).

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

Stories Not discussed.

3.4 Guards

```
f x | x < 0 = -1
f x | x > 0 = 1
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, each with its own redex trail, for the values of all the guards that were needed to reduce the redex. See preceeding example. This is not discussed in any paper.

EDT There is no special node for guards.

Freja Guards are not discussed. Expect guard derivations as children of the redex whose definition contains the guards.

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).

Stories Not discussed

3.5 List Comprehensions

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

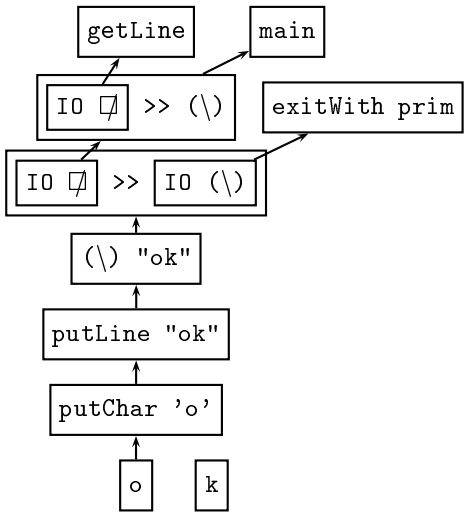
Freja A list comprehension is a redex. A simple, intuitive semantics of list comprehensions, which reduces a comprehension to a similar one, is used for the construction of the EDT. Not yet implemented.

Buddha Not yet dealt with.

Stories Not part of the language.

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 (\)` (see preceding) partial redex trail). The output is divided into fixed parts. Each part was created by a single output action and has this action as parent. It is not possible to select several parts 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.

Stories Not part of the language.

HOOD Also values of type IO (actions) can be observed. Only the return value of an action is shown, not the action itself.

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.

Stories A transformer transforms the program into another one which creates a trace besides the original output. The transformation is purely syntax-directed.

HOOD A programmer annotates his program with the combinator `observe`, which is defined in a library that has to be imported. When the program is run, the occurrences of the combinator create the trace by writing to a file as a side-effect. The combinator `observe` does not influence the program semantics in any other way.

4.2 Parts of Debugger

Redex Trails The system consists of three pieces: a source-to-source program transformer, which is currently integrated into the nhc compiler, an extension of the runtime system and a browser. When the compiled transformed program stops, the extension of the runtime system takes over and connects to the separately running browser, transfering trace information to it.

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.

Stories The system consists of a soure-to-source program transformer and a browser for walking through a story. Walking through the story with the browser forces the execution of the respective parts of the program. Unclear how debugger and language execution (Gofer interpreter?) is implemented.

HOOD The system consists of the library with combinators such as `observe` and a separate browser that reads the trace file.

4.3 How is Trace Represented?

Redex Trails As Haskell data structure with links to C tables for information about names and source references. All data type definitions of the original program are transformed. The data structures of the program are interwoven with links to the trace structure. There are no links from the trace to data structures of the program.

Freja As C data structure with links to the data structures of the original program. (opposite direction with respect to redex trails!). The normal heap nodes also contain additional information.

Buddha As Haskell data structure, separate from the data structures of the original program. However, it is separate only at the end when trace has been built. At runtime the trace has links to the expressions of the original program that are under evaluation.

Stories Seems to be represented by a Gofer data structure. It is separate from the data structures of the original program. However, it is separate only when full trace has been built. At runtime the trace has links to the expressions of the original program that are under evaluation.

HOOD As an XML file.

4.4 How is Printable Form Obtained?

Redex Trails Some trace nodes contain a string and a source reference. The printable form of a node is constructed from the strings of several nodes.

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.

Stories A value node of a story contains a value which is printable. The printable form of an application node is constructed from the printable form of its three stories (value of these stories).

HOOD An observation contains the observed constructor as 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.

Stories Additional evaluation does occur, because only values are printed.

HOOD Evaluation triggers creation of the trace. Expressions that are not evaluated do not contribute to the trace.

5 Intended/Possible Use of Trace

Redex Trails Main application of the trace and only one discussed in papers is locating bugs. Also intended for understanding why/how a correct program works.

Freja Only debugging is stated as intended use of the system.

Buddha Only debugging is stated as intended use of the system.

Stories Only debugging is stated as intended use of the system.

HOOD Only debugging is stated as intended use of the system.

Use of systems for teaching functional programming is not mentioned anywhere.

5.1 How is Trace Used for Locating Bugs?

Redex Trails By following redex trails backwards from erroneous values. No general rule for locating bugs (Memo needed!).

EDT Either by algorithmic debugging, answering yes/no questions, or by stepping through the strictified computation like in debugging of imperative languages.

Stories According to the authors by both "algorithmic debugging and faulty function investigation". The examples resemble stepping through the strictified computation like in debugging of imperative languages.

HOOD Trace enables programmer to locate differences between what the computer has been told to do and what the programmer thinks the computer should be doing. No general rule for locating bugs.

5.2 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 first application of a specific function, skipping all computation needed to reach this application. Functions can be made trusted or untrusted in the navigator.

Buddha No.

Stories No.

HOOD Not specific function definitions, but data at specific program locations. Likewise a function can be observed at a given location.

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".

Stories Yes. Debugging will follow result of `main`.

HOOD Yes.

6.2 Termination with an Error?

Redex Trails Yes. If an error occurs, then process connects to browser. An error has value $\perp$. The error is the 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.

Stories No. Non-erroneous termination is required.

HOOD Applicable but not necessarily helpful. The trace does not show where error occurred. Programmer must observe the right locations.

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.

Stories Yes. Debugger forces evaluation and creates story only up to given depth.

HOOD Applicable but not necessarily helpful. The trace does not show where the infinite recursion occurred. Especially difficult if the infinite recursion does not produce any data. Programmer must observe the right locations.

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.

Stories No.

HOOD Same as non-termination.

7 Performance

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 (either to given depth or to global heap size limit). 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.

Stories Not discussed.

HOOD Only data at programmer selected locations are observed. Because trace is written in file, the increase in required heap space should be modest (no measurements).

7.2 Incremental Archiving Feasible?

Redex Trails Yes, see Memo 3.

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

Buddha Not discussed.

Stories . Not discussed.

HOOD Yes, always done. XML trace file may be compressed by a standard tool.

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.

Stories Time is not discussed.

HOOD Time is not discussed. However, if data at only a few program locations is observed, then additional cost should be limited.

8 Practical Side of Debugger

8.1 Portability to Different Haskell systems

Redex Trails Although the transformational approach was chosen to acquire portability, the system is currently not portable. The transformation is performed on the abstract syntax within nhc. The compiler and especially the runtime system have been modified in several parts. The graphical browser is portable, because it is separate and written in Java.

Freja Not portable, because it would require modification of nearly every phase of a Haskell compiler/system. It is a Haskell system of its own.

Buddha Buddha really performs a source-to-source transformation. The declarative debugger is a library written in Haskell which is imported by the program that is to be debugged. These parts are perfectly portable. However, the function `dirt`, which is used by the debugging library, is not portable and may be very difficult to implement for all Haskell compilers, because the required information is not available at runtime.

Stories If debugger is implemented as library, then system should be portable.

HOOD Highly portable (proven in practise), because the library only requires Haskell 98, except for the special function `unsafePerformIO`, which is provided by all Haskell implementations. The graphical browser is portable, because it is separate and written in Java.

8.2 Which influence have compiler optimisations on trace construction?

For all methods the semantics of the program is not changed by compiler optimisations.

Redex Trails Optimisations which duplicate or share expressions change the observed sharing properties. The trace reflects the real sharing of the program, not the sharing of a simple semantics of the original program.

EDT Transformations that change sharing may indirectly show up in how far arguments are evaluated.

Freja Is a comiler of its own with its own optimisations. Every compiler transformation must be implemented so that a mapping between original program text and final abstract machine code is preserved.

Stories Should have no effect.

HOOD Optimisations which duplicate non-value expressions may duplicate occurrences of the `observe` combinator and thus lead to irritating repeated observations.

8.3 Availability

Redex Trails Distributed as 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.

Stories Does not seems to be available. Supports a tiny toy language.

HOOD Available from the haskell.org CVS repository. Covers full Haskell 98 plus extensions.

9 Specific Advantages and Disadvantages

EDT Provides a method for locating bugs (algorithmic debugging). However, a large number of questions may have to be answered and they may be difficult to answer because of the size of data structures.

10 Others

Buddha Considers soundness and completeness of a debugging method.

A Debugging with Freja in York

1. You have to login on a Solaris 2 machine (tested on doozer)
2. You must have `/grp/fp/bin/solaris2` in your `Path` (good idea anyway)
3. Compile a Freja program with `ifc -g` *`name`*`.fr`
4. Run the resulting program with *`name`* `-- -d`
5. Use the command `help` to obtain a list of debugger commands

Several example programs are in `/grp/fp/lib/FrejaDist/Examples`.

Stolen from the readme file:

Please be aware of the following:

- The Freja compiler does currently not implement Haskell, only a Haskell subset. In particular:
  - No user-defined type classes.
  - No support for monadic programming (do-notation).
  - No (monadic) I/O. Only output in the form of a list of characters.
- There are many other bugs and shortcomings. Some bugs to watch out for:
  - Top-level pattern bindings does not work.
  - It is not verified that main has type [Char]. If it has some other type, the program will crash when executed.

The compiler is normally invoked via the compiler driver ifc. ifc expects to find Freja source code in files with the extension ".fr". Interface files have the extension ".fi", and Runtime System Dependence files the extension ".rsd". There are no limitations regarding the number of modules that can be put in a file. Nor are there any naming conventions regarding what a file containing a module must be called. Thus the Freja compiler needs to be invoked on all files which are relevant; i.e., if one module depends on another, either its source must be present in one of the ".fr" files given on the command line, or, when doing separate compilation, its interface must be present in one of the ".fi" files given on the command line.

Example. If the current working directory contains a set of ".fr" files containing a set of modules making up a program, the following command could be used to compile it, placing the executable in the file foo:

```
ifc -o foo *.fr
```

In order to compile for debugging, supply the -g flag:

```
ifc -g -o foo *.fr
```

In order to invoke the debugger, the binary has to be invoked in debugging debugging mode:

```
foo -- -d
```

The debugger has a built-in help facility.

Both ifc and fcomp, the compiler proper, have fairly extensive descriptions of their command line options. Use the flag -h.

References

[BJ97] Simon P Booth and Simon B Jones. Walk backwards to happiness — debugging by time travel. Technical Report Technical Report CSM-143, Department of Computer Science and Mathematics, University of Stirling, 1997. This paper was presented at the 3rd International Workshop on Automated Debugging (AADEBUG'97), hosted by the Department of Computer and Information Science, Linkping University, Sweden, May 1997.

[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.

[JB95] Simon B. Jones and Simon P. Booth. Towards a purely functional debugger for functional programs. In *Proceedings Glasgow Workshop on Functional Programming 1995*, Ullapool, Scotland, July 1995.

[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] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 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.

[Pen99] Alastair Penney. *Augmenting Trace-based Functional Debugging.* PhD thesis, Department of Computer Science, University of Bristol, September 1999.

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

[SR97a] Jan Sparud and Colin 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] Jan Sparud and Colin 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.

[Wat96] R. D. Watson. *Tracing Lazy Evaluation by Program Transformation.* PhD thesis, Southern Cross, Australia, October 1996.