A toolkit for multi-view tracing of Haskell programs

von
Thorsten Brehm
Matr. Nr. 205526

Diplomarbeit

in Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen
im November 2001

angefertigt am
Department of Computer Science
University of York

Gutachter:
Professor Dr. Colin Runciman
Professor Dr. Klaus Indermark

Hiermit versichere ich, daß ich die Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Aachen, den 26. November 2001

Abstract

A team at the University of York is developing the Hat tracer for the lazy functional language Haskell. A Haskell program is transformed into a traced version, which will produce a trace file, representing its evaluation.

Evaluation of a lazy functional program can be viewed in various ways. Hat began as a prototype of a tool, providing one such view by displaying redex trails.

This thesis makes three contributions:

- Two new views are provided. For each desired view a tool was designed and implemented, accessing and interpreting the trace file appropriately. The tools are based on Algorithmic Debugging and on Observation Debugging. It is investigated whether a single trace structure can support these several different views effectively.
- A low-level interface is defined in C as a common basis for efficient access to the trace. The trace file can become very large (several hundreds of megabytes) making it important to access the contained graph structure efficiently. For the development of more complex tools, a Haskell interface is layered on top. The Haskell interface allows concise and convenient expression of tool-design. Among the issues investigated are, what the optimal boundary between these interfaces is, and if all the tools can be made to fit the common interface.
- Each of the three debugging views has strengths and weaknesses. Interaction between the tools might enable the user to take advantage of the strengths of all three tools by switching between different views in the middle of viewing an evaluation trace. It is investigated how the three separate tools may be connected to form a single debugging toolkit for the lazy functional language.

Acknowledgements

This thesis is part of my degree at the RWTH Aachen. But I was working on this project at the Department of Computer Science at the University of York, where I stayed from March to September 2001.

First of all, I would like to thank Prof. Colin Runciman for inviting me to York, supervising me as a foreign, visiting student, for uncounted constructive comments on the project and this thesis, and for so much patience in proof-reading many parts of this thesis and correcting my grammar. I am afraid, readers spotting remaining errors immediately identify parts on which I have been working on since I left York.

Also special thanks to the other members of the ART project team in York, to Dr. Olaf Chitil and Dr. Malcolm Wallace, also for many constructive comments, for so many explanations, bugfixes, and all sorts of other help with my project.

Thank you all for providing such excellent working conditions at your department!

Also thanks to Prof. Klaus Indermark and Dr. Frank Huch at the RWTH Aachen for giving me the opportunity and encouraging me to work on my project at the University of York.

All of you, *thank you very much* – for your help, for a great experience and an unforgettable stay in England!

Thorsten Brehm
Aachen, November 26th, 2001

Contents

Chapter 1

Introduction

Programming in functional languages offers many advantages to a program developer. The language abstracts from the specific underlying computer architecture. Algorithms are formulated on a higher level. Programming in functional languages saves time compared to imperative languages.

If the evaluation of a program does not show the intended behaviour, it is desirable to debug a program by following its evaluation. Even tracing the evaluation of a correct program can be of interest. A programmer may want to learn more about how a program works – for example when he has not written the program himself. Or he may be interested in seeing how an own program is evaluated, in order to find ways to improve it.

Strict functional languages force the evaluation of all arguments to a function before evaluating the function itself.

Lazy or non-strict functional languages in contrast may pass unevaluated arguments to functions. These arguments are only evaluated if their value is really needed for the computation. Values of a complex structure may only be partially evaluated, hence allowing infinite data structures to be used.

Difficulties in tracing functional programs: The abstraction introduced by functional languages makes it more difficult to trace and debug a program. The evaluation of a program in a *lazy* functional language is especially hard to comprehend. The actual evaluation order of expressions cannot easily be seen from the program's definition.

Tracing tools: Tools are necessary to assist in understanding the evaluation of a lazy functional program. Various methods have been proposed and implemented allowing tracing and debugging of such programs. One of these methods is the Haskell Tracer *Hat*. Hat records the evaluation of a program and provides a browser to trace its computation, once the evaluation has finished. Two further tracing views are also described in this thesis. It is an important requirement for all tracing tools that their use must not influence the semantics of the traced program.

This thesis has three goals:

- The first goal is to show that two additional debugging and tracing views can be provided, based on the data structure which is generated by Hat. This data structure is already used for Hat's Redex-Trail browser. Tools will be implemented supporting these two new views.

- A second goal is to support interaction between all three tracing tools. It is discussed how switching between different tracing views is possible and how this helps the programmer in gaining from the advantages of all three tracing views.

- A third goal is to provide high-level interfaces for Haskell to the three different tracing views. These interfaces are used for the implementation of the presented tracing tools. But the interfaces may also be used to conveniently implement further tools based on the supported tracing views.

Structure of this thesis:

- In chapter 2 three tracing views are introduced. For each of these views an existing tool is presented. The strengths and weaknesses of these tracing views in general and disadvantages of the existing tools in particular are discussed.

- Chapter 3 describes a slightly modified approach for a tracing view named *Observational Debugging*. It is shown how this view can be implemented based on Hat's Redex-Trail structure.

- Chapter 4 shows that the basic data structure necessary for a tracing view named *Algorithmic Debugging* can also be derived efficiently from Hat's Redex-Trail structure.

- Chapter 5 describes the implementation of a new tool for Observational Debugging. Several features are introduced which support the programmer in his observations. These features help to overcome some weaknesses of Observational Debugging.

- Chapter 6 describes the implementation of a new tool for Algorithmic Debugging. It is described how the convenience of Algorithmic Debugging is improved by providing additional options.

- Chapter 7 shows that interaction between all three debugging views based on the common data structure is possible. It is shown that the presented tracing tools are complementary and that the programmer may gain by switching between the tracing views.

- Chapter 8 presents the interfaces to the different tracing views and describes specific aspects of their implementation.

- Chapter 9 is a case study of the implemented debugging toolkit. It demonstrates some of its advantages and disadvantages on a moderate sized program example.

Chapter 2

Current Tracing and Debugging Tools

This chapter explains why tracing tools for lazy functional languages are necessary. Three tracing and debugging views and an existing tool for each view are described:

- Observational Debugging is introduced as provided by the Haskell library HOOD [GIL00] (section 2.2).
- Algorithmic Debugging on lazy functional programs is introduced and an existing tool named Freja [NIL98] is discussed (section 2.3).
- The Redex-Trail view and the Redex-Trail browser of Hat named `hat-trail` is discussed (section 2.4). It is the original tool implemented for the Haskell Tracer (Hat) developed at the University of York.

Strengths and disadvantages of each tool are briefly discussed. A complete evaluation of these tools is given in [CRW00]. The comparison in this chapter shows that the different debugging views are complementary. Hence, a unified approach is advantageous, which provides all three debugging views on top of a single data structure.

2.1 Why tracing tools are necessary

In imperative languages programmers often use statements to display values of variables during evaluation. The printed values provide him with basic information about the evaluation. Programmers might attempt to use a similar approach in functional languages. Two different methods of printing in functional languages exist:

Monadic output: The introduction of monadic output to print values in functions which previously returned basic types is not straightforward. It may even lead to extensive modifications to the entire program, as all occurrences of the function need to be modified (see example 2.1).

Unsafe output: With unsafe output it is possible to print values as a side effect of evaluation. Usually side effects are not allowed in functional languages but here it is only used as a debugging method and not part of the original program. The advantage compared to monadic output is that the type of a function does not change when unsafe output is introduced. This approach requires less modifications to the original program. But the disadvantage of unsafe output is that the due to lazy evaluation the order of printed values is hard to comprehend. Output of different unsafe operations may even interleave and result in unreadable data (see example 2.2, page 7).

Example 2.1 (Tracing with monadic output)

```
insert              :: Int → [Int] → [Int]
insert x []         = [x]
insert x (y : ys) | x > y = x : insert x ys
                  | otherwise = x : y : ys
main = putStrLn (show (take 5
                             (insert 4 [1..])))
```

Function `insert` is erroneous. The program's output is not `[1,2,3,4,4]` as intended but: `[4,4,4,4,4]`
The erroneous program is modified in order to print the result of the recursive call to `insert`. Due to the monadic type of `putStrLn` the type of function `insert` is altered. All modifications are shown in bold text:

```
insert              :: Int → [Int] → IO [Int]
insert x []         = return [x]
insert x (y : ys) | x > y = do
                              result ← insert x ys
                              putStrLn (show result)
                              return (x : result)
                  | otherwise = return (x : y : ys)
main = do
          result ← insert 4 [1..]
          putStrLn (show (take 5 result))
```

The modified program does not terminate as an infinite list is printed:
`[4,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,...` □

Common drawbacks of primitive tracing approaches: Both simple methods to display values have common drawbacks.

- The use of output is inconvenient as the program's source-code needs to be altered in order to introduce the printing functions.
- Printing values can be very dangerous as it may alter the program's semantics. For example an infinite data structure must not be printed. Printing a value may also result in a failing evaluation which is not performed by the original program.

Especially the problem of influencing a program's semantics shows that more sophisticated ways of debugging are necessary, which allow to trace program evaluation without affecting it. Furthermore the tools should provide comprehensible information. Printing simple values only makes it difficult to conclude where the error of a program is located.

Example 2.2 (Tracing with unsafe output)

> Printing values using unsafe output is demonstrated using a combinator:
> `trace :: String → a → a`
> The combinator eagerly evaluates its first argument and prints it as a side effect. The function's result is simply its second argument, so the combinator can easily be inserted anywhere in a program. To print the result of the recursion only a single line in the erroneous program (example 2.1) needs to be modified:
>
> ```
> insert x (y : ys) |x > y  =  x : let  result = (insert x ys) in
>                                       trace (show result) result
> ```
>
> Using unsafe output is less effort compared to example 2.1 but the unsafe output of three `trace` instances interleaves. Also the program does not terminate as an infinite list is printed:
> `[4[4[4,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,...` □

2.2 Observation Debugging

A debugging method already implemented by HOOD [GIL00] allows observations of values within the computation. The programmer annotates expressions of interest within the source-code of the program with a specific combinator `observe`. When the program is recompiled and rerun then the combinator records each value observed as a side effect of its evaluation. The implementation of `observe` ensures that the observed value is evaluated exactly as far as the unmodified program evaluates it. The semantic of a program is therefore unaffected by the introduction of the combinator, except for the additional recording of the observed values. Observed values may therefore contain unevaluated subexpressions, which are represented by a special symbol.

The recorded information is displayed after the program's execution. HOOD may even observe higher-order values and display them as a partial table of arguments with their respective results (see example 2.3).

Observing complex data structures: To use the combinator `observe` for user-defined data structures a special class instance for `observe` needs to be defined. The definition is straightforward but still results in some work for the programmer. This is especially expensive if user-defined data structures are used recursively, as the class instance needs to be defined for each of these structures (see example 2.4).

Example 2.3 (Making an observation)

The combinator `observe` is introduced to the erroneous function `insert` in example 2.1 (page 6). Its first argument is only a label. The second argument is the observed value. Only a single line needs to be modified:

```
insert x (y : ys) |x > y  =  x : (observe "insert" insert) x ys
```

When the program is rerun then the normal output is shown, as well as the recorded arguments and results of `insert`.

```
[4,4,4,4,4]
-- insert
  { \ 4  (2 : 3 : 4 : _) -> 4 : 4 : 4 : 4 : _
  }
-- insert
  { \ 4  (3 : 4 : _) -> 4 : 4 : 4 : _
  }
-- insert
  { \ 4  (4 : _) -> 4 : 4 : _
  }
```

□

Example 2.4 (Observing values of a user-defined data structure)

A class instance of `Observable` for the user-defined data structure `MyList` needs to be defined in order to observe values of this type:

```
data MyList a = MyList a (MyList a) | MyEmpty deriving Show

instance (Observable a) => Observable (MyList a) where
observer (MyList a list) = send "MyList" (return MyList << a << list)
observer MyEmpty         = send "MyEmpty" (return MyEmpty)
```

□

As the debugging tool HOOD is implemented as a combinator library, which can be imported into a Haskell program, it automatically supports the full Haskell language standard.

GHOOD: An extension to HOOD was presented as GHOOD [REI01]. It offers a graphical back-end for HOOD and allows animated visualisation of the observations. The programmer can follow the intermediate evaluation steps of the observed values. He can therefore tell when and in which sequence substructures of a larger data structure are evaluated.

Advantages of Observational Debugging:

- The use of Observational Debugging is easy to learn. Using HOOD is similar to the primitive debugging method using "print" statements in imperative languages [CRW00]. But also functions may be observed, which are presented to the user as a list of its observed applications and their respective results.
- The programmer can use his knowledge about the program to immediately inspect parts of the computation where he suspects an error. Parts which are known to be correct need not be considered.

Disadvantages of Observational Debugging:

- Finding the cause of an error can be difficult using observations as no guidance is provided. The programmer needs to decide himself which functions to observe. This can be easy if the programmer has a correct suspicion about the error's cause. But otherwise the programmer may need many observations to be successful.
- Each observation may result in an extensive result list of values. The programmer may be flooded with information and be unable to find a particular value of interest.

Specific disadvantages of HOOD:

- The source-code of a program needs to be modified in order to make an observation. These modifications are inconvenient and also force the programmer to recompile and rerun his program. Modifications may also introduce new errors to the program. Eventually the programmer has to remove all observation statements when his tracing session is finished.
- In order to find the cause of a program error the programmer often needs more than a single observation. He rather makes a whole series of observations, where each observation gives him some hints which values to observe next. Efforts are therefore renewed with each new observation.
- Additional effort is necessary to observe user-defined data structures (see example 2.4). The modifications are straightforward but result in some inconvenience. Programmers may therefore avoid the observation of user-defined data structures.
- The observation of local functions is possible but HOOD does not show their context. Though observations may belong to different contexts they are mixed into a single table. Such observations may confuse the programmer and lead to wrong conclusions.

2.3 Algorithmic Debugging

The goal of Algorithmic Debugging is to provide an almost automated tool to locate programming errors which lead to wrong results of a program's evaluation.[1] The Algorithmic Debugger is based on a data structure named the *Evaluation Dependency Tree* describing the program evaluation.

Evaluation Dependency Tree: The evaluation of a functional program can be described by equations, where each equation is given by a redex and its respective result. For a *strict* functional language it is enough to consider redexes which are CAFs[2] or applications with fully evaluated arguments (the idea is extended to *lazy* functional languages on page 13). For the evaluation of a redex a reduction rule is chosen, matching the outermost function symbol. The evaluation of a redex therefore *directly depends* upon the evaluation of all CAFs and applications introduced within the body of the chosen rule. This *dependency* (named *direct evaluation dependency*) is used to order the equations into an *Evaluation Dependency Tree* (EDT). The root node of this EDT is labeled by the reduction of the CAF `main` to the program's result. A formal definition of the EDT is presented in section 4.1. Figure 2.1 is an example for an EDT, describing the evaluation of the program in example 2.5.

Example 2.5 (Erroneous program for the symmetric difference of sorted lists.)

```
symdif                      :: Ord a ⇒ [a] → [a] → [a]
symdif [] ys                = ys
symdif xs []                = xs
symdif xs@(x : xs') ys@(y : ys') = symdif' (compare x y) xs ys

symdif'                     :: Ord a ⇒ Ordering → [a] → [a] → [a]
symdif' cmp xs@(x : xs') ys@(y : ys') =
  case cmp of
  LT → x : symdif xs' ys
  EQ → symdif xs' ys'
  GT → y : symdif xs' ys'

main = (symdif [1,4,6] [1,3,6])
```

Function `symdif` is intended to return the symmetric difference of its two ascending argument lists. However, the evaluation of this example yields the *unexpected* result `[3]` rather than the *expected* result `[3,4]`. □

Reasons for errors: Algorithmic Debugging for functional languages is based on the idea that if any evaluation of a redex with fully evaluated arguments yields an incorrect result, it must be for one of the following two reasons:

[1]Of course, even if a program's evaluation resulted in correct output, the program may still contain errors which did not influence the program's final result. Such errors cannot be found by an Algorithmic Debugging tool.

[2]constant applicative form

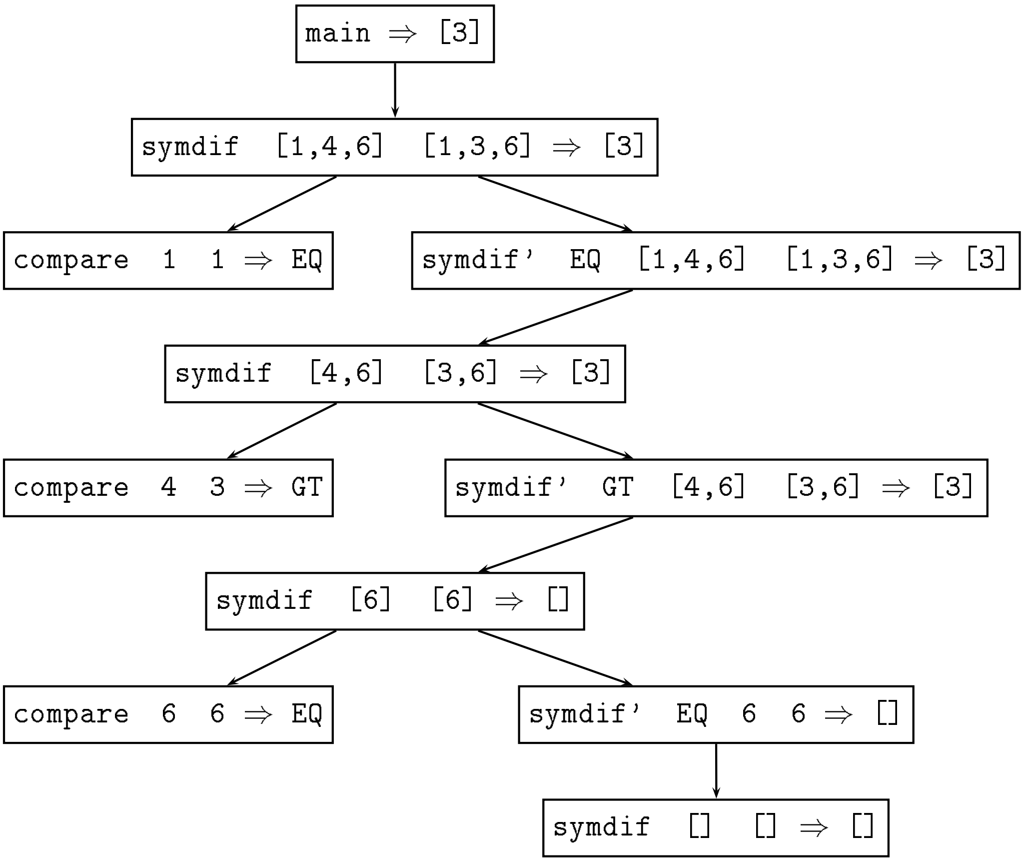

Figure 2.1: Evaluation Dependency Tree for Program in example 2.5

(1) The rule used to reduce the outermost function symbol is defined incorrectly.

(2) The evaluation of a CAF or an application introduced by the reduction rule for the outermost function has an erroneous result.

Locating an error: If an equation is wrong, but all its child equations within the EDT are correct, then the rule used to reduce the outermost function symbol must be erroneous. The goal of Algorithmic Debugging is to find such an erroneous node which does not have erroneous children. Such a node is called *buggy*.

Definition 2.1 (Buggy Node)
"A node in a tree is *buggy* if it is erroneous but has no erroneous children."
(taken from [NAI97], Definition 1, page 4) □

Instead of presenting the entire EDT to the programmer and asking him to use this knowledge to locate a *buggy* node, this search can be automated.

The Algorithmic Debugger: An algorithm named the *Algorithmic Top-Down Debugger* traverses the EDT and asks an oracle about the correctness of each equation. Usually the programmer acts as an oracle and is assumed to be able to answer all questions truthfully. *Correctness* here means *correct* according to the programmer's original intentions[3].

- If the programmer answers that an equation is correct then its child equations in the EDT do not need to be considered. Some of its child equations could indeed still be erroneous, but if their parent equation is correct then these errors do not cause further errors within the program's evaluation.
- If the programmer identifies an erroneous equation then the algorithm continues to ask about the correctness of all its child equations. If there are no child equations or all child equations are correct then the equation describes a *buggy* node, which is presented to the programmer. The reduction rule used to reduce the outermost function symbol of the redex on the equation's LHS[4] must be erroneous.

Starting the Algorithmic Top-Down Debugger: The Algorithmic Debugger starts by asking about the root node of the EDT. The root node is the equation describing the reduction of the constant `main` to its result. But it would also be possible to start at any other point within the EDT if an error is only to be sought in a certain part of the computation.

Correctness of Algorithmic Top-Down Debugger The correctness of the described Top-Down Debugger can be proven. When the evaluation of a program has an erroneous result, hence the root node of the EDT is erroneous, then the Top-Down Debugger finds a buggy node:

Lemma 2.2 (Topmost buggy node)
"A finite tree *(an EDT)* has at least one topmost buggy node if and only if the root is erroneous."
(taken from [NAI97], Proposition 2, page 8) □

Lemma 2.3 (Correctness of the Top-Down Debugger)
"If the child relation defines a finite tree then Top-Down Debugger returns all topmost buggy nodes and only topmost buggy nodes."
(taken from [NAI97], Proposition 3, page 8) □

[3]The underlying runtime system is supposed to be reliable and not to introduce any mistakes when applying the program's reduction rules. Errors are therefore always assumed to result from an erroneous definition of a reduction rule.

[4]left hand side

Example 2.6 (Algorithmic Debugging for example 2.5)

```
1 > main = [3]                          (Y/N) : no
2 > symdif [1,4,6] [1,3,6] = [3]        (Y/N) : no
3 > compare 1 1 = EQ                    (Y/N) : yes
4 > symdif' EQ [1,4,6] [1,3,6] = [3]    (Y/N) : no
5 > symdif [4,6] [3,6] = [3]            (Y/N) : no
6 > compare 4 3 = GT                    (Y/N) : yes
7 > symdif' GT [4,6] [3,6] = [3]        (Y/N) : no
8 > symdif [6] [6] = []                 (Y/N) : yes

Error located! Bug found in :
symdif' GT [4,6] [3,6] = [3]
module "Main", line 7, column 1
```

Traversing the EDT: In questions 1,2,4 and 7 the programmer identifies equations which seem erroneous. Hence the Algorithmic Debugger continues to ask about their child equations (see EDT in figure 2.1). The equations in questions 3,6 and 8 are identified to be correct, so their child equations are ignored.

Eventually, the algorithm halts as the equation in question 7 is erroneous, but its only child equation is correct according to question 8. The error is therefore identified immediately at the definition of function `symdif'`.

Fixing the bug: It is left to the programmer to inspect the definition of this particular function and to determine how to fix the problem. The given arguments for which the function results in an erroneous result provide him with further information. In this case the programmer should inspect the `GT` alternative of the `case` expression, as the parameter `cmp` has the value `GT`. The bug here is an incorrect argument to the function `symdif`. The subexpression `symdif xs' ys'` needs to be replaced by `symdif xs ys'` in order not to lose the value of `x`. □

EDT for lazy functional languages: The presented EDT (page 10) refers to *strict* functional languages, as only applications with fully evaluated arguments are considered. In a *lazy* functional language argument evaluation is interleaved with the reduction of the outermost function itself. But an EDT can also be constructed for a lazy functional program [NIL94]. Equations may describe the *partial* evaluation of redexes instead of *full* evaluation as in *strict* languages. Considered applications may also have *partially* evaluated arguments, where the arguments are evaluated exactly as far as is demanded. The EDT for a lazy functional program describes the program evaluation by a call-by-value strategy with a perfect guess how far arguments need to be evaluated.

Freja - An Implementation of Algorithmic Debugging

An implemented example for Algorithmic Debugging in lazy functional languages is the system named Freja [NIL98]. It is a partial implementation of the Haskell language standard. The system consists of an own compiler and runtime system to evaluate the program to be debugged. During its evaluation the runtime system collects necessary data about reductions and their dependencies. Once the result of the main CAF is evaluated, the system asks about its correctness. In case the result is wrong, it continues to traverse the described Evaluation Dependency Tree until the error is located.

EDT in bounded memory: The information about the reductions gathered during the evaluation might exceed the system's memory. Freja therefore bounds the amount of memory used for debugging information. Each execution only supplies a pruned subtree of the EDT. When a debugging session needs any of the pruned parts then the program is reexecuted. A debugging session in general may require several automated reexecutions.

The system has shown the possibility of Algorithmic Debugging in lazy functional programs. It has proved to be a useful tool for debugging certain kinds of Haskell programs, but in other cases programs are less suitable for Algorithmic Debugging.

A drawback of Freja: The system only supports a subset of the Haskell language, which is a considerable drawback for use as a standard tool on complex Haskell programs.

Advantages of Algorithmic Debugging:

- The programmer is guided by the debugging tool until the cause of the erroneous program result is found. He only needs to answer questions about equations by deciding whether the reduction of the redex on the LHS to the result on the RHS[5] meets his intentions.
- The programmer is only presented one equation at a time. This makes the tool easy to use. For example with Observational Debugging the programmer may not be able to decide the correctness of a function when too many equations are shown.
- Implementations of Algorithmic Debuggers will usually be separate tools and not require modifications of the program's source-code.

Disadvantages of Algorithmic Debugging:

- Though the tool's guidance can be helpful especially for inexperienced programmers, it can also be annoying. Programmers may feel restricted.
- The programmer has to follow the program's evaluation from the very beginning, though he may already know which parts of the program may have caused the erroneous result and which parts are reliable.

[5]right hand side

- The programmer may not be able to decide the correctness of a reduction – for example if a presented data structure is too large to comprehend. Then the programmer is stuck in the debugging session.

- The programmer's answers must be correct. Any mistake results in a bug found at a wrong place of the program.

2.4 Hat - The Haskell Tracer

Hat is a system which records information about program evaluation, allowing separate tracing tools to use this information after the evaluation has finished. Initially the recorded information was intended for use by a single tracing tool only.

Graph reduction: Computation of a functional program can be seen as graph reduction [SR97a]. The reduction of redexes results in replacing a subgraph representing a redex by a new subgraph describing the reduction's result.

The Redex-Trail: Hat creates a directed graph describing an abstract view of the entire program computation. It is based on graph reduction but it does not lose the information about a subgraph when it is overwritten. Instead the new subgraph representing the reduct has a link to the original subgraph representing the original redex. Hence, the resulting *Redex-Trail graph* records the entire program evaluation. Starting from the program's result each reduct has a link to its creating redex (called the *parent* redex). Even each subexpression of a reduct has a link to its *parent* redex.

Recording the Redex-Trail: Hat generates a program's Redex-Trail by applying an automatic *source-to-source transformation* to the program [SR97a]. The transformation introduces combinators, which record the original program's Redex-Trail during the execution of the transformed program. The transformed program still produces the same output as the original and its semantics are unchanged, apart from the additional Redex-Trail generation.

An example of a Redex-Trail: The Redex-Trail for example 2.7 is shown in figure 2.2. In figure 2.2 it can be seen that all nodes of the Redex-Trail are reachable from the node representing the program's result `6`, using the nodes' links to their parents and subexpressions.

Example 2.7 (Factorial of 3)

$$
\begin{array}{lll}
\texttt{fac} & :: \ \texttt{Int} \to \texttt{Int} & \\
\texttt{fac}\ 0 = & 1 & \\
\texttt{fac}\ 2 = & 2 & \text{— added to keep example computation short} \\
\texttt{fac}\ \texttt{n} = & \texttt{n} * \texttt{fac}\,(\texttt{n} - 1) & \\
\texttt{main} = & \texttt{fac}\ 3 &
\end{array}
$$

□

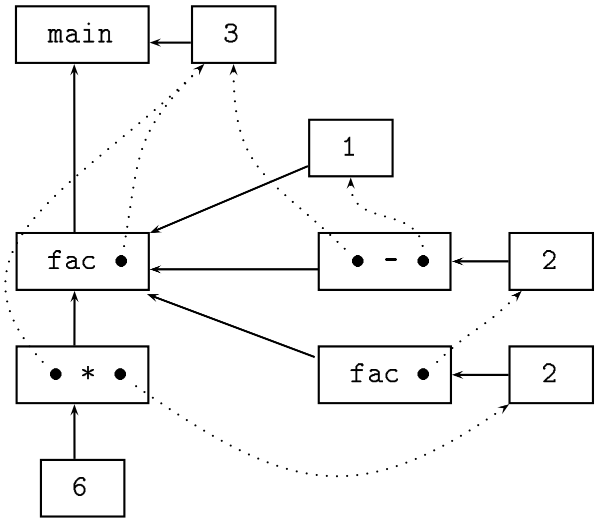

Figure 2.2: Redex-Trail for program in example 2.7. Solid arrows represent links from reducts to their creating redexes. Dotted arrows link nodes representing an expression.

Hat's Redex-Trail Browser: The original tool using Hat's Redex-Trail graph is the Redex-Trail Browser called `hat-trail`. It offers a graphical interface to explore a Redex-Trail interactively. Tracing starts by displaying the program's result or a redex where the evaluation was aborted or failed. The programmer can then choose to see the parent of any expression or any subexpression. Each expression is shown in a separate line. When the programmer chooses to see the parent of an expression it is shown in a line below. Indentation is used to distinguish between parents of an entire expression and parents of subexpressions. Color coding is used to distinguish parents of subexpressions when the parents of several subexpressions of a single expression are shown. Example 2.8 shows a snapshot of a browser session where several parents of expressions and subexpressions are displayed.

Example 2.8 (Snapshot of a browser session for Redex-Trail graph in example 2.7)

```
• 6
  3 * 2
    • fac 2
    ▼ ▼ 3 - 1
  fac 3
  main
```

A parent of an expression is shown in a line below it with the same alignment. The parent of the subexpression `2` in `3*2` is the redex `fac 2` shown with indentation. The parent of the subexpression `2` of `fac 2` is shown with further indentation. □

Browsing strategy: The entire Redex-Trail is usually too large to be displayed at once, even when a short computation is considered only. The programmer therefore needs an exploration strategy. The programmer chooses to see the parent of a (sub)expression whenever he wonders "Where does this expression come from?". If he follows a trace and eventually finds it to be correct he can collapse the entire trace and follow the parents of a different (sub)expression. When the Redex-Trail of an erroneous program is traced he will try to identify erroneous expressions and follow their parents until the cause of the problem is found.

The Hat Structure

In this section a simplified definition of the data structure holding a Redex-Trail graph is given. The Redex-Trail graph is only defined by the implementation of the Redex-Trail source-to-source transformation [SR97a]. A formal definition of the Redex-Trail graph is currently being developed by the ART[6] team in York. An informal explanation of Redex-Trail data structure is presented here. Chapters 3 and 4 require some properties of the Redex-Trail in order to derive two different views from the Redex-Trail graph. These properties are stated below. A formal proof that these properties hold for the implementation of the Hat's Redex-Trail generation is beyond the scope of this dissertation.

Definition 2.4 (Redex-Trail Graph Data Structure)

```
data SrcPos = SrcPos {file :: FilePath; moduleName :: String;
                      line :: Int; column :: Int } |
              NoPos

data Atom   = Id         String SrcPos |
              Constr     String SrcPos |
              CharVal    Char          |
              IntVal     Int           |
              IntegerVal Integer       |
              FloatVal   Float         |
              DoubleVal  Double        |
              ...

data Trace  = Const {value :: Atom;
                     parent :: Trace; src :: SrcPos}  |
              App {fun :: Trace; args :: [Trace];
                   parent :: Trace; src :: SrcPos}    |
              SATA { trace :: Trace }                 |
              SATB { trace :: Trace }                 |
              SATC { trace :: Trace }                 |
              Proj { trace :: Trace; parent :: Trace } |
              Root                                    |
              ...
```

□

[6] Augmented Redex-Trails

The three different data types `SrcPos`, `Atom` and `Trace` are used as follows within a Redex-Trail graph. Variables of these types form the nodes of the graph.

Data type `SrcPos`: The data type `SrcPos` is used to associate expressions with occurrences in the source-code. The module and row/column position are stored in nodes of this type. The constructor `NoPos` is used to indicate that an expression did not occur in the source-code (for example a value that was computed).

Data type `Atom`: Nodes of type `Atom` are used to represent basic values. Nodes for function identifiers (`Id` nodes) and for constructors (`Constr` nodes) hold the identifier or constructor name and additionally have a source reference. The source reference points to the definition of the function identifier or constructor within the source-code. Each basic Haskell type (for example `Int`, `Integer`, `Float`, `Double`, ...) is represented by an its constructor of type `Atom` (for example `IntVal`, `IntegerVal`, ...).

Data type `Trace`: The Redex-Trail graph of a program evaluation is described by a structure of type `Trace`. Several different constructors for the data type `Trace` exist:

- **`Const` nodes:** Basic values in the Redex-Trail graph are not directly represented by an `Atom` node but by a `Const` node which references an `Atom`. This saves space as an `Atom` may be shared by several `Const` nodes with differing parents. The field `src` describes where the value occurred in the source-code or `NoPos` otherwise.

- **`App` nodes:** Applications are represented in the graph by an `App` node. The field `fun` contains a representation of the application's function and the `args` field holds a list of the arguments. Parent and source-code occurrence are given just like for the `Const` node.

- **`Proj` nodes:** When a function application results in a value which was (part of) an application's argument a special node is necessary to describe this projection. The argument value (or the part of it) already has a parent. But the application's result needs to reference the function application as its parent. Hence, an additional parent needs to be attached to the representation of the result. The `Proj` node holds the new `parent` and the original `tracefield` of the returned value.

- **SAT nodes:** The result of a redex is described by one of the following three `SAT` node kinds:

  (a) **`SATA` node:** If the result of the application was never demanded during the evaluation then the result is represented by a `SATA` node. It is called a `SAT` node in state *applied* denoting an unevaluated value. The `SATA` node references the node representing the unevaluated redex.

  (b) **`SATB` node:** If the result is demanded but its evaluation failed or was aborted then the result is given by a `SATB` node representing $\perp$. The node is called a `SAT` node in state *blackholed*. The `SATB` node references the redex which could not be evaluated.

(c) `SATC` **node:** Otherwise, if the result was demanded by the evaluation then the result is given by a `SATC` node. It is a `SAT` node in state *completed*. A `SATC` node references the representation of a value in WHNF[7].

- `Root` **nodes:** The constructor `Root` is used as a null `parent`. The redex `main` of a program is the initial redex to be evaluated and does not have a parent redex. The `parent` field of its graph node is therefore marked with `Root`.

The Hat Redex-Trail graph contains a number of additional node types in order to describe the entire variety of Haskell expressions, for example `if` expressions, guards, lambda abstractions. These nodes are omitted from the description here for reasons of simplicity.

Properties of `App` and `Const` nodes: Some additional properties of the `App` and `Const` nodes are necessary for chapters 3 and 4:

(1) Consider a reduction of a redex r: $r \rightarrow ...name...$, where $r = f\ x_1..x_n$ and $name$ is an instance of a function identifier, constructor or basic value occurring in the body of f's definition. If $name$ is a function identifier it has an arity greater than zero (CAFs are considered separately in (2)) and this instance of $name$ is represented in the Redex-Trail graph by a `Const` node which has the parent r. The `value` field of the `Const` node contains the appropriate `Atom` node describing the function identifier, constructor or basic value.

(2) CAFs are represented by `Const` nodes. Top-level CAFs are evaluated at most once during a program's evaluation, even though they may be used at many places within the program. Therefore they cannot reference a *single* source position or a *single* parent redex. Therefore top-level CAFs do not reference any parent redex and all `Const` nodes representing top-level CAFs have a `parent` field marked by `Root` and a source reference set to `NoPos`.

(3) Consider a reduction of a redex $r = f\ x_1..x_n$: $r \rightarrow ...(g\ y_1..y_m)...$, where $g\ y_1..y_m$ is an instance of an application occurring in the body of f's definition. Then this instance of the application $g\ y_1..y_m$ is represented in the Redex-Trail graph by an `App` node. The `parent` of this `App` node references the node representing the redex $f\ x_1..x_n$.

Figure 2.3 shows part of the Redex-Trail graph of figure 2.2 in more detail, as represented by Hat.

Linearised Redex-Trail graph: Hat stores the Redex-Trail graph in a file allowing other tools to use the data structure. The Redex-Trail graph is linearised when it is written to a sequential file. The nodes are written to the file in the sequence of their creation. This ensures that all nodes only reference earlier nodes, so all references within the file are backwards. The only exception to this case are `SATC` nodes as they refer to a result of a redex's evaluation. The `SATC` node is created immediately after the node representing

[7] Weak head normal form

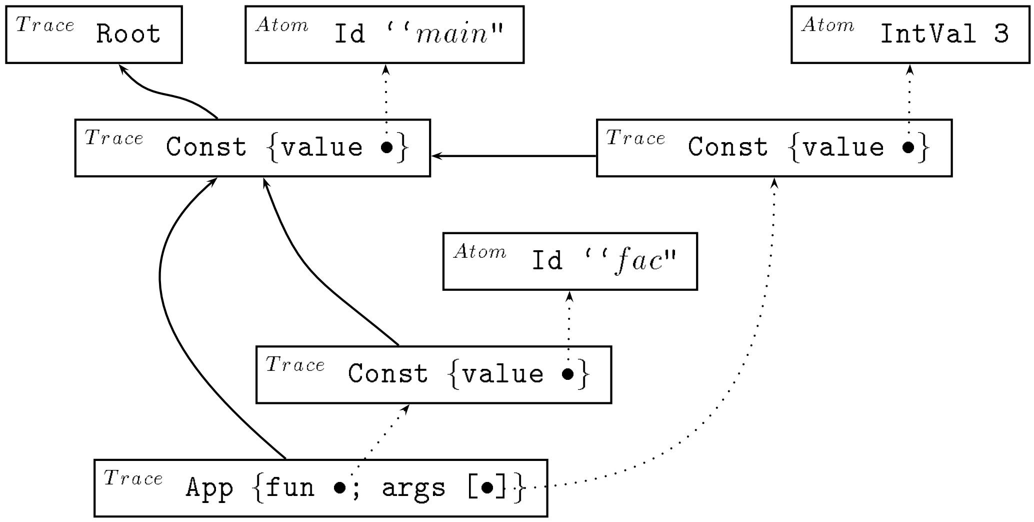

Figure 2.3: Part of the Redex-Trail graph in figure 2.2 showing the details of Hat's graph structure (definition 2.4). (The `parent` field of a node is not explicitly shown but represented by a solid arrow. Source references are omitted.)

the redex. As the redex is only evaluated when demanded, it is always evaluated at a *later* time. The `SATC` node therefore needs to point forward in the linearised Redex-Trail structure.

Advantages of Hat's Redex-Trail:

- The programmer is free to trace the evaluation of any (sub)expression he is interested in. He is not restricted by the tool's guidance.
- When a large and hardly comprehensible data structure is shown, parts of it can be traced selectively. Though the programmer may not be able to decide the correctness of the entire structure he can trace subexpressions of it and analyse their evaluation.
- No modification to the source-code is necessary.
- The Redex-Trail only needs to be generated once and the programmer can use it for multiple tracing sessions.

Disadvantages of Hat's Redex-Trail:

- The tool's interface is more complex than the usage of Algorithmic or Observational Debugging. The programmer first needs to learn about it.
- The programmer may *get lost* when tracing a program as the tool does not offer guidance. This is a problem especially for inexperienced programmers.

- Tracing always starts from the program's result or a point where the evaluation aborted. The programmer cannot start tracing a specific function in which he is interested.

2.5 Summary

Three tracing and debugging views and an example tool for each of these views have been presented. All views have advantages for certain kinds of programs. A number of problems were described which were specific for the presented implementations of the debugging and tracing views:

- Observational Debugging with HOOD suffers from the effort necessary to make observations. The program's source-code needs to be modified.
- Algorithmic Debugging with Freja can hardly be used in practice as only a subset of the Haskell 98 language is supported.

An improved implementation: The problems could be solved by a different implementation of the tools. An implementation of the tools based on the Redex-Trail structure generated by Hat overcomes the need to make any source-code modifications or to rerun the program for every new observation. Hat also supports almost the full Haskell 98 language. Tools based on the Redex-Trail structure can therefore be used for programs in practice.

Complementary tracing views: General advantages and disadvantages of the three tracing and debugging views show that the views are complementary:

- With Observational Debugging the programmer may immediately start observing any suspected function.
  Algorithmic Debugging and the Redex-Trail browser force the programmer to start tracing from a fixed point of the evaluation.
- With Observational Debugging it is difficult to find the cause of an erroneous observation. The programmer needs to inspect the source-code to decide which function to observe next. Eventually the programmer needs to conclude from different observations what really caused the erroneous behaviour.
  Algorithmic Debugging guides the programmer to the cause of an incorrect evaluation.
- With Observational Debugging the programmer needs to decide the correctness of an entire list of observed values or functions at once.
  In Algorithmic Debugging the programmer only needs to decide the correctness of a single equation at a time.
  In the Redex-Trail browser only the correctness of single redexes is considered.
- The use of an Algorithmic and an Observational Debugging tool is easy to learn.
  The Redex-Trail browser needs some experience to be used successfully.

- A programmer may feel restricted by the guidance of an Algorithmic Debugging session.

  In the Redex-Trail browser the programmer decides which parts of the computation to trace. But inexperienced programmers may also get lost in a Redex-Trail and want more guidance.

  In Observational Debugging the programmer is free to observe any value.

A unified approach: These complementary properties of the three tracing and debugging views show that a programmer may benefit from using all three tools as a single tracing toolkit. So far all three tools are implemented in three completely separate systems. Each tool requires special program compilation. With a unified approach, where all three tools are implemented in a single system, the programmer may easily swap between the tracing views. In a single system interaction between the tools can be supported. The programmer may find an erroneous equation or a wrong redex in one tracing tool and may continue to trace its cause with a different tool.

Implementation: Hat generates a data structure, which is stored in the file system, making it a non-volatile entity. It was initially used for the Redex-Trail browser `hat-trail` only. This structure can also be used as a basis of further tracing tools. Chapters 3 and 4 show that it is possible to provide Observational and Algorithmic Debugging based on Redex-Trails. The actual tools are presented in chapters 5 and 6 respectively. Interaction between all three tracing tools is discussed in chapter 7.

Chapter 3

Observation Debugging View

This chapter presents a new approach for Observational Debugging, which is described in section 3.1. The new approach provides a very similar Observational Debugging method to that already provided by HOOD, described in section 2.2. It is shown that all necessary information for these observations can be obtained immediately from Hat's Redex-Trail structure. A basic problem of Observational Debugging as such is discussed in section 3.7 motivating a restriction so that certain functions may not be observed.

3.1 A new approach for observations

With HOOD the programmer has to annotate the source-code of the program in order to indicate which values are to be observed. Several disadvantages of this solution were listed in section 2.2.

Separating observations from the source-code: Instead of annotations in the source-code, a separate tool is introduced, which allows to make a series of observations without modifying and rerunning the inspected program. After the program is run and its Redex-Trail recorded, the programmer starts the observational tool, and states in which values he is interested.

Values to be observed: HOOD allows the programmer to observe any value within the program. The solution presented here however restricts the programmer to the observation of functions. Observations of basic values would simply result in a list of these basic values. In imperative languages programmers often add statements to print the value of a variable. The order of the printed values is then considered to understand program execution. In lazy functional languages the evaluation order is hard to predict. The order of observed values is therefore of little help to the programmer. The observation of a function results in a list of equations. The programmer can consider each single equation and decide its correctness. Hence the restriction to observations of functions is not really a disadvantage. The programmer is especially interested in observing functions.

Observing functions: The observation of functions results in information which is easy to comprehend. All applications of a function that occur during program evaluation are presented to the programmer with their respective results. Each application and its result forms a self-contained equation. When the programmer finds an erroneous equation he may examine the source-code to find the cause. He may also continue to observe further functions, until he finally locates the cause of the program's incorrect behaviour.

3.2 Observations on Redex-Trails

Observation debugging can be provided by observing applications of a particular function identifier. As the Redex-Trail structure (see page 17) represents the reduction steps during the program's execution, all necessary information should be contained within the structure.

- All applications of the program's function identifiers are recorded within the Redex-Trail.
- All results of these applications are stored in the Redex-Trail structure.

Therefore two questions need to be answered, in order to allow observations to be made immediately from the Redex-Trail structure:

1. How can all application nodes of a particular function be found within the structure effectively?
   This question will be answered in section 3.4.
2. How can the result of an application node be found?
   The solution to this problem is given in section 3.3.

3.3 Equations within the Hat Structure

Both new views introduced in this thesis, the Observational Debugging view described in this chapter, and the Algorithmic Debugging view described in chapter 4, need to extract equations from the Redex-Trail graph. In order to form an equation it is necessary to find the result for a given redex within Hat's Redex-Trail graph. It is the purpose of this section that this basic requirement of both debugging views can be fulfilled.

The most evaluated form

In a lazy functional program expressions may not be evaluated at all, may only be partially evaluated or fully evaluated. During the evaluation of an entire program, a redex may become more and more evaluated, as additional parts of its structure are demanded.

When extracting an equation from the Redex-Trail structure, it has to be defined what exactly the result of a redex is. The result of a redex refers to the redex's *most evaluated form*:

Definition 3.1 (Most evaluated form)
(adapted from [NIL98], definition 4.5)
The *most evaluated form* of an expression is its representation once execution has stopped, assuming a lazy language implementation.
If a redex r was reduced to a result v in the *most evaluated form*, it is written $r \Rightarrow_M v$. □

So, the result of a redex does not just depend on its syntactic structure and the program's reduction rules. The result of a redex depends on the context in which the redex is used, as shown in example 3.1.

Example 3.1 (Dependency of a redex's result on its context)

```
fromN   :: Int -> [Int]
fromN n = n : fromN (n + 1)

main    = show (take 1 (fromN 1) ++ take 2 (fromN 1))
```

The redex `fromN 1` is used as an argument to `take 1`. The most evaluated form of this occurrence of the redex `fromN 1` is `[1,_]`. The symbol "`_`" represents an unevaluated expression. A second occurrence of the redex `fromN 1` is an argument to `take 2`. The second occurrence is not shared with the first, so it is evaluated separately, to its most evaluated form `[1,2,_]`. □

A mapping to the result: As the result of a redex depends on the context, there is no mapping from a redex to its result, but a mapping from a redex and its context to its result.

In practice the redex and its context are already described by a particular node within the Redex-Trail structure. Each element in the Redex Trail structure is referred to by a node number within the Redex-Trail file. This node number represents a particular redex in its context.

A well defined mapping "$\Rightarrow_M$" can be given, mapping from one node number representing a redex to another node number representing its most evaluated form within the file. In order not to add overhead to the descriptions, a redex within the Redex-Trail is equated with a particular node number in the file, which represents the redex. We shall continue to refer to the *result of a redex* and write *redex* $\Rightarrow_M$ *result*, bearing in mind that a particular redex including its context is denoted by a node number.

Finding a redex's result

Hat's Redex-Trail graph was initially introduced to support only one view of a Haskell program's evaluation. This is why expressions in the Redex-Trail graph have only backward links to their creating parents (see page 17). Links to traverse the graph in the opposite direction, from a redex to its result, are not included. However, for the two new views introduced it is necessary to extract equations by finding the result of a given redex. A forward link from an expression to its result needs to be found.

A primitive approach: In principle a search for a redex's result is possible by searching the entire graph structure. All nodes which have a parent link to the original redex can be identified. It could then be determined which of these nodes are substructures of the redex and which node represents the entire redex. By iterating this search the most evaluated form of the redex can be found. But this solution would be inefficient and is not feasible.

An efficient approach: Even though the formal definition of the Redex-Trail graph (page 17) does not include a link from a redex to its result, such a link is implicitly given when the graph structure is linearised and written to a file. This fact was initially revealed by analysing the low-level file structure.

For each application or CAF node representing a redex a `SAT` node is introduced within the Redex-Trail graph. The `SAT` node represents the redex's result. The Hat runtime system recording a program's evaluation creates the `SAT` node immediately after a new redex is recorded. Linearisation ensures that those nodes are written one after another to the file. This way a link is implicitly given from the low-level representation of the graph, by considering the node immediately following a redex in the file. Evaluation results are represented by `SAT` nodes in the Redex-Trail graph (see page 18). The implementation of Hat's Redex-Trail generation guarantees that `SAT` nodes are stored in the Redex-Trail file in the following way:

- To save space the `SAT` node for an application is omitted if it is identical to the `SAT` node of the application's parent.
- Otherwise the `SAT` node representing the result of an application/CAF node is stored in the Redex-Trail file immediately after the application/CAF node.
- Functions (for example Prelude functions) may be considered *trusted* during the computation in which case applications in their bodies are not recorded in the Redex-Trail graph. But `SAT` nodes representing the results of unrecorded applications are still stored in the Redex-Trail. But these `SAT` nodes are specially marked. They can be distinguished from `SAT` nodes belonging to an application. This is important as a *lonely* `SAT` node may incidentally be written immediately after an application node, which does not have an own `SAT` node.

Given this information it is easy to define an algorithm to find the result of a redex in the Redex-Trail file:

Algorithm 3.2 (Finding the result of a redex)

getResult ($\mathcal{A}$)
  if `isRedex`($\mathcal{A}$) **then**
    $\mathcal{B}$ = `hatSequentialNext`($\mathcal{A}$)
    if `isSAT`($\mathcal{B}$) **then**
      return $\mathcal{B}$
    else
      return `getResult(hatParentFor(`$\mathcal{A}$`))`

□

Augmented Redex-Trail graph

Instead of using the low-level information described in the previous section, an explicit link could be included in the graph, immediately pointing from every redex to its most evaluated form.

The original definition of the Redex-Trail graph (page 17) is modified to the *Augmented Redex-Trail graph* [WCBR01]. The new structure unifies an application or CAF node with its resulting `SAT` node to a single node.

This new solution is not yet implemented in Hat[1]. The current version of the basic interface to the Redex-Trail structure (section 8.2) uses the original definition and finds the results of redexes as explained (page 26).

3.4 Finding the Application nodes - the Needles in the Haystack

How are applications of a given function identifier found within the Redex-Trail graph?

Searching along the graph's links: One way to search the Redex-Trail graph is by following the links of the graph. The program's result is the result of the CAF `main`, which can be determined as described in section 3.3.

A full search of the Redex-Trail graph can be done by following all links within the graph, starting from the program's result. All redexes involved in the evaluation of the program's result can be visited this way. Applications of a certain function may be found in any part of the graph, so no restriction of the search is possible.

Disadvantages: Attempting to search the entire graph along its links does have disadvantages. Multiple searches of subgraphs have to be explicitly avoided as sharing of structures occurs in the graph. Some parts of the graph may even be inaccessible due to a problem described in section 4.5 (page 52). The main problem with this approach arises from the very low level implementation. Searching the entire graph along its links would mean following from one node to the next by jumping all over the file. As the entire structure may be too large to be held in memory, this would be inefficient.

Sequential searching: A better solution is to scan the entire file containing the Redex-Trail graph and to check every stored node in a sequential order. Reading a large file sequentially can be implemented very efficiently, compared to the random access. A sequential scan through the file visits all application nodes of the Redex-Trail graph exactly once. It has only to be determined which of these nodes represent applications of the given function identifier. However, there is a vast number of application nodes to be considered. Therefore checking a node needs to be very efficient.

[1]This thesis refers to the release 1.10 of Hat. A coming release of Hat is already planned to support the modified structure.

What an application looks like

How an application is represented in the Redex-Trail graph can be seen from the graph type definition 2.4 (page 17). Applications in the Redex-Trail graph consist of the following three nodes:

- a node `Id` *identifier* of type `Atom` holding the name of the identifier explicitly. There is exactly one `Atom` node within the entire Redex-Trail graph for each function identifier. It is shared by all its applications.
- a node `Const` of type `Trace` which references the `Atom` node for the function identifier. There can be many `Const` nodes for each function identifier.
- a node `App` of type `Trace` referencing one of these `Const` nodes with its field `fun`.

An example of a subgraph describing such a simple application is shown in figure 3.1.

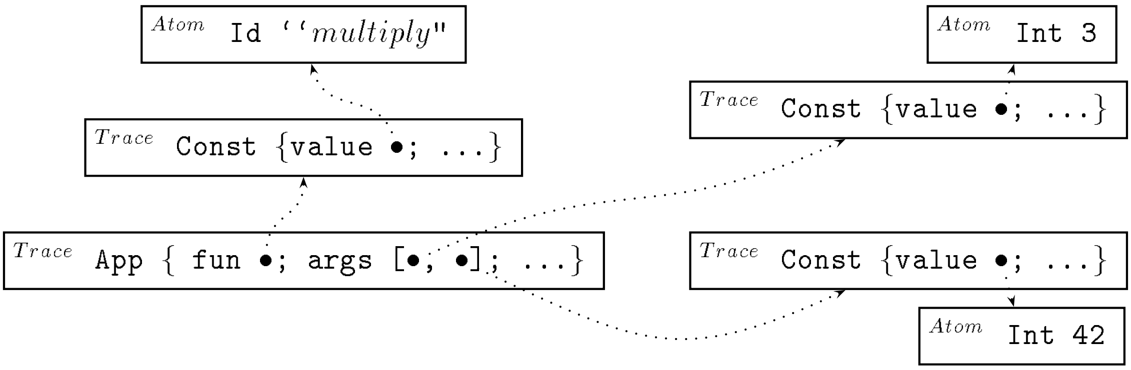

Figure 3.1: Structure of the application `(multiply 3 42)`. Nodes to the left describe the application of `multiply`, nodes to the right the arguments to the application.

Applications of a slightly more complex structure may consist of several nested application nodes as shown in Figure 3.2. The only difference to the simple example above is that the outer application node now references an application node, which represents a partial application.

Searching the Redex-Trail

The following nodes need to be found:

1. the unique node `Id` *identifier* holding the name of the given function identifier,
2. all `Const` nodes referencing the above `Id` node with their `value` field,
3. all application nodes (`App` nodes) referencing any of these `Const` nodes as their function. An application node's function is linked by a special field named `fun`.

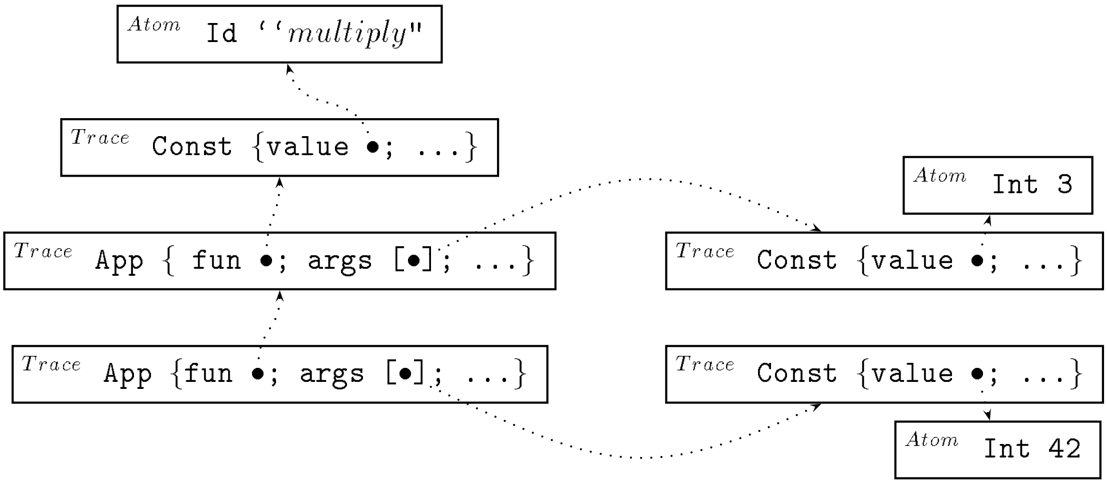

Figure 3.2: Simplified structure of the application `((multiply 3) 42)`

4. all application nodes with a `fun` link to any other application node, that in turn matches criteria 3 or 4.

It might appear that several separate scans through the entire Redex-Trail file are necessary.

- a first scan to find the unique `Id` node for the identifier.
- a second scan to find all `Const` nodes referencing the `Id` node.
- a third scan would allow to find all applications referencing the found `Const` nodes.
- additional scans if necessary. The scanning for application nodes needs to be iterated, as every scan may find application nodes referencing other application nodes found in the previous iteration. Eventually the iteration stops, when no additional application nodes are found.

Searching for applications in a single scan: However, the Redex-Trail graph is linearised when it is written to the file. The linearisation ensures. Each node in the file can only reference earlier nodes within the file. The one exception to this rule is discussed in "Searching forward links" (page 32). A single scan through the entire file will suffice. The nodes in the file are read sequentially.

- When the `Id` node for the identifier is found, its address is stored. `Const` nodes referencing this `Id` cannot exist in the file before this `Id` node. Hence, the search can continue from this point.
- When a `Const` node matching criterion 2 is found, its address is stored. All application nodes referencing any such node must appear in the file at a point beyond it. Again the search can always continue from the position where the node was found.

- When an `App` node is found, which has a function reference to any of the nodes stored, then the `App` node's address itself is stored as well. This way all application nodes matching criteria 3 or 4 are found.

Eventually all matching application nodes are found and the list of the applications with their results can be presented to the programmer.

Checking nodes efficiently

Checking whether a `Const` or `App` node matches any of the criteria means checking to which node its `fun` link points. Within the file representing the Redex-Trail graph these links are simply addresses. If the `fun` link matches any address of any previously found node, then the address of the `Const` or `App` node containing the link is stored as well.

The Redex-Trail graph may contain a large number of nodes. Some functions might have been applied very often during the program's evaluation. So the number of matching application nodes might be large. This leads to the problem where a large number of nodes has to be compared to a large number of addresses to see whether the nodes reference any of the previously found addresses. This is a standard problem which can be solved efficiently by using a hash table to store the addresses. Both storing a new address and looking up whether a node references any previously found address can be done in almost constant time. The table is enlarged and rehashing is used to keep the utilization of the table below 90%. This ensures a low average number of steps necessary to store and find addresses. According to [WIR86] less than 6 average search steps are necessary assuming an evenly distribution of the addresses.

3.5 Higher-order functions

In section 3.4 only identifier, constant and application nodes were considered. An additional node kind is now considered separately, as it needs to be treated differently to the other nodes.

Functions may return higher-order values. A higher-order value in turn can be used as a function in an application node.

Example 3.2 (A function with a higher-order result)

```
h     :: Int → Int
h x y = x + y

g     :: Int → (Int → Int)
g x   = h x

main  :: Int
main  = (g 5) 6
```

□

In example 3.2 the application `g 5` returns a higher-order result. This result is applied to the argument `6`. The corresponding application node in the Redex-Trail graph does not match the examples considered in section 3.4, but uses an additional kind of node. The Redex-Trail graph of this application node is given in figure 3.3.

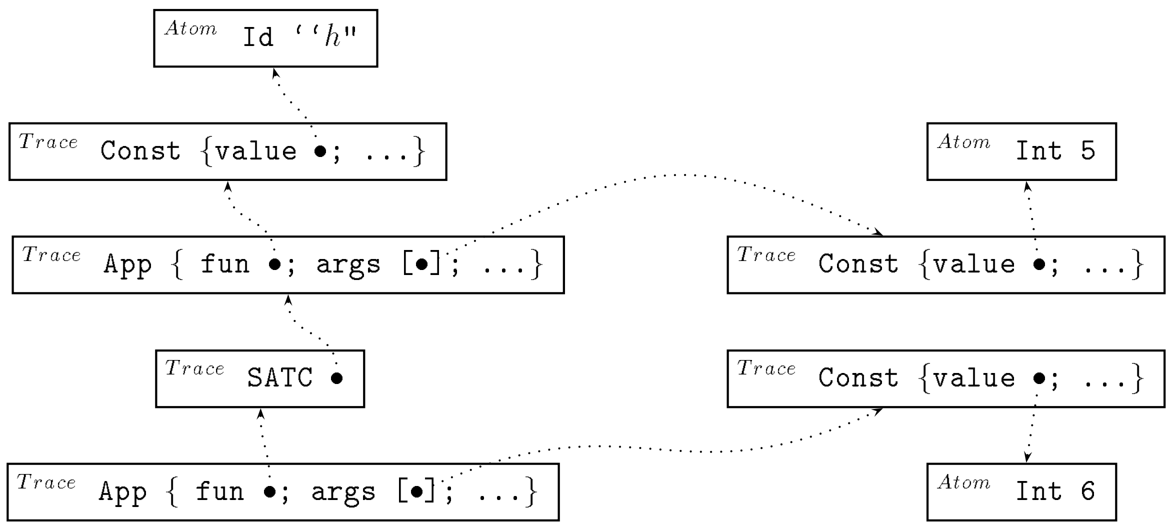

Figure 3.3: Partial Redex-Trail graph for example 3.2, showing the expression `(h 5 6)` where `(h 5)` was a higher-order result.

Applications referring to `SAT` nodes: When the application node with argument `6` is created, the result of `g 5` is unevaluated. The function of the application node therefore refers to a `SAT` node instead, which represents a placeholder for the result of the saturated application `g 5`. When `g 5` is evaluated, its corresponding `SAT` node is updated, and refers to the node representing the result. (see example figure 3.3).

There are three kinds of `SAT` nodes. The function of an application node may reference all three of these `SAT` nodes, apart from the nodes already considered in the previous section 3.4:

- An application node may reference a `SATA` (*saturated application in applied state*) node as its function. This means its function was not evaluated when the execution stopped. But as the application's function was never evaluated, neither can the application itself have been evaluated, as redexes are evaluated in a leftmost-outermost strategy. Hence, as the application node itself is unevaluated, it cannot contribute to an observation of a particular function identifier. The application node is ignored for the observation.

- The function of an application node may be represented by a `SATB` (*saturated application in blackholed state*) node. This means the evaluation of the application's function has failed and resulted in $\perp$. Therefore the application represents an application of $\perp$ to some arguments, with $\perp$ as result. An equation "$(\perp\ a_1..a_n) = \perp$" does not contribute to the observation of any function identifier. Such application nodes are ignored for the observation.

- An application node may reference a `SATC` (*saturated application in completed state*) node as its function. This means its function was indeed evaluated and resulted in a higher-order value. This higher-order value is in turn referenced by a link of the `SATC` node. Application nodes of this kind may contribute to the observation if the higher-order value equals the searched function identifier or is a partial application of it.

Searching for applications of `SATC`: Hence, the only extra case which has to be taken into account is where an application node references a `SATC` node. Unfortunately `SATC` nodes cannot be dealt with in the same way as the other kinds of nodes. A `SATC` node represents the result of an application, and the result is always evaluated *after* the `SAT` node is created. For this reason, `SATC` nodes are the only kind of nodes which may have forward links to nodes within the *linearised* Redex-Trail. The fact that the other kinds of nodes could only have backward references has until now allowed us to search the file sequentially.

Searching forward links: To solve this problem a modification has to be made to the purely sequential approach. Following every `SATC` node's forward link would be costly, as there is a large number of such nodes and following a `SATC` nodes may involve jumping within the file. Instead `SATC` nodes can be ignored during the sequential scan. Only if an application node is found, which indeed references the `SATC` node with its function link, the forward reference of the `SATC` node is checked. This will be the case for a small minority of the `SATC` nodes only, as most `SATC` nodes will describe first-order values and not be referenced as an application's function[2].

3.6 Observation of oversaturated applications

It can be seen in example 3.2 that `main` was defined as `(g 5) 6` rather than `g 5 6`. Both expressions are semantically equivalent. Unfortunately these two expressions lead to different Redex-Trails when evaluated.

Saturated application: The brackets in `(g 5) 6` cause the creation of two saturated application nodes in the Redex-Trail as shown in figure 3.3. The observation of the identifier `g` according to the description "Searching forward links" on page 32, finds the appropriate application node representing `g 5`. As `g 5` was reduced to `h 5` the equation `g 5 = h 5` is observed.

[2]The proportion will certainly vary with the style of the program. More higher-order functions will involve more of such cases.

Oversaturated application: The expression `g 5 6` represents an oversaturated application, as `g`'s arity is one. The second argument is an argument to the higher-order result of `g 5`. When `g 5 6` is evaluated the Redex-Trail does not record an application node for `g 5`. As the brackets are missing only a single oversaturated application node for `g 5 6` is created. The observation of the identifier `g` finds this oversaturated application node. The equation `g 5 6 = 11` is observed. Hence, expressions bracketed in different ways may result in different observations.

Splitting oversaturated applications: The problem is introduced when the Redex-Trail is recorded. Instead of recording an oversaturated application, Hat should split an oversaturated applications into several applications. But this requires Hat to check the arity of a function identifier when creating an application, in order to decide where the application list needs to be split. A different solution is to avoid representing applications to multiple arguments in single nodes, and to have application nodes of a single argument only. Both solutions would make the Redex-Trail recording less efficient and were therefore not implemented. The inconvenience of slightly different observations caused by different uses of brackets remains.

3.7 Local Function Definitions

Haskell allows the local definition of functions. In a function definition within the body of another function definition local variables may be used. Example 3.3 shows a problem that arises from such local function definitions.

Example 3.3 (Example with a locally defined function)

```
f x y = g x
  where g x = x + y

main = [(f 1 1), (f 1 2), (f 1 3)]
```

Observing function `f` results in the expected behavior but observing `g` results in the following equation list :

```
g 1 = 2
g 1 = 3
g 1 = 4
```

□

The observation in example 3.3 clearly violates the idea of a well defined function. But the real problem is that the shown applications on the LHS of the equations are not applications of the same function. These are applications of three different functions, which only have the same name.

Grouping equations: It is possible to group equations so that each group only consists of applications of the same function. So in the equations in example 3.3 could be divided into three separate groups, each consisting of a single equation. But this solution would not help at all in deciding whether any of these equations are correct or not.

Observing the context: In order to decide the correctness of equations including a locally defined function, it is necessary to show all local variables defined in the same local context as the local function. The observations in example 3.3 should be presented with their local context, which states the value of the local variable y. The resulting observations are shown in example 3.4.

Example 3.4 (Observations with context)

```
g 1 = 2 where y = 1
g 1 = 3 where y = 2
g 1 = 4 where y = 3
```
□

Extending the context: The problem with local functions is even worse if functions of the same name are defined locally within the body of different functions. In this case it is not enough to show the value of local variables used. The particular function in which the local function was defined has to be added to the context, as shown in example 3.5.

Example 3.5 (Multiple local functions definitions of the same name)

```
f x y = g x
  where g x = x + y

h x y = g x
  where g x = x + y + 5

main = [(f 1 1), (f 1 2), (h 1 1), (h 1 2)]
```

Observations of g need to show the full context, including the functions they were defined in.

```
g 1 = 2 within f, where y = 1
g 1 = 3 within f, where y = 2
g 1 = 7 within h, where y = 1
g 1 = 8 within h, where y = 2
```
□

Missing context information: Currently Hat does not record any information about the value of local variables in the Redex-Trail graph. Hence, the necessary context information allowing to decide about the correctness of local functions is not available. Showing equations of local functions without giving this information could mislead the programmer, as he might not be aware of the problem. Therefore the observation of local functions is disabled. This is a drawback of the current Observational Debugging tool.

A programmer's workaround: Observations will at least allow the programmer to isolate an error within particular top-level function. Once an erroneous top-level function is found he may continue by moving some of its local functions to the top-level. Local variables used by the former local functions may be passed as arguments instead. This workaround however, spoils one of the original goals for the new approach of observation debugging. It forces the programmer to make modifications to the source-code, in order to observe the functions.

3.8 Summary

This chapter has shown the following results:

- Hat's implementation of the Redex-Trail graph supports efficient access to a redex's result, though this is not immediately apparent in the high-level definition of the graph. Extracting equations from the Redex-Trail graph is a basic requirement for the Observational Debugging view, and for the Algorithmic Debugging view presented in chapter 4.

- A new approach for Observational Debugging was presented. The approach allows the observation of applications of a particular function identifier. The observation results in a list of equations, where each equation has an application of the function identifier on the LHS and its respective result on the RHS. A programmer can identify a function to be erroneous if any of its applications resulted in an erroneous result.

- The new Observational Debugging approach can be implemented efficiently, using the information given in Hat's Redex-Trail structure only. Modifications to the source-code in order to make observations are unnecessary with this approach.

Though a major inconvenience of the original observation library HOOD was overcome, some problems persist:

- Only the observation of functions is permitted, whereas HOOD allowed the observation of any value. This is a minor disadvantage, as the observation of basic values is less help compared to observing functions.

- The observation of local functions is not permitted with the new approach, as necessary information about local variables is currently not recorded in the Redex-Trail. HOOD does allow observations of any function or value, but it does not show the context of such functions. It is up to the programmer to interpret information about local functions correctly. If the programmer is not aware of the problem he may be misled by the observations.

- An observation of all applications of a particular function identifier may result in a large number of equations. Flooding the programmer with too many equations can make it hard to decide whether a function is erroneous or not. Hence, a tool is needed to select only those applications in which the programmer is really interested. Several ways to reduce the number of equations are presented in chapter 5. Still, in some cases this problem will remain.

- The programmer is also flooded with information when equations have large data structures on the left- or right-hand side. With large and complex data structures involved, it is hardly possible to determine whether such an equation is correct or not. But this is a fundamental problem of all debugging tools. If the programmer is not able to decide whether a function application resulted in the correct result or not, all tools are of little help. In this case the programmer needs to use a different program or different program input resulting less complex data structures.

A difference concerning unevaluated subexpressions exists between observations in HOOD and those derived from the Redex-Trail structure. HOOD's observations show exactly which parts of the expressions were demanded at the *point of observation*. Expressions in Redex-Trails are shared. Observations derived from the Redex-Trail therefore show how far the expressions were evaluated in the context of the *entire program*. Observations of HOOD are more precise in that respect.

Chapter 4

Algorithmic Debugging View

Algorithmic Debugging as implemented by Freja (page 14) has shown its usefulness. It is desirable to allow Algorithmic Debugging of Haskell programs by using the information given in the Hat Redex-Trail structure only. Hat already supports almost full Haskell 98 and can be made independent of the underlying Haskell system. It is supported by NHC[1] and will be available for GHC[2] soon.

This chapter shows that Algorithmic Debugging based on Redex-Trails is possible:

- The dependency between equations already mentioned in section 2.3 is formally defined.
- The Evaluation Dependency Tree (EDT), which is the basic data structure for Algorithmic Debugging (as described in section 2.3), is formally introduced.
- It is shown that all necessary information to build the EDT is stored in the Redex-Trail structure.
- Several problems in deriving the EDT from the Redex-Trail structure are described and solutions are discussed.
- An algorithm is presented which efficiently builds an EDT from the Redex-Trail structure.

4.1 The Evaluation Dependency Tree

As presented in section 2.3 the evaluation of a lazy functional program can be described by a series of equations. Each equation represents the reduction of a redex to its most evaluated form. A reduction to the most evaluated form may depend on the further reductions, which in turn describe equations. Therefore a dependency between the equations is given. This dependency is now formally defined:

[1]Nearly a Haskell Compiler
[2]Glasgow Haskell Compiler

Definition 4.1 (Direct evaluation dependency)
(Based on [NIL98])
Let $f\ x_1..x_m$ be a redex for some function identifier f (of arity $m \geq 0$)[3] with arguments $x_i, 1 \leq i \leq m$.
Suppose $f\ x_1..x_m \rightarrow ...(g\ y_1..y_n)...$ is a reduction step given by a rule for the outermost function identifier f. So $g\ y_1..y_n$ is an instance of an application occurring in the body of f's definition. The expression $g\ y_1..y_n$ is a redex if g is a placeholder for a function identifier or any expression, resulting in a higher-order value. g (of arity $n \geq 0$) is applied to the arguments $y_i, 1 \leq i \leq n$. Should the $g\ y_1..y_n$ redex ever become reduced, then the reduction of the $f\ x_1..x_m$ redex is *directly evaluation dependent* on the reduction of the $g\ y_1..y_n$ redex. □

Given this definition of dependency a tree structure can be defined, which orders the equations of a program evaluation:

Definition 4.2 (Definition of Evaluation Dependency Tree, EDT Node)
An *Evaluation Dependency Tree* consists of EDT *nodes*.
Each *EDT node* consists of a label given by an equation. An equation is given by a redex and the result of the reduction to its most evaluated form.
An *EDT node* $\mathcal{B}$ is a child of an *EDT node* $\mathcal{A}$, if the reduction labeling the node $\mathcal{A}$ is *directly evaluation dependent* on the reduction labeling the node $\mathcal{B}$.
The root node of the EDT is an *EDT node* labeled with the reduction of the main CAF to its result. □

4.2 Deriving an EDT

The EDT definition (definition 4.2) shows that three kinds of information need to be obtained from the Redex-Trail structure in order to construct an EDT:

1. The label for each node is to be found. A node is labeled with an equation. The equation's LHS consists of a redex, its RHS is the result of the redex's reduction to its most evaluated form.

2. The root node of the tree needs to be determined.

3. All children of each node need to be found. An initial solution is presented in this section (page 39) and a more efficient solution is presented in section 4.3.

Labeling a node: Labeling an EDT node means to find the *most evaluated form* of a given redex. The Redex-Trail structure does not immediately give a link from a redex on the LHS to its result of the reduction on the RHS. But as shown in section 3.3 it is possible to find a result given a redex. Furthermore, all arguments of an application on the LHS

[3] CAFs are implicitly included in this definition, as functions of zero arity are allowed.

need to be in their most evaluated form. This is easy to achieve as redexes stored in the Redex-Trail structure already fulfill this property. When a redex is evaluated during program execution, it is substituted by its reduction result. The original redex is only referred to by a parent link which points from the new value to the original redex.

An EDT node is already determined by the LHS redex of its label. The RHS can be derived from the LHS. To simplify the following descriptions, EDT nodes are referred to by the LHS redex only - or by referring to the node within the Redex-Trail structure which represents this LHS redex. Hence, the following definition:

Definition 4.3 (Direct evaluation dependency of redexes and Redex-Trail nodes)
If the reduction of a redex r is directly evaluation dependent on the reduction of a redex s, the redex r is *directly evaluation dependent* on the redex s.

Suppose two nodes $\mathcal{A}$ and $\mathcal{B}$ within the Redex-Trail structure represent redexes r and s respectively:

If the reduction of the redex r is *directly evaluation dependent* on the reduction of the redex s, the node $\mathcal{A}$ is called *directly evaluation dependent* on the node $\mathcal{B}$. □

The root node of an EDT: By definition the root node of the tree is a node labeled by the program's main CAF. As the main CAF is the first redex to be demanded in a computation of a program, and the Redex-Trail file is written incrementally, its representation can easily be found at the very beginning of the Redex-Trail file.

Obtaining the children of an EDT node

Finding the children for an EDT node is a more complex task. The description is divided into the following steps:

- It is initially assumed that all redexes are represented by application nodes. The special case for CAFs, which are represented by `Const` nodes, is separately considered in section 4.5.
- It is shown that the direct evaluation dependency on equations consisting of an application on the LHS is easy to decide.
- A primitive approach to build an EDT is presented.
- A more sophisticated and efficient way to build an EDT is given in section 4.3.

Lemma 4.4 (Deciding direct evaluation dependency for application nodes)
Suppose a node $\mathcal{A}$ represents a redex $f\ x_1..x_n$ $(n \geq 0)$ and a node $\mathcal{B}$ represents a redex $g\ y_1..y_m$ $(m \geq 1)$ within the Redex-Trail structure.

Then the application node $\mathcal{B}$ has a parent link to $\mathcal{A}$ if and only if the reduction of redex $f\ x_1..x_n$ is *directly evaluation dependent* on the reduction of redex $g\ y_1..y_m$. □

Proof:

($\curvearrowleft$) Suppose the reduction of $f\ x_1..x_n$ is directly evaluation dependent on the reduction of $g\ y_1..y_m$.

According to the definition of *direct evaluation dependency* (definition 4.1) $g\ y_1..y_m$ must be an instance of an application within the body of f's definition. According to (3) the application node $\mathcal{B}$ has a parent link to the node $\mathcal{A}$ representing its creating redex.

($\curvearrowright$) Suppose a redex $g\ y_1..y_m$ ($m \geq 1$) is represented by an application node $\mathcal{B}$. Its parent link must reference some node $\mathcal{A}$.

According to (3) the `parent` link of application node $\mathcal{B}$ references its creating redex. The creating redex must be a CAF or an application of a function identifier f. So, the application node $\mathcal{B}$ representing $g\ y_1..y_m$ is an instance of an application within the body of f's definition. According to the definition of *direct evaluation dependency* (definition 4.1) the reduction of the redex represented by $\mathcal{A}$ is evaluation dependent on the reduction of the redex $g\ y_1..y_m$. □

Identifying redexes: Lemma 4.4 assumed that application nodes represent redexes. But application nodes may represent constructor applications or unsaturated function applications, which are not redexes. Identifying constructor applications is easily done by considering the outermost application symbol. Unsaturated applications can be identified by considering their result: if no `SAT` node exists for the application then the application is not a redex.

Algorithm: An algorithm can now be formulated which uses lemma 4.4 to decide whether the reduction of a node $\mathcal{A}$ is evaluation dependent on the reduction of a node $\mathcal{B}$:

Algorithm 4.5 (Deciding direct evaluation dependency)

isDirectEvaluationDependent ($\mathcal{A}$, $\mathcal{B}$)
  if `isRedex(`$\mathcal{B}$`)` **then** — avoid constructor applications
    return (`parent(`$\mathcal{B}$`)` `==` $\mathcal{A}$) — identify *direct evaluation dependency*
— according to lemma 4.4
  else
    return `False`
  fi

□

Primitive method of building an EDT: With lemma 4.4 and the resulting algorithm 4.5 the goal of finding all EDT children for a given redex is almost achieved. For a redex represented by a node $\mathcal{A}$, all EDT children can be determined by considering all nodes in the entire Redex-Trail structure. Algorithm 4.5 can identify all nodes on which the evaluation of $\mathcal{A}$ directly depended.

Considering all nodes within the Redex-Trail structure is technically easy to achieve by scanning the entire Redex-Trail file. But in contrast to the task which is presented in section 3.4 a solution with a sequential scan is not acceptable in this case.

Disadvantages of the primitive approach: In practice, it is necessary to determine parts of the EDT according to the programmer's decisions. Each search for the children of a particular node would involve searching the entire structure. Scanning a large trace file whenever the children of a node need to be determined would not allow the programmer to explore the EDT interactively with an Algorithmic Debugger. Responses would be too slow.

This shows the necessity of a different approach. And a much better solution where only a small part of the Redex-Trail has to be searched can be achieved, as described in section 4.3.

4.3 Efficient EDT generation

The primitive approach in section 4.2 searches the entire Redex-Trail file in order to find all redexes on which a given redex is *directly evaluation dependent*[4]. But considering the entire Redex-Trail graph to find the children of a single EDT node is too expensive.

In [PPRS01] it was proposed to *reverse* the whole Redex-Trail graph in order to generate an EDT, which also is an expensive operation. A better solution is to generate the EDT piecemeal. This can be done with a search in a limited part of the trace using the links in the Redex-Trail graph. This search method is described in this section.

For the presented algorithm it is an essential requirement that it is possible to find starting from the *most evaluated form* of a redex, the original (unevaluated) redex, using only the links given in the Redex-Trail graph. (The case where an intermediate step of the reduction results in a CAF, $redex \to^* caf \to^* result$, is not considered here. Due to a specific problem with CAFs this is described separately in section 4.5.)

Definition 4.6 (Ancestor)
A node $\mathcal{A}$ in the Redex-Trail graph is called an *ancestor* of a node $\mathcal{B}$

- if $\mathcal{A}$ is the parent of $\mathcal{B}$ or
- if the parent of $\mathcal{B}$ is a node $\mathcal{C}$ and $\mathcal{A}$ is an ancestor of $\mathcal{C}$. □

Lemma 4.7 (Ancestors of a given most evaluated form)
Suppose a node $\mathcal{B}$ represents the *most evaluated form* of a redex represented by $\mathcal{A}$, and no intermediate step in the evaluation $\mathcal{A} \Rightarrow_M \mathcal{B}$ is a CAF.
Then the node $\mathcal{A}$ is an *ancestor* of the node $\mathcal{B}$. □

Proof:

(1) Given a redex r represented by a node $\mathcal{R}$. The first evaluation step of r in the Redex-Trail results in an expression e represented by a node $\mathcal{E}$. As e cannot be a CAF there are two cases:

[4] Note that direct evaluation dependency was extended from reductions to redexes in definition 4.3.

- e may be a subexpression of r In this case the node $\mathcal{E}$ is a node of type `Proj`, which references the subexpression with its `tracefield` field and its `parent` field references the node $\mathcal{R}$.
- otherwise e must be an application, which is not part of the redex r. So, the application is introduced by the reduction of r. Therefore $\mathcal{E}$ must be an application node, which references the node $\mathcal{R}$ with its `parent` field.

So in all cases the parent of node $\mathcal{E}$ is node $\mathcal{R}$.

(2) The *most evaluated form* of the redex a represented by the node $\mathcal{A}$ is the expression b represented by the node $\mathcal{B}$. So, the redex a is evaluated in n evaluation steps to its *most evaluated form* b. By using (1) n times it follows that $\mathcal{A}$ is an ancestor of $\mathcal{B}$. □

Given this lemma 4.7 an algorithm can be described, which finds the EDT children of a redex more directly.

Algorithm to find all EDT children of a given redex r

(a) First the *most evaluated form* e of the given redex r is considered. If e equals r then e is already most evaluated and does not depend on the evaluation of any other redex. Otherwise r was reduced to e. Then the node $\mathcal{R}$ representing r is an ancestor of the node $\mathcal{E}$, which represents e, as was shown in lemma 4.7. By searching all ancestors of $\mathcal{E}$ a node $\mathcal{C}$ can be found, which has a parent reference to the node $\mathcal{R}$. If the node $\mathcal{C}$ represents a redex c (and not a basic value, a constructor application or an unsaturated function application) then r is *directly evaluation dependent* on r as was shown in lemma 4.4.

(b) According to the definition of *direct evaluation dependence* (definition 4.1) r may be *directly evaluation dependent* on subexpressions which are created by the reduction of the outermost function of r. But all subexpressions of the redex c are already in the *most evaluated form*. According to lemma 4.7 the original subexpressions created by the reduction of r are ancestors of their *most evaluated forms*. So, all ancestors of all subexpressions of c need to be searched.

(c) The search in step (b) has to be done recursively. Whenever a node is found, which has a parent link to the node $\mathcal{R}$, all ancestors of its subexpressions need to be searched.

Searching in vain: When searching the ancestors of a subexpression not created by the original redex r, none of its ancestors has a parent link to $\mathcal{R}$. But the search will eventually stop as the top-most ancestor is always a top-level CAF which does not have any further parent.

Avoiding multiple searches: In practice nodes are never searched more than once. Hence, shared arguments are only searched once. This limits the problem of searching subexpressions in vain. All visited nodes are stored in a hash table. Before the algorithm considers a node, it checks whether the node is already in the hash table and therefore does not need to be searched again. This check with the hash table is omitted in the simplified algorithm shown here.

So the following algorithm, finds the EDT children of a node more effectively:

Algorithm 4.8 (Finding EDT children)

```
edtChildren (parent)
  e = MostEvaluatedFormOf(parent);
  if (parent == e) then
    return []
  else
    return (searchForEDTchildren(parent, e));

searchForEDTchildren (parent, e)     — search all ancestors of e
  if (ParentOf(e) == parent) then
    childrenList = []
    if (isApplicationNode(e)) then
      childrenList = searchForEDTchildren(parent,
                                          ApplicationNodesFunction(e)))
      for i = 1 to ApplicationNodesArity(e) do
        childrenList = append(childrenList,
                              searchForEDTchildren(parent,
                                                   ApplicationNodesArgument(e, i)))
      od
    fi
    if (isRedex(e)) then
      childrenList = append(childrenList, [e])
    fi
    return childrenList
  else                                       — search further ancestors
    p = ParentOf(e)
    if (p ≠ Root) then
      return (searchForEDTchildren(parent, p))
    else
      return []
    fi
  fi
```
□

4.4 Higher-Order Examples

The case of higher-order functions has not yet been addressed specifically. Algorithm 4.8 finding all EDT children for a given redex does work for these cases, but an additional node kind of the Redex-Trail structure needs to be considered.

A current problem of the Redex-Trail structure is shown, which leads to the generation of an incorrect EDT. The problem can be solved by a slight modification to algorithm 4.8.

Example 4.1 (Two examples involving higher-order functions)

```
h        :: Int → Int → Int → Int
h x y z  = x + y + z

j        :: (Int → Int → Int) → Int
j f      = f 2 3

g        :: Int → (Int → Int)
g x      = h x 2

(a) main1    = j (h 1)
(b) main2    = (g 1) 3
```

The reductions of `main1` and `main2` are as follows:

```
(a) main1 → j (h 1) → h 1 2 3 → 1 + 2 + 3 → 6
(b) main2 → (g 1) 3 → h 1 2 3 → 1 + 2 + 3 → 6
```

The Redex-Trail graphs for both examples `main1` and `main2` are given in figure 4.1 (a) and (b) respectively. The graphs are simplified as follows: all basic values are included in the constant and application nodes. Only the application, constant and `SATC` nodes are shown. The field names of all nodes are omitted. Constant nodes are only labeled by their value, application nodes are labeled by their function reference to the left and their arguments to the right. □

First example `main1`: The first example in 4.1, the evaluation of `main1`, involves a higher-order argument which is passed to the function `j`. The resulting EDT is shown in figure 4.2 (a). The EDT can be derived from the Redex-Trail graph in figure 4.1 (a) as follows:

- The root node for the EDT is immediately given by the reduction of the program's main CAF (for this example `main1`).
- The children for `main1` can be found by using algorithm 4.8. The algorithm starts by considering `main1`'s result node representing `6`. All ancestors of the node `6` are considered until the node which has the parent `main1` is found. This is the node representing `j (h 1)`, a redex and its reduction `j (h 1)` $\Rightarrow_M$ `6` is therefore included in the EDT.

- All subexpressions of `j (h 1)` are searched for nodes having the parent `main1`. The nodes for `j` and for `h 1` have the parent `main1` but they are not redexes: `j` is a constant for a function symbol and `h 1` is an unsaturated application. The search also considers `h` itself, but it is not a redex.

- Algorithm 4.8 is used again to find the children of the EDT node `j (h 1)` $\Rightarrow_M$ `6`. Starting from its result node `6` the ancestors are searched. The node representing `h 1 2 3` has a parent link to `j (h 1)` and is saturated, so it is an EDT child of `j (h 1)`.

- The subexpression `h 1` is considered, but it does not have the parent `j (h 1)`. Its ancestor is the CAF `main1`, which does not have a parent, hence the search is stopped.

- The only EDT child `1 + 2 + 3` of `h 1 2 3` is found in the same way.

- `1 + 2 + 3` does not have any EDT children. Its result node `6` is considered and it does have the parent node `1 + 2 + 3`. But `6` is not a redex.

Second example `main2`: The second example in 4.1 is the evaluation of `main2`. Note the brackets in the definition `main2 = (g 1) 2`. Even though they seem unnecessary, but they are given for a reason revealed in the section "Oversaturated Applications" (page 46). The Redex-Trail graph for this evaluation is shown in figure 4.1 (b). The EDT for this example is given in figure 4.2 (b). The EDT can be derived from the Redex-Trail graph as described before, but an additional node needs to be considered.

- The root node of the EDT is given by the reduction of the CAF `main2`.

- The ancestors of `main2`'s result node `6` are considered. The node representing the application of `3` to `h 1 2` does have the parent `main2` and so `h 1 2 3` $\Rightarrow_M$ `6` is the first EDT child of `main2`.

- The subexpressions of `h 1 2 3` are considered. The function link of the application node references a `SATC` node, a possibility not considered in algorithm 4.8. The `SATC` node indicates an expression that is the result of a saturated application. So the algorithm only needs to skip this node by considering the node to which the `SATC` node points. The referenced node representing `h 1 2` does not have the parent `main2` so its parent is considered. The node `g 1` has the parent `main2`, is a redex and results in `h 1 2`. The reduction `g 1` $\Rightarrow_M$ `h 1 2` is therefore included in the EDT as a child of `main2`'s reduction.

- The remaining child `1 + 2 + 3` $\Rightarrow_M$ `6` shown in figure 4.2 (b) is derived in the same fashion.

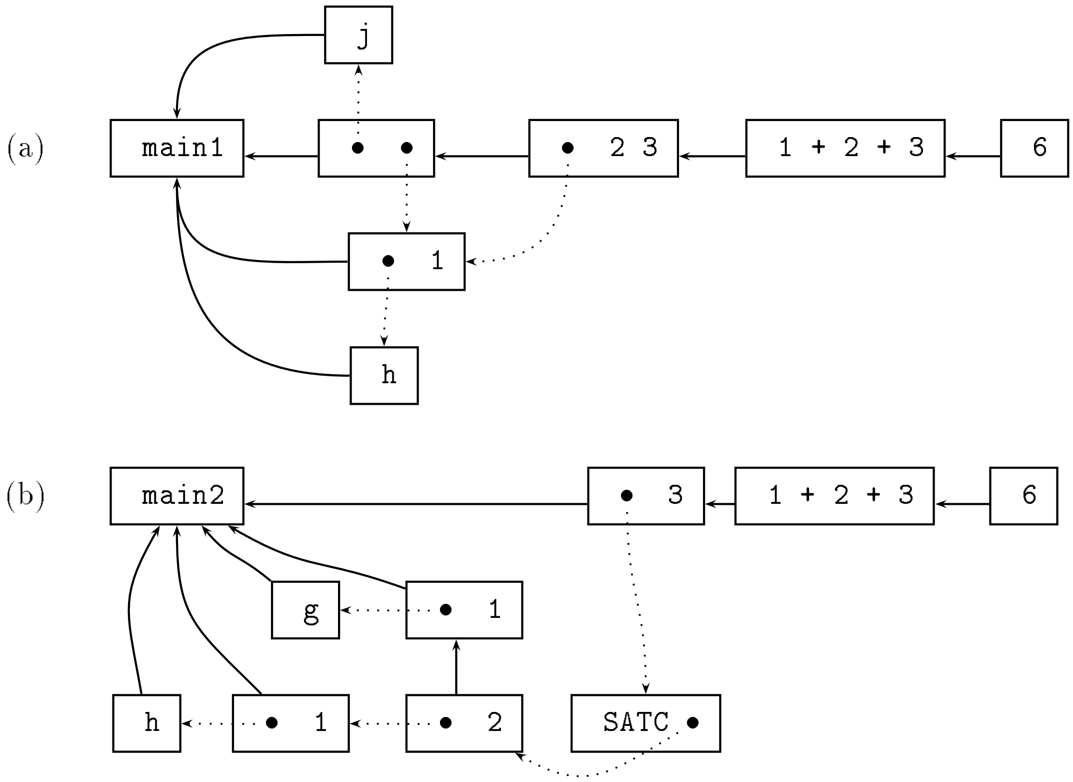

Figure 4.1: Redex-Trail structures for the evaluation of `main1` and `main2` in example 4.1

Oversaturated Applications

In this section a specific problem of the current Redex-Trail structure is discussed, which is illustrated by example 4.2.

Example 4.2 (Oversaturated application)

```
h       :: Int → Int → Int → Int
h x y z = x + y + z

g       :: Int → (Int → Int)
g x     = h x 2

main    = g 1 3
```

The reduction of `main` is as follows:
`main → g 1 3 → h 1 2 3 → 1 + 2 + 3 → 6`

This example is almost identical to the one given in example 4.1 (b) for `main2`. The subtle difference is missing brackets. In this example `main` is defined as `g 1 3`; in example 4.1 `main2` was defined as `(g 1) 3`. □

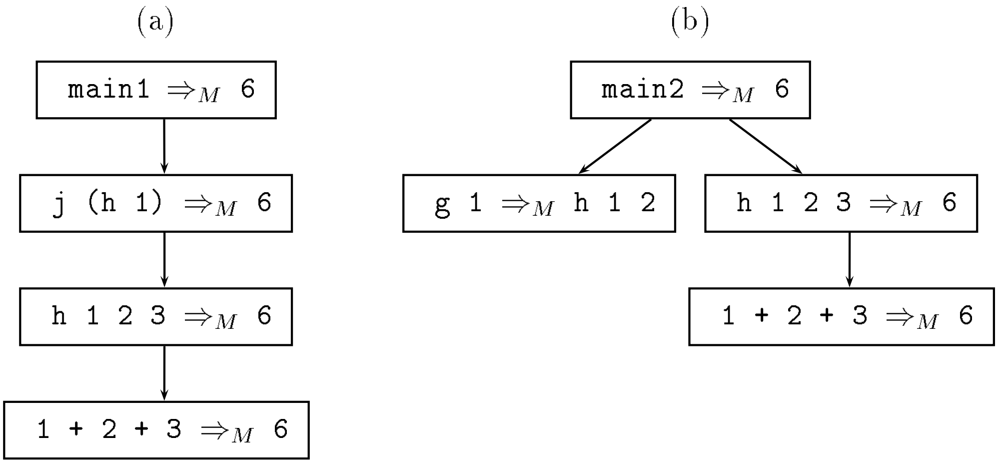

Figure 4.2: Two EDTs (Solid arrows represent *direct evaluation dependency*)

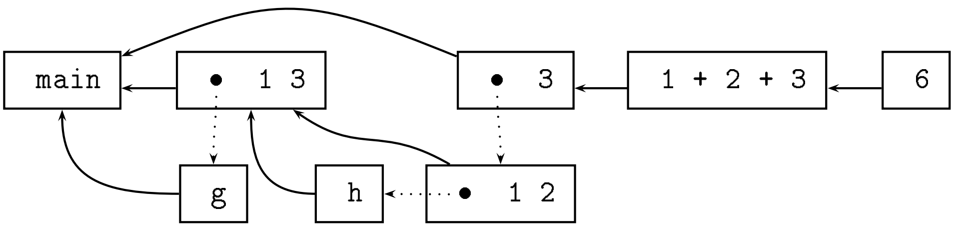

Figure 4.3: Redex-Trail structure for evaluation of main in example 4.2

Oversaturated application: The two examples 4.2 and 4.1 (b) only have a minor syntactic difference. Unfortunately this syntactic difference leads to a different structure of the Redex-Trails, as can be seen by comparing figures 4.1 (b) and 4.3. The problem arises as Hat uses the syntactic structure of the program to create the Redex-Trail graph. The brackets in example 4.1 (b) cause the creation of two separate application nodes, one for the redex `g 1` and one for the application to the remaining argument `3`. The brackets are missing in example 4.2 and Hat does not detect that `g 1` is already a saturated application. Instead an *oversaturated* application node for `g 1 3` is created.

During the evaluation `g 1` is reduced to `h 1 2`. An application node for `h 1 2` is created in the Redex-Trail graph. The remaining argument `3` needs to be applied to this application. Therefore a *new* application node is created which applies the remaining argument to `h 1 2`.

Splitting an application: The Redex-Trail graph in figure 4.2 is not quite correct. The application `g 1 3` is strictly speaking not a redex. Instead, the application needs to be split into two parts. But as Hat is currently not aware of function arity, applications cannot be split correctly. A correct solution for the Hat system is already planned, but not yet implemented. The EDT generation therefore has to deal with oversaturated application nodes.

Incorrect EDT: Algorithm 4.8, which finds EDT children for a given node, results in an incorrect EDT when used on the Redex-Trail in figure 4.2. This incorrect EDT is shown in figure 4.4 (I) and can be compared to the correct EDT in figure 4.2 (b). Instead of `g 1` $\Rightarrow_M$ `h 1 2` the reduction `g 1 3` $\Rightarrow_M$ `6` is shown for the example with the oversaturated application. But this is not really a problem of the EDT generating algorithm. There is no node in the Redex-Trail in figure 4.2 which represents the redex `g 1`. It is therefore impossible to somehow derive the reduction `g 1` $\Rightarrow_M$ `h 1 2` from the Redex-Trail graph.

Fixing the EDT: The correct EDT cannot be derived from the Redex-Trail graph in figure 4.2. But it is possible to lessen the problem by rearranging the EDT nodes. A major problem in the EDT in figure 4.4 (I) is that the evaluation of `g 1 3` does not depend on any other evaluation. Rearranging the EDT nodes as shown in figure 4.4 (II) reflects the fact that the evaluation of `g 1 3` depends on the evaluation of `h 1 2 3`. This still does not match the *direct evaluation dependency* of definition 4.1, but it helps an Algorithmic Debugging tool using the EDT if the programmer spots an error in the evaluation of an oversaturated application. Without this modification the EDT would never include any dependencies of an oversaturated application evaluation.

Rearranging the EDT node: The problem now is to rearrange the reduction of `h 1 2 3` as an EDT child to the reduction of `g 1 3`. In normal circumstances the reduction of `h 1 2 3` is an EDT child of the reduction of `main`. But it is to be rearranged as the application to the argument `3` was oversaturated. The inner application node of `h 1 2 3` is the application `h 1 2`, with the parent `g 1 2`. If algorithm 4.8 considered the parent of the whole application `h 1 2 3` to be the parent of the inner application then the node is already rearranged as shown in figure 4.4 (II).

But by default the parent of a redex is the parent of the outermost application. So, it needs to be determined exactly when this exceptional rule applies.

Deciding when to rearrange the EDT: It is possible to detect that the application to `3` in the redex `h 1 2 3` was created to represent an oversaturated application. When considering an application the following can be stated about the parents of the outer and inner application nodes:

- If an inner node and an outer node have the same parent, these nodes were created by the reduction of the same redex.
- An inner node may belong to an *earlier* redex then the outer application. So a part of an *earlier* redex was passed as an argument to some function. This case can be detected as the parent of the inner node is an ancestor of the outer, but the parent of the outer node is not an ancestor of the inner.

For example see figure 4.1 (a). When the application of `h 1 2 3` is considered in this figure, the parent of `h 1` is `main1`, which in turn is an ancestor of the outer application node of `h 1 2 3`. But the parent of the outer application node is `j (h 1)`, which is not an ancestor of `h 1`. Therefore `h 1` must be a part of a redex which was created earlier than the outer application of `h 1 2 3`.

- An inner node may belong to a *later* redex then the outer application. So the parent of the outer node is an ancestor of the inner node, but the parent of the inner node is not an ancestor of the outer node. So the inner node was created by the evaluation of a redex, which in turn was created by the reduction of the outer application. But in this case the inner node cannot be referenced immediately by the outer application. When the outer application node was created, the value of the inner node was not evaluated. Therefore the outer application references a `SAT` node, which is a placeholder for an unevaluated value. When this value is evaluated then the `SAT` node is updated to reference the result.

  For example see figure 4.1 (b). The outer node representing the redex `h 1 2 3` references a `SATC` node, which in turn has a link to `h 1 2`. The inner application `h 1 2` has the parent `g 1`, which is not an ancestor of the outer node.

Considering the example in figure 4.2 reveals that an oversaturated application causes an exception to the default Redex-Trail structure. The outer application node of `h 1 2 3` has the parent `main` which is an ancestor of the inner node `h 1 2`. So, the inner node is created by a later redex than the outer node. But in contrast to the default case above, the inner node is not referenced by an intermediate `SATC` node. The reason for this is that the evaluation of `g 1 3` causes the creation of the redex `h 1 2`. For the application to the argument `3` a new application node is created which copies the remaining arguments from the original oversaturated application node, in order to apply them to `h 1 2`. The redex `h 1 2` already exists and is therefore referenced immediately.

Detection Rule: The rule to fix the EDT problem with oversaturated applications is as follows:

Whenever the EDT algorithm 4.8 considers a parent of an application node, it is compared to the parent of its inner node if the inner node is not behind a `SATC` node. If the parent of the outer application is an ancestor of the inner node's parent, then the inner node's parent is chosen instead. If nested inner nodes exist then their parents are compared in the same way, in order to find the parent which is not an ancestor of any other node.

4.5 Treatment of CAFs

So far *constant applicative forms* (CAFs) have not been considered, but they might have been included as normal applications with zero arity. The actual representation of a CAF in the Redex-Trail structure however, is different to the representation of an application. This different representation demands a special treatment of CAFs to arrange their reduction correctly within an EDT.

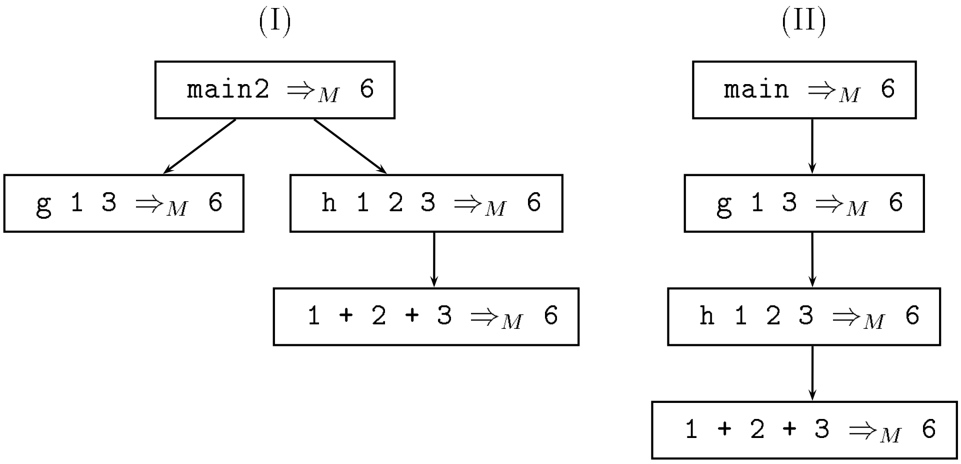

Figure 4.4: EDT for example 4.2: (I) is an incorrect EDT, (II) a *fixed* version of the EDT. (The correct EDT is the one shown in figure 4.2 (b).)

This section will present the following:

- A short discussion explaining why CAFs are represented in a different way.
- Two solutions are given to the problem of how CAFs can be considered in an EDT.
- A specific deficiency of the current Redex-Trail implementation is described and a possible solution is given.
- An algorithm is given which considers CAFs according to one of the presented solutions.

Top-level CAFs are Orphans

A difference between functions and CAFs originates from the way Haskell lazily evaluates CAFs. As the result of a CAF does not depend on any argument, each CAF is evaluated at most once by Haskell, even though the CAF might be used at different places within the source-code. In contrast, distinct function applications with identical arguments are earch evaluated separately.

So there is at most a single representation of a top-level CAF within the Redex-Trail structure. CAFs are not represented by an application node with no arguments, but by a special type of node. The question arises where the parent link of a node representing a CAF points. The idea of the Redex-Trail is to have a link from every redex to the redex which created it. But CAFs are an exception as each CAF is only represented once, but may be used at several points within the program. It seems there may be many parent nodes for a CAF. As a node can have only one parent at a time, it was defined that a node representing a top-level CAF does not have a parent. The parent link of a CAF is marked with a special value `root`.

Arranging CAFs within an EDT

As the node representing a top-level CAF does not have a parent, its creating redex is not referenced. Lemma 4.4 (Deciding evaluation dependency for application nodes) did not consider CAFs, as it needed the parent of a redex to decide the evaluation dependency. Therefore algorithm 4.5 (deciding direct evaluation dependency) does not consider any redex to be evaluation dependent on a top-level CAF.

There are at least two ways to deal with reductions of CAFs in an EDT:

1. **A forest of trees:** A first solution is instead of having a single EDT which describes the entire computation of the `main` CAF, to rather have one EDT for each top-level CAF. The computation of the entire program is then represented by a *forest of trees*. This approach was chosen for the Freja system (see page 14).

   Disadvantages:

   - There are dependencies between the CAFs. The evaluation of a CAF may depend on the evaluation of other CAFs. Therefore an ordering of these EDTs has to be introduced. An Algorithmic Debugging tool using these EDTs must verify the correctness of each EDT in this order. The EDT having `main` as its root node would always be the last to be verified. In some cases two or more CAFs may mutually depend on each other, making a total ordering impossible. In this case this approach can lead to incorrect results of the debugging tool. The tool could find an erroneous node within the EDT for one CAF, whereas the *real* error is within another, not yet considered.
   - A second disadvantage is that the user of a debugging tool is forced to answer questions about the correctness of all CAF reductions, before starting to debug the `main` CAF. In some cases, these answers are in vain, if no part of the `main` EDT depending on these CAFs is considered.
   - The previous problem gets worse if the user only intends to debug a small part of a computation. Suppose he only wishes to debug the evaluation of the redex `insert 5 [1,2,6,7]`, which occurs within the evaluation of a complex program. The Algorithmic Debugging tool has no way to tell if this redex depends on any top-level CAF. But not to verify the top-level CAFs has the danger of potentially finding a bug within the computation of `insert` when the bug is in an incorrect CAF. On the other hand, verifying all CAFs means that the user has to answer many questions about CAFs of a large program, whereas `insert` might not even depend on any of the CAFs.

2. **A single tree:** A different solution is to stick to a single EDT and to include the CAFs within `main`'s EDT. The definition of *direct evaluation dependency* (definition 4.1) already includes the case of CAFs, as functions of zero arity (both arities n and m mentioned in the definition may be zero) are allowed. The obvious advantage of this approach is that the EDT now reflects the real dependencies of the evaluation.

A single EDT represents the entire computation of the program. A tool using the EDT for an Algorithmic Debugging session only asks about the validity of a CAF's reduction when a reduction really depends on the CAF's value. This solution is presented on page 53.

Disadvantages:

- As a CAF may be used multiple times within the program, it can appear at multiple places within the EDT. This can lead to a case where the user of a debugging tool is asked several times about the validity of the same CAF. This problem has to be avoided by the debugging tool using the EDT, as described in section 6.2.

An advantage of Hat: The previous discussion shows an advantage of the Hat Redex-Trail system, which generates a stored structure representing the entire computation. Debugging tools are only invoked on this structure after the entire computation has finished. The tools can use this structure without interfering with the original computation of the program itself. The Freja system could not choose this approach because debugging and program evaluation were done at the same time. Including CAFs at multiple places within the EDT would therefore mean to evaluate the CAFs multiple times. Multiple evaluation of CAFs in Haskell programs is too severe a penalty to take.

A deficiency of the current Redex-Trail

A specific problem with CAFs exists where the current Redex-Trail structure does not support construction of the entire EDT. The following example shows such a case:

Example 4.3 (Example showing a specific problem with a CAF)

```
const4 :: Int
const4 = 2 * 2

f      :: Int -> Int
f x    = const4

main   = 1 + (f 1)
```

The Redex-Trail graph (figure 4.5) for the evaluation of `main` shows a severe problem, which cannot be overcome with the current Redex-Trail structure. The result of the CAF `const4` is referenced by the application node for `1+4`. However, there is no possiblity to get from the result node for `4` back to the application `f 1`, as the CAF `const4` does not have a parent. It cannot be determined from the Redex-Trail graph that the value `4` resulted from the evaluation of `f 1`. The node for `f 1` is unreachable. There is no link from *any* node within the Redex-Trail structure pointing to `f 1`. □

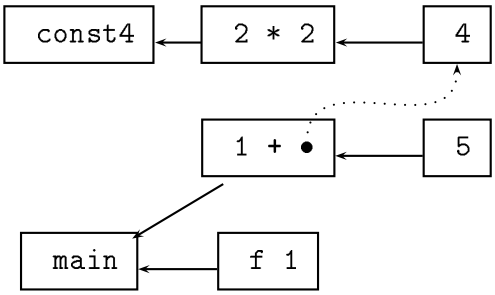

Figure 4.5: Redex-Trail structures for evaluation of main in example 4.3

Problem in general: The problem occurs whenever the following situation is given:
$a \to^* caf \Rightarrow_M b$
That is, a reduction of a redex a to its result b involves an intermediate CAF redex. In the Redex-Trail structure, the nodes representing redexes for the reduction steps between a and the CAF are unreachable and cannot be considered in the EDT.

A simple fix: The evaluation of $a \Rightarrow_M b$ is evaluation dependent on the reduction of caf, but not necessarily *directly* evaluation dependent. The reductions involved between a and caf cannot be considered in the EDT. In order to lessen the problem the evaluation of caf can be lifted in the EDT, by considering the evaluation of caf to be an EDT child to the evaluation of $a \Rightarrow_M b$.

A problem for the Redex-Trail browser: This problem is not a specific problem of the task to derive an EDT from a Redex-Trail structure. Rather, this problem shows a deficiency of the current definition of the Hat Redex-Trail structure. Hat's Redex-Trail browser (see page 16), which is the original tool for tracing on the Redex-Trail structure, suffers from the very same problem.

A possible solution: A solution to this problem is to modify the Redex-Trail generation so that the result of a CAF is never referred to immediately. Instead of using the CAF's result a new type of node is introduced, which refers both the CAF's result and the redex which introduced the occurrence of the CAF. A similar modification is already planned for Hat's Redex-Trail structure, but not yet implemented.

Building an EDT with CAFs

Two ways were presented (page 51) to treat CAFs in Algorithmic Debugging. The best solution is to include CAFs within the EDT according to the evaluation dependency. However, there is no perfect implementation of this solution, as some necessary information cannot be obtained from the Redex-Trail structure. Therefore some flaws have to be accepted. In practice these flaws are minor and do not cause severe problems.

Example 4.4 (Examples involving CAFs)

$$
\begin{array}{lll}
\texttt{const}4 & :: & \texttt{Int} \\
\texttt{const}4 & = & 2 * 2 \\
\\
\texttt{g} & :: & \texttt{Int} \rightarrow \texttt{Int} \\
\texttt{g x} & = & 1 * \texttt{const}4 \\
\\
\texttt{h} & :: & \texttt{Int} \rightarrow \texttt{Int} \\
\texttt{h x} & = & \texttt{x} * \texttt{x} \\
\\
\texttt{j} & :: & \texttt{Int} \rightarrow \texttt{Int} \\
\texttt{j x} & = & \texttt{const}4 * \texttt{x} \\
\\
\texttt{main}1 & = & \texttt{g}\ 1 \\
\texttt{main}2 & = & \texttt{h const}4 \\
\texttt{main}3 & = & \texttt{j const}4
\end{array}
$$

The Redex-Trail structure for the evaluation of `main1`, `main2` and `main3` is shown in figure 4.6 (a), (b) and (c) respectively.

First example `main1`: As can be seen in figure 4.6 (a) the representation of the CAF `const4` does not include a link to the redex which created it. What can be told from the structure though, is that `const4`'s result is an argument to the application node representing `1*4`. The application node has a parent indicating which redex created it. An assumption can be made that the redex creating the application node for `1*4` created the CAF `const4`, too. It can be seen from `g`'s definition that the assumption is correct in this case. But the second example shows a case where this assumption is wrong.

Second example `main2`: The Redex-Trail graph for the second example (figure 4.6 (b)) is similar to the first one. Again the result of `const4` is an argument to an application node (see node representing `4*4`). But this time the occurrence of `const4` is passed as an argument to the function `h`. The assumption that a CAF is created by the same redex as the application in which its result is referenced is wrong in this case: The result of `const4` is an argument to the application node `h 4` *and* `4*4`. The assumption is that the parents of `h 4` *and* `4*4` are both evaluation dependent on the evaluation of `const4`. But only `main2`, which is the parent of `h 4`, is evaluation dependent on `const4` as can be seen in the definition of `main2` and `h`.

Third example `main3`: The third example is similar to the second. The structure of its Redex-Trail graph (figure 4.6 (c)) is the same as for the second example. But this time the CAF `const4` is introduced within the body of the function `j`'s definition *and* it is introduced in the definition of `main3` and passed as an argument to the application of `j`. The assumption is correct in this case. Parents of both application nodes where `const4`'s result is used are evaluation dependent on `const4`. □

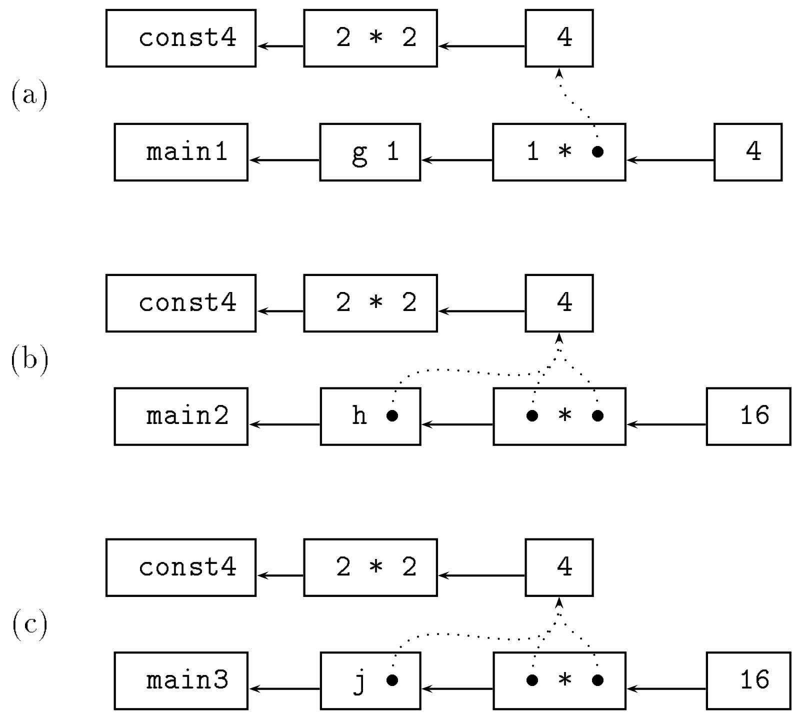

Figure 4.6: Redex-Trail structures for evaluation in example 4.4

Assuming a parent for CAFs: As described on page 50 there is only a single representation of a CAF in the Redex-Trail and a CAF may therefore have many parents. But the Redex-Trail structure does not record what the parent redex of a CAF is. The parent is needed to determine the evaluation dependency. If the most evaluated form of a CAF is referenced by an application node (as an argument, or in case of higher orderness as the function) then it is *assumed* that the parent of the application node is a parent of the CAF, too.

Flaws in the assumption: As shown by example 4.4 the assumption is not always correct and may lead to wrong EDTs. But the flaws in the EDT are minor. The following two flaws may occur:

- A reduction of a redex may incorrectly be considered evaluation dependent on a CAF, where the CAF is actually an EDT child of an *earlier* EDT node.

This can only happen if the CAF is passed as an argument to a function. Example 4.4 for `main2` explained this case: importantly the evaluation of the CAF is also considered an EDT child of the correct reduction.

Consequence: In practice the EDT is used by an Algorithmic Debugger, which considers reductions in a innermost-redex-first order (see section 4.7) and avoids asking questions about identical equations multiple times (*memoisation*: see section 6.2). As the evaluation of the CAF in this case is indeed an EDT child of an *earlier* EDT node, it is considered at the correct place *first*. Even though the CAF's evaluation is considered again at a wrong place, the memoisation of the Algorithmic Debugger ensures that the CAF is not presented again. The flaw of the EDT is not visible in practice.

- A reduction of a redex may incorrectly be considered evaluation dependent on a CAF, where the CAF is actually an EDT child of a *later* EDT node.

  This can only happen if the evaluation of a redex resulted in the CAF. So, the parent of the application node is not *directly* evaluation dependent on the evaluation of the CAF, but on the evaluation of another redex. The evaluation of this redex in turn depends on the evaluation of the CAF.

  An example: Example 4.3 and figure 4.5 (page 53) illustrate this problem. With the assumption that the CAF `const4` has the same parent as the application node `1+4` the redex `main` seems to be *directly* evaluation dependent on the evaluation of `const4`. But in this case `main` is *directly* evaluation dependent on the evaluation of `f 1`, which in turn is *directly* evaluation dependent on `const4`. As described in section 4.5 (page 52), it is impossible to reach the node for `f 1` and the only workaround is to skip the intermediate reductions.

  Consequence: This flaw of the EDT can be noticed in practice, and it is the proposed fix to the problem described in section 4.5. Due to this problem a part of the EDT which dependent on the CAF's evaluation cannot be derived. The CAF is therefore lifted in order not to lose too much information.

The assumption about the parent of a CAF therefore only introduces minor flaws to the EDT. Algorithm 4.8, which searches for all EDT children of a given EDT node, can be modified to include CAFs.

Modification to the algorithm building the EDT: Algorithm 4.8 searches the Redex-Trail structure in a specific way: the argument references and the function reference of an application node are only searched for further EDT children if the parent of the application itself is the given parent redex.

So, whenever the algorithm finds a CAF, it must have been an argument (or function in case of higher-orderness) to an application node with a parent link to the given parent. Hence, with the introduced assumption, the CAF is considered to have the same parent and is therefore an EDT child of the given parent redex.

Only a simple modification needs to be done to algorithm 4.8 (page 43):

Algorithm 4.9 (Modification to Algorithm 4.8 to include CAFs)

...
else — search further ancestors
p = *ParentOf*(e)
if (p ≠ Root) **then**
return (searchForEDTchildren(parent, p))
else
return [e]
fi
fi

The modification of algorithm 4.8 is very simple. Only the last return statement was changed. Instead of returning the empty list, the CAF itself is returned as an EDT child. □

4.6 Treatment of Local and Trusted Functions

Local function definitions: Section 3.7 discussed problems that occur when applications of a locally defined function are considered. As a programmer cannot decide the correctness of such applications, they need to be omitted. Omitting the entire branch of the EDT would be too severe. The EDT children of the omitted equation have to be rearranged, in order not to lose valuable information for the debugging tool. The rearrangement of the EDT children is described in "Rearranging EDT nodes" (page 58).

Lambda abstractions: There is a similar problem for lambda abstractions. Hat represents lambda abstractions in the Redex Trail by nodes of type `lambda`. These nodes currently only carry information about the source position where the lambda abstraction is defined. It is possible to access the source-code of the program to extract information about the definition of the lambda abstraction. But this solution raises the same problem as for the local function definitions, as lambda abstractions may use local variables.

Again, there is not sufficient information for the user to decide a questions about applications of lambda functions. These applications have to be omitted. Their EDT children are rearranged within the EDT as described on page 58.

Trusted functions: Hat distinguishes two kinds of functions: *trusted* and *untrusted* functions.

- User-defined functions are likely to be the reason for erroneous program behaviour. User-defined functions are therefore considered *untrusted* and their entire reductions are recorded in Hat's Redex-Trail graph.
- The idea is to consider functions *trusted* if they are known to be implemented correctly and cannot be responsible for erroneous program behaviour. By default

only functions defined within Haskell's Prelude are considered *trusted*. Redexes of *trusted* functions in *untrusted* function definitions are still recorded in the Redex-Trail, but redexes of *trusted* functions used within *trusted* function definitions are omitted. This considerably decreases the size of the Redex-Trail graph.

Omitting reductions of trusted function applications: When a programmer is trying to find an error in his program he is not interested in the reduction of *trusted* function applications. Questions like "*Is 18 * 75 = 1350 correct (Y/N):*" only annoy the programmer using the debugging tool, as the function symbol "*" is almost certainly correctly implemented in the Prelude and therefore *trusted*. And a program is likely to use many *trusted* functions defined in Haskell's Prelude. The programmer should not be flooded with endless questions about correct equations. To avoid this, reductions of *trusted* function applications are omitted in the EDT. However, such reductions of *trusted* function applications can still be evaluation dependent on applications of *untrusted* functions, as shown in example 4.5. Their EDT children need to be rearranged in the EDT (see page 58).

Example 4.5 (Simple example showing trusted code depending on untrusted code.)

$$\begin{aligned}&\texttt{untrusted} &&:: \texttt{Int} \to \texttt{Int}\\ &\texttt{untrusted x} &&= \texttt{x} + 1\end{aligned}$$

$$\texttt{main} = \texttt{map}\ \texttt{untrusted}\,[1,2,3]$$

The function `map` is defined in the Prelude and of course to be trusted. The application `map untrusted [1,2,3]` is an application of a trusted function. Its evaluation depends on the evaluation of three *untrusted* function applications: `untrusted 1`, `untrusted 2` and `untrusted 3`. The equations `untrusted 1` $\Rightarrow_M$ `2` etc. need to be included in the EDT. □

Rearranging EDT nodes

So in several cases reductions cannot or should not be included in the EDT. Pruning the EDT could render a debugging tool using the EDT almost useless, as too much information about the evaluation is lost. Instead of pruning the entire branch of the EDT only the node itself needs to be omitted, while all its children are lifted by one level. So far, the *direct evaluation dependency* (definition 4.1) was used to define the child relation between EDT nodes. As some nodes of the original EDT shall be omitted and their children are lifted, a new relation between the EDT nodes is defined:

Definition 4.10 (Hat evaluation dependency)
Let $\mathcal{R}$ and $\mathcal{S}$ be two reductions.
The reduction $\mathcal{R}$ is called *Hat evaluation dependent* on the reduction $\mathcal{S}$,

- if $\mathcal{R}$ is *direct evaluation dependent* on $\mathcal{S}$.
- or if a reduction $f\ x_1..x_m \Rightarrow_M r$ $(m \geq 0)$ exists where f is a trusted function identifier, a local function identifier or a lambda abstraction, $\mathcal{R}$ is *direct evaluation dependent* on $f\ x_1..x_m \Rightarrow_M r$ and $f\ x_1..x_m \Rightarrow_M r$ is *Hat evaluation dependent* on $\mathcal{S}$. □

Note the recursive nature of the definition. This ensures that reductions are lifted even over a series of recursive applications of trusted functions, local functions, and lambda abstractions.

This is the actual dependency used to build the EDT from the Hat Redex-Trail. The EDT built is called the *Hat EDT* and is now formally defined. In the definition redexes of trusted function, local function and lambda applications can be omitted, as their children were rearranged within the EDT in definition 4.10.

Definition 4.11 (Hat Evaluation Dependency Tree)
The *Hat Evaluation Dependency Tree* is a tree structure over EDT nodes.

- The root node of the Hat EDT is the node labeled with the reduction of the `main` CAF.
- A node labeled $f\ x_1..x_m \Rightarrow_M r$ $(m \geq 0)$ is considered an EDT child of a node labeled with $s \Rightarrow_M s'$ if the reduction $f\ x_1..x_m \Rightarrow_M r$ is *Hat evaluation dependent* on $s \Rightarrow_M s'$ and f is a top-level and untrusted function identifier. □

4.7 Ordering the EDT children

So far no specific ordering of the children of an EDT node was given. However, the ordering of the children influences the default sequence in which an Algorithmic Debugger presents questions about these nodes to the programmer. Two orders are reasonable:

Outermost redex first: Ordering the EDT children so that the evaluation of outermost redexes is shown first may surprise the programmer. Outermost redexes often have *wrong* arguments resulting from the erroneous evaluation of an inner redex. Hence, by presenting outermost redexes first the programmer often needs to decide the correctness of an evaluation of an *incorrect* redex. In some cases he may not even be able to tell what the evaluation's result of an *incorrect* redex should be (see section 6.4). At this point the programmer may naturally only want to find out where the *wrong* argument to the redex came from.

Innermost redex first: Presenting equations about innermost redexes first has the advantage that redexes which result in arguments for outer redexes are shown first. If the evaluation of an inner redex is erroneous then the programmer does not need to consider the outer redexes. The situation where the programmer needs to decide the evaluation of an *incorrect* redex is avoided in many cases. Though both approaches are correct, it seems more reasonable to choose this second solution.

Common problems: Even when using the second solution the programmer may still face redexes with *wrong* arguments. Such *wrong* arguments can still occur: for example, correct arguments may be applied in a wrong order, or an argument may be an erroneous constant value. Examples and ways to deal with questions about redexes with incorrect arguments are shown in section 6.4.

Implemented ordering: Algorithm 4.8 already finds the children in a specific sequence. The algorithm uses a *depth-first-search* approach to search the graph for redexes being children of an EDT node. As subexpressions of a redex are searched first, before the redex itself is added as an EDT child, the children are already ordered in an *innermost-first* order. Arguments of applications are searched left to right, so a *leftmost-innermost-first* ordering of the redexes is given. Numbering the redexes in the sequence in which they are found makes this ordering explicit.

4.8 Summary

This chapter has shown how an EDT can be derived from Hat's Redex-Trail structure. An algorithm was given which finds all child nodes of a given EDT node. As the EDT is the basic data structure for Algorithmic Debugging, it is possible to implement a tool supporting this debugging view. Such a tool using the EDT is described in chapter 6.

A number of problems were encountered in extracting the EDT. Not all these problems could be solved correctly.

- The problem with oversaturated application nodes was discussed. A modification for the algorithm was explained, which lessens the problem by reflecting a different evaluation dependency. However, this dependency does not match the defined *direct evaluation dependency*.

  A future version of the Hat Redex-Trail generation will avoid the creation of oversaturated applications and split them correctly to saturated ones. This specific problem with the EDT will then be solved.

- For the evaluation of functions returning the value of a CAF the EDT child nodes cannot be derived from the Redex-Trail. Instead only the dependency on the CAF is shown. This problem cannot be solved efficiently as links to the intermediate redexes are missing. Only a full search of the Redex-Trail graph could identify these intermediate redexes. The same problem also affects the original Hat Redex-Trail browser. A solution is planned which helps both tools by providing access from the evaluation's result to currently unreachable redexes.

- Applications of locally defined functions and lambda abstractions are omitted in the EDT. The EDT children of omitted equations are lifted and still included in the

EDT. To decide the correctness of the evaluation of a local function or a lambda abstraction context information is necessary. But this context information cannot be derived from the Redex-Trail. This is a disadvantage to the Algorithmic Debugging tool Freja (see page 14). Freja records context information and supports debugging of locally defined functions and lambda abstractions.

- Functions defined in Haskell's Prelude are considered *trusted* and their reductions are omitted in the EDT as the programmer will rely on the correctness of the predefined functions. As their EDT children may still contain *untrusted* functions, these children are lifted and included in the EDT.

- The EDT may contain several identical equations. The solution presented for including the reductions of CAFs in the EDT resulted in a flaw which may result in mistakenly considering a CAF an EDT child of several nodes in an EDT branch. The tool using the EDT needs to avoid repeating questions.

Chapter 5

A Tool for Observation Debugging

Chapter 3 showed how the entire list of applications of a particular function identifier can be obtained from the Hat Redex-Trail structure. A tool could simply pretty print and present the entire list to the programmer, showing each application and its result. The programmer is then able to decide which applications resulted in wrong results and investigate the source-code to find the cause of the error.

But some functions may be applied very often within the evaluation of a program, resulting in an extensive list of equations. In this case it does not seem reasonable to present the entire list of equations to the programmer. This could flood him with too much information.

This chapter presents means to reduce the number of observed equations, while not losing any information the programmer is really looking for.

Furthermore this chapter describes the implemented Observational Debugging tool named `hat-observe` and shows how filters are integrated in the tool.

5.1 Most General Equations

One way to reduce the number of equations presented to the user is to avoid showing redundant equations. An obvious example is the case of completely identical equations. As a function may be applied to the same arguments several times during a program's evaluation this may lead to the observation of identical equations. It is desirable to show unique equations only.

But not just identical equations may be omitted. Due to Haskell's lazy nature arguments to applications and the application's result may be only partially evaluated. Expressions and equations therefore may have different degrees of definiteness, as shown in example 5.1.

Example 5.1 (Some observations of the function `or`)

(1) `or [True, _] = True`
(2) `or [True, False, False] = True`
(3) `or [True, _] = _`

1. The first equation shows that the function `or` only demands the first element `True` of its argument list in order to compute its result.

2. The argument list in the second example is fully evaluated, even though its first element has the same value as in the first example. The argument list may have been shared with another expression, which demanded the full evaluation, or the entire list may have been given as a constant.

   The extra information given in the second equation is of no interest. The first example shows that only the first element of the list is demanded by `or` if the list's first element is `True`. Hence, the extra information given in the second example cannot have influenced the evaluation of `or`. Therefore the second equation is redundant.

3. The third example shows that equations with less defined results can be omitted. The entirely unevaluated result is an extreme case, which renders the whole equation useless. Loosely speaking the programmer is interested in seeing as much of the RHS as possible while only seeing as much as necessary on the LHS of the equations shown.

A tool observing these three equations should only display the first equation. The first equation is *more general* than the second and third. The second and third equations are redundant, they are *less general*. □

A partial ordering for equations

What exactly is meant by *more* and *less general* can be defined formally. A non-flat partial ordering is stated for expressions, and extended to a partial ordering for equations. The partial ordering is introduced for expressions over a simplified signature Σ. The simplified signature consists of untyped identifiers, constructors, a symbol for the undefined value $\perp$ and a symbol for an unevaluated value $\_$.

$$\Sigma = Identifiers \bigcup Constructors \bigcup \{\perp, \_\}$$

T_Σ is then defined as the smallest set fulfilling:

- $\Sigma \subseteq T_\Sigma$
- $(t\ t_1..t_n) \in T_\Sigma$, for $t, t_1, ..., t_n \in \Sigma$

Definition 5.1 (Partial Ordering of Expressions)
A partial ordering $\leq_{exp}$ for expressions is given by:

$\forall t \in T_\Sigma : \_ \leq_{exp} t$
$\forall t \in T_\Sigma : t \leq_{exp} t$
$\forall t, t_1, ..., t_n, t'_1, ..., t'_n \in T_\Sigma : (t\ t_1..t_n) \leq_{exp} (t'\ t'_1..t'_n)$, iff $t \leq t'$ and $\forall 1 \leq i \leq n :\ t_i \leq_{exp} t'_i$

An expression t_1 is called *more general* than an expression t_2 when $t_1 \leq_{exp} t_2$ and $t_1 \neq t_2$. By analogy t_2 will be called *less general* than t_1. □

Definition 5.2 (Partial Ordering of Equations)
Suppose two equations $(l = r)$ and $(l' = r')$.
A partial ordering $\leq_{eq}$ for equations is defined by:
$(l = r) \leq_{eq} (l' = r')$ iff $l \leq_{exp} l'$ and $r' \leq_{exp} r$.

If $(l = r) \leq_{eq} (l' = r')$ then $(l = r)$ is called *more general* than $(l' = r')$. Also $(l' = r')$ is called *less general* than $(l = r)$. □

When a list of equations is considered, only the most general of these equations need to be shown.

5.2 Refined Observations

So the number of equations can be reduced by avoiding redundant equations. But even then, some observations may still result in a large number of equations.

Further improvements can be achieved by refining the observations. The programmer gives additional information about the kind of applications he is interested in. Often a programmer has suspicions about exactly when a function may return erroneous results. He is interested in validating his suspicions. Therefore additional methods are provided to cut the number of presented equations down, while still presenting the information the programmer is really looking for.

Omitting Recursive Function Calls

When a program has an erroneous result then the programmer starts observing a series of functions until an erroneous function is found. Once an erroneous function is found he continues to analyse this function to find the cause of the error.

First step of Observational Debugging: In the first step the programmer only wants to find out whether any application of a certain function has caused erroneous program behaviour. For this purpose it suffices to consider non-recursive calls to this function only. If all non-recursive calls to the function have correct results then this function cannot have caused erroneous program results.

Second step of Observational Debugging: In the second step the programmer wonders why an application of a certain function has an erroneous result. Hence, equations about recursive function calls must not be omitted in this step. But other means to refine the search are implemented to assist in this case, as is shown on page 66.

Optional filtering: Filtering recursive calls is therefore provided as an option within the observation tool, allowing the programmer to choose whether recursive calls are shown or not.

Example 5.2 (Sorted insertion)

```
insert                :: Ord a ⇒ a → [a] → [a]
insert x []           = [x]
insert x all@(y : ys) | x > y = y : insert x ys
                      | otherwise = x : all

main                  = insert 4 (insert 3 [1,2])
```

The observation of function `insert` results in the following list of equations:

```
insert 3 [1,2] = [1,2,3]
insert 3 [2] = [2,3]
insert 3 [] = [3]
insert 4 [1,2,3] = [1,2,3,4]
insert 4 [2,3] = [2,3,4]
insert 4 [3] = [3,4]
insert 4 [] = [4]
```

If the programmer is only interested in whether function `insert` caused the program to be erroneous then it suffices to show the non-recursive calls. Only the following equations need to be shown:

```
insert 3 (1:2:[]) = 1:2:3:[]
insert 4 (1:2:3:[]) = 1:2:3:4:[]
```

□

Missing erroneous functions: Erro It needs to be emphasized that cases may occur where all non-recursive calls to the function return correct results, despite the fact that some recursive calls were erroneous. Erroneous functions may therefore be missed if the observations of non-recursive calls are omitted. An intermediate erroneous result may not affect the final result of a function application or several errors may compensate. Still, the debugging goal to find the cause of incorrect program behaviour is met. An erroneous function definition can only cause the program's result to be erroneous if at least one of the non-recursive calls to this function results in an erroneous value.

Identifying recursive calls: To show non-recursive calls to a function only, a way to distinguish application nodes representing recursive and non-recursive calls is to be found. Distinguishing between recursive and non-recursive calls can be done immediately by considering the Redex-Trail graph:

- A recursive call to a function may originate immediately from the body of the function's definition. The application node for the recursive call has a parent reference to its creating redex. In this case, according to the Redex-Trail structure (see (3) on page 19), the creating parent redex is another application of the same function identifier.
- A recursive call to a function f may originate from the body of a different function, which in turn originates from the body of function f or so on recursively. Hence, all *ancestors* (see definition 4.6, page 41) of an application node need to be considered. If any ancestor of an application of the function f is another application of the same function f then the former application represents a recursive call to function f.

Therefore a method is found to filter all non-recursive calls to a function.

Applications in function bodies

In the description so far observations considered all applications within the entire scope of the program. Restricting the observation of function applications to the body of a given second function may reduce the number of observations significantly. For example, when an erroneous function is found, the programmer may decide to observe further functions to analyse the cause of the error. In this case it suffices to observe those function applications only, which originate from the erroneous function's body.

Example 5.3 (Restricting the scope of an observation)

```
import Char(digitToInt)

add_digit         :: Int → Int → Int
add_digit x y     = 10 * x + y

stringToInt       :: String → Int
stringToInt l     = foldl add_digit 0 (map digitToInt l)

main              = ... (map stringToInt ["123", "456", "789"]) ...
```

The function `map` may be used often in a large program. In this example it is used in the bodies of function `stringToInt` and of CAF `main`. When analysing this example program the programmer may be interested in observing the applications of `map` originating from the function body of `main`, instead of observing *all* applications of `map` in the entire program:

query: `observe "map` *in* `main"`
`map stringToInt ("123":"456":"789":[]) = 123:456:789:[]` □

Associating an observation with a function body

In example 5.3 it is quite clear that the observed equation "`map stringToInt` ("123": "456": "789": []) = `123:456:789:[]`" origins from the body of the CAF `main`'s definition. But with higher-orderness multiple ways to associate an observed equation with an originating function body are possible.

Equations with a CAF on the LHS: When a CAF is observed it cannot be assigned to any originating function body, as no creating parent redex for a CAF is recorded in the Redex-Trail graph (see (2) on page 19). But as a CAF is only evaluated once there is only a single occurrence of such an equation. Hence, there is no need for further refinement to limit the number of equations shown.

Equations with an application on the LHS: When the LHS of an observed equation is given by an application then the equation can be associated in two ways to an originating function body:

1. An equation can be associated with the function where the outermost *application* on its LHS was introduced. This is the solution used in section 4.2 to order equations according to their *direct evaluation dependency*. The same solution can be used here but it has a disadvantage. The programmer needs to find out where a function is applied in order to submit an appropriate observation query. With higher-orderness this can turn out to be difficult and it is not immediately visible from the source-code.

   Example 5.4 (Observing `add_digit` in example 5.3)
   The programmer may want to observe the use of `add_digit` only in the body of function `stringToInt` (see example 5.3).
   Since `add_digit` is passed as a higher-order argument to `foldl` it may be applied immediately within `foldl` but it may also be passed on to some other internal function used by `foldl`. The programmer cannot tell from the source-code of his program where `add_digit` is eventually applied. □

2. A better solution is to associate an equation with the function which introduced the outermost *function symbol* on the equation's LHS. The programmer does not need to find out where the function is eventually applied. He can simply refer to the function where the function symbol is introduced, which is immediately visible from the source-code.

   To observe `add_digit` as it is used in function `stringToInt` (example 5.3) an observation query *observe* `add_digit` *in* `stringToInt` can be submitted. More details about how the observational tool `hat-observe` supports such queries is given in section 5.3.

Implementation: To implement the second solution it is necessary to identify which function body introduced the outermost function symbol of an equation's LHS. This is immediately given by the Redex-Trail graph as described in (1) on page 19. Each function symbol is represented by a `Const` node referencing its creating parent redex. This means the function symbol occurred in the body of the definition of the parent redex's outermost function symbol.

Equations of a Specific Pattern

The number of observations presented to the programmer can be lessened by describing more specifically the observations of interest. Often functions are defined correctly for the general case but are erroneous for specific arguments. A programmer may want to observe function applications meeting such a specific case only. This can be done by defining a pattern for the observation. The tool initially generates the entire list of equations for the given function identifier as described in section 3.4. It then filters all observed equations according to the defined pattern and shows matching equations only.

Patterns for observations: Two kinds of patterns for observations are possible.

- A pattern for the LHS of an observed equation. This allows to filter equations with special arguments on the LHS (see example 5.5 (a)).
- A pattern for the LHS and RHS of an observed equation. This allows further refinement of the search to filter equations with special cases of results (see example 5.5 (b)).

Example 5.5 (Observation with a pattern)

```
  — insert defined in example 5.2
sort      :: [Int] → [Int]
sort list = foldr insert [] list

main      = sort [1,8,4,1,9,7,5]
```

(a) Only the value `1` is inserted twice. So instead of observing all applications of `insert` the programmer may decide to observe only the special case for inserting `1`. Therefore a pattern is given to observe all applications of `insert` where the first argument is `1`. The second argument may be any value: the symbol "`_`" is used as a wildcard matching any value.

query: `observe "insert 1 _"`
```
insert 1 [] = [1]
insert 1 [1,4,8] = [1,1,4,8]
```

(b) It is also possible to state a pattern for the RHS which is matched against each observed equation. The following query shows all applications of `insert` which resulted in a list of exactly three elements.

query: `observe "insert _ _ = [ _, _, _]"`
```
insert 4 [1,8] = [1,4,8]
```
□

Matching an observed equation against a pattern: The pattern for the LHS and the optional pattern for the RHS are each parsed into an expression tree. A wildcard "_" in a pattern is represented by an unevaluated value in the pattern's expression tree. The partial ordering of *expressions* (definition 5.1, page 64) is reused for the pattern matching. However, the following definition has nothing in common with the partial ordering of *equations* (definition 5.2, page 64). Here, both sides of the equation have to be at least as defined as the pattern. The partial ordering of equations required the LHS to be equal or *less* defined and the RHS to be at least as defined as the second equation.

Definition 5.3 (Pattern Matching of Equations)

- An equation $(l = r)$ matches a pattern $(pl = pr)$ when $pl \leq_{exp} l$ and $pr \leq_{exp} r$.
- An equation $(l = r)$ matches a pattern pl when $pl \leq_{exp} l$. □

Pattern Matching Implementation: Definition 5.3 is used to compare the expression trees of the LHS and RHS of an equation with the defined pattern.

Parsing a pattern into the required expression tree involves a problem. The Observational Debugging tool does not know the sets of constructors and identifiers of the observed program. Furthermore, numeric values in patterns do not have types. In the Redex-Trail every numeric value is of a certain numeric value, for example `Int`, `Integer`, `Float`, etc. Still, the effort to obtain the entire sets of identifiers and constructors from the Redex-Trail or to force the programmer to state a specific numeric type for every numeric value in a pattern is unnecessary. Instead the pattern matching is slightly modified so that any numeric value matches any other *equal* value, regardless of type. Furthermore, the pattern matching does not distinguish identifiers and constructors and only relies on comparing their names.

5.3 Using the Observational Tool

This section outlines how the observational tool `hat-observe` is used and how it allows the programmer to deal with a large number of equations and large data structures. A complete list of supported commands is given in the appendix (page 113).

Making Observations: The observational tool is started after the Redex-Trail for the program in question was recorded. The tool is interactive, allowing the programmer to make successive observations and to choose between different options:

- The programmer can toggle the mode for filtering recursive calls as was described on page 64. If the programmer is only interested in finding out whether a function definition is erroneous at all then it is useful to omit recursive calls.
- Unevaluated subexpressions have no influence to a program's computation. By default the tool hides unevaluated subexpressions[1], and instead shows the symbol

[1]Unevaluated expressions are represented by a `SATA` node in the Redex-Trail graph (see page 17 for more detail).

"_" only. The programmer can still choose to see unevaluated subexpressions by switching to *verbose mode*.

- An observation is initiated by a query. A query simply consists of a function identifier or a pattern (see page 114 for query syntax). The outermost symbol of the pattern must specify a function identifier. If the scope of the observation is restricted to a specific function body (see page 66) then the keyword `in` followed by the second function identifier needs to be appended to the query. The tool then observes all applications of the given function identifier (see section 3.4) and, in case a pattern was defined, matches each observed equation against the pattern (see page 68).
- The results of a query is a list of equations. A small (and adjustable) number of equations is presented to the programmer at a time. The programmer can analyse these equations and then choose to see more equations or to submit a new query.

Lazy Observations: In section 5.1 it was proposed to avoid redundant equations by showing the most general equations only.

1. A first implementation of this solution is to filter the entire list of equations before showing any observations to the programmer. As the very last equation observed may be *more general* than all earlier equations, all equations need to be considered before any of them may be presented to the programmer. Searching the entire Redex-Trail structure and finding the *most general* equation may take a long time in extreme cases.
2. An alternative solution provides a much better response time than the first solution. The solution is not to show *most general* equations only but to show equations one by one and omit equations which are *less general* than previous equations. If many of the observed equations are wrong, then even the first equations observed may allow the programmer to decide what the error is.

   The disadvantage of this approach is that it may result in more equations being presented. But a major advantage is a much better usability of the tool. This solution allows the tool to make observations lazily. Only as many observations are done by searching the Redex-Trail structure as the programmer really demands to see. The maximum time penalty of searching an entire Redex-Trail structure only occurs when the programmer indeed demands to see all equations.

   Filtering all equations before showing them, and therefore guaranteeing to show the most general equations only, is still available to the programmer as an option.

Expression printing: The equations are printed by a pretty printer (see [CHI01]), which uses indentation of subexpressions to make larger expressions more readable. However, expressions may be too large to be displayed entirely. The programmer can therefore determine how far an expression tree is descended. The expression tree is pruned at the given depth. The pruned subexpressions are displayed by a special symbol "`CUT`". This method limits the space used for printing an expression, but may hide important information. A better solution is allows the programmer to choose interactively which subexpressions to show in full and which to prune, as implemented in the Redex-Trail browser (see page 16). This method is not straightforward with a textual interface and it is not implemented for the observational tool `hat-observe`.

5.4 Summary

In this chapter a tool using the observational view has been described. Several features were presented which make the tool convenient to use, even with large numbers of observed equations.

- A partial ordering for equations is the basis for a method to avoid redundant information by omitting *less general* equations.
- An optional filter eliminates equations describing the evaluation of recursive calls to the observed function. As recursion is common in a functional language this may effectively limit the number of observed equations, while still allowing the programmer to decide whether a function is responsible for erroneous program behaviour.
- The scope of function applications may be restricted to observe the use of functions within a specific function body.
- The implementation of the tool offers a low response time and observations are reported lazily, searching only for as many observations as are really demanded by the programmer.

Filtering *less general* equations may result in additional time overhead when large numbers of similar equations are filtered. However, the overhead is worse if all equations were shown and the programmer had to consider all equations manually.

A number of problems persist, which limit the usability of the tool in certain situations:

- In some cases the filters are of little help: for example when the programmer wants to verify whether all applications of a function had correct results. Even after filtering all *less general* equations there may be too many equations left to be considered manually. Then the programmer can only verify a limited number of them. A workaround in practice is to modify the program or to use simpler test data so that less applications of the function in question occur.
- In other cases patterns are helpful. But their use becomes inconvenient when the pattern of interest is too complex, for example to observe applications where one of the argument matches a certain complex data structure.
- Though a method for pretty printing is implemented and expressions may be pruned at a given depth, equations containing large expressions are still a problem. An interactive method is necessary to allow the programmer to explore all subexpressions of interest. This is implemented for the Redex-Trail browser but not yet for the observational tool.

Chapter 6

A Tool for Algorithmic Debugging

In chapter 4 it was explained how an EDT can be derived from the Redex-Trail structure. An Algorithmic Debugging Tool named `hat-detect` using this EDT is presented in this chapter. The tool relies on the programmer as an oracle to decide whether reductions resulted in the intended results.

As with the observational tool presented in chapter 5, means have to be provided to avoid flooding the programmer with too many equations. The convenience and usability of the tool depends on the number of questions the programmer has to answer in order to locate a bug in his program.

Another problem to be discussed arises from complex equations. It is a requirement for Algorithmic Debugging that the programmer is able to answer all questions correctly. But in some cases the programmer may be unsure whether an equation is erroneous or not. For example an equation may be too large to be considered manually.

A specific problem of Algorithmic Debugging is discussed regarding the evaluation of redexes which must not occur during evaluation. This causes a difficult situation for the programmer in order to use the Algorithmic Debugger for locating the cause of the error.

6.1 Traversing the EDT

The Algorithmic Debugger traverses the EDT, which can be derived piecemeal from the Redex-Trail structure using algorithm 4.8 (page 43). The EDT is searched in a top-down fashion in order to find a *buggy* node (see definition 2.1, page 11). The scheme guarantees that a *buggy* node is found when the root node of the EDT is erroneous [NAI97]. Algorithm 6.1 shows an implementation of the Algorithmic Debugging scheme (see page 12) including the user interaction and piecemeal EDT generation.

- Function `algorithmicDebugging` starts an Algorithmic Debugging session by verifying the root node of an EDT.

- Function `verifySiblings` presents the first sibling to the user and asks about its correctness. If the programmer answers that an equation is correct then function `verifySiblings` continues to consider the next sibling.

Algorithm 6.1 (Debugging Algorithm)

```
data DebugResult = NoBugFound | BugFound String  deriving Eq
data YesNo       = Yes | No deriving Eq
data RedexTrail  = ...
data EDTNode     = ...

prettyPrint :: EDTNode    -> String
edtRootNode :: RedexTrail -> EDTNode
edtChildren :: EDTNode    -> [EDTNode]
askOracle   :: EDTNode    -> IO YesNo

verifySiblings                  :: [EDTNode] -> IO DebugResult
verifySiblings []               = return NoBugFound
verifySiblings (node : siblings) =
  do
    yesno <- askOracle node
    if yesno == Yes then
      verifySiblings siblings
    else
      do
        debugresult <- verifySiblings (edtChildren node)
        if debugresult == NoBugFound then
          return (BugFound ("Bug located in EDT node : " ++
                               (prettyPrint node)))
        else
          return debugresult

algorithmicDebugging             :: RedexTrail -> IO DebugResult
algorithmicDebugging redextrail =
  verifySiblings [edtRootNode redextrail]
```

The algorithm is given in Haskell as the actual tool is also implemented in Haskell. Some of the data structures and functions are not defined in detail. The function `edtChildren` returns a list of the argument's EDT child nodes and is the Haskell interface to the function given in algorithm 4.8 (page 43). The function `askOracle` presents the equation of an EDT node to the user and returns his decision about its correctness. □

- If the programmer identifies an equation to be erroneous then the EDT children of the current node are verified. The children are determined lazily by function `edtChildren`.
- If there are no siblings left in `verifySiblings`, all siblings were correct or none existed, then `NoBugFound` is returned.
- If a node was identified to be erroneous but all its children are correct then the current EDT node is *buggy*. The location of the bug is returned by `verifySiblings`. If the root node itself is identified to be correct, so the program does not contain a visible bug, then `NoBugFound` is returned.

In [NAI97] it is shown that the algorithm terminates when the EDT is finite and has no loops. The EDT derived from the Redex-Trail graph is finite but may indeed have loops as the evaluation of a CAF may be self-dependent. Hence, algorithm 6.1 may not terminate if the evaluation of a self-dependent CAF is included in the EDT. This problem is avoided by *memoisation*, which is described in section 6.2.

6.2 Handling the Flood of Questions

The Algorithmic Debugging Tool asks the programmer about the correctness of the equation of a considered EDT node. The tool needs to avoid unnecessary questions for example repeating identical questions. The less questions the programmer has to answer to locate a bug in his program the more convenient the tool is. Two techniques are presented which may reduce the number of questions asked.

Memoisation

For the Observational Debugging tool a method was presented to omit redundant equations (section 5.1). A similar approach is used for the Algorithmic Debugging Tool. The idea was introduced in [SHA82] as the *mechanisation of an oracle*. Each equation considered by the programmer, and his answer, is memoised by the tool. Whenever the Algorithmic Debugger reaches an EDT node labeled with an equation which is identical or *less general* (see definition 5.2, page 64) than a previous equation then the programmer's previous answer is immediately reused. The question does not need to be presented to the programmer.

Memoisation of CAFs: A specific solution is necessary for self-dependent CAFs. Self-dependent CAFs result in a loop within the EDT, as the evaluation of a CAF is only represented once. When the evaluation of a CAF is considered erroneous then the Algorithmic Debugger continues to verify its EDT children. If another EDT node describing the evaluation of the same CAF is found then the standard Top-Down Debugger enters a EDT loop in the EDT and does not terminate (see "*Correctness of the Top-Down Debugger*" lemma 2.3, page 12).

The solution is to omit the repetition of the EDT node describing the CAF's evaluation. Once the evaluation of a CAF is considered it does not need to be considered again. This solution is a modification to the original Algorithmic Top-Down Debugger. It may now consider a node *buggy* though it depends on an erroneous child, when the erroneous child describes a self-dependency. But this is the only way to keep the Algorithmic Debugger from non-termination when the EDT contains a loop.

Memoisation also avoids a flaw in EDT generation, described in section 4.5 (page 55).

Example 6.1 (Memoisation)

```
inc    :: Int → Int
inc x  = x + 1

f      :: [Int] → [Int]
f l    = map inc l

ones   :: [Int]
ones   = 0 : ones

list   :: Int → [Int]
list x = (f x) : (list x)

(a) main = f [1, 1, 1, 1, 1, 1]
(b) main = take 5 ones
(c) main = take 2 (list 0)
```

(a) The first example shows the use of memoising answers where the programmer answered "*Yes*". The programmer has other intentions about what function `f` is supposed to do. Function `inc` is implemented correctly though. Only a single question about `inc` needs to be answered as all its six applications are identical.

```
1 > main  = [2, 2, 2, 2, 2, 2]                        (Y/N) : no
2 > f [1, 1, 1, 1, 1, 1]  = [2, 2, 2, 2, 2, 2]      (Y/N) : no
3 > inc 1  = 2                                        (Y/N) : yes
Error located! Bug found in :
f [1, 1, 1, 1, 1, 1]  = [2, 2, 2, 2, 2, 2]
```

(b) The second example shows a case where the programmer identifies the equation "`ones = a where a = 1:a`" to be incorrect[1]. The evaluation of `ones` is self-dependent. The Algorithmic Debugger does not consider the second occurrence of the equation, otherwise it would enter an infinite loop:

```
1 > main  = [0, 0, 0, 0, 0]          (Y/N) : no
2 > ones  = a where a = 1 : a      (Y/N) : no
Error located! Bug found in :
ones  = a where a = 1 : a
```

[1]A special notation is used by the pretty printer to show cyclic data structures [PEN99].

(c) In the third example the programmer gains from memoising an incorrect equation. A third question about "`list 0 = 1:_`" is avoided as the equation is *less general* than the previously considered erroneous equation "`list 0 = 1:1:_`".

```
1 > main   = [1,1]              (Y/N) : no
2 > list 0  = 1 : 1 : _         (Y/N) : no
Error located! Bug found in :
list 0  = 1 : _
```

□

Trusting

Another way to cut down the number of questions being presented to the programmer is given by *trusting*. Functions defined in the Prelude of Haskell are automatically considered trusted by Hat and their evaluation is omitted in the EDT while the evaluation of untrusted higher-order arguments is still included (see section 4.6).

Manual trusting: But the programmer may be convinced that some of the functions defined in his program are also correct and cannot have caused an incorrect program behaviour. He can therefore *manually trust* these functions during an Algorithmic Debugging session. When the programmer trusts a function identifier the Algorithmic Debugger assumes all applications of this function to reduce to correct results. All EDT nodes describing the evaluation of such function applications are omitted, which has the same effect as answering "*Yes*" to all questions about the evaluation of such function applications. This feature is very useful to avoid inconveniencies when a correct function is used very often with differing arguments (see example 6.2).

Example 6.2 (Manual trusting)

```
  — see definition of "f" and "inc" in the previous example 6.1
main = f [0,1,2,3,5,6]
```

The program was intended to result in the list `[1,2,3,4,5,6,7]`. The programmer manually trusts the function `inc` when asked the first question about an application of `inc`:

```
1 > main  = [1,2,3,4,6,7]                    (Y/N) : no
2 > f [0,1,2,3,5,6]  = [1,2,3,4,6,7]         (Y/N) : no
3 > inc 0  = 1                               (Y/N) : trust
Error located! Bug found in :
f [0,1,2,3,5,6]  = [1,2,3,4,6,7]
```

□

Danger of manual trusting: Manual trusting is dangerous as the programmer may not be aware that the function he trusts causes a problem. In this case the tool may incorrectly locate an error at an EDT node which depends only on the evaluation of an application of the *manually trusted* erroneous function.

Another problem concerns functions with higher-order parameters. An application of such a function may still reduce to an incorrect result though the implementation of the function itself is correct (see example 6.3). This problem does not apply to automatically trusted Prelude functions. As described in section 4.6 (page 57) EDT children of an evaluation of a Prelude function are *lifted*. Using the same solution for the functions trusted manually would be too inconvenient. The implementation of a function defined by the programmer is likely to depend on several other functions also defined by the programmer. By lifting the EDT children for a *manually trusted* function application almost the entire evaluation would still be visible. The programmer would hardly gain from manual trusting.

Improving manual trusting: The manual trusting mechanism can be improved by searching for EDT children which describe the evaluation of an application of a higher-order argument to the manually trusted function. Hence, the entire evaluation of an application of a manually trusted function is omitted except the evaluation of an application of a higher-order argument. This solution is possible using the information given in the Redex-Trail structure but currently not implemented in the Algorithmic Debugging tool. Therefore functions with higher-order parameters should not be *manually trusted*.

Example 6.3 (Danger of manual trusting)

```
erroneous         :: Int -> Int
erroneous x = x * x * x

mymap             :: (a -> b) -> [a] -> [b]
mymap _ []        = []
mymap f (x : xs)  = f x : mymap f xs

main              = mymap (+1) [1,2,3] ++ mymap erroneous [1,2,3]
```

During a debugging session the programmer decides to *manually trust* the function `mymap`, when he is asked about the equation "`mymap(+1)[1,2,3]=[2,3,4]`". Therefore the tool also trusts the evaluation of "`mymaperroneous[1,2,3]=[1,8,27]`" though the function `erroneous` does not meet the programmer's intentions.

Note that it is essential in this example that `mymap` is not defined in Haskell's Prelude (in contrast to `map`). This problem does not occur with trusted Prelude functions (compare to example 4.5, page 58). □

6.3 Being Indecisive

In some cases the programmer may be unsure whether an equation is correct or not. This can be due to a complex data structure in the equation. The tool provides two ways to proceed at this point. Both options were already implemented in Freja [NIL98].

"Maybe Yes": The programmer may postpone a question by answering "*maybe yes*" (short "*?yes*"). The tool proceeds as if the programmer had answered "*yes*" and continues to verify the siblings of the current EDT node. Only if all its siblings are identified to be correct is the postponed question reconsidered. The answer "*?yes*" changes the order in which the Algorithmic Debugging tool asks about the children of an EDT node. By default the EDT children of a node are sorted in a *leftmost-innermost-first* order (see section 4.7).

Answering "*?yes*" is useful as the cause of the program's erroneous result might be in a sibling of the currently considered EDT node. If an error is found in a sibling then the programmer saves the effort of dealing with the complex equation. Otherwise, if all siblings are correct then he knows that it is indeed essential to consider the complex question.

"Maybe No": By analogy "*maybe no*" (short "*?no*") is provided. The question about an equation is also postponed but the tool initially continues as if the programmer had answered "*no*". The children of the postponed EDT node are considered first. Only if all these children are identified to be correct is the question about the postponed equation reconsidered. An advantage of this possible answer is that the programmer actually learns about how the redex on the LHS of the complex equation is evaluated. Afterwards the programmer may be able to answer the question about the postponed equation.

6.4 Incorrect redexes

A specific problem occurs with Algorithmic Debugging when the tool asks about the evaluation of a redex which should not have occurred in the program's evaluation or which even must not be evaluated at all (see example 6.4). This causes a problem for the programmer as he has to answer each question in order to locate a bug.

Example 6.4 (Evaluating an incorrect redex)

```
fibonacci        :: Int -> Int
fibonacci n      = fibiter 1 0 n

fibiter          :: Int -> Int -> Int -> Int
fibiter n1 _  0  = n1
fibiter n1 n2 n  = fibiter (n1 + n2) n1 (n - 1)

fib7             = fibonacci 77

main             = (2 * fib7)
```

The program contains a simple typing mistake as the CAF `fib7` was accidentally defined as `fibonacci 77` rather than `fibonacci 7`. The result of `fibonacci 77` causes an overflow in the Haskell type `Int`. An Algorithmic Debugging session leads to a difficult situation for the programmer:

$1 >$ `main` $= 751639760$ (`Y/N`) : *no*
$2 >$ `fib7` $= 375819880$ (`Y/N`) : *no*
$3 >$ `fibonacci 77` $= 375819880$ (`Y/N`) :

The first two questions can easily be answered with *no* as the intended result of `main` is `26` and `fib7` is `13`. But the third question is puzzling. The result of `fibonacci 77` surely is not `375819880`. So, the programmer could answer *no*. On the other hand the fact may be accepted that the result of `fibonacci 77` causes an overflow, as it is too large for the Haskell type `Int`. Hence, the programmer might also answer *yes*; he intends that the evaluation of `fibonacci 77` results in an overflow.

But the real problem with this equation is that the redex `fibonacci 77` should never have occurred in the evaluation of `fib7`. The programmer may be unsure how to answer the third question in order to determine the location of the bug responsible for creating the erroneous redex. □

Causes for wrong redexes: When the Algorithmic Debugging tool asks about an equation "$r = v$", where the redex should not even have occurred during the evaluation, one of the following cases must apply:

1. A subexpression of r is the result of an erroneous evaluation. The EDT node describing this erroneous evaluation must be a sibling of the "$r = v$" node. The Algorithmic Debugger asks about equations in an *innermost-redex-first* order (section 4.7) by default. Hence, the erroneous equation which resulted in the erroneous subexpression of r is considered first and the equation "$r = v$" is not presented to the programmer, unless the programmer has deliberately postponed the question about the erroneous sibling.

   Workaround: This problem shows that postponing questions using "*maybe yes*" can be dangerous. To avoid this problem the programmer should also postpone the question about the evaluation of the wrong redex, and first reconsider all previously postponed questions.

2. If no sibling of the node with equation "$r = v$" is erroneous then a bug is to be found at the parent of r. The reduction of the r's parent created the redex r. As r is already wrong and should never have occurred in the program's evaluation there must be a bug in the rule used to reduce r's parent.

While there is a workaround to avoid the first case, there is a severe problem about the second. What if a *wrong* redex r evaluates to v but none of the siblings of the node describing the equation "$r = v$" is erroneous? Then the Algorithmic Debugging tool may not necessarily succeed in locating the bug which caused the creation of the erroneous redex. There are three possibilities regarding the equation "$r = v$":

- The programmer may identify "$r = v$" to be a correct equation. Given the redex r then the intention of the reduction rules defined in the program is indeed that r reduces to v.

  Hence, the only problem is that the redex r is created and evaluated during the program evaluation at all. In this case the programmer should truthfully answer "*yes*" to the question about the correctness of "$r = v$". All siblings of the node describing "$r = v$" are correct but the EDT parent node of the equation "$r = v$" is erroneous (only the children of erroneous nodes are ever considered). This means the parent the equation "$r = v$" in the EDT is a *buggy* node (definition 2.1, page 11) and the Algorithmic Debugging tool presents it as the location of a bug.

  Drawback: Though the tool is indeed able to locate this bug an irritating situation for the programmer is caused: He has to answer "*yes*" to the question about the equation though he may be tempted to answer "*no*". The equation is *correct* but it is *wrong* that the equation occurred at all.

  This is a situation where the Algorithmic Debugging tool is tricky to use. An inexperienced user of the tool may be tripped.

- The programmer may identify "$r = v$" to be an erroneous equation. Given the redex r then it is *not* the intention of the reduction rules defined in the program that r reduces to v.

  The programmer has to answer the question about the correctness of "$r = v$" with "*no*". The Algorithmic Debugging tool then continues to search for the bug which caused the redex r to evaluate to an incorrect result. At this point there are already two bugs in the program. A first bug which caused the creation of the redex r, and a second bug which occurred in the evaluation of r. The later bug in the evaluation of r may be found using the tool.

  Drawback: The first bug is located at the EDT node which is the parent of the node describing "$r = v$". As the equation "$r = v$" is erroneous its parent is *not buggy*. Hence, the Algorithmic Debugger (Top-Down Debugger) is unable to find the error causing the creation of the redex r as the responsible node is not *buggy* (see "*Correctness of the Top-Down Debugger*", lemma 2.3, page 12). The second bug has to be corrected first. The tool can then be used again to locate the bug causing the creation of r.

  Again, the programmer may be irritated. He may already spot the problem of the incorrect redex but he cannot find the cause of the error. Instead he has to chase the second bug, which occurred in the evaluation of r.

- The programmer may not even know what the result of the *wrong* redex r should be. Suppose the programmer imports a module he has not written himself. The specification of the module may not allow certain arguments to a function (in example 6.4 the specification of `fibonacci` may only permit argument values from `0` to `45` and not mention any defined behaviour for other arguments). In this case the redex r does not have *any* intended result. Redex r is not only wrong in the sense that it should not have occurred (as described by the first two alternatives) but it is even wrong in a sense that the redex r *must not* be reduced.

Workarounds: There are two ways to proceed at this point.

1. He may simply answer "*yes*" to the question as the evaluation of redex which must not be evaluated cannot be erroneous. The evaluation of r must be omitted, so the EDT children of the equation are not considered. The programmer is only interested in locating the bug at the parent node of the equation "$r = v$". Answering "*yes*" is the way to return to the parent of this equation in the EDT. The programmer is not aware of the EDT structure, he only perceives a list of questions being presented by the Algorithmic Debugger. So he needs the tool to tell him what the parent of the wrong redex is.

   Drawback: This is a basic problem of Algorithmic Debugging. It is assumed that the programmer always knows to which value a redex should reduce. But if he has the intention that a specific redex must not be reduced at all then he is unable to answer a question about such an equation with *yes* or *no*. Intuitively any equation describing the evaluation of a redex that should not occur seems wrong. But the only way to successfully proceed with Algorithmic Debugging at this point is to accept that the evaluation in this case may yield *any* value.

2. The workaround which is intuitively comprehensible is to switch to a different debugging tool when this problem occurs. The Redex-Trail browser is the ideal tool to determine where an incorrect redex comes from (see "Browsing strategy" page 17 and chapter 7).

6.5 Using the Algorithmic Debugging Tool

The tool is started after the Redex-Trail of a program is recorded. Using the tool is straightforward and needs little learning as the programmer is strictly guided. The programmer only needs to answer a series of questions with *yes* or *no*, deciding whether the evaluation of redexes resulted in the intended result. A complete command list is given in the appendix (page 115).

Starting points: The first equation describes the evaluation of the `main` CAF. When a large program is considered the programmer may want to start the debugging session at a specific point within the evaluation where he already suspects a program error. Starting an Algorithmic Debugging session at a point within the evaluation is provided by interaction between all three debugging tools of the Hat system and is separately described in chapter 7.

Memoisation: Memoisation is a very useful option during the debugging session, as it may significantly reduce the number of questions asked. Memoisation is therefore activated by default. But in certain situations the programmer may want to see all questions – for example to learn more about the evaluation itself. Memoisation may therefore be manually deactivated.

Navigation: Navigation during a debugging session is automatic as the tool guides the programmer through the EDT. However, during the session the programmer may notice that he made a mistake in deciding the correctness of an earlier equation. Therefore all questions of the debugging tool are numbered and the programmer is allowed to go back to any earlier question by entering the appropriate number. When the programmer chooses to go back to an earlier question it is important for the tool to remove all equations which were memoised since the incorrect decision was made. The same applies to function identifiers which were manually trusted.

6.6 Summary

This chapter has described several features which improve the convenience of the Algorithmic Debugging tool:

- Memoisation is provided to avoid unnecessary questions.
- Manual trusting avoids questions about functions which are known to be correct.
- Two ways of postponing questions may help the programmer when he is uncertain about an equation.
- The programmer may change earlier decisions.

But even with this support Algorithmic Debugging may turn out to be inconvenient in certain cases:

- The Algorithmic Debugging tool suffers from the same problem with large equations as the Observational Debugging tool (section 5.4). The same methods for pretty printing and pruning are provided. But a more sophisticated way to deal with large equations is necessary. For example a graphical interface allowing interactive exploration of larger expressions.
- Though questions may be postponed, eventually the decision of a complex equation may be necessary. Algorithmic Debugging relies on the programmer's ability to answer all questions correctly. If the programmer is not able to decide the correctness of a complex equation, or if he even makes an incorrect decision, then the Algorithmic Debugging tool is of little help or may even spot a bug at an incorrect location.
- Postponing questions by using "*maybe yes*" is a dangerous alternative as it means to temporarily assume that the equation is correct. The evaluation's result may be a subexpression of another redex. Postponing with "*maybe yes*" modifies the default *innermost-redex-first* order in asking about the evaluation of redexes. Hence, the tool may ask about the evaluation of a wrong redex, which would not be considered if the postponed question had been answered correctly.

- As already described in the introduction to Algorithmic Debugging (page 14) the guidance of the Algorithmic Debugging may be convenient, especially for inexperienced programmers. But it may also make the programmer feel restricted during the debugging session.

- The Algorithmic Debugging session always starts with the reduction of the `main` CAF. The programmer may already know that the cause of the error is in a specific part of his program, but he still cannot immediately start the debugging session at this point. This is a very specific and also very common case of the problem with unwanted guidance.

- The option to allow *manual trusting* of functions can be dangerous as the programmer may not be aware that the implementation of a *manually trusted* function is still erroneous, which might only be visible in a specific case. Furthermore, functions with higher-order parameters cannot safely be trusted manually in the current tool. An improved solution to this problem was presented but has not yet been implemented.

- A problem already noticed in deriving an EDT from Redex-Trails limits the implemented Algorithmic Debugging tool: applications of local functions and lambda abstractions cannot be included in the EDT as information about their context is not available. The Algorithmic Debugging tool therefore locates errors in local functions and lambda abstractions at the enclosing top-level function. This is a disadvantage compared to the original Algorithmic Debugger of the Freja system (page 14).

The problem with too complex equations is a basic problem of Algorithmic Debugging. Also the strict guidance is a basic feature, which can hardly be overcome by an Algorithmic Debugging tool itself. But these problems show that the programmer may gain from interaction between different tools. This interaction is especially useful when the tool asks about the evaluation of redexes, where the programmer already spots an error in the redex itself. Possible interaction between the debugging tools is described in chapter 7.

Chapter 7

Interaction of Debugging Tools

It was shown in chapters 3 and 4 that Observational and Algorithmic Debugging can be supported using Hat's Redex-Trail structure only (section 2.4). Actual debugging tools providing these two views were presented in chapters 5 and 6. The Redex-Trail browser (page 16) is a third debugging tool which was previously implemented for Hat's Redex-Trail structure [SR97b].

All three debugging tools are based on top of a single and common data structure. The strengths and weaknesses of these tools are complementary (section 2.5). The common data structure already allows the programmer to conveniently use all three tools. But it also suggests linking the tools in order to provide immediate interaction. The programmer is supported in his task to trace or debug a program's evaluation, as he may quickly switch between all three different debugging views. In this chapter some problems of the three debugging tools are reviewed and it is shown how the interaction between the different tools can overcome these problems.

7.1 Interaction from hat-observe

The Observational Debugging tool `hat-observe` extracts details of function applications and their respective results and shows them as a list of equations. When the programmer analyses the equations he may consider the following cases:

- **Erroneous Equation:** The goal of Observational Debugging is to find an erroneous equation. The programmer then needs to find the cause of this error. The Observational Debugging tool already provides some features to assist the programmer in locating an error. The search scope of observations can be restricted to a particular function body (see page 66). However, this method is not very convenient as the programmer needs to find out himself which other functions were used in the body of the erroneous functions. And the observation of those functions may give too many equations to be verified. Instead he may choose to use a different debugging tool at this point:

  1. **Invoking the Algorithmic Debugger:** When an erroneous equation is found then the Algorithmic Debugger `hat-detect` may be a helpful tool to

find the cause of the error. The programmer may invoke `hat-detect` immediately on an observed equation. In this case the Algorithmic Debugger does not consider the entire EDT describing the program's evaluation, but it only considers the subtree describing the evaluation of the redex on the LHS of the erroneous equation (example given in case study, page 103).

2. **Invoking the Redex-Trail browser:** Instead of using the guidance of the Algorithmic Debugger the programmer may use the Redex-Trail browser in order to find the cause for an erroneous equation. The Redex-Trail browser may be launched from the observational tool. In this case the Redex-Trail browser `hat-trail` starts tracing from the expression on the RHS of the observed erroneous equation. The programmer may then find out which redex caused the RHS of the equation to be erroneous.

- **Suspicious Redex:** A problem which cannot easily be solved by using Observational Debugging is an equation where the programmer spots a suspicious redex on the LHS. The programmer may find the equation itself correct, but he may wonder why a certain argument occurred at all (see section 6.4). Observational Debugging alone cannot show what created the suspicious redex.

  Invoking the Redex-Trail browser: This is an ideal task to switch to the Redex-Trail browser `hat-trail`. But this time the tool is to be used to trace the LHS redex of an observed equation (see case study, page 102).

Switching to a different tool: Switching to a different tool from `hat-observe` is simple despite the textual interface. Each displayed equation is numbered. By using the command "`debug` n" an Algorithmic Debugging session for equation number n is initiated. The Redex-Trail browser is invoked by "`trace` n" (complete command list of `hat-observe` is given on page 113). Before the Redex-Trail browser is started the tool asks whether to trace the LHS or RHS of the equation. The chosen debugging view is presented in a separate window, so the programmer may also continue the Observational Debugging.

Starting points: Interaction between the tools not only overcomes disadvantages of the Observational Debugging. It also allows the programmer to use a new tracing strategy when he intends to use the Algorithmic Debugger or the Redex-Trail browser.

Both tools, the Algorithmic Debugger and the Redex-Trail browser, suffer from a similar problem with the starting point of a tracing session. The Algorithmic Debugger always starts with the evaluation of the `main` CAF. The Redex-Trail browser only allows to start tracing from the program's result (or from a redex where the evaluation aborted). But the programmer often already knows which part of his program is responsible for an error. He may not know what exactly goes wrong but he has suspicions about the implementation of certain functions.

If he does not want to start the Algorithmic Debugger from the very beginning or the Redex-Trail browser from the very end of the program, he may instead start to use the Observational Debugging tool. He then observes the functions where he suspects an error and searches for an erroneous equation. He may then switch to the tracing tool of his choice to start the session at this point. This way large parts of the program which the programmer knows to be correct may not need to be considered.

7.2 Interaction from hat-detect

The Algorithmic Debugging tool `hat-detect` guides the programmer through the EDT by asking questions about the correctness of equations.

Incorrect redex: The debugger may ask a question about an equation where the LHS redex is already incorrect. This is the same problem as described for the Observational Debugger. The programmer may switch to the Redex-Trail browser `hat-trail` on the equation's LHS redex to find out about its creation.

Switching to the Redex-Trail browser: The programmer may decide to switch from the Algorithmic Debugger to the Redex-Trail browser – for example, to avoid the guidance of the Algorithmic Debugger. This is easily possible when the Algorithmic Debugger shows an erroneous equation. Then tracing may immediately be continued using the Redex-Trail browser on the RHS of an equation. Usually it does not make sense to trace the RHS of a correct equation, unless the programmer wants to learn about the steps of a correct evaluation. Switching to the Redex-Trail browser may help with equations which contain large data structures. The subexpressions of the data structures may be traced piecemeal with the Redex-Trail browser.

Manual Trusting: The Algorithmic Debugger allows the programmer to *manually trust* functions (page 76). This can be dangerous as the programmer may not be aware that the function is still erroneous. The Observational Debugging tool may be used to observe all applications of a given function identifier. The programmer may then make sure that all applications yield correct results, or he may only check special cases of applications. Hence, it may be useful to use the Observational Debugger before *manually trusting* a function. Therefore the `hat-observe` tool may be launched immediately from the Algorithmic Debugging tool to observe all applications of the outermost function symbol of the redex on the equation's LHS.

Switching to a different tool: Switching to a different tool from `hat-detect` is done by certain commands. Instead of answering the question about an equation with "*yes*" or "*no*" the programmer may choose to answer "*observe*" or "*trace*" to launch the Observational Debugger or the Redex-Trail browser in a separate window (complete command list of `hat-detect` is given on page 115). When the Redex-Trail browser is chosen the programmer has to decide whether to trace the redex on the LHS of the equation or the RHS expression.

7.3 Interaction from hat-trail

The Redex-Trail browser `hat-trail` allows the programmer to trace a computation by following the parent redexes of any (sub)expression. For completeness links to the other two debugging tools were added to `hat-trail`:

Invoking the Algorithmic Debugger: Inexperienced programmers may get lost when searching a large Redex-Trail for an error. They may decide to switch to the Algorithmic Debugger and use its guidance to locate the error. They may switch back to the Redex-Trail browser once they isolate the error more exactly. Hence, a link is provided from the tool `hat-trail` to `hat-detect`. Any redex in the Redex-Trail browser may be chosen and the Algorithmic Debugger is launched. The Algorithmic Debugging session then only considers the evaluation of the selected redex.

Invoking the Observational Debugger: For convenience and completeness the observational tool may also be launched from the Redex-Trail browser. This can be useful when the programmer spots an application of a certain function identifier and wonders about other applications of the same function. Therefore the Observational Debugger `hat-observe` may immediately be launched by choosing a function identifier in the Redex-Trail browser.

Switching to a different tool: Switching to a different tool from `hat-trail` is conveniently possible through the tool's graphical interface. The tool already allows to select single redexes or identifiers using the mouse. A double-right-click on a redex starts an Algorithmic Debugging session for the evaluation of the selected redex. A double-right-click on an identifier starts the Observational Debugger showing all applications of the selected identifier.

7.4 Summary

This chapter has shown how the programmer may gain from the offered links between all three debugging tools using the common Hat Redex-Trail structure. Each tool can be invoked immediately from each of the two other tools.

Different views of a part of the computation: Switching between the Algorithmic Debugger and the Redex-Trail tool may offer two different views of the same part of the computation. Both tools consider particular redexes of the evaluation, but the Algorithmic Debugger additionally shows each redex's result.

Magnification of the computation: The Observational Debugger always considers all matching applications of the entire program evaluation, resulting in a list of observations. When switching *from* the Observational to the Algorithmic Debugger or to the Redex-Trail browser only a single observation may be considered. This switching can be seen as a sort of magnification. It allows the programmer to see the evaluation details of a chosen equation and to find the cause of the error.

Overview of the computation: Conversely switching *to* the observational tool allows the programmer to get an overview of the entire computation. When the evaluation of a particular function application is considered in the Algorithmic Debugger or Redex-Trail browser, switching to the observational tool puts this one application in the context of the entire computation by considering all applications of the same function identifier. This allows to quickly switch between different parts of the computation. With the Algorithmic Debugger and the Redex-Trail browser moving between different parts of the computation is much slower.

Further advantages of the interaction:

- The programmer may choose to avoid the guidance of the Algorithmic Debugger by switching to the Redex-Trail browser, while the evaluation of the same redex is still considered.
- It is difficult to decide the correctness of *equations* containing large data structures. Switching from the Observational or the Algorithmic Debugger to the Redex-Trail browser may allow the programmer to verify the correctness of a data structure piecemeal.
- The evaluation of *wrong* redexes is a problem for the Algorithmic Debugger. Switching to the Redex-Trail browser at this point is a workaround. The cause of the incorrect redex may then be found without a decision about the correctness of the equation itself, which may not be intuitive (see section 6.4).
- The observation of a function allows to verify its correctness in the context of the particular program. If all observations are correct then it is safe to *manually trust* the function in the Algorithmic Debugger.

Remaining problems, future work: Though the interaction offers many advantages and unites the three separate tools to form a more powerful debugging and tracing system, some problems remain:

- The tools are still implemented separately and interaction is achieved by opening a new window whenever the view is switched. The three tools could be integrated into a single tracing tool, which would offer all three views. This integration is only hindered by the implementation. The Redex-Trail browser should ideally be implemented in Haskell like the other two debugging views to allow the integration. However, there are still problems with powerful and reliable graphical interfaces for Haskell, which is a requirement for the complex interface of the Redex-Trail browser.
- To take advantage of tool interaction the programmer now needs to learn about all three debugging views. The Redex-Trail browser is probably the most complex to be learned [CRW00]. But all three tools need experience to be used efficiently.

- Switching between the tools is a workaround which may help in certain situations. However, improvements are still necessary to the individual tools. The observational tool still suffers from problems when too many matching equations are found. The Algorithmic Debugger is of little use when large or complex data structures are involved. Means to customize the display of those data structures would be helpful. For example tree structures and other two-dimensional structures could be shown in a graphical form. A variant of HOOD named GHOOD [REI01] provides such a graphical interface which pretty prints tree structures.

Chapter 8

Implementation and Interfaces

This chapter describes the design and implementation of several interfaces to Hat's Redex-Trail structure. The interfaces support three different views of the Redex-Trail.

The goal is to allow convenient implementation of tools in Haskell. Access to the structures needs to be very efficient to handle large amounts of data in minimal time. The solution is to divide the implementation into a low-level part programmed in C and a high-level part programmed in Haskell.

8.1 Structure of the Interfaces

Accessing low-level information coded in a binary file is possible in a functional language like Haskell [WR98]. But reading and decoding such a file is still slower than in a low-level language as C. This is due the fact that programs compiled with current Haskell implementations still suffer from a slow-down factor compared to native C implementations of the same algorithm. Decoding a binary file with bit operations is a task which is especially easy to optimize in a low-level language like C. This speed-up advantage of C compared to Haskell is merely a constant factor and not one of complexity. But for the programmers' acceptance of a debugging and tracing tool it indeed matters whether the tools needs several seconds or several minutes to show a result. An initial test to decode an entire file of about 200MB holding the Redex-Trail graph took about 30 seconds to decode with a C program, compared to about 90 minutes for a Haskell program compiled with NHC. Though this was just a very specific test and it certainly cannot suggest any general slow-down factor for a Haskell program, it shows that the implementation should be split into several parts:

- A basic interface to the file holding the Redex-Trail structure is implemented in C.
- The algorithms providing the Observational Debugging view and deriving the EDT both involve searching large amounts of data and are therefore implemented in C.
- A Haskell interface layered on top of these interfaces provides a more abstract representation. It allows the actual debugging tools to be implemented in Haskell and also allows convenient implementation of additional tools.

The interface structure can be seen in figure 8.1. The details of the interfaces are described in sections 8.2 and 8.3.

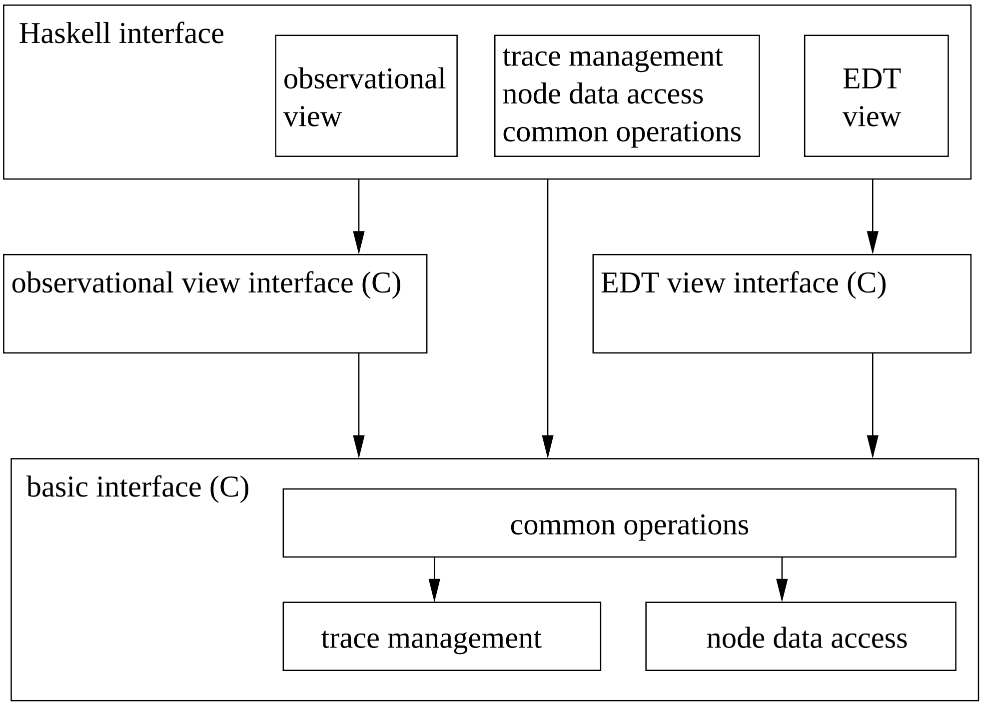

Figure 8.1: Dependencies of the interfaces to the Hat Redex-Trail file

8.2 The Low-Level Interfaces

Two important points need to be considered for the basic interfaces to the Redex-Trail structure:

1. The implementation details need to be hidden. This makes it possible to modify the underlying implementation without affecting the modules depending on the interface. Abstract data types are provided, which are only accessible by the functions defined in the interface.

2. The interface needs to be very efficient, both in terms of complexity and even in terms of a low constant factors. The Redex-Trail files easily grow very large (for example several hundred of megabytes). Just reading this amount of data from a local disk may consume considerable time (for example several seconds up to a few minutes). The acceptance of the tools using these interfaces may depend on the time they need to produce tracing information.

The basic interface

The basic interface to the Redex-Trail file is divided in three parts, as shown in figure 8.1.

1. **Trace management:** This part of the basic interfaces defines an abstract data type `HatFile` representing an entire Redex-Trail file. Operations on this type allow opening and closing files, and provide sequential access to the Redex-Trail (which is necessary for the observational tool). It is also possible to get information about the Redex-Trail, such as the location of the node representing the `main` CAF, a node representing a failed redex of an aborted evaluation and the error message of an aborted evaluation. A number of additional operations exist (see part 1 of the basic interface, appendix, page 116).

2. **Node data access:** This part of the interface is responsible for accessing nodes in Redex-Trail structure. Each node has a certain type. Depending on the type certain fields may be read. But the type of a node is only determined at runtime. So the program using the interface needs to test the type of a node at runtime and use the functions to access its fields accordingly.

   - One possibility is to define a function which returns a data structure holding the entire data stored in a node. The disadvantage is that each accessed node then needs to be decoded and copied to a separate structure. Often a program only needs to read a single field of a node, for example to determine the parent of a node.
   - A simpler and more efficient solution was chosen for the interface. The function `getNodeType` determines the type of node and returns it as an identifying value. At the same time it ensures that the entire node is within an internal buffer. Additional functions then provide access to fields of this node. Which node is accessed is only determined by the call to function `getNodeType`, hence the program needs to consider the type of a node before accessing it. However, it is still the responsibility of the programmer to consider the type correctly.

   The chosen solution is efficient as no unnecessary copying of data is involved. Only those fields are accessed and decoded which are really needed by the program. Still, it is possible to hide implementation details. For example the current interface abstracts some specific variants of nodes. The Redex-Trail files contains special nodes for *isolated* `SAT` nodes. Such variants are hidden by the interface as the difference only matters for the internal implementation of the function `getResult`, which determines the result of a redex. A similar approach was chosen for special *top-level* identifiers, which the interface treats as normal identifier nodes. The difference only matters for the function `getTopLevelFlag` which determines whether the identifier is defined locally or at the top level. Further functions of this part of the interface are listed on page 116.

3. **Common operations:** Some functions for frequently used operations were added. These include functions to determine the node representing the outermost symbol

of a nested application, functions following a trail of `SATC` nodes or to determine whether an identifier/constructor was defined within a *trusted* module. These functions were only defined for convenience as they do not add new functionality to the interface. They are implemented on top of the functions provided by the first two parts of the interface (see part 3 of the basic interface in the appendix, page 118).

Buffering: Each open `HatFile` is associated with an internal file buffer. The buffer always holds a number of successive nodes. Whenever a node is accessed by the function `getNodeType` it is ensured that the entire node is within the buffer.

The buffer is ideal for sequential access to the file, as needed for the observational view. But the buffer even proves to be effective when the Redex-Trail graph is searched along its links. Nodes in the Redex-Trail graph often reference nodes which were created shortly before or after the node itself was created. Due to the linearisation of the Redex-Trail graph these nodes are stored *close* to each other. Further buffering is left to the underlying operating system.

Interface to the Observational View

The interface for the observational view of the Redex-Trail file only needs to fulfill a single task. The list of all nodes describing applications for a given identifier needs to be accessed. The result of each application can be determined by the function `getResult` of the basic interface in order to form an equation. Two additional options described in chapter 4 are already implemented at this level. They allow restriction of the observation's scope and omitting recursive calls. These options are enabled by special arguments. The algorithm finding these applications was described on page 29.

Laziness in C: The language C does not support lazy evaluation. But *lazy* observations make the observational tool much more convenient to use. Searching the entire Redex-Trail file may take considerable time. Often the programmer may only demand to see a small number of equations and then decide to observe a different function identifier or to use a different tool. So only as many observations should be searched in the Redex-Trail file as are really demanded.

The interface therefore imitates a form of lazy evaluation. Instead of searching the entire file and returning the entire list of nodes a different approach using queries is implemented:

- An observation is initiated by the function `newObserveQuery`. It returns an abstract data type `ObserveQuery` representing the observation.
- A separate function `nextObserveQueryNode` returns a single observation, described by the node for the redex on the equation's LHS.

The interface to the Observational View is concise and shown in full in the appendix (see page 119).

Implementing Laziness: As the observing algorithm described on page 29 is purely iterative it can easily be suspended and resumed. Whenever a matching application is found it is immediately returned by the function. The entire state of the iteration is stored in the internal representation of the query. When the program needs another observation the state of the algorithm is restored and further iterations are done. The query is finished when no additional node is found or when the query is freed, as no more observations are demanded.

Interface to the EDT

The interface to the EDT view is superficially very similar to the one for the observational view. However, there are major differences in its implementation.

Each EDT node is described by a redex in the Redex-Trail graph. In order to form the entire label of an EDT only the result of the redex needs to be determined, which is possible through the basic interface.

- A search for all EDT children of a node is initiated by a query, created by the function `newEDTQuery`.
- The nodes describing the EDT children can be demanded one by one using the function `nextEDTQueryNode`.

The interface to the EDT View is concise and shown in full in the appendix (see page 120).

Laziness of a recursive algorithm: The algorithm finding all EDT children of a node consumes less time than the observation algorithm as only a part of the Redex-Trail file needs to be searched. Still, to offer best response time and to avoid unnecessary evaluation a lazy implementation is provided. An *iterative* algorithm can easily be suspended and resumed as described for the observational interface (page 93). But algorithm 4.8 (page 43) determining the EDT children of a node is *recursive*. The problem with a recursive algorithm is that the state of the recursion cannot easily be stored and restored. Access to the call stack is impossible, so the state of the recursion needs to be saved in a different way.

Transforming the recursive algorithm: The solution is to transform the recursive algorithm into an iterative one. Instead of using the call stack by making recursive calls an internal buffer is used. Algorithm 4.8 searches a graph in a specific way and for specific nodes. Instead of making recursive calls to search subgraphs the search is delayed. The subgraphs are stored in a buffer. Each iteration considers an entire recursion step and buffers all further recursion steps. When a recursion step has finished a new subgraph is taken from the buffer and the next iteration is begun.

Ordering the results: Care needs to be taken not to change the sequence in which the algorithm finds the nodes representing the EDT children. The recursive algorithm returns the EDT children in a leftmost-outermost-first order, which must not be altered. This is achieved in the iterative solution by the following steps:

- In each iteration a separate list is maintained, holding the new subgraphs to be searched. Subgraphs are appended to this list as they are found, so they are ordered in the same way as they are observed by the recursive version.
- After each iteration the separate list is attached to the beginning of a global list of subgraphs. Hence, the first subgraph observed in an iteration is the next one to be searched after the current iteration has finished.
- When an iteration finds a matching EDT child node it must *not* consider it immediately. Instead this result is delayed by storing it in the same list which keeps the delayed subgraphs. This ensures that the subgraphs which were found in the same iteration before the node was found are searched first.
- When an iteration has finished the first subgraph is taken from the global list and searched. When the first element in the list is a result node and not a subgraph the result node is returned by the function and the iteration is suspended.

An example of the transformation of a recursive search algorithm to an iterative solution is given in figure 8.2 (page 96).

8.3 The High-Level Interface: Interfacing from Haskell

As described in section 8.1 a Haskell interface is layered on top of the low-level interfaces to allow more convenient tool design. The Observational and Algorithmic Debugging tools (chapters 5 and 6) are programmed in Haskell using the Haskell interface.

Connecting two alien worlds: The low-level interfaces implemented in C are accessed from the Haskell world using the Green Card foreign-language interface. Green Card allows Haskell functions to have function bodies implemented in C. Code for marshalling function results and unmarshalling parameters of basic types is automatically introduced. Differences between different Haskell compilers are therefore hidden by Green Card.

But the access to the C world needs to be implemented carefully. Functions in a functional language need to be well defined. Each function must represent a proper mapping from arguments to result with no (visible) side effects to the evaluation of the Haskell program. But in general functions in C use many side effects, which must not affect the Haskell world. Where side effects are visible in Haskell, such as effects of IO operations, monadic types need to be used in Haskell to force correct sequencing of such operations.

IO Operations in the Haskell world: Accessing the Redex-Trail file through the basic interface involves IO operations, which are side effects of calls to the interface's functions. On the Haskell side monadic types are used for IO operations, which guarantees their correct sequencing and avoids side effects in a functional language. But using function of monadic types is less convenient than functions of basic types, and it also limits the advantages of laziness in Haskell. Though the result of an IO operation may not be needed

```
procedure rekSearch(subgraph);
begin
  ...
  if ... then ...
    rekSearch(graphA);
    ...
    if foundNode(X) then
      considerResult(X);
    fi
    ...
    rekSearch(graphB);
    ...
  fi
end
```

```
procedure iterativeSearch(subgraph);
begin
todo := [subgraph];
while (todo != []) do
  if isResult( head(todo) ) then
    considerResult( fromResult( head(todo) ) );
    todo := tail(todo);
  else
    subgraph := head(todo);
    todo := tail(todo);
    local := [];
    ...
    if ... then ...
      local := append(local, Graph graphA);
      ...
      if foundNode(X) then
        local := append(local, Result X);
      fi
      ...
      local := append(local, Graph graphB);
      ...
    fi
    todo:=concat(local, todo);
  fi
od
end
```

Figure 8.2: Example transformation of an recursive to an iterative search algorithm

by the computation, it must still be performed if it is sequenced with another operation, which yields a demanded result.

The Redex-Trail structure as a large constant: The basic interface to the Redex-Trail structure does not provide any operations modifying the file. Hence, the Redex-Trail structure can be seen as one large constant. Operations accessing this constant are projections, returning a part of this constant – for example a certain field of a certain node. So, there is no need for functions accessing this constant to be of monadic type. Only the function initially opening the trace file and returning a representation for the entire constant needs to be of monadic type as its side effect is visible in Haskell: every open operation returns a *different* representative. Explicitly closing a file is not necessary; closing is done automatically when the representation of the constant is garbage collected.

Haskell Interface to the Redex-Trail: The Haskell interface provides two abstract data types to access the Redex-Trail:

1. A representation of an entire Redex-Trail structure is given by a value of type `HatTrail`. Its internal structure is only known to the C world. But such a C data structure may still be used in the Haskell world as a *foreign object*.

2. A single node of the Redex-Trail structure is represented by a value of type `HatNode`. Its internal structure is only accessible within the Haskell module defining the interface. Internally `HatNode` contains a tuple of the Redex-Trail representation `HatTrace` and a specific node address.

Two kinds of functions for `HatTrace` are briefly described:

- A representation for a Redex-Trail file is obtained by the function
  `openTrace :: String → IO (Maybe HatTrace)`
  of the Haskell interface. It tries to open a Redex-Trail file and returns a `HatTrace` object when successful.

- Several functions provide representations of specific nodes in the Redex-Trail: for example the first node in sequential order, the node representing the `main` CAF or a redex where the evaluation aborted:
  `hatTraceFirst :: HatTrace → HatNode`

The important functions for `HatNode` allow sequential access to the Redex-Trail structure and access to the node data:

- Sequential access to the trace is possible with the function
  `hatTraceNext :: HatNode → HatNode`

- Access to the node data is provided in a very similar fashion to the basic C interface. Different functions access the different fields of a node, such as the parent of a node. The type of a node, determining which fields may be accessed, is also obtained by a function.
  ```
  hatNodeType :: HatNode → HatNodeType
  hatParent   :: HatNode → HatNode
  ```

  The functions accessing data of a `HatNode` are implemented as methods of a class `HatRep`. The type `HatNode` is an instance of this class. This allows different Haskell representations of a redex trail node to be mixed when an instance of the class is defined for this different representation.

Disadvantage of the interface: This implementation of the Haskell interface to the Redex-Trail does have two disadvantages:

- Accessing the field of a node several times may result in multiple file accesses. Each call to a function of the interface immediately makes a call to the underlying basic interface implemented in C.

- Each field needs to be accessed by a separate function call. A data structure which allows pattern matching for a node and determines the fields of a node lazily may be more convenient for a Haskell programmer.

Different interfaces to the Redex-Trail: These disadvantages are easily overcome by a definition of an additional Haskell data type. The described Haskell interface is used to construct an entire expression tree. Each node in this tree holds the information given in the Redex-Trail. As Haskell evaluates the fields of the nodes lazily the file is accessed at most once for every field and the expression tree is build only as far as necessary.

```
data HatExpressionTree = HatConstant    {parent :: HatExpressionTree,
                                          fun    :: HatExpressionTree,
                                          result :: HatExpressionTree} |
                         HatConstructor {name      :: String,
                                          infixType :: HatInfixType} |
                         ...
```

A class instance of `HatRep` is also introduced for this type. Hence, the same functions accessing `HatNode` may also be used. Both representations may be mixed when only the class methods are used for access.

The disadvantage of this type is that keeping whole expression trees in memory may exceed the system's memory, especially when many expressions of a file are considered. The programmer has to choose between the different representations.

Additional representations of the Redex-Trail may be introduced to optimize access for a specific purpose. For example a representation may define a data type which keeps all basic data of a node but where the links between the nodes are still represented by `HatNode`. This way the basic data of a node is read lazily but the links between the nodes do not lead to linked Haskell representations, so the construction of an entire expression tree in memory is avoided. Again, each introduced representation may implement the class methods of `HatRep` so the different representations may easily be exchanged.

Observational and EDT View

The Haskell interface to the observational and EDT view is simple as the query system implemented for the C interfaces is unnecessary. Instead simple functions are provided which return the entire list of observed applications or all EDT children respectively:

```
data ObserveResult = IdentifierNotFound    |
                     TopIdentifierNotFound |
                     Found [HatNode]

observe     :: HatTrace → String → String → Bool → ObserveResult
edtChildren :: HatRep a ⇒ a → [HatNode]
```

- The arguments to the function `observe` are the representation of the entire Redex-Trail file to be searched, the function identifier to be observed, an optional second identifier restricting the scope (may be an empty string) and a flag determining whether to omit recursive calls. Its result may indicate that the first or second identifier never occurred in the Redex-Trail, otherwise the entire list of observed applications is returned.

- The only argument to the function `edtChildren` is a representative of a Redex-Trail node, so any class instance of `HatRep` can be used. Its result is the list of all EDT children described as `HatNode` elements.

The result lists of both functions are build lazily. The C-level query system is hidden by the Haskell interface. Whenever an additional list element is demanded on the Haskell side then the interface translates this to a call to the query across the C-level interface.

8.4 Summary

This chapter has introduced a low-level interface in C and a more abstract interface in Haskell, providing access to the Redex-Trail file and to both derived debugging views.

- The basic interfaces in C are optimized for speed. They still provide abstract data types which hide implementation details. So any modifications to the Redex-Trail file structure can be hidden even at this level.

- The Haskell interface requires some overhead for the conversion of the C data types to Haskell representations. It still offers full access to the Redex-Trail. Two examples of facilities implemented entirely in Haskell are pretty printing of expressions and pattern-matching for observed equations.

- The Observational and the Algorithmic Debugging views of a Redex-Trail are implemented in C using the basic interface to the Redex-Trail. Both search the Redex-Trail graph lazily. The algorithm deriving the Algorithmic Debugging view of the Redex-Trail was transformed into an iterative solution so that the algorithm can be suspended and resumed after each iteration.

- The Observational and Algorithmic Debugging view may be interfaced from Haskell elegantly. Lazy observations and lazy EDT generation is provided in the natural way of Haskell by lazily generating the result list containing all observations or EDT children respectively. The use of C-level queries to support laziness is hidden by the Haskell interface.

Chapter 9

Case Study

In this chapter a larger example is presented showing the use of the tracing toolkit in practice. The example tracing sessions focus on using the implemented Observational and Algorithmic Debugging tool and the advantages of tool interaction.

9.1 Adjoxo - the example program

As an example the program *Adjoxo*[1] is used which analyses positions of the game *noughts-and-crosses*. The game board consists of a gird of three times three squares. Two players alternately mark one field with a cross or a nought respectively. The player first marking three fields in a vertical, horizontal, or diagonal line wins.

The program considers a game position defined within the source-code. It determines whether the player "X" or his opponent "O" can force a win, guarantee a draw or loses assuming both players make optimal moves.

How the program works: The basic idea of the program is to test all possible moves for the player in turn and to determine the result of the best move. For each move all possible moves of the opponent are tested recursively. The recursion ends when a player achieved a line of three squares or the grid is filled.

Fields in the grid are numbered 1 to 9 from left to right and top to bottom. The grid position is represented by two ordered lists of numbers: one list for the fields marked by the respective player. An important function in the program is `analysis`. Its arguments are the two lists representing the current grid position. The first argument list describes the fields marked by the active player who needs to make the next move. Its result is the value of the best possible move: `Win`, `Draw`, or `Loss`.

A more detailed description of the program is unnecessary to follow the example sessions. This shows that the tools also help in understanding an unknown program.

[1] *Adjoxo* is written by Prof. Colin Runciman and used by courtesy.

9.2 Example debugging sessions

Four bugs were deliberately introduced in the program's source-code and need to be found using the tracing tools. The source-code is shown in the appendix (page 124) where all bugs are clearly marked.

In the example debugging sessions[2] programmer input is printed in *italic text*. Output of debugging tools and the analysed program is shown in `typewriter font`. A detailed description of the commands for the Observational and Algorithmic Debugging tool is shown in the appendix (pages 113 and 115).

Running the example program

The program (page 124) is compiled and run. The option `-T` causes a compilation for tracing. The compiled program generates the Redex-Trail structure during evaluation.

```
thorsten:~/York/adjoxo # hmake -T Adjoxo
nhc98 -T -c -o Adjoxo.T.o Adjoxo.hs
nhc98 -T -o Adjoxo Adjoxo.T.o

thorsten:~/York/adjoxo # ./Adjoxo
if X to play:  WIN for X
if O to play:  WIN for O

Program exited normally.
```

The program output is erroneous. With the example grid defined in the program (see `input`, page 126) the player "X" *cannot* force a win if he makes the first move. The optimal strategy for player "X" only guarantees a draw.

First observation session

Why does the program have erroneous output? We try to find the error by starting with Observational Debugging. It is also possible to use the Algorithmic Debugger or Redex-Trail browser to start debugging.

```
thorsten:~/York/adjoxo # hat-observe Adjoxo

Welcome to hat-observe 1.11
Type 'help' for list of commands

command> observe adjudicate
searching for:  adjudicate
#1:  adjudicate ([1],[2]) =
     "if X to play:  WIN for X\nif O to play:  WIN for O\n"
```

[2]All example sessions can be tested immediately using the attached CD. It contains pregenerated Hat Redex-Trail files and precompiled debugging tools for Linux/386 machines. For other operating systems/machine types an installation of NHC98 is necessary. See CD's index file for more details.

There is only one observation of function `adjudicate`. The function is called immediately from `main`. Its two arguments show that the defined example grid was parsed correctly into the grid representation. But the equation is erroneous as "X" can only achieve a draw if to play. Now we observe function `analysis` as it is the main function determining the value of the best move for a given grid position.

```
command> observe analysis
searching for:  analysis
#1:  analysis [2] [1] = Win
#2:  analysis [1] [2] = Win
#3:  analysis [1] (3:_) = Loss
#4:  analysis [2] (3:_) = Loss

command> verbose
verbose mode is now ACTIVE

command> go 1
#1:  analysis [2] [1] = Win
#2:  analysis [1] [2] = Win
#3:  analysis [1] (3:insert 3 []) = Loss
#4:  analysis [2] (3:insert 3 []) = Loss
```

The observation of `analysis` resulted in four equations only. All equations except the second are erroneous. The last two equations are especially interesting. Function `analysis` managed to reduce to its result without ever considering the entire argument list. It is *wrong* that the general grid position `[1] (3:_)` forces a `Loss` for the first player.

And an additional error is visible. The grid position `[1] (3:_)` must never occur during evaluation. In the initial grid position player "X" has already marked field `2` and player "O" marked field `1`. The game does not allow to remove fields. Therefore every occurring grid position needs to be represented by two lists starting with `1` and `2`. So we have found an evaluation of a redex which should not occur.

The use of *verbose mode* is also demonstrated. Unevaluated expressions are shown when *verbose mode* is active. In *verbose mode* it is visible that the *unevaluated* expression even represents an entirely illegal grid position: `(3:insert 3 [])`. A field must not be marked twice.

Tracing an incorrect redex: Finding the first bug

Where did this incorrect argument `(3:_)` come from? The Observational Debugger cannot answer this question. But with interaction we can switch to the Redex-Trail browser to trace what created the incorrect expression. We invoke a Redex-Trail browser session for the LHS of the third equation:

```
command> trace 3
Equation 3:  Trace lhs or rhs of equation?  (L/R): lhs
```

A separate Redex-Trail browser is launched where we select the subexpression `3:insert 3 []` in order to see its creating parent:

```
•   analysis (1:[]) (3:insert 3 [])
▼   ▼ insert 3 (2:[]) ▷ | True
```

The first bug in the program is found. The rule used to reduce the redex `insert 3 (2:[])` created the expression `3:insert 3 []`. This rule must be erroneous. In the definition of `insert` (page 124) we need to consider the `x > y` alternative as `x` is `3` and `y` is `2`. The error marked as `Bug 2` is found and can be fixed.

Following an erroneous evaluation: Finding the second bug

After fixing the previous bug it is unnecessary to recompile and rerun the program. We already found a second error in the program. The bug in `insert` does not explain why the evaluation of `analysis [1] (3:_)` resulted in `Loss`. We return to the window of the Observational Debugging session and invoke the Algorithmic Debugger to find the error in the third equation's evaluation:

```
command> debug 3
```

A separate window opens and provides an Algorithmic Debugging session:

```
Welcome to hat-detect 1.11
1> analysis [1] (3:_) = Loss (Y/?Y/?N/N): no
2> hasLine (3:_) = True (Y/?Y/?N/N): no
3> subset [1,2,3] (3:_) = True (Y/?Y/?N/N): no
4> dif [1,2,3] (3:_) = [] (Y/?Y/?N/N): no
5> dif [2,3] (3:_) = [] (Y/?Y/?N/N): no
6> dif [3] (3:_) = [] (Y/?Y/?N/N): yes

Erroneous reduction:  dif [2,3] (3:_) = []
Bug found within the body of function:  "dif"
line 31, column 1 in module "Main", file:  "Adjoxo.hs"
```

It is *wrong* in general that `3:_` has a line of three marked fields, so `hasLine (3:_) = True` is erroneous. The answers to the remaining questions are straightforward. Eventually the Algorithmic Debugger locates a bug in `dif`'s definition (page 124). In its body the `LT` alternative needs to be analysed. The bug marked as `Bug 3` is found and needs to be fixed in the source-code.

After this bug is fixed the program is recompiled and rerun. But unfortunately the program still results in the same erroneous output.

Second observation session

The Observational Debugger is used to find the remaining errors. This time an observation of `analysis` results in many equations (350 unique equations). Many of them are erroneous. We therefore decide to observe applications only where the grid position is represented by two lists of three elements. The advantage of first considering these applications is that their evaluation is much shorter. The considered grid positions have six marked fields and only three fields remain for further moves. The evaluation of these applications is therefore easier to analyse.

```
Welcome to hat-observe 1.11
Type 'help' for list of commands
command> observe analysis [_,_,_] [_,_,_]
searching for:  analysis
#1:  analysis [2,3,5] [1,4,6] = Win
#2:  analysis [2,3,5] [1,4,7] = Loss
#3:  analysis [2,3,6] [1,4,5] = Loss
#4:  analysis [2,3,7] [1,4,5] = Win
#5:  analysis [2,3,7] [1,4,6] = Win
command (<RETURN> for more equations)>
```

This observation with a pattern still results in many equations (81 equations) but only the first five are shown. The third equation is erroneous. Again we use the Algorithmic Debugger to find the cause.

Following the erroneous evaluation: Finding the third bug

The Algorithmic Debugger is invoked from the Observational tool to debug the third equation:

```
command (<RETURN> for more equations)> debug 3
```

This starts the following Algorithmic Debugging session:

```
Welcome to hat-detect 1.11

1> analysis [2,3,6] [1,4,5] = Loss (Y/?Y/?N/N): no
2> hasLine [1,4,5] = False (Y/?Y/?N/N): yes
3> gridFull [2,3,6] [1,4,5] = False (Y/?Y/?N/N): yes
4> insert 7 [2,3,6] = [2,3,6,7] (Y/?Y/?N/N): trust
   Ok, "insert" is trusted from now on.
5> analysis [1,4,5] [2,3,6,7] = Win (Y/?Y/?N/N): yes
6> inverse Win = Loss (Y/?Y/?N/N): yes
8> analysis [1,4,5] [2,3,6,8] = Win (Y/?Y/?N/N): yes
10> dif [1,2,3,4,5,6,7,8,9] [2,3,6] = [1,4,5,7,8,9] (Y/?Y/?N/N): trust
    Ok, "dif" is trusted from now on.
13> analysis [1,4,5] [2,3,6,9] = Win (Y/?Y/?N/N): no
14> hasLine [2,3,6,9] = False (Y/?Y/?N/N): no
15> subset [1,2,3] [2,3,6,9] = False (Y/?Y/?N/N): yes
16> subset [4,5,6] [2,3,6,9] = False (Y/?Y/?N/N): yes
17> subset [7,8,9] [2,3,6,9] = False (Y/?Y/?N/N): yes
18> subset [1,4,7] [2,3,6,9] = False (Y/?Y/?N/N): yes
19> subset [2,5,8] [2,3,6,9] = False (Y/?Y/?N/N): yes
20> subset [3,7,9] [2,3,6,9] = False (Y/?Y/?N/N): yes
21> subset [1,5,9] [2,3,6,9] = False (Y/?Y/?N/N): yes
22> subset [3,5,7] [2,3,6,9] = False (Y/?Y/?N/N): yes
```

```
Erroneous reduction:  hasLine [2,3,6,9] = False

Bug found within the body of function:  "hasLine"
line 43, column 1 in module "Main", file:  "Adjoxo.hs"
```

To demonstrate *trusting* (see page 76) we have answered the questions about functions `insert` and `dif` with "*trust*". After fixing the previous bugs in these functions and thoroughly checking their definitions we are sure that there are no remaining bugs. The tool therefore omitted questions 7,11, and 12.

The questions about applications of `analysis` are the most difficult to answer. It would be helpful to see a graphical representation of the grid instead of two separate lists of numbers. When necessary the gird can be visualized manually.

We have spotted the bug already at question 20 but did answer the remaining questions so that the Algorithmic Debugger shows the bug location. The equation `subset [3,7,9] [2,3,6,9] = False` is correct. But why did the program evaluate the redex on the LHS of this equation? The grid positions `[3,7,9]` do not represent any horizontal, vertical or diagonal line. The correct line is the grid's last column `[3,6,9]`. Instead of answering the last two questions we could have also used the Redex-Trail browser to determine what created the incorrect redex `subset [3,7,9] [2,3,6,9]`.

We can now fix the bug in the definition of `hasLine` (`Bug 4`, page 125).

Searching the last bug

After repairing the previous bug we again recompile and rerun the program. Unfortunately the program still results in the very same erroneous output. Again we can use the Observational Debugger to observe function `analysis`. To determine whether `analysis` has caused the program to be erroneous we use the *recursive-call-filter* (see page 64) to omit all recursive calls to `analysis`.

```
Welcome to hat-observe 1.11
Type 'help' for list of commands
command> recursive
recursive-call-filter is now ACTIVE
command> observe analysis
searching for:  analysis
#1:  analysis [2] [1] = Win
#2:  analysis [1] [2] = Win
command>
```

Only two equations remain (The tool omitted 348 recursive calls.). We notice that function `analysis` is still erroneous as the first equation is wrong. So we could now observe *all* applications of `analysis` to find another erroneous application where the grid is almost filled which eases Algorithmic Debugging. But this time almost all applications of `analysis` reduce to correct results. The available filters in the Observational Debugging tool are of little help in order to find the few erroneous equations. And of course we do not want to consider *all* applications of `analysis` (including the 348 recursive calls). So, we need to check the entire evaluation of the erroneous equation `analysis [2] [1] = Win`. We invoke the Algorithmic Debugger for the first observed equation:

```
command> debug 1
```

The resulting Algorithmic Debugging session is as follows:

```
1> analysis [2] [1] = Win (Y/?Y/?N/N): no
2> hasLine [1] = False (Y/?Y/?N/N): trust
   Ok, "hasLine" is trusted from now on.
3> gridFull [2] [1] = False (Y/?Y/?N/N): trust
   Ok, "gridFull" is trusted from now on.
4> insert 3 [2] = [2,3] (Y/?Y/?N/N): trust
   Ok, "insert" is trusted from now on.
5> analysis [1] [2,3] = Win (Y/?Y/?N/N): yes
6> inverse Win = Loss (Y/?Y/?N/N): trust
   Ok, "inverse" is trusted from now on.
8> analysis [1] [2,4] = Win (Y/?Y/?N/N): no
12> analysis [2,4] [1,3] = Win (Y/?Y/?N/N): yes
15> analysis [2,4] [1,5] = Loss (Y/?Y/?N/N): no
19> analysis [1,5] [2,3,4] = Win (Y/?Y/?N/N): yes
22> analysis [1,5] [2,4,6] = Win (Y/?Y/?N/N): yes
25> analysis [1,5] [2,4,7] = Win (Y/?Y/?N/N): yes
28> analysis [1,5] [2,4,8] = Win (Y/?Y/?N/N): yes
30> dif [1,2,3,4,5,6,7,8,9] [2,4] = [1,3,5,6,7,8,9] (Y/?Y/?N/N): trust
    Ok, "dif" is trusted from now on.
33> analysis [1,5] [2,4,9] = Win (Y/?Y/?N/N): no
37> analysis [2,4,9] [1,3,5] = Loss (Y/?Y/?N/N): no
41> analysis [1,3,5] [2,4,6,9] = Win (Y/?Y/?N/N): yes
44> analysis [1,3,5] [2,4,7,9] = Draw (Y/?Y/?N/N): yes
49> analysis [1,3,5] [2,4,8,9] = Win (Y/?Y/?N/N): yes
51> bestOf Draw Loss = Loss (Y/?Y/?N/N): no

Erroneous reduction:  bestOf Draw Loss = Loss

Bug found within the body of function:  "bestOf"
line 8, column 1 in module "Main", file:  "Adjoxo.hs"
```

This is a long debugging session with many difficult questions about function `analysis`. This shows that the Algorithmic Debugging tool is not always convenient to use. We trusted the functions `insert`, `dif`, and `hasLine` as we have considered them before. Also `gridFull` is trusted as its implementation is trivial. The tool therefore asked only 20 questions. Without *memoisation* (page 74) and *trusting* all 51 questions had to be answered. Still, this debugging session was not very convenient as 14 out of 20 questions involved function `analysis`.

But eventually the last remaining bug is found in function `bestOf`. Its definition needs to be corrected so that `bestOf Draw Loss` results in `Draw` instead of `Loss` (see `Bug 1`, page 124).

9.3 Summary

This example has emphasized some advantages and also some problems in practical use.

- The advantages of interaction between the Observational Debugging tool and the other two tools were demonstrated. We quickly succeeded in finding the cause of an incorrect redex. We also managed to find two bugs in short Algorithmic Debugging sessions. Instead of starting Algorithmic Debugging at the beginning of a program we picked specific equations with the Observational Debugging tool.
- We used the Algorithmic Debugger without detailed knowledge about how the program works. For the Observational Debugger it is necessary to consider the source-code in order to choose functions to observe.
- Questions about function `analysis` needed some effort to answer. It would be much easier if there was an automatic graphical visualization of the grid position. In this example it is still possible to do so manually. But for examples with a larger grid of six times six squares manual visualization would be very inconvenient.
- The filters in the Observational Debugging tool were helpful in several situations – for example the *recursive-call-filter* and *pattern matching*. But in the last observational session these filters did not help much in finding a specific equation of interest. There was no method to find an erroneous equation involving `analysis` with the longest possible argument lists in order to minimise the effort of Algorithmic Debugging. There were too many correct equations.
- The final Algorithmic Debugging session has shown that a lot of effort may be necessary. *Trusting* and *memoisation* were useful as they reduced the number of questions asked from 51 to 20. But the questions omitted considered the evaluation of simple or even trivial functions. The remaining questions were the ones difficult to answer.

Chapter 10

Conclusion

A goal of this thesis was to support two new tracing and debugging views based on Redex-Trails: Observational and Algorithmic Debugging. Also interaction between these tools and the existing Redex-Trail browser was to be supported. Furthermore an efficient and convenient interface to the Redex-Trail and the derived debugging views should allow the implementation of further debugging tools.

Supporting Observational Debugging: A slightly modified approach to Observational Debugging was introduced, which restricts to observations of functions rather than basic values. In practice this is a minor disadvantage only. Programmers are especially interested in observing functions as the resulting list of equations is much more expressive than a list of observed basic values.

It was shown that the necessary information to support this modified Observational Debugging approach can be derived efficiently from the Redex-Trail structure. The main problem to be solved was to find a *forward* link from a redex to its evaluation result. The original browsing tool for the Redex-Trail structure only needed *backward* links pointing from a redex to its creating parent. It was shown that the necessary *forward* links are implicitly given in Hat's Redex-Trail structure.

Observational Debugging tool: A tool for Observational Debugging was implemented which provides several filters to refine the programmer's observations. The filters reduce the number of equations observed by omitting redundant or unwanted equations. They support the programmer in finding the information he is really interested in and are an advantage compared to the original Observational Debugger HOOD. An important improvement of the new approach for Observational Debugging is that a series of observations can be done conveniently without any modifications to a program's source-code.

However, HOOD still has an advantage concerning the preciseness of observations: it shows expressions evaluated exactly as far as demanded at the point of observation. Observations based on the Redex-Trail structure show expressions as far evaluated as they were demanded by the entire program evaluation. Due to sharing of expressions these two approaches may differ.

A remaining problem of Observational Debugging is the potentially large number of equations observed. The presented filters may not help enough in some situations. For example the use of pattern matching for equations is limited. Patterns matching complex data structures can hardly be defined manually by a programmer.

Supporting Algorithmic Debugging: The basic data structure necessary for Algorithmic Debugging in functional languages is the *Evaluation Dependency Tree*. It was shown that an EDT can be derived immediately from the Redex-Trail structure. It is possible to build the EDT piecemeal so that only parts demanded by an Algorithmic Debugging session need to be derived. Finding the children for an EDT node is efficient as only a part of the Redex-Trail graph needs to be searched.

There were no fundamental problems in deriving an EDT from a Redex-Trail. But a number of problems were discussed which resulted from the specific Redex-Trail representation chosen for Hat:

- CAFs unlike other redexes do not refer their creating parents. A solution was presented so that CAFs are still included in the derived EDT. But possible flaws in the generated EDT need to be accepted. However, it was shown that these flaws are hardly noticeable in the implemented Algorithmic Debugging tool.
- Another problem results from the absence of context information for locally defined functions and lambda abstractions. Therefore it is impossible to decide the correctness of equations involving such functions. The concerning EDT nodes have to be omitted. The children of such nodes are lifted to mitigate the problem.

Algorithmic Debugging tool: The Algorithmic Debugging tool does not allow to find errors in the definition of local functions or lambda abstractions, as their applications were omitted in the EDT. Instead the tool locates an error at the enclosing top-level function. This is a drawback compared to the original implementation of Algorithmic Debugging by the Freja system.

But a major advantage is that the new tool can be used for programs in practice. Additionally the tool was improved compared to Freja by implementing *memoisation* and *manual trusting*. Both options reduce the number of questions a programmer needs to answer in order to locate an error.

Manual trusting shows to be dangerous when a user-defined function has an higher-order parameter. A solution to this problem was presented by lifting the EDT nodes describing the evaluation of applications of the higher-order argument. But this solution was not implemented and programmers should therefore avoid to *manually trust* functions with higher-order parameters.

Common problem of both debugging tools: Both implemented debugging tools suffer from a problem when equations contain large data structures. A programmer may not be able to decide whether such an equation is correct or erroneous. This especially affects the Algorithmic Debugger as a programmer may get stuck in a debugging session and postponing such questions may not help. More sophisticated ways to display complex data structures could be implemented – for example to layout tree structures. But eventually the size of equations always limits a debugging tool, as programmers can only comprehend an evaluation of a limited complexity. As a solution programmers need to use smaller program input resulting in less complex data structures for testing purposes.

Extending the Redex-Trail: Both implemented tools could be improved by recording additional information in the Redex-Trail – for example the creating parents of CAFs and the context of locally defined functions. But the absence of this information results from design decisions made for Hat. A trade-off between recording necessary information and the efficiency of Redex-Trail generation needs to be considered. Additional information can only be recorded if an efficient solution is found which does not cause severe slow-down of the Redex-Trail generation.

Interaction between tracing and debugging tools: The main advantage of supporting three separate tracing tools based on a common data structure is the possible interaction between the tools. It was shown that the two implemented debugging tools and the Redex-Trail browser are complementary. It was discussed how a programmer gains from interaction. He may use a new tracing strategy and a number of problems of the individual tools are overcome:

- If the programmer knows which function in his program is erroneous he may avoid the fixed starting points of the Algorithmic Debugger or the Redex-Trail browser: he initially observes the function, picks an erroneous equation and switches to the Algorithmic Debugger or the Redex Trail browser.
- A programmer may swap between the guidance of Algorithmic Debugging and the unguided exploration with the Redex-Trail browser.
- A programmer may be puzzled when considering the evaluation of an already incorrect *redex*. He may not be able to continue an Algorithmic or Observational Debugging session. The cause of the incorrect redex can be found easily by switching to the Redex-Trail browser.

Interfaces: Interfaces to the Redex-Trail structure and to both derived debugging views were implemented in C and a high-level interface in Haskell was layered on top. The presented debugging tools were already implemented in Haskell. For the implementation of additional tools a programmer can choose between the low-level interface in C optimized for speed and the convenient Haskell interface.

Bibliography

[CHI01] Olaf Chitil. Pretty Printing with Lazy Dequeues. In *2001 ACM SIGPLAN Haskell Workshop*, 2001.

[CRW00] Olaf Chitil, Colin Runciman and Malcolm Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, LNCS 2011, pages 176–193. Springer, 2001.

[GIL00] Andy Gill. Debugging Haskell by observing intermediate data structures. In *2000 ACM SIGPLAN Haskell Workshop*, 2000. Technical Report NOTTCS-TR-00-1, University of Nottingham.

[NAI97] Lee Naish. A Declarative Debugging Scheme. In *The Journal of Functional and Logic Programming*, MIT Press, Volume 1997, Article 3.

[NIL94] Henrik Nilsson and Peter Fritzson. Algorithmic Debugging for lazy functional languages. In *The Journal of Functional Programming*, Volume 4 (3), July 1994.

[NIL98] Henrik Nilsson. Declarative Debugging for Lazy Functional Languages. PhD Thesis, Linköping University, Sweden, 1998.

[OKA98] Chris Okasaki. Purely Functional Data Structures. Cambridge University Press 1998.

[PEN99] Alastair Penny. Augmented Trace-based Functional Debugging. PhD Thesis, University of Bristol, September 1999.

[PPRS01] C. Pareja, R. Peña, F. Rubio and C. Segura. Adding Traces to a Lazy Monadic Evaluator. Departmento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain.

[REI01] Claus Reinke. GHood - Graphical Visualisation and Animation of Haskell Object Observations. In *2001 ACM SIGPLAN Haskell Workshop*, 2001.

[SHA82] Ehud Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, MA, 1982.

[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 and P. Hartel, editors, *Proceedings of the Ninth International Symposium on Programming Languages: Implementations, Logics and Programs, Lecture Notes in Computer Science*, volume 1992, pages 291-308. Springer, September 1997.

[WCBR01] Malcolm Wallace, Olaf Chitil, Thorsten Brehm and Colin Runciman. Multiple-View Tracing for Haskell: a New Hat. In *2001 ACM SIGPLAN Haskell Workshop*, 2001.

[WIR86] Niklaus Wirth. *Algorithms and data structures.* Prentice Hall, 1986.

[WR98] Malcolm Wallace and Colin Runciman. The Bits Between The Lambdas: Binary Data in a Lazy Functional Language. In *Proceedings of the International Symposium on Memory Management*, Vancouver, Canada, Oct 1998.

Appendix

Using the Observational Debugging tool hat-observe

The Observational Debugging tool is invoked by the following command stating the Redex-Trail file to be searched:
`hat-observe hatfile[.hat]`
The tool is controlled by a number of textual commands:

- "[`observe`] *query*" initiates a new query. The syntax for *query* is given on page 114. The keyword "`observe`" is only necessary when the query starts with an identifier which is also a command in `hat-observe`.
- `<RETURN>` shows additional observed equations of the last query. By default only the first ten observed equations are shown.
- "`lines` n" to set the number of observed equations shown at a time to n.
- "`go` n" to show equations starting with observed equation number n.
- "`show`" shows the list of observable functions in the Redex-Trail file.
- "`debug` n" to start the Algorithmic Debugger on equation number n.
- "`trace` n" to start the Redex-Trail browser on equation number n (asks for LHS or RHS).
- "$+n$" or "$-n$" to increase/decrease the precision depth for pretty printing of expressions. Expressions beyond the adjusted depth are pruned.
- "`verbose`" to toggle verbose-mode (active: show unevaluated subexpressions).
- "`unique`" to toggle unique-mode (active: guarantee to show unique equations only (low response time), passive: only omit equations which are less general than previous observations (very fast response time))
- "`recursive`" to toggle recursive-filter-mode (active: omit recursive calls to the observed function).
- "`help`" to show a manual page.
- "`quit`" to leave the tool.

Query syntax in EBNF:

```
query      = (identifier | pattern) ["in" identifier]

pattern    = identifier expression* ["=" expression]

expression = identifier | constructor | numericvalue | "_" | "_|_"
             "(" expression expression* ")" |
             "[" expression ("," expression)* "]" | "[]" |
             "(" expression ("," expression)* ")" |
             "'" anychar "'" |
             """ anychar* """
```

`identifier`, `constructor`, `numericvalue`, `anychar` are not defined here. Patterns/expressions are not type-checked. Illegally typed patterns result in empty observations.

Using the Algorithmic Debugging tool hat-detect

The Algorithmic Debugging tool is invoked by the following command stating the Redex-Trail file to be debugged:
`hat-detect hatfile[.hat]`
The tool is controlled by answering questions about the correctness of presented equations. Possible answers and commands are:

- "`yes`" to state that the equation is correct. The equation's LHS is indented to reduce to its RHS.
- "`no`" to state that the equation is erroneous.
- "`?no`" to postpone the question and to continue as if answered `no` (*maybe no*).
- "`?yes`" to postpone the question and to continue as if answered `yes` (*maybe yes*).
- "`go` n" to go back to question number n.
- "`trust`" to manually trust all applications of the outermost function on the equation's LHS.
- "`untrust`" to untrust all manually trusted functions.
- "`observe`" to start the Observational Debugging tool and to observe all applications of the outermost function on the equation's LHS.
- "`trace`" to start the Redex-Trail browser on the current equation (asks for LHS or RHS).
- "$+n$" or "-n" to increase/decrease the precision depth for pretty printing of expressions. Expressions beyond the adjusted depth are pruned.
- "`verbose`" to toggle verbose-mode (active: show unevaluated subexpressions).
- "`memoize`" to toggle memoize-mode (default: active).
- "`help`" to show a manual page.
- "`quit`" to leave the tool.

Basic Interface to the Redex-Trail

```
typedef struct hiddenFilePointer* filepointer;  // abstract data type for a node
#define InvalidFilePointer (filepointer) 0      // THE node specifying invalidity

typedef struct hiddenHatFile* HatFile;          // abstract data type for a file

/******************************************************************************/
/* Part 1: Trace management: Operations for abstract data type "HatFile"      */
/******************************************************************************/
HatFile      hatOpenFile  (char* name);  // open file, return internal handle
void         hatCloseFile (HatFile h);   // close file in internal handle

filepointer  hatErrorRedex(HatFile h);   // last redex of an aborted evaluation
                                         // (otherwise InvalidFilePointer)
char*        hatErrorMessage(HatFile h); // error message, otherwise NULL
filepointer  hatMainCAF   (HatFile h);   // return the main CAF of the trace

filepointer  hatSeqFirst  (HatFile h);   // go to the first node in seq. order
filepointer  hatSeqNext   (HatFile h,    // jump to next node in seq. order
                           filepointer nodenumber);
BOOL         hatSeqEOF    (HatFile h,    // check for EOF
                           filepointer nodenumber);


/******************************************************************************/
/* Part 2: Node data access                                                   */
/******************************************************************************/
char         getNodeType  (HatFile h,               // get type of node and
                           filepointer nodenumber); // prepare node for access

 // for all Trace nodes:
filepointer  getParent      ();     // trace to node's parent
filepointer  getResult      ();     // result node of a redex
                                    // InvalidPointer returned otherwise
 // for application nodes only:
char         getAppArity    ();     // arity of an application node
filepointer  getAppArgument (int i);// argument "i" of an application node
filepointer  getAppFun      ();     // function trace of an application

 // for constructor/identifier nodes:
char         getInfixType   ();     // infix type of constructr/identifier
char         getInfixPrio   ();     // infix priority of constr./ident. node

 // for constructor/identifier/module nodes:
char*        getName        ();     // constructor, identifier or module name
```

```
 // for identifier nodes:
BOOL          getTopLevelFlag ();     // is identifier defined at toplevel?

 // for constant nodes:
filepointer   getAtom         ();     // Atom referenced by a constant node

 // for projection and SAT nodes:
filepointer   getProjValue    ();     // projected value of projection/SAT nodes

 // for constant/application/constructor/identifier nodes:
filepointer   getSrcRef       ();     // get source reference of a node

 // for Atom nodes of according type
char          getCharValue    ();     // value of a char node
char*         getStringValue  ();     // value of a CString node
int           getIntValue     ();     // value of an int node
int           getIntegerValue ();     // value of an integer node
char*         getRationalValue();     // value of a rational node
float         getFloatValue   ();     // value of a float node
double        getDoubleValue  ();     // value of a double node

 // for source reference nodes:
filepointer   getModInfo      ();     // returns a module node
int           getPosnColumn   ();     // column position in source code
int           getPosnRow      ();     // row position in source code
char*         getPosnStr      ();     // source position as a formatted string

 // for module nodes:
char*         getModuleSrcName();     // source module name of a module node
BOOL          getModuleTrusted();     // 0 for untrusted module, 1 for trusted
```

```
/*****************************************************************************/
/* Part 3: Common operations                                                 */
/*****************************************************************************/

/* testing                                                                   */
BOOL          isTopLevel     (HatFile handle,          // (redex of an) toplevel
                              filepointer nodenumber); // identifier?

BOOL          isTrusted      (HatFile handle,          // (redex of an) trusted
                              filepointer nodenumber); // function?

BOOL          isCAF          (HatFile handle,          // is a redex describing
                              filepointer nodenumber); // a CAF?

/* following trails                                                         */
filepointer   hatTopAncestor(HatFile handle,          // find top-most ancestor
                             filepointer fileoffset); // (is always a CAF)

filepointer   hatOutermostSymbol(HatFile handle,      // outermost symbol
                              filepointer nodenumber);// of an application

filepointer   hatOutermostName(HatFile handle,        // outermost name node
                    filepointer nodenumber);          // representing a symbol

filepointer   hatFollowHidden(HatFile handle,         // follow trail of
                              filepointer nodenumber);// hidden traces

filepointer   hatFollowSATCs (HatFile handle,         // follow trail of SATC
                              filepointer fileoffset);// nodes

filepointer   hatFollowTrace (HatFile handle,         // follow trail of SATC
                              filepointer nodenumber);// and projection nodes

/* Locations as strings                                                      */
char*         hatLocationStr(HatFile handle,          // show source location of
                             filepointer nodenumber); // application/constant
```

Interface to Observational View

```
typedef struct hiddenObserveQuery* ObserveQuery;
  /* abstract data type: ObserveQuery                                          */

ObserveQuery newObserveQuery(
             HatFile handle,
             filepointer identifierNode,
             filepointer topIdentifierNode,
             BOOL recursiveFilter,
             BOOL showProgress);
  /* newObserveQuery: creates a query, searching for all applications of
     identifierNode. Search scope may be restricted by topIdentifierNode.      */

ObserveQuery newObserveQueryIdent(
             HatFile handle,
             char* identifier,
             char* topIdentifier,
             BOOL recursiveFilter,
             BOOL showProgress);
  /* same as newObserveQuery but identifier and topIdentifier are passed
     as strings rather than as file nodes.                                     */

ObserveQuery newObservableQuery(HatFile handle,
                                BOOL showProgress);
  /* Query searching for all observable identifiers and modules.               */

void         freeObserveQuery(ObserveQuery query);
  /* end query, free memory                                                    */

filepointer  nextObserveQueryNode(ObserveQuery query);
  /* get next observation from query.
     InvalidFilePointer is returned when query has finished.                   */
```

Interface to EDT View

```
typedef struct hiddenEDTQuery* EDTQuery;
  /* abstract data type: EDTQuery                                                 */

filepointer  getEDTroot       (HatFile handle);
  /* get root node of EDT                                                         */

EDTQuery     newEDTQuery      (HatFile handle,
                               filepointer parent);
  /* create new query: get EDT children of parent                                 */

void         freeEDTQuery     (EDTQuery query);
  /* end query, free memory                                                       */

filepointer  nextEDTQueryNode (EDTQuery query);
  /* get next EDT child in query (InvalidFilePointer signals end of query)        */

BOOL         isEDTChild       (HatFile handle,
                               filepointer maybechild,
                               filepointer parent);
  /* determine whether maybechild is an EDT child of parent                       */
```

Haskell Interface

```
module HatTrace (...) -- all types/functions/classes below are exported
 where

type HatTrace        = ...      -- type, representing a Redex Trail file
data HatNode         = ...      -- type, representing a Redex Trail node

-- type: information about what a node within the Redex
data HatNodeType     = HatApplNode      | HatConstantNode  | HatSAT_ANode  |
                       HatSAT_BNode     | HatSAT_CNode     | HatHiddenNode |
                       HatDummyNode     | HatProjNode      | HatConstrNode |
                       HatIdentNode     | HatCaseNode      | HatLambdaNode |
                       HatIntNode       | HatCharNode      | HatIntegerNode|
                       HatRationalNode  | HatFloatNode     | HatDoubleNode |
                       HatCStringNode   | HatIfNode        | HatGuardNode  |
                       HatContainerNode | HatInvalidNode   | HatModuleNode
                       deriving (Show,Eq)

-- type: type of application node
data HatApplType     = HatApplied       | HatBlackholed    | HatComplete   |
                       HatValue deriving (Show,Eq)

-- type: infixtype of an application node.
data HatInfixType    = HatInfix Int     | HatInfixR Int    | HatInfixL Int |
                       HatNoInfix deriving (Show,Eq)

data ObserveResult   = IdentNotFound    | TopIdentNotFound | Found [HatNode]

-- type for values returned as source reference
data HatSourceRef    = HatSourceRef {row           :: Int,
                                     column        :: Int,
                                     moduleName    :: String,
                                     moduleFile    :: String,
                                     moduletrusted :: Bool} |
                       HatNoSourceRef

-- class for all structures representing Hat redexes
class Show a => HatRep a where
  toHatNode        :: a -> HatNode     -- convert to type HatNode
  toUndefined      :: a -> a           -- return undefined node of same instance
  hatNodeType      :: a -> HatNodeType -- return type of redex trail node
  hatParent        :: a -> a           -- return parent of Redex Trail node
  hatApplFun       :: a -> a           -- function of application/constant node
  hatApplArgs      :: a -> [a]         -- list of arguments of application node
```

```
  hatInfix         :: a -> HatInfixType  -- infix priority of
                                       -- constructor/identifier node
  hatResult        :: a -> a           -- result of application (if redex)
  hatName          :: a -> String      -- name of name/identifier/constructor
                                       -- node
  hatProjValue     :: a -> a           -- projected value projection/SATA/SATB/
                                       -- SATC node
  hatValueInt      :: a -> Int         -- Int value of an HatIntNode
  hatValueChar     :: a -> Char        -- Char value of an HatCharNode
  hatValueInteger  :: a -> Integer     -- Integer value of an HatIntegerNode
  hatValueRational :: a -> Rational    -- Rational value of an HatRationalNode
  hatValueFloat    :: a -> Float       -- Float value of an HatFloatNode
  hatValueDouble   :: a -> Double      -- Double value of an HatDoubleNode
  hatValueString   :: a -> String      -- String value of an HatCStringNode
  hatSourceRef     :: a -> HatSourceRef -- source position of application/
                                        -- identifier/constructor node

instance Eq     HatNode where ...
instance HatRep HatNode where ...

isValidNode      :: HatRep a => a -> Bool
isInvalidNode    :: HatRep a => a -> Bool

openTrace        :: String -> IO (Maybe HatTrace) -- open a trace file

hatTraceName     :: HatTrace -> String        -- name of a Redex Trail file

hatVersionNumber :: String                    -- current Hat version number

hatMain          :: HatTrace -> HatNode       -- node of "main" CAF

hatErrorRedex    :: HatTrace -> HatNode       -- last redex of aborted evaluation

hatErrorMessage  :: HatTrace -> Maybe String  -- message of aborted evaluation

hatTraceFirst    :: HatTrace -> HatNode       -- first node in Redex Trail file
hatTraceNext     :: HatRep a => a -> HatNode  -- sequentially following node

hatLeftmost      :: HatRep a => a -> HatNode  -- leftmost-outermost constructor/
                                              -- identifier

hatIsCAF         :: HatRep a => a -> Bool     -- does node represent a CAF?

hatIsTopLevel :: HatRep a => a -> Bool -- (outermost) symbol defined at toplevel?
```

```
hatIsTrusted  :: HatRep a => a -> Bool -- (outermost) symbol defined in Prelude?

-- EDT View
edtChildren        :: HatRep a => a -> [HatNode]
 -- returns list of all EDT children of the argument

-- Observational View
observe            :: HatTrace -> String -> String -> Bool -> ObserveResult
 -- observe identifier (2nd argument) in HatTrace, restricting the scope to
 -- the identifier in 3rd argument (empty string = no restriction)
 -- progress indicator is shown if 4th argument is True

-- predicates for result returned by observe
isIdentNotFound    :: ObserveResult -> Bool
isTopIdentNotFound :: ObserveResult -> Bool
isFound            :: ObserveResult -> Bool
fromFound          :: ObserveResult -> [HatNode]
```

Adjoxo - Example Program for Case Study

```
-- Adjoxo.hs: Adjudicator for noughts-and-crosses positions
-- Program by Prof. Colin Runciman.
-- Four bugs were deliberately introduced and marked in the
-- source-code.

data GameValue = Loss | Draw | Win deriving (Eq,Ord,Show)

bestOf :: GameValue -> GameValue -> GameValue
bestOf Win  v   = Win
bestOf Loss v   = v
bestOf Draw Win = Win
bestOf Draw v   = v  -- Bug 1, correct is:
                     -- besfOf Draw v = Draw (as v must be Loss or Draw)

inverse :: GameValue -> GameValue
inverse Loss = Win
inverse Draw = Draw
inverse Win  = Loss

-- Positions in the playing grid are numbered 1..9 with
-- top row 1,2,3; middle row 4,5,6; bottom row 7,8,9.
-- Ordered lists of such numbers represent O's or X's.

type Region = [Int]
type Player = Region

insert :: Int -> Region -> Region
insert x []                 = [x]
insert x xs@(y:ys) | x > y     = x : insert x ys -- Bug 2, correct is:
                                                  -- ... = y : insert x ys
                   | otherwise = x : xs

dif :: Region -> Region -> Region
dif [] ys = []
dif xs [] = xs
dif xs@(x:xs') ys@(y:ys') =
  case compare x y of
  LT ->     dif xs' ys    -- Bug 3, correct is:      LT -> x : dif xs' ys
  EQ ->     dif xs' ys'
  GT -> x : dif xs  ys'   -- also Bug 3, correct is: GT ->     dif xs  ys'

subset :: Region -> Region -> Bool
subset xs ys = null (dif xs ys)
```

```
hasLine :: Player -> Bool
hasLine p = subset [1,2,3] p || subset [4,5,6] p || subset [7,8,9] p ||
            subset [1,4,7] p || subset [2,5,8] p || subset [3,7,9] p || --Bug 4
              -- Bug 4: last vertical column is [3,6,9] and not [3,7,9]
            subset [1,5,9] p || subset [3,5,7] p

gridFull :: Player -> Player -> Bool
gridFull ap pp = length ap + length pp == 9

-- The 'ap' argument of 'analysis' is a position list for the active player
-- (to move next); and the 'pp' argument is the list for the passive player.
-- The result represents the outcome with best play on both sides.

analysis :: Player -> Player -> GameValue
analysis ap pp =
  if      hasLine pp then Loss
  else if gridFull ap pp then Draw
  else    foldr1 bestOf (map moveval (([1..9] `dif` ap) `dif` pp))
  where
  moveval m = inverse (analysis pp (insert m ap))

-- The argument to 'parsed' is an input string, and its result
-- is the corresponding pair of position-lists for O's and X's.
-- Example:  O | O |
--          ---+---+---
--             |   |      ==> ([1,2],[7,8])
--          ---+---+---
--           X | X |

parsed input = (region 'O' input, region 'X' input)
  where
  region = region' 1
  region' _ _ [] = []
  region' n p (c:cs) =
    if c=='|' then region' (n+1) p cs
    else if c=='-' then region' (n+1) p (dropWhile (`elem` "-+") cs)
    else if c==p then n : region' n p cs
    else region' n p cs
```

```haskell
-- The argument to 'adjudicate' represents positions of O's and X's.
-- Its result is a short adjudicator's report.

adjudicate :: (Player, Player) -> String
adjudicate (os,xs) =
  case compare (length os) (length xs) of
  GT -> report (analysis xs os) 'X'
  EQ -> if hasLine xs then report Win 'X'
        else if hasLine os then report Win 'O'
        else "if X to play: " ++ report (analysis xs os) 'X' ++
             "if O to play: " ++ report (analysis os xs) 'O'
  LT -> report (analysis os xs) 'O'

report Loss p = report Win (opp p)
report Win  p = "WIN for " ++ [p] ++ "\n"
report Draw _ = "DRAW\n"

opp 'O' = 'X'
opp 'X' = 'O'

-- Test input for program.
-- (If 'X' is to play then 'X' can achieve a draw only.
--  If 'O' is to play then 'O' can force a win.)
input =
  "  O | X |   \n\
  \ ---+---+---\n\
  \    |   |   \n\
  \ ---+---+---\n\
  \    |   |   \n\
  \"

main = putStr (adjudicate (parsed input))
```