Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs

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, Hat and Hood. We evaluate their usefulness in practice by applying them to a number of moderately complex programs in which errors had deliberately been introduced. We identify the strengths and weaknesses of each system and then form ideas on how the systems can be improved further.

1 Introduction

The lack of tools for tracing and debugging has deterred software developers from using functional languages [13]. 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 a wave of research into tracing methods for lazy functional languages started and has been increasing since. In this paper we compare the tracing systems that (a) cover a large subset of a standard lazy functional language, namely Haskell 98 [9], (b) are publicly available and (c) are still actively developed. Freja[1] [7, 5] is a system that creates an evaluation dependency tree as trace, a structure based on the idea of declarative/algorithmic debugging from the logic programming community. Hat[2] [12, 11] creates a trace that shows the relationships between the redexes (mostly function applications) reduced by the computation. The most recent system, Hood[3] [2], enables the programmer 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 Freja 1.1, Hat 1.0 and Hood July 2000 release. We evaluate the systems in practice by applying them to a number of moderately complex programs in which errors are deliberately introduced. Tracing systems are interactively used tools. In this paper we concentrate on the usefulness of the systems for the programmer. Runtime and space usage measurements are reported in other papers [5, 6, 11]. We do not aim for a quantitative comparison to crown a winner. Only with a large number of programmers could we have obtained statistically valid data about, for example, how long it takes to locate a specific error with a specific system. Even these data depend for example on how well the programmers are trained for a system, especially because the systems are rather different. Our aim is to explore the design space of tracers and gain insights for the future development of tracing and debugging systems. Our experiments highlight and sometimes even uncover previously unnoticed similarities and distinguishing features of the three systems. The experiments enable us to evaluate the usefulness of system features and lead us to new ideas for how the current systems can be improved or even be combined.

The paper is structured as follows. Section 2 gives a short introduction to each of the three systems. Section 3 compares the systems with respect to their approach to tracing, design and implementation. Section 4 reports on our practical experiments and the insights they gave us into the systems' distinguishing properties and their usefulness. Section 5 briefly describes other systems for tracing and debugging. Section 6 concludes.

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 three systems are still under rapid development we try to avoid details that may change soon.

We demonstrate the use of each system with the following example program.[4]

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

head (x:xs) = x

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

Note that the evaluation in Section 4 is based on experiments with far larger programs.

[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.

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.

The following is a debugging session with Freja for our example program. The symbol ⊥ represents an error and the symbol `?` represents an expression that has never been evaluated and whose value hence cannot have influenced the computation.

```
main ⇒ (8,⊥)      no
4*2 ⇒ 8           yes
head [8,?] ⇒ 8    yes
last [8,?] ⇒ ⊥    no
last [?] ⇒ ⊥      no
last [] ⇒ ⊥       yes
Bug located! Erroneous reduction: last [?] ⇒ ⊥
```

2.2 Hat

Hat 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 alongside the normal computation it builds a redex trail in heap and instead of terminating at the end it waits for the 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 Hat works by going from a faulty output or error message *backwards* until the error is located.

The browser has a graphical user interface which we do not discuss here. Basically the system is used as follows to locate the error in our example program. The program aborts with an error message and the browser directly shows its

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

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 presents the redex trail as shown in the following figure. To demonstrate how the parent of a subexpression is presented (`4*2` is the parent of `8`), more of the redex trail is shown than is needed for locating the error.

```
• last []
  last (3+6:[])
  last (8:3+6:[])
  ▽ 4*2
  main
```

The browser can also show where in the program text for example `last` is called with the argument `[]` 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
  _ : _ : []
```

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>
  }
```

So `last` is called with an empty list. We draw the conclusion that `last` applied to the one element list caused this erroneous call, but strictly the information provided by Hood does not imply this.

3 Comparison in Principle

At first sight the three systems do not seem to have anything in common except the goal of aiding debugging. However, all three systems take a two 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 Hat 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 behaviour and locating errors which should make these systems also interesting for strict functional languages or even non-functional languages.

All three systems 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 can 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 three systems show values instead of arbitrary expressions as far as possible. Hood only shows values anyway. Both Freja and Hat show an argument in a redex not as it was passed in the actual computation but as a value. Only (a part of) an argument that was never evaluated is shown as an unevaluated redex in Hat (`3+6` in the previous example) whereas Freja and Hood represent it by a special symbol (`?` in Freja and _ in Hood). Freja and Hat show an expression only up to a given depth (for example `map succ (0 : succ 0 : □)` in Hat; □ represents the elided subexpression). A subexpression beyond that depth is only shown on demand. None of the systems changes the usual observable behaviour of a program. In particular, 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 Hat show how far values are evaluated in the whole computation, including the effect of sharing. Hence in the previous example Freja and Hat show the first element of the list argument in the first call of `last` as `8` whereas Hood only represents that element by _.

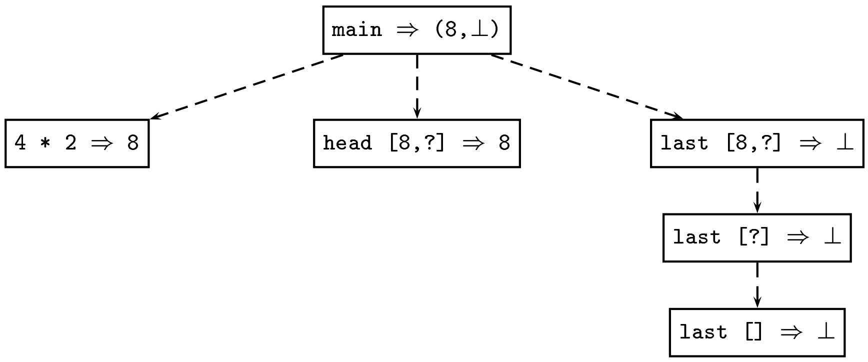

Fig. 1. Evaluation Dependency Tree

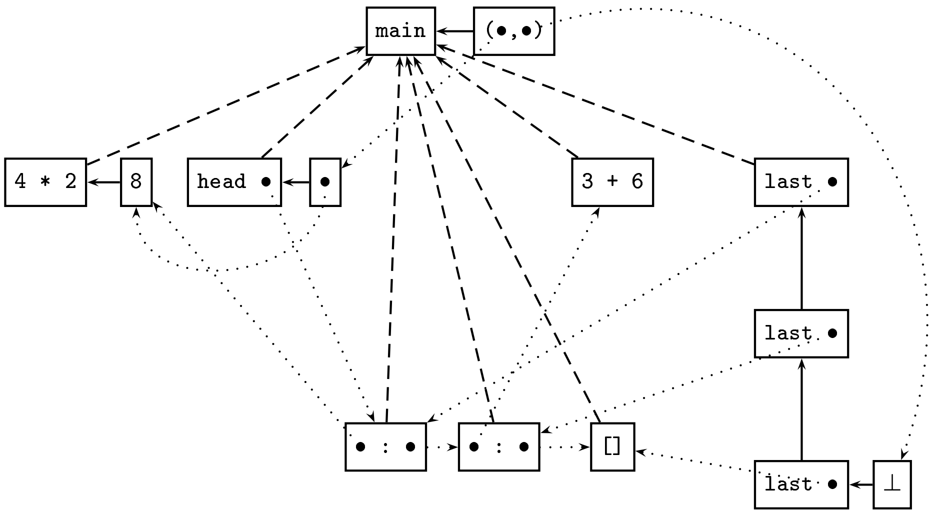

Fig. 2. Redex Trail

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 Hat create a single large trace structure for a program run. It is impossible to show such a trace in full to the user. The browser of each system permits the programmer 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 1 shows the EDT for our example program of Section 2. The symbol $\perp$ represents the value of the error message.

Hat 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 2 shows the redex trail for our example program of Section 2. Dotted arrows point to subexpressions. Both dashed and solid arrows denote the parent relationship. $(8,\perp)$ is the result value of the computation. As in Freja, $\perp$ represents the value of the error message.

The graphs of the two trace structures are laid out to stress their similarity. 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 (however, see also the next paragraph and Section 4.1 about free variables). In contrast, the redex trail contains more information than the EDT, because it additionally links every value with its parent redex and describes how expressions are shared.

The redex trail shown in Figure 2 is a simplified version of the one that is really created by Hat. The real redex trail has an additional node `xs` with parent `main` and children `4 * 2`, `3 + 6`, the two `• : •` nodes and `[]`. That is, the redex trail also records the reduction of the `let` expression. The whole `let` expression is a redex, but in the redex trail it is represented by the defined variable `xs`. Similarly a node `xs` $\Rightarrow$ `[8,?]` that records the reduction of the `let` expression could be added to the EDT. So recording a `let` reduction is an option for both the EDT and the redex trail and the implementors of Freja and Hat made different decisions with respect to this option. On the one hand recording `let` reductions leads to larger traces with an unusual kind of redex. On the other hand it enables more fine grained tracing (cf. Section 4.3).

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.

3.3 Implementation and Portablility

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 modified instructions of the abstract graph reduction machine. Hat transforms the original Haskell program into another Haskell program. Running the compiled transformed program yields the redex trail 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[6]. The transformation of Hat 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 Hat construct traces of the complete computation in the heap.

To reduce the size of the trace, both Freja and Hat 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 Hat 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 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.

[6] The version of Hood which can handle not only terminating programs but also those that abort with an error message or do not terminate requires the non-standard exception library supplied with the Glasgow Haskell compiler.

[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.

Hat can produce partial traces by limiting the length of the redex trails. Because a redex trail is browsed backwards, the system prunes away those redexes that are further than a certain length away from the live program data or output. Hat does not provide any mechanism like re-execution in Freja to recreate a pruned part of the redex trail.

Requiring less heap space may reduce garbage collection time, but Hat 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 desirable to add a feature of one system to another system? Does an alternative design decision make sense? 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. The errors caused all three kinds of faulty observable behaviour mentioned earlier: wrong output, abortion with error message and non-termination.

Our evaluation experiments use the following protocol: At least two programmers are involved. First the author of a correctly working program explains how the program 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.

All the participants are experienced Haskell programmers.

The programs used in the experiments are of moderate complexity. The largest program, PsaCompiler, a compiler for a toy language, consists of 900 lines in 13 modules and performs 20,000 reductions for the input we provided. The longest running program, Adjoxo, an adjudicator for noughts and crosses (tic tac toe), consists of only 100 lines but performs up to 830,000 reductions for our inputs. In our choice of programs we were restricted by the subset of Haskell that Freja supports. For example, Freja does not implement classes and unfortunately not even every Freja program is a valid Haskell program. Freja had been applied to a mini compiler with 16 million reductions [6] and Hat had been applied to a version of nhc98 with 14,000 lines and 5.2 million reductions and a chess end-game program with 20 million reductions [11]. These papers give performance figures but do not indicate how easy debugging programs of this size is. We cannot make such statements either, but our programs are definitely beyond toy examples and of a size often occurring in practise. Our programs also do not perform monadic input/output. Freja does not implement it and Hat only supports a few operations. It would be interesting to see if Hood's ability to show the return value of an executed input/output action is sufficient in practice.

4.1 Readability of Expressions

In contrast to our preliminary fears that the expressions shown by the browsers—reductions, redexes and values — would be too large to be comprehensible, for our programs they are mostly of moderate size and easily readable.

As we will discuss in Section 4.2 the user of a tracing system not only views the trace but also the program. Nonetheless in Freja and Hat informative variable (function) names, that convey the semantics of the variable well, substantially reduce the need for viewing the program and thus increase the speed of the debugging process substantially.

Unevaluated Expressions Freja shows unevaluated expressions as `?` and the undefined value as $\perp$. This property makes expressions even shorter and more readable. This also holds for Hood. Only in some cases more information would be desirable for better orientation. In Hat the display of the unevaluated redexes in full sometimes obscures 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 Hat is not based on questions, it is less clear if showing only demanded values would be suitable for it. Finally, the fact that Freja and Hat show values to the extent to which they are evaluated in the whole computation whereas Hood shows them to the extent 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. Because the mapping contains only expressions that were demanded during the computation, the representation is short in most cases. However, for functions that are called often and especially for higher-order functions the representation is unwieldy. The representation requires some time to get used to. In return, it permits a rather abstract, denotational view of program semantics which is useful for determining the correctness of part of a program.

In Freja and Hat 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 easily 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 Hat. 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, reading expressions involving λ-abstractions is hard. We conjecture that with Freja or Hat debugging programs that make substantial use of λ-abstractions, as commonly done for stylised abstractions such as continuation passing, higher-order combinators and monads, is rather difficult. Our programs hardly use stylised abstractions. In fact, PsaCompiler uses only named functions, even in the definitions of its parser combinators, where most Haskell programmers would use λ-abstractions. During tracing, Freja and Hat show very readable expressions for PsaCompiler.

Free Variables Both λ-abstractions and the definition bodies of locally defined functions often contain free variables. To answer a question in Freja the values of such free variables must be known. Hence Freja shows this information in a `where` clause. The following question from an evaluation experiment 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*.

Hat 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 the corresponding arguments of parent redexes it is necessary to follow links to the program source.

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 one observation of a local function `moveval` is presented as follows

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

4.2 Locating an Error

With all three systems we successfully locate all errors in our programs. For locating an error in our largest program we answer between 10 and 30 questions in Freja, look at 0 to 6 parents in Hat and add `observe` up to 3 times for Hood. The relation between these numbers is typical. However, the numbers cannot be compared directly to determine speed of use, because the counted operations are completely different. A major difference between the systems is the time the user has to spend thinking about what to do next, and the effort required to do it. For example, the time required in Hood for deciding where to add `observe` annotations, modifying the program (discussed further in Section 4.4), recompiling the program and reexecuting it is substantially higher than answering a question or selecting an expression for viewing its parent. Furthermore, the amount of data produced by a single `observe` annotation is usually substantial.

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.

Hat 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 seems arbitrary. In general Hood users apply a top-down strategy in their placement of `observe` combinators, if the faulty behaviour does not point to any program location, for example when the program does not terminate. Then the questions the Hood users asks are similar to those asked by Freja. If, on the other hand, the position where the observable fault is caused can be identified, for example when the program aborts with an error message occurring only once in the program, then a Hood user tries to apply a bottom-up strategy reminiscent of Hat.

Our programs contain several errors. Users of Hat and Hood locate the errors in the same order, because they always locate the error that causes the observed faulty behaviour. In contrast, the questions of Freja sometimes lead to the location of a different error. It is possible to tackle a specific faulty behaviour by answering some questions incorrectly, but that requires care. One may easily steer into irrelevant regions of the EDT.

General Usability Hat 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 Hat.

Information Used A Hood user has to modify the program and hence look at it. Sometimes just the process of searching for a good placement of `observe` reveals the error. Users of Freja and Hat, 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 Hat are not intended 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 Hat 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 Hat also gives information about which expressions are shared. This information is useful in some cases, usually when expressions are shared unexpectedly.

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 observe functions to obtain at least a relation between arguments and result. In particular, the representation of an observed function shows clearly which (part of an) argument is *not* demanded by the function for determining its result. This feature is helpful for locating errors.

Wrong Subexpressions Often, in the questions posed by Freja, a specific subexpression of a result is wrong. For example in the following program the *1* in the second list element should be a *2*. But there is no way to give Freja this information. We can only confirm or refute the reduction as a whole.

```
translateStatement
  (TableImp
    (newTableFunction
     where
       newIndex = "y",
       newEntry = 2,
       oldTableFunction = newTableFunction
                          where
                            newIndex = "x",
                            newEntry = 1,
                            oldTableFunction = implTableEmpty))
  ?
  (Assignment "x" (Compound (Var "x") Minus (Var "y")))
=>
_Tuple_2 [Lod 1,Lod 1,Sb,Sto 1] 4
```

In contrast, the redex trail contains the parent of every subexpression. A Hat user seldom asks 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 look at far less parents with Hat than we answer 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 mechanisms in Freja and Hat not only save space but also reduce 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 is useful. For Hat a corresponding feature seems desirable. In Freja, sometimes a question is repeated, because the same reduction is 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 Overhead With respect to the time overhead caused by the creation of traces the low-level implementation of Freja pays off. The overhead is not noticeable. In contrast, in Hat traced computations are more than ten times slower. For some inputs adjoxo seems to be non-terminating but it is only slow! We experience the same with Hood when we observe at positions that are 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.

Compiler Messages A helpful error message from a compiler can reduce the need for a tracer. 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[8], but not if it was compiled with Freja or nhc98. However, in that case Hat directly shows the function with its arguments whereas Freja requires the answers to numerous questions before locating the error.

4.3 Redexes and Language Constructs

A computation does not only consist of reductions of function applications. We noted already in Section 3.2 for `let` expressions that there are other kinds of redexes. This aspect only concerns Freja and Hat, because Hood only shows values.

CAFs A constant applicative form (CAF) is a top-level variable of arity zero, in other words a top-level function without arguments. Its value is computed on demand and shared by its users. Both Freja and Hat take the view that a CAF has no parent. Hence the trace of a program in Freja is generally not a single

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

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 experiment programs containes 35 CAFs. We have to confirm the correctness of evaluation for all CAFs before reaching the question about `main`, although none of these CAFs are related to any of the errors. Freja can be instructed to 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 Hat a corresponding modification without losing sharing of CAFs 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 is not noticeable, because none of the introduced errors concernes CAFs. However, programs can be constructed where this lack of information hinders locating an error:

```
nats :: [Int]
nats = 0 : map succ nats

main = print (last nats)
```

The computation of this program does not terminate. When the programmer interrupts the computation, Hat may show `map succ (0 : succ 0 : □)` as next redex to be evaluated. The parent of this redex is `nats`, which has no parent. The error may well be that the programmer intended to call another function than `last` in the definition of `main`, but unfortunately the redex `last nats` is unreachable.

We stated in Section 3.2 that Hat has a special kind of redex for locally defined variables of arity zero (defined in `let` expressions and `where` clauses). The parent of such a variable redex is the redex that created the definition and not — as for function application redexes — the redex that created the application. So as for CAFs redexes may become unreachable.

Guards, `cases` and `ifs` 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, 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.

Similarly, Hat uses special redexes for case and if expressions. On the one hand, these special redexes complicate the system. On the other hand, they are useful for large function definitions. The special redexes enable more fine grained tracing up to the level of guards, cases and ifs, whereas Freja only identifies a whole function reduction as faulty. Similar to the situation for locally defined variables it is possible to extend the definition of Freja's EDT by special nodes for guard, case and if reductions. For Hat, special redexes for these reductions are important to make parts of the redex trail reachable by backward traversal that otherwise would be unreachable.

4.4 Modification of the Program

Whereas Freja and Hat are applied to the original program, requiring only special compilation, Hood is based on modifying the program. Sometimes 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 can only be observed if it is an instance of a class Observable. Some of our experiment programs define many data types; because we want to observe most of them, we have 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 Hat cannot save any information from a tracing session for future versions of the program.

5 Other Tracers and Debuggers

Buddha [4, 10] is a tracing system which like Freja constructs an EDT. Its implementation is based on a source-to-source transformation, but unlike the transformation of Hat this transformation is not purely syntax-directed but requires type information. Buddha is still actively developed.

Booth and Jones [3, 1] sketch a system 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 [14], Chapter 11 of [5] and Chapter 2 and Section 7.5 of [8] review a large number of tracing and debugging systems for lazy functional languages.

We could not include any of these systems in our experiments, because there are only limited prototypes, not publicly available.

6 Summary and Conclusions

We have compared and evaluated the tracing and debugging systems Freja, Hat and Hood by applying them to a number of programs.

Tracing and debugging systems for lazy functional languages have made considerable progress in recent years: all three systems prove to be effective tools for debugging our programs. Though none of our programs is very large, some of them are large enough to show that the scope of application for the tools goes well beyond easy exercises. Unfortunately the practical usability of Hat and especially Freja is currently limited by the fact that they do not support full Haskell 98.

Each of the tracing tools takes a unique approach with specific strengths. In particular, Freja has a systematic fault-finding procedure; Hat starts at the observed error and enables exploring backwards the history of every subexpression; Hood observes the data flow at specific program points by need.

Based on our experiments we identify in Section 4 the strengths but also the weaknesses of each system. For some weaknesses we already suggest improvements, often based on the convincing solutions of the problems in other systems. Other weaknesses are linked either to the tracing method or the implementation, which we discuss in Section 3. Hence they are more difficult to address and require further research. For example, Freja cannot take advantage of the common case that only a subexpression of a reduction is wrong, Hat is slow and Hood gives almost no indication of how values are related. We claim that an integration of Freja into Hat is feasible whereas Hood's approach is rather different from the approaches of the other two systems.

Finally, good tools are not sufficient for debugging. The user needs advice on how to effectively use each system; a strategy needs to be developed for Hat and especially for Hood, but even Freja would benefit from advice on how to employ its advanced features. Also a strategy for using several systems together, taking advantage of their respective strengths, is desirable.

Acknowledgments

We thank Henrik Nilsson and Jan Sparud for taking part in the evaluation experiments and making valuable observations. The work reported in this paper was supported by the Engineering and Physical Sciences Research Council of the United Kingdom under grant number GR/M81953.

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. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 36–47. ACM Press, 1999.
7. 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.
8. Alastair Penney. *Augmenting Trace-based Functional Debugging*. PhD thesis, Department of Computer Science, University of Bristol, September 1999.
9. Simon L. Peyton Jones, John Hughes, et al. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, February 1999.
10. Bernard Pope. Buddha: A declarative debugger for Haskell. Technical report, Dept. of Computer Science, University of Melbourne, Australia, June 1998. Honours Thesis.
11. 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.
12. 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.
13. Philip Wadler. Functional programming: Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, August 1998. Functional programming column.
14. R. D. Watson. *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, Southern Cross, Australia, October 1996.