Tracing and Debugging of Lazy Functional Programs — A Comparative Evaluation of Three Systems

Olaf Chitil, Colin Runciman and Malcolm Wallace

University of York, UK
{olaf,colin,malcolm}@cs.york.ac.uk

Abstract. In this paper we compare three systems for tracing and debugging Haskell programs: Freja, the Redex Trail System and Hood. We identify the similarities and differences of these systems and we evaluate their usefulness in practice by applying them to a number of small to medium programs in which errors had deliberately been introduced.

1 Introduction

The lack of tools for tracing and debugging has deterred software developers from using functional languages [12]. Conventional debuggers for imperative languages give the user access to otherwise invisible information about a computation by allowing the user to step through the program computation, stop at given points and examine variable contents. This tracing method is unsuitable for lazy functional languages, because their evaluation order is complex, function arguments are usually unwieldy large unevaluated expressions and generally computation details do not match the user's high-level view of functions mapping values to values.

In the middle of the 1980's research into tracing methods for lazy functional languages started and has been increasing since. Today there exist three systems for practical use, that is, they are publically available and they cover at least a large subset of a standard lazy functional language, namely Haskell 98 [8]. Freja[1] [6, 5] is a system that creates an evaluation dependency tree as trace, a structure based on the idea of declarative debugging from the logic programming community. The Redex Trail System[2] [11, 10] creates a trace that shows the relationships between the redexes (mostly function applications) reduced by the computation. The most recent system, Hood[3] [2], permits to observe the data structures at given program points. It can basically be used like `print` statements in imperative languages, but the lazy evaluation order is not affected and functions can be observed as well.

[1] `http://www.ida.liu.se/~henni`
[2] `http://www.cs.york.ac.uk/fp/ART`
[3] `http://www.haskell.org/hood`

In this paper we compare the three systems Freja, the Redex Trail Tracer and Hood, we identify their similarities and differences and we evaluate the usefulness of the systems in practice. Our aim is to explore the design space of tracers for lazy functional languages and to improve the understanding of tracing to obtain new ideas for how the current systems can be improved or even be combined.

The paper is structured as follows. In the next section we give a short introduction to each of the three systems. In Section 3 we compare the systems with respect to their approach to tracing, design and implementation. In Section 4 we report on our practical experiments and the insights they gave us into the systems distinguishing properties and their usefulness. We shortly mention other systems for tracing and debugging in Section 5 before concluding with Section 6.

2 Learn Three Systems in Three Minutes

To give an idea about what the three tracing systems provide and how they are used we give a short introduction here. Because all systems are still under rapid development we try to avoid details that may change soon.

2.1 Freja

Freja is a compiler for a subset of Haskell 98. A debugging session consists of the user answering a sequence of questions. Each question concerns a reduction of a redex, that is, a function application, to a value. The user has to answer *yes*, if the reduction is correct with respect to his intentions, and *no* otherwise. In the end the debugger states which reduction is the cause of the observed faulty behaviour, that is, which function definition is incorrect.

The first question always asks if the reduction of the function `main` to the result value of the program is correct. If the question about the reduction of a function application is answered with *no*, then the next question concerns a reduction for evaluating the right-hand-side of the definition of this function. Freja can be used rather similarly to a conventional debugger. The input *no* means "step into current function call" and the input *yes* means "go on to next function call". If the reduction of a function application is incorrect but all reductions for the evaluation of the function's right-hand-side are correct, then the definition of this function must be incorrect for the given arguments.

We demonstrate the use of each system at the hand of the simple program[4] given in Figure 1. The following is a debugging session with Freja. The symbol $\perp$ represents an error and the symbol `?` represents an expression that has never been evaluated and whose value hence cannot have influenced the computation.

[4] Freja actually expects `main` to be of type `String` and the other two systems expect it to be of type `IO ()`. Here we abstract from the details of input/output.

`main` $\Rightarrow (8, \bot)$ *no*
`4*2` $\Rightarrow 8$ *yes*
`head [8,?]` $\Rightarrow 8$ *yes*
`last [8,?]` $\Rightarrow \bot$ *no*
`last [?]` $\Rightarrow \bot$ *no*
`last []` $\Rightarrow \bot$ *yes*
`Bug located! Erroneous reduction: last [?]` $\Rightarrow \bot$

2.2 The Redex Trail System

The Redex Trail System consists of a modified version of the nhc98 Haskell compiler[5] and a separate browser program. A program compiled for tracing executes as usual except that instead of terminating at the end it waits for the Redex Trail Browser to connect to it. The browser shows the output of the program. The user selects a part of it and asks the browser for its parent redex. The parent redex of an expression is the redex that through its own reduction created the expression. Each part of the redex has again a parent redex which the browser shows on demand. A trail ends at the function (redex) `main`, which has no parent. Debugging with the Redex Trail System works by going from a faulty output or error message *backwards* until the error is located.

The Redex Trail Browser has a graphical user interface which we do not discuss here. Basically the system is used as follows to locate the error in the program of Figure 1. The program aborts with an error message and the browser directly shows its parent redex: `last []`. The user is surprised that the function `last` is ever called with an empty list as argument and asks the browser for the parent redex of `last []`. The answer, `last [3+6]`, makes clear that the definition of `last` is not correct for a single element list. The browser can also show where in the program text `last` is called with an empty list in the equation for `last (x:xs)`.

2.3 Hood

Hood currently is simply a Haskell library. A user annotates some expressions in a program with the combinator `observe`, which is defined in the library. While the program is running, information about the values of the annotated expressions is recorded. After program termination the user can view for each annotation the observed values.

We annotate the argument of `last` in our example program:

```
main = let xs = [4*2, 3+6]
         in (head xs, last (observe "last arg" xs))
```

When the modified program terminates it gives us the following information:

```
-- last arg
  _ : _ : []
```

[5] `http://www.cs.york.ac.uk/fp/nhc98`

The symbol _ represents an unevaluated expression. Note that the first element of the list `xs` is evaluated by the program, but not by the function `last`.

To gain more insight into how the program works we observe the function `last`, including all its recursive calls:

```
last = observe "last" last'

last' (x:xs) = last xs
last' [x]    = x
```

The value of the function is shown as a finite mapping of arguments to results:

```
-- last
  {  (_ : _ : [])  -> throw <Exception>
  ,  (_ : [])  -> throw <Exception>
  ,  []  -> throw <Exception>
  }
```

3 Comparison in Principle

At first view the three systems do not seem to have anything in common except the goal of aiding debugging. However, all three systems take a 2-phase approach: while the program is running, information about the computation process is collected. After termination of the program the collected information is viewed in some kind of browser. In Freja, the browser is the part that asks the questions, in the Redex Trail System the program that lets the user view parents and in Hood the part that prints the observations. This approach should not be confused with classical post-mortem debugging where only the final state of the computation can be viewed. Having a trace that describes aspects of a full computation enables new forms of exploring program behavior and locating errors which should make these systems also interesting for strict functional languages or even non-functional languages.

All systems[6] are suitable for programs that show any of the three kinds of possible faulty observable behaviour: wrong output, abortion with error message, non-termination. In the latter case the program can be interrupted and subsequently the trace be viewed.

3.1 Values and Evaluation

All three systems are source-level tracers. They mostly show Haskell-like expressions which are built from functions, data constructors and constants of the program. To improve comprehensibility, all systems show values instead of arbitrary expressions as far as possible. Hood only shows values anyway, marking

[6] Hood requires the non-standard exception library supplied with the Glasgow Haskell compiler to handle programs that abort with an error message or do not terminate.

unevaluated parts by a special symbol `_`. Both Freja and the Redex Trail System show the arguments in redexes not as they were passed in the actual computation but in most evaluated form. Only subexpressions that were never evaluated are shown as unevaluated redexes in the Redex Trail System but not in Freja, which represents them by a special symbol `?`, similar to Hood. None of the systems changes the usual observable behaviour of a program, especially they do not force the evaluation of expressions that are not needed by the program.

However, the systems differ in that Hood shows values as far evaluated as they are *demanded in the context* of the observation position whereas both Freja and the Redex Trail System show how far values are evaluated in the whole computation, including the effect of sharing.

3.2 Trace Structures

In Hood a trace is a set of observations. These observations are shown in full to the user. In contrast, each of Freja and the Redex Trail System create a single large trace structure for a program run. It is impossible to show such a trace in full to the user and hence the browser of each system permits to walk through the structure, always seeing only a small local part of the whole trace.

Freja creates an Evaluation Dependency Tree (EDT) as trace. Each node of the tree is a reduction as shown in the browser. The tree is basically the derivation/proof tree for a call-by-value reduction with miraculous stops where expressions are not needed for the result. The call-by-value structure ensures that the tree structure reflects the program structure and that arguments are maximally evaluated. Figure 2 shows the EDT for our program of Figure 1.

The Redex Trail System creates a Redex Trail as trace. A Redex Trail is a directed graph of value nodes and redex nodes. Each node, except the node for `main`, has an arrow to its parent redex node. Because subexpressions of a redex may have different parents or may be shared, redex nodes may contain arrows to nodes of their subexpressions. Figure 3 shows the Redex Trail for our program of Figure 1. Dotted arrows point to subexpressions. Both dashed and solid arrows denote the parent relationship. Note that browsing the Redex Trail starts with the marked result value.

The graphs of the two trace structures are layouted to stress their similarity. We note that all arrows of the EDT are also present in the Redex Trail but point in the opposite direction. If the Redex Trail held information about which parent relations correspond to reductions (these are shown as solid arrows), then the EDT could be constructed from the Redex Trail. In contrast, the Redex Trail is more complex than the EDT, because it additionally links every value with its parent redex and describes how expressions are shared.

Because Hood observations contain values as they are demanded in a given context whereas both the EDT and the Redex Trail contain values in their most evaluated form, it is not possible to gain Hood observations from either the EDT or the Redex Trail. Conversely, even observing every subexpression of a program with Hood would not enable us to construct an EDT or Redex Trail, because there is no information about the relations between the observations.

```
main = let xs = [4*2, 3+6]
         in (head xs, last xs)

head (x:xs) = x

last (x:xs) = last xs
last [x]    = x
```

Fig. 1. Example program

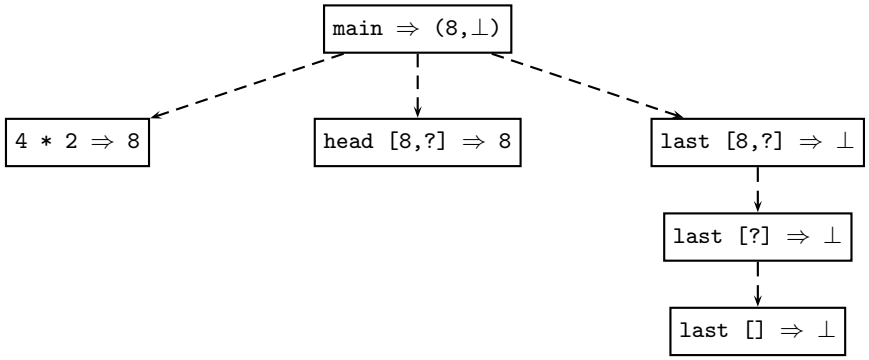

Fig. 2. Evaluation Dependency Tree

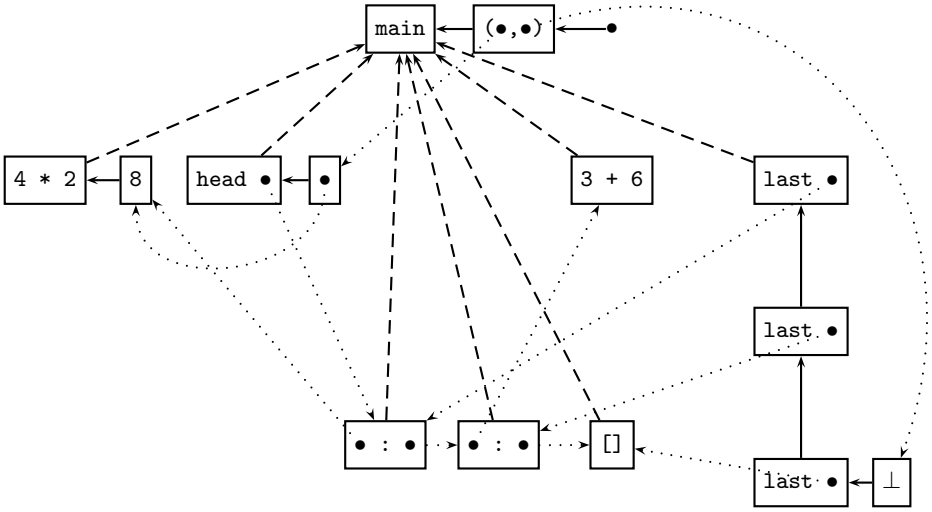

Fig. 3. Redex Trail

3.3 Implementation

Each system consists of two parts, the browser and a part for the generation of the trace. We will discuss the browsers in Section 4.

The developers of the three systems made different choices about the level at which they implemented the creation of the trace. In Freja the trace is created in the heap directly by the modified instructions of the abstract graph reduction machine. The Redex Trail System transforms the original Haskell program into another Haskell program. Running the compiled transformed program yields the trace in addition to the normal result. Finally, in Hood the trace is created as a side effect by the combinator `observe`, which is defined in a Haskell library.

The level of implementation has direct effects on the portability to different Haskell systems. Hood can be used with different Haskell systems, because the library only requires a few non-standard functions such as `unsafePerformIO` which are provided by every Haskell system. The transformation of the Redex Trail System is currently integrated into the nhc98 compiler but could be separated. A transformed program uses a few non-standard unsafe functions to improve performance. Furthermore, some extensions of the Haskell run-time system are required to retain access to the result after termination or interruption and to connect to the browser. Finally, Freja is a Haskell system of its own. Adding its low-level trace creation mechanism to any other Haskell system would require a major rewriting of this system.

3.4 Reduction of Trace Size

In Hood the trace consists only of the observations of annotated expressions. Hence its size can be controlled by the choice of annotations[7]. In contrast, both Freja and the Redex Trail System construct traces of the complete computation in the heap.

To reduce the size of the trace, both Freja and the Redex Trail System enable marking of functions or whole modules as trusted. The reduction of a trusted function itself is recorded in the trace, but not the reductions performed to evaluate the right-hand-side of its definition. The details of the trusting mechanisms of both systems are non-trivial, because the evaluation of untrusted functions which are passed to trusted higher-order functions have to be recorded in the trace. Usually at least the Haskell Prelude is trusted.

To further reduce the space consumption, both Freja and the Redex Trail System support the construction of partial traces. In Freja, first only an upper part of the EDT may be constructed during program execution. When the user reaches the edge of the constructed part of the EDT in the browser, this part is deleted and the whole program is re-executed, this time constructing the part

[7] A variant of Hood allows the annotated running program to write observed events directly to a file, so that the trace does not need to be kept in primary memory. However, to obtain observations, the events in the file need to be sorted. Hence the browser for displaying observations reads the complete file and thus has problems with large observations.

of the EDT that can be reached next by the questions. So, except for the time delay caused by re-execution, the user has the impression that the whole EDT is present.

The Redex Trail System can produce partial Redex Trails by limiting the length of the Redex Trails. Because a Redex Trail is browsed backwards, the system removes those redexes that are further than a certain length away from the live program data or output (pruning). The Redex Trail System does not provide any mechanism like re-execution in Freja to recreate a pruned part of the Redex Trail.

Note that requiring less heap space may reduce garbage collection time, but the Redex Trail System still spends the time for constructing the whole trace whereas Freja does not need to spend time on trace construction after construction of an upper part of an EDT.

4 Evaluation of the Systems

Differences between the systems directly raise several questions. Is it sensible to add a feature of one system to another system? Does an alternative design decision make sense? In how far is a distinguishing feature inherent to a system, possibly determined by its implementation method or its tracing model? Because the design space for a tracer is huge, it is sensible to evaluate system features in practice early. We applied the three systems to a number of programs in which errors had deliberately been introduced. Our experiences from these tests first made us aware of some distinguishing features of the systems which we had not noticed before and then enabled us to evaluate the usefulness of system features.

Our evaluation exercise required at least two programmers. First the author of a correctly working program explains how the program basically works. Then one programmer secretly introduces several deliberate errors into the program, of a kind undetected by the compiler. Given the faulty program, the other programmers use a tracing system to locate and fix all the errors, thinking aloud and taking notes as they do so. We performed this exercise with several programs of 100 to 900 lines. The introduced errors caused all three kinds of faulty observable behaviour mentioned earlier: wrong output, abortion with error message and non-termination.

4.1 Readability of Expressions

In contrast to our preliminary fears that the expressions shown by the browsers, that is, reductions, redexes and values, would be too large to be comprehensible, they were mostly of moderate size and well readable in our experiments.

As we will discuss in Section 4.2 the user of a tracing system does not only view the trace but also the program. We noted however in Freja and the Redex Trail System that informative variable (function) names, that convey the semantics of the variable well, substantially reduced the need for viewing the program and thus increased the speed of the debugging process substantially.

Unevaluated Expressions Freja shows unevaluated expressions as `?` and the undefined value as $\perp$. This property made expressions even shorter and more readable. We made the same observation for Hood. Only in some cases more information would have been desirable for better orientation. In the Redex Trail System the display of the unevaluated redexes sometimes obscured higher level properties, for example the length of a list. All in all our observations suggest that unevaluated expressions should be collapsed into a symbol by default but should be viewable on demand.

Hood shows even less of a value than Freja, because it only shows the part demanded in a given context. Note that this amount of information would suffice for answering the questions of Freja. Because the Redex Trail System is not based on questions, it is less clear if showing only demanded values would be suitable for it. Finally, we note that the fact that Freja and the Redex Trail System show values to the extend to which they are evaluated in the whole computation whereas Hood shows them to the extend to which they are demanded is closely linked to the respective implementations of the systems and thus not easily changeable.

Functions In Haskell, functions are first-class citizens and hence function values may appear for example as arguments in redexes or inside data structures.

For the representation of function values, Hood deviates from the principle of showing Haskell-like expressions. It shows function values as finite mappings from arguments to results. This representation requires some time to get used to. However, it permits are rather abstract, denotational view on program semantics which is useful for determining the correctness of a part of a program. Especially, the representation shows clearly which (part of an) argument is *not* demanded by a function for determining its result. This feature was particularly instructive. Also, because only demanded parts are shown, the representation is short in most cases. However, for functions that are called often and especially for higher-order functions the representation is unwieldy.

In Freja and the Redex Trail System a function value is shown as a function name, a λ-abstraction, or as a partial application of a function name or a λ-abstraction. Function names and their partial applications are well readable but λ-abstractions are not. Both systems do not show a λ-abstraction as it is written in the program but represent it by a new symbol: `<lambda#`n`>` for a number n in Freja and `(\)` in the Redex Trail System. Both systems can show the full λ-abstraction on demand. However, because of the necessary additional step and because λ-abstractions are often large expressions (the main reason why they are represented by symbols) make reading expressions involving λ-abstractions very hard. One of our test programs used named functions where most Haskell programmers would have used λ-abstractions. During tracing, Freja and the Redex Trail System showed very readable expressions for this program.

Free Variables Both λ-abstractions and the definition bodies of locally defined functions often contain free variables. To answer a question of Freja the values of

such free variables must be known. Hence Freja shows this information in a `where` clause. The following question from our evaluation exercise demonstrates that this information usually adds to the comprehensibility of a question considerably:

```
tableRead
  "y"
  (TableImp
    (newTableFunction
     where
       newIndex = "x",
       newEntry = 1,
       oldTableFunction = implTableEmpty))
=>
Just 1
```

The correct answer is obviously *no*.

The Redex Trail System does not show the values of free variables. This information can be obtained only indirectly by following the chain of parent redexes of such a function. To realise that a function has free variables and to see to which arguments of parent redexes they correspond it is furthermore necessary to study the program.

In Hood an observation of a locally defined function can be misleading. The observation is really for a family of different functions, with different values for free variables. In our experiments the observation of a local function `moveval` looked as follows

```
-- moveval
{ ... , 8 -> Draw, ... , 8 -> Win, ... }
```

4.2 Locating an Error

With all three systems we successfully located all errors in our programs. To locate an error the number of questions answered in Freja was always larger than the number of parents looked at in the Redex Trail System which again was larger than the number of times `observe` annotations were added for Hood. For example, for locating an error in our largest program we answered between 10 and 30 questions in Freja, looked at 0 to 6 parents in the Redex Trail System and added `observe` up to 3 times. However, these numbers do not imply that an error was located quickest with Hood.

First of all, the time required in Hood for modifying the program (discussed further in Section 4.4), recompiling the program and reexecuting it is substantially higher than answering a question or viewing a parent. Furthermore, the amount of data produced by a single `observe` annotation is usually substantial. However, the major difference between the systems is the time the user has to spend thinking about what to do next.

Guidance and Strategies Freja asks questions which the user has to answer whereas in both other systems the user also has to ask the right questions. Freja guides the user towards the error.

The Redex Trail System at least starts with the program output, an error message or the last evaluated redex in an interrupted program and the main operation is to choose a subexpression and ask for its parent. There are usually many subexpressions to choose from and the system never states that an error has been located at a given position in the program. Wrong parts in the output or wrong arguments in redexes are candidates for further enquiry. Nonetheless, for the less experienced user it is easy to get lost examining an irrelevant region of the redex trail. Hood gives the complete freedom to observe any value in the program. The initial choice of what to observe is difficult and often seemed arbitrary.

We noticed, however, that Hood users applied a top-down strategy in their placement of `observe` combinators when the faulty behaviour did not point to any program location, for example when the program did not terminate. So the questions the Hood users asked were similar to those asked by Freja. If, on the other hand, the position where the observable fault was caused was identified, for example when the program aborted with an error message occurring only once in the program, then a bottom-up strategy reminding of the Redex Trail System was employed.

Our programs contained several errors. Users of the Redex Trail System and Hood located the errors in the same order, because they always located the error that caused the observed faulty behaviour. In contrast, the questions of Freja sometimes lead to the location of a different error. It would be possible to tackle a specific faulty behaviour by answering some questions incorrectly, but this certainly requires care to not to get directed to an irrelevant region of the EDT.

General Usability The Redex Trail System with its complex browser has the steepest learning curve for a new user. In contrast, the principle of questions and answers of Freja is easy to grasp and Hood has the advantage of using the idea of `print` statements, which are well-known from imperative languages. Hence a mode that would hide some features from the beginner seems desirable for the Redex Trail System.

Information Used A Hood user has to modify the program and hence look at it. Sometimes already searching for a good placement of `observe` reveals the error. Users of Freja and the Redex Trail System, especially the former, tend to neglect the program. As long as the user knows the intended meaning of functions he can use Freja without ever looking at the program. This does however imply that the user does not try to follow Freja's reasoning and to understand how the finally located error actually caused the observed faulty behaviour. Redexes as shown by the Redex Trail System are too terse to be the only source of information for locating an error. Viewing the program part where a redex is created gives valuable context information and at the end the

program is needed to locate the error. Both Freja and the Redex Trail System provide quick access to the part of the program relating to the current question or redex. Nonetheless, it seems worthwhile to test, if automatically showing the relevant part of the program when a new question or parent is shown would improve usability.

In contrast to the other two systems the Redex Trail System also gives information about which expressions are shared. This information was useful in several cases, usually when expressions were shared that were not expected to be so.

A trace of Hood is a set of observations. The trace unfortunately contains no information about the relations between these observations. Hence, with a few exceptions, we observed functions to obtain at least a relation between arguments and result.

Wrong Subexpressions Often, in the questions posed by Freja, a specific subexpression of a result was wrong. For example the last element of a list. There is no way to tell Freja this information. In contrast, the Redex Trail contains the parent of every subexpression. A user of the Redex Trail system seldom asked for the parent of a complete expression but usually for the parent of some subexpression. We believe that this is the major reason why we looked at far less parents with the Redex Trail System than we answered questions of Freja for locating the same error. A Hood user obviously also tries to use information about wrong subexpressions but it is not easy to decide where to place the next `observe` combinator.

Reduction of Information In Hood, the user determines the size of the trace by the placement of `observe` combinators. It is, however, sometimes not easy to foresee how large an observation will be. The trusting mechanism in Freja and the Redex Trail System is not only good for saving space but also for reducing the amount of information presented to the user. The ability of the Freja browser, to dynamically trust a function and thus avoid further questions about it, was useful. For the Redex Trail System a corresponding feature seems desirable. In Freja, sometimes a question was repeated, because the same reduction was performed again. Hence memoisation of questions and their answers is desirable. It would also be useful to be able to generalise an answer, to avoid a series of very similar questions all requiring the same answer.

Runtime With respect to the time overhead caused by the creation of traces the low-level implementation of Freja payed off. The overhead was not noticeable. In contrast, in the Redex Trail System traced computations are more than ten times slower, too slow for large computations. Once we believed a program to be non-terminating but it was only slow. We made the same experience for Hood when we observed at position that were computed very often and that lead to large observations. So in Hood the time overhead is considerable but it is only proportional to the amount of observed data.

Haskell System We noticed in our experiments that a Haskell system can reduce the need for a tracer. Both the Glasgow Haskell compiler[8] and Freja warned about overlapping patterns in one of our programs whereas nhc98 did not. If a function is called with an argument for which no matching equation exists, then the aborting program gives the function name, if it was compiled with the Glasgow Haskell compiler, but not, if it was compiled with Freja or nhc98. However, in that case the Redex Trail Browser at least directly shows the function with its arguments whereas Freja requires the answers to numerous questions before locating the error.

4.3 Language Constructs

A constant applicative form (caf) is a top-level variable of arity zero. Its value is computed on demand and then shared by its users. Both Freja and the Redex Trail System take the view that a caf has no parent. Hence the trace of a program in Freja is generally not a single EDT but a set of EDTs, an EDT for each caf including `main`. These EDTs are sorted so that a caf only uses those cafs about which questions have already been asked and which are hence known to be free of errors. Unfortunately one of our test programs contained 35 cafs. We had to confirm the correctness of evaluation for all cafs before reaching the question about `main`, although none of these cafs were related to any of the errors. Freja permits to directly start with the question about `main`. However, that implies stating that the evaluation of all cafs is correct, which may not be the case and thus lead Freja to give a wrong error location. An alternative definition of the EDT could imply that all users of a caf are its parents. Then a question about a caf would be asked only if it were relevant and memoisation of the question and its answer could avoid asking the same question when another reduction using the caf were investigated.

For the Redex Trail System a corresponding modification seems to be more difficult, because the Redex Trail is browsed by going backwards from an expression to its unique parent. In our experiments the fact that a caf has no parent in a Redex Trail was not noticeable, because none of the introduced errors concerned cafs. However, programs can be constructed were this lack of information would hinder locating an error.

In Haskell the selection of an equation of a definition may not only be determined by pattern matching but may also depend on the value of a guard:

```
test :: (a -> Bool) -> a -> Maybe a

test p x | p x       = Just x
         | otherwise = Nothing
```

In Freja the reduction of a guard (`p x`) is a child of the reduction of the function (`test`). Redex Trails are, however, traversed backwards from the result value (`Just` x or `Nothing`). To hold the information about the reduction of a guard,

[8] `http://www.haskell.org/ghc`

Redex Trails have an additional sort of redexes. In the example, if the first equation were chosen, then the value `Just` x would have the parent `| True` $\lhd$ `test` p x, and if the second equation were chosen, then the value `Nothing` would have the parent `| True` $\lhd$ `| False` $\lhd$ `test` p x. By asking for the parents of the truth values `True` and `False` in the redexes, the user can obtain information about the evaluation of the guards. On the one hand, this special redex complicates the system. On the other hand, it enables more fine grained tracing up to the level of a guard, whereas Freja only identifies a whole reduction as faulty. This feature is useful when a pattern is associated with many guards.

4.4 Modification of the Program

Whereas Freja and the Redex Trail System are applied to the original program, requiring only special compilation, Hood is based on modifying the program. In practice these modifications are not neglectable. Already the introduction of the `observe` combinator requires modifications which are non-trivial, if an operator is observed (because of its infix position) or if not a specific call but all calls of a function are observed as in our example in Section 2.3. Furthermore, the `main` function has to be modified and the library has to be imported in every program module that uses its entities. Most importantly, a data type that shall be observed has to be an instance of a class `Observable`. Some of our test programs defined many data types and, because we wanted to observe most of them, we had to write many instance definitions. Writing these instance definitions is easy but time consuming. Additionally, all these modifications potentially introduce new errors in the program and also make the program less readable.

On the other hand it might be useful to leave the modifications for Hood in the program. They could be en-/disabled during compilation by a preprocessor flag for a debug mode. Then most modifications, especially writing instances of the class `Observable`, require only a one-time effort. The `observe` combinator may even be placed to observe the main data structures of the program. Thus debugging is integrated more closely into program development. In contrast, Freja and the Redex Trail System cannot save any information from a tracing session for future versions of the program.

5 Other Tracers and Debuggers

Buddha [4, 9] is a tracing system which constructs an EDT and is hence used similarly to Freja. Its implementation is based on a source-to-source transformation, but unlike the transformation of the Redex Trail System this transformation is not purely syntax-directed but requires type information. Buddha can handle only a subset of the language handled by Freja and is not publically available.

In [3, 1] a system is sketched which creates a trace quite similar to an EDT. The main difference is that a parent node is only connected directly to one child. All sibling nodes are connected with each other according to the structure of the definition body of the parent node. Thus the trace has the nice property that

all connecting arrows denote equality, unlike the arrows in an EDT or a Redex Trail. The authors describe a browser which gives more freedom in traversing the trace than the questions of Freja.

There also exist several systems for showing the actual computation sequence of a lazy functional program. Section 2.2 of [13], Chapter 11 of [5] and Chapter 2 and Section 7.5 of [7] give overviews over a large number of tracing and debugging systems for lazy functional languages.

6 Conclusions

In this paper we compared three systems for tracing and debugging Haskell programs: Freja, the Redex Trail System and Hood. These systems take surprisingly different approaches to the task. We evaluated the systems by applying them to a number of programs. We identified distinguishing features and for each system we determined which features are most valuable in practice and where the system is still inadequate. Each system has its virtues and drawbacks but all of them proved useful for locating errors.

Although Hood arrived latest on the market, it is the only system that covers the full Haskell, because of its implementation as a library. Unfortunately, this implementation method seems to be impossible to use for Freja and the Redex Trail System, because their traces describe reductions, not only values. Hood is rather different from Freja and the Redex Trail System whereas the latter have more in common. We noted that the EDT of Freja could be constructed from a Redex Trail with only minor extensions. Thus the Redex Trail System could incorporate the question and answer system of Freja. We fear, however, that the features of a combined system would be overwhelming for the user.

We noted that it is not obvious how to use the Redex Trail System and especially Hood for locating an error. These systems would benefit considerably, and even Freja to some extend, from documentation informing the user how to apply the system for locating errors, which strategies exist and what should be avoided.

In our experiments we avoided programs that made substantial use of continuation passing, higher-order combinators and monads. Already a limited number of λ-abstractions were disturbing, because Freja and the Redex Trail System present them so poorly. It would be interesting to investigate, if using named functions instead of λ-abstractions would make tracing and debugging programs written in the mentioned styles feasible. Especially in this context, it would also be interesting to see how trusting a tested combinator library would improve tracing a program that uses it. It is still unclear to us how Hood could be used for tracing programs using the mentioned styles, because these are based on effecting the control flow, which is not observable by Hood.

We did not use any programs that performed monadic input/output, because Freja does not implement it and the Redex Trail System only for a few operations. It would, however, be interesting to see if Hood's ability to show the return value of an executed input/output action is sufficient in practice.

We conclude that today useful tracing and debugging systems for Haskell are available, but that there is still much to do to make them more useful.

Acknowledgments

We thank Henrik Nilsson and Jan Sparud for taking part in the experiments and making valuable observations.

References

1. 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, Linköping University, Sweden, May 1997.
2. Andy Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop*, 2000. Technical report of the University of Nottingham.
3. 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.
4. Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. In *Proc. 19th Australasian Computer Science Conference*, January 1996.
5. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.
6. 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.
7. Alastair Penney. *Augmenting Trace-based Functional Debugging*. PhD thesis, Department of Computer Science, University of Bristol, September 1999.
8. Simon L. Peyton Jones, John Hughes, et al. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, February 1999.
9. Bernard Pope. Buddha: A declarative debugger for Haskell. Technical report, Dept. of Computer Science, University of Melbourne, Australia, June 1998. Honours Thesis.
10. 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.
11. 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.
12. Philip Wadler. Functional programming: Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, August 1998. Functional programming column.
13. R. D. Watson. *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, Southern Cross, Australia, October 1996.