Multiple-View Tracing for Haskell: a New Hat

Malcolm Wallace, Olaf Chitil, Thorsten Brehm and Colin Runciman

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

Abstract

Different tracing systems for Haskell give different views of a program at work. In practice, several views are complementary and can productively be used together. Until now each system has generated its own trace, containing only the information needed for its particular view. Here we present the design of a trace that can serve several views. The trace is generated and written to file as the computation proceeds. We have implemented both the generation of the trace and several different viewers.

1 Introduction

Usually, a computation is treated as a black box that performs input and output actions, but whose internal workings are invisible. As programmers, however, we may want to look into the black box to understand how the different parts of the program cause the computation to perform the observed actions. Often the computation does not perform the intended actions and we have to determine which parts of the program cause the erroneous behaviour. Even if a program is correct, we may desire to understand its parts better by seeing "how it works"; especially when we have to modify a program that we did not write ourselves. Also for teaching it is sometimes useful to "see" a computation.

In [2] we compared the Haskell tracing systems Freja[1] [5,6] and HOOD[2] [3] with our Haskell tracer Hat[3] [9,10,11]. The main conclusion of our comparison was that each system gives a unique view of a computation and these views are usefully complementary. In experiments, we discovered that after using one system to help track a bug to a certain point, users often wanted to change to another system to continue the search, or to confirm their suspicions.

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

All three tracing systems take a two-phase approach to tracing: during the computation information describing the computation is collected in a data structure, the trace. After termination of the computation the trace is viewed. An advantage of a trace as a concrete data structure is that it liberates the viewer from the time arrow of the computation. However, each system creates its own trace, containing only the information required for its particular view. We noted in [2] that Hat's trace, called a Redex Trail, contains nearly all the information contained in Freja's trace. Hence we decided to extend the Redex Trail structure to the Augmented Redex Trail structure (ART). With separate tools we can view an ART trace in at least three different ways: à la Freja, à la Hat and à la HOOD. Whereas Freja, HOOD and the old Hat system generated their traces in main memory, the new Hat writes the ART trace to file as computation proceeds. Hat's new architecture has the following advantages:

- As a stand-alone description of a computation, the ART trace serves as interface between trace generation and trace viewing. The two phases become completely separate.
- A trace in file supports sequential access and forms of indexed search that were not feasible for heap-based traces.
- The ART trace clarifies the relationships between the different views of a computation. It suggests ways for integrating different views and creating new views.
- The size of the trace is no longer bound by the size of the main memory but only by the far larger size of the file store.
- The trace is no longer transient but can be archived for later viewing.
- Trace system developers only need to implement trace generation once for several views.
- The user only pays the cost of generating a trace once for several views.

In Section 2 we review the Redex Trail of the old Hat system, while Section 3 briefly illustrates some alternative tracing views. In Section 4 we develop in several steps the new Augmented Redex Trail structure. In Section 5 we describe how several tools for different views obtain their information from the ART trace. In Section 6 we outline the generation of the ART trace. In Section 7 we discuss ideas for future work. Section 8 concludes.

We have modified Hat to produce the ART trace and have implemented new tools for viewing the trace in the style of Freja and HOOD. The system has been publicly released as Hat 1.04.

2 The Redex Trail Model

Let us view a computation abstractly as a series of rewrite steps. Starting from a single expression (main), at each step a reducible expression (redex) is replaced by another expression (its reduct), by instantiating the LHS of an equation, and replacing it with the corresponding instance of the RHS. Eventually, only irreducible expressions (values) remain.

The original Redex Trail structure is a directed graph, recording copies of all values and redexes, with a backward link from each reduct (and each proper subexpression contained within it) to the parent redex that created it.

2.1 Example

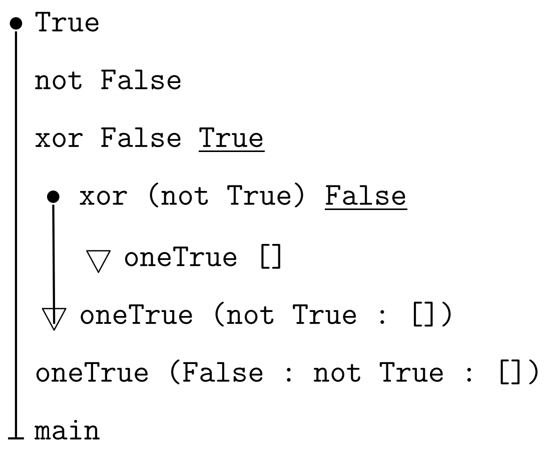
```
oneTrue            :: [Bool] → Bool
oneTrue []         = False
oneTrue (x : xs)   = xor x (oneTrue xs)

xor                :: Bool → Bool → Bool
xor x True         = not x
xor _ False        = True

main               = print (oneTrue [False, not True])
```

Fig. 1. Example program

The small example program shown in Figure 1 produces the Redex Trail illustrated in Figure 2. A subexpression with a different parent is represented as a box within a box. A solid arrow denotes the parent relationship. Of course, the user is not expected to see and understand a complete graph of this nature. A tool called the Redex Trail viewer permits the whole graph to be explored interactively one expression at a time, as illustrated in this snapshot:



```
● True
  not False
  xor False True
   ● xor (not True) False
      ▽ oneTrue []
   ▽ oneTrue (not True : [])
  oneTrue (False : not True : [])
  main
```

Each redex is shown on a separate line. The parent of an expression is shown below it. (Because parents are shown below their children in the viewer, we have drawn the full graph in Figure 2 similarly.) The parent of a whole

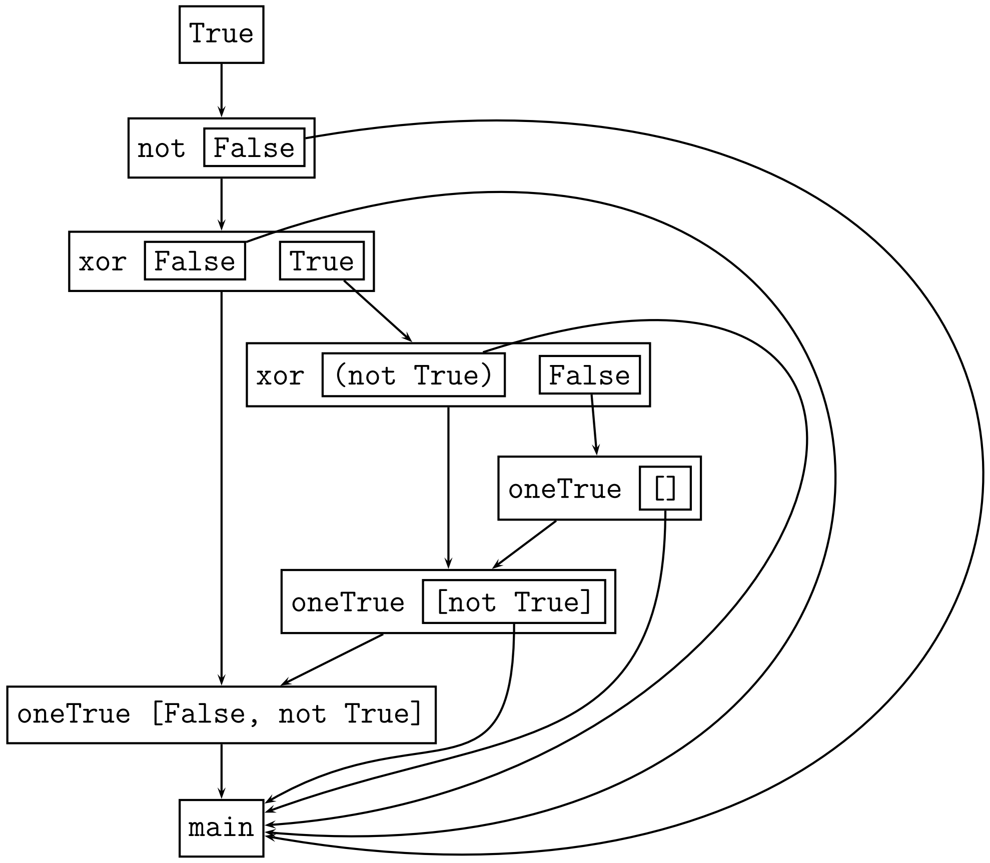


Fig. 2. An example of a Redex Trail

redex starts in the same column, whereas the parent of a proper subexpression is further indented. Underlining, and colouring if available, is used to show to which subexpression a parent belongs.

2.2 Structure

Figure 3 formalises the Redex Trail structure as a set of concrete Haskell types. An *App* node represents a redex in the obvious way, with a trace for the function and separate traces for each argument. It also contains the *parent* trace describing the redex that created it. Each function and argument is likewise of *Trace* type and therefore has its own, possibly different, parent. The *SrcPos* type records the location in the source code of the relevant application site on the RHS of a definition.

A *Const* node represents an irreducible value (an *Atom*), such as an integer, character, or a constructor or named function from the program, represented simply as a string identifier. (In the latter case, a *SrcPos* is associated with the identifier to record its static definition site.) A *Const* node also has a

```
data Trace  = App     { fun    :: Trace, args :: [Trace]
                      , parent :: Trace, src  :: SrcPos }
            | Const   { value  :: Atom
                      , parent :: Trace, src :: SrcPos }
            | Root
data SrcPos = SrcPos  { file :: FilePath
                      , line :: Int       , column :: Int }
            | NoPos
data Atom   = Id String SrcPos | IntVal Int | CharVal Char | ...
```

Fig. 3. The original Redex Trail structure.

parent redex and a *SrcPos* to indicate its dynamic origin. For instance, two uses of the function *f* in a computation may have different positions in the source code, because they are introduced to the trace by rewriting different redexes.

Finally, it is possible for a redex to have no parent, represented simply as *Root*. This clearly occurs at the very start of the computation, namely for the *main* function. It also applies in the case of other top-level pattern-bindings (CAFs).

3 Alternative Views

We would like to adapt the original Redex Trail structure to support additional styles of viewing, such as Algorithmic Debugging and Observations. So what do these views look like? And what information do they require from the trace?

3.1 Algorithmic Debugging

Algorithmic Debugging is a well-known technique in declarative languages [8], implemented for a subset of Haskell by the Freja system [5]. The algorithm locates an error in a program, given a user who can provide answers to a sequence of questions. Each question concerns a reduction of a redex to a result, presented as an equation. The user should answer *yes* if the equation is correct with respect to his intentions, and *no* otherwise. After some number of questions, the system identifies an incorrect function definition.

Here is such a session for the example program of Figure 1. The symbol '_' represents an expression that has never been evaluated and whose value hence cannot have influenced the computation.

```
1> oneTrue (False:_:[]) = True   (Y/?/N): n
2> oneTrue (_:[]) = True   (Y/?/N): n
3> oneTrue [] = False   (Y/?/N): y
4> xor _ False = True   (Y/?/N): n
```

```
Error located!
Bug found in: xor _ False = True
```

Freja creates an Evaluation Dependency Tree (EDT) as its trace structure. Figure 4 shows the EDT for this example. Each node of the tree is a reduction. The tree is basically the derivation/proof tree for a call-by-value reduction with miraculous stops where expressions are not needed for the result. The call-by-value structure ensures that the tree structure reflects the program structure and that arguments are maximally evaluated.

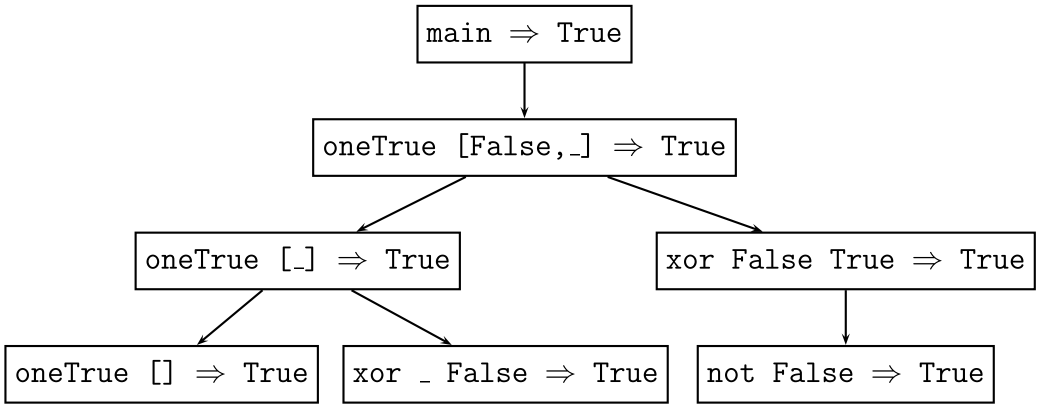


Fig. 4. An Evaluation Dependency Tree

The viewer dialogue walks the tree, presenting each node as a question – some answers permit some branches of the tree to be ignored.

To allow algorithmic debugging starting from a Redex Trail rather than an EDT, we need to add the ability to extract redexes from deep within the Trail – for instance, the very first question is about the reduction of *main*, which lies at the farthest tip of the Redex Trail graph. Furthermore, we need to record dependency information in the opposite direction – the reduction of an expression depended on what sub-reductions?

3.2 Observations

HOOD [3] allows observation of individual values within computations. The programmer annotates the expression(s) of interest in the program source with the combinator `observe`. For each annotation, HOOD records the value in all of its intermediate stages of evaluation, so that after termination of the computation the observed expression can be shown to exactly the degree to which it was demanded.

By a few clever tricks, HOOD can record not only data values, but also functional values, again only to the degree they are really used in the computation. Thus a functional value is recorded as a bag of actual argument/result

```
data StoredEvent    = Value  { value :: String
                             , inside :: Ref    , position :: Int }
                    | Constr { name :: String, arity    :: Int
                             , inside :: Ref    , position :: Int }
type Observation    = Ref  → StoredEvent
data ObservedValue  = Value  { value :: String }
                    | Constr { name      :: String
                             , arguments :: [ObservedValue] }
```

Fig. 5. The two HOOD observation structures. StoredEvents are recorded as a simple sequence during the computation, but the viewer must later traverse the StoredEvents to construct an ObservedValue that can be displayed.

mappings [4].

Here we illustrate the output when observing the functions `oneTrue` and `xor` in our example program. The symbol '_' again represents an unevaluated expression.

```
oneTrue (False:_:[]) = True
oneTrue (_:[]) = True
oneTrue [] = False

xor False True = True
xor _ False = True
```

The HOOD trace structure (sketched in Figure 5) is a sequence of individual 'events'. Every event represents the creation of a data value or constructor (in WHNF) during the computation. It has a backwards link to identify the enclosing data structure of which it is a component, and the particular argument position it occupies. Each constructor also has a note of how many argument 'holes' it can accept. A functional value is treated like a data constructor with two components, an argument and a result.

At viewing time, the HOOD viewer transforms the stored structure internally by constructing forward links from each constructor to the final value of each of its components. This can be expensive for a large structure.

The most notable difference between these structures and the Redex Trail is that HOOD stores only irreducible values, not reducible expressions. Another major difference is that HOOD records only individual annotated values, not a full trace of the whole computation.

In another paper at this workshop Reinke describes a graphical viewer for HOOD observations [7].

4 The Augmented Redex Trail Structure

The EDT structure used in Freja's algorithmic debugging is very similar to a substructure of the Redex Trail, but with pointers reversed. Most of the information in HOOD's Observations can be derived from either an EDT or a Redex Trail by searching through those structures for named values and source positions. The Redex Trail lacks some small pieces of information that would permit the reconstruction of an EDT by the reversal of pointers. This same lack also prevents the reconstruction of Observations.

In this section we extend the original structure in stages to become the Augmented Redex Trail, or ART structure. [4]

4.1 Linearisation and Explicit Sharing

The original Redex Trail is an ephemeral heap-based structure, but we want to store the trace in a file so it is persistent, does not depend on connecting a viewer to a live program, and can be accessed many times by different tools.

The original Redex Trail type *Trace* (in Figure 3) is self-recursive, so to place it in file requires the structure to be linearised. Linearisation gives two benefits: we can write the trace into the file one node at a time; and we can access a part of the trace piecemeal without necessarily following all possible paths. In both cases, efficiency is important: when generating, sequential writing is best; when viewing, the viewer tool should need to read only a small fragment of the whole structure.

Another benefit of linearising the structure is that it makes sharing of nodes explicit. In the original graph model, sharing and cycles are implicit via the self-recursive type, but in the new model this information is revealed directly to the viewing tool. There is a great deal of sharing in a trace of a typical program.

```
data Expr  = App    { fun    :: Ref, args :: [Ref]
                    , parent :: Ref, src  :: SrcPos }
           | Const  { value  :: Atom
                    , parent :: Ref   , src :: SrcPos }
           | Root
type Trace = Ref  → Expr
data Ref   = NoRef  | Ref FilePos deriving Eq
```

Fig. 6. The linearised Redex Trail structure.

Figure 6 shows how the trace structure changes to accommodate linearisation. We rename the original *Trace* datatype to *Expr*, and all self-recursive

[4] The concrete types presented in this section are a slightly abstracted view of the real trace structures in our current implementation.

uses of the type become explicit pointers *Ref*. The new *Trace* type defines a unique mapping from *Ref*s to *Expr*s. The *Root* constructor is replaced by the null reference *NoRef*. Every *Expr* node of the graph can be written to or read from a file individually. A *Ref* can also be updated in-place (see Section 4.2).

The trace structure as a whole is a sequence of *Expr* nodes, and the evaluation order of the program is apparent from the implicit ordering of *Expr* nodes in the file.

4.2 Redexes with Results

The original Redex Trail structure records all of the intermediate steps in a reduction sequence, but the links are made only in one direction – backwards – allowing exploration only from 'effects' to 'causes' in a viewer. However, both Algorithmic Debugging and Observations present equations in their viewers, and hence require a 'forward' link from every redex (LHS) to its reduct (RHS).

In Figure 7 we once again modify the concrete Haskell representation of the trace structure, this time to incorporate the 'forward' or *result* links.

```
data Expr = App     { fun    :: Ref , args   :: [Ref]
                    , parent :: Ref , src    :: SrcPos
                    , status :: Eval, result :: Ref }
          | Const   { value  :: Atom
                    , parent :: Ref  , src    :: SrcPos
                    , status :: Eval , result :: Ref }
data Eval  = Applied | Blackholed | Completed | Value
```

Fig. 7. The core of the Augmented Redex Trail structure.

Although it may appear that the *parent* and *result* pointers simply represent the same relation with directions reversed, this is not so. A redex has at most one outgoing result arc – to the single expression it is rewritten to – but it can have many incoming parent arcs, because it is the creator of all subexpressions within the reduct. Hence, a result arc represents equality whereas a parent arc represents only inclusion.

A *Const* representing a basic value (e.g. integer/character) has a *parent* but no *result*; a *Const* which is an identifier (e.g. a top-level pattern-binding) could have a *result*, but no *parent*; often a *Const* identifier (e.g. local pattern-bindings) has both a *parent* and a *result*.

4.3 Unevaluated Expressions

Together with the *result* pointer we introduced a *status* :: *Eval* field. Even though an *Expr* is created, the expression it represents may never be evaluated.

This has consequences for how a viewer should interpret the *result* pointer. [5]

Every *Expr* can potentially go through three possible states. Initially, it is created by rewriting another expression according to some equation (its *status* is *Applied* and its *result* pointer is undefined). At some later point in the computation, the value of the expression may be demanded, or 'entered' (*status* now becomes *Blackholed*). At that point the *result* pointer is set to the newly written *Expr* that represents the reduct. Later again, the expression may become evaluated to a result expression (although this of course may still contain unevaluated subexpressions). At this point, the lazy evaluation mechanism does an 'update', overwriting the original expression with its result. In the trace, however, we do not overwrite the *Expr*, but update only the *status* to *Completed* (see §6.2). The final possible *Eval* status of *Value* is for irreducible expressions: an *Atom*, or an *App* with a data constructor in the function position. The *result* pointer for a *Value* is undefined.

4.4 Example

The augmented version of the Redex Trail graph from Figure 2 is shown in Figure 8. The subexpression relationship is now shown as a pointer (solid line). Parents are shown as dotted lines, and results as dashed lines. Expressions are annotated with their *status*.

4.5 Entry Points to the Trace

Every ART viewing tool needs an entry point at which to begin its presentation to the user. These entry points can be different for different tools.

The entry point for algorithmic debugging is the 'beginning' of the computation, the evaluation of the *main* function. In every ART trace, the *Expr* for *main* is the first *Expr* in the generated sequence.

The entry point for some other viewers is at the 'end' of computation, for instance when reconstructing a virtual stack-trace from an error message, or when exploring a Redex Trail backwards from the program output. This suggests that both the program output and any error messages must be recorded in the trace, since they are the 'end-points' of the program. Output and errors are easily added to the ART structure as strings with parent pointers. The output need not be monolithic; it can be spread across many strings; however we do not discuss here the various possible ways to split the output, nor how to store it in the file in a manner that permits quick access.

Other viewers may have variable entry points. For instance, a HOOD-style observation may need a named function or source position, and retrieve the relevant information by linear search through the trace.

[5] The original Redex Trail structure had a kind of node (*Sat*) which incorporated aspects of both the *result* pointer and the *status* marker, but these nodes were transient, removed from the graph once an expression was evaluated. The *Sat* node did not permanently record the full information required to forward-link every redex to its reduct.

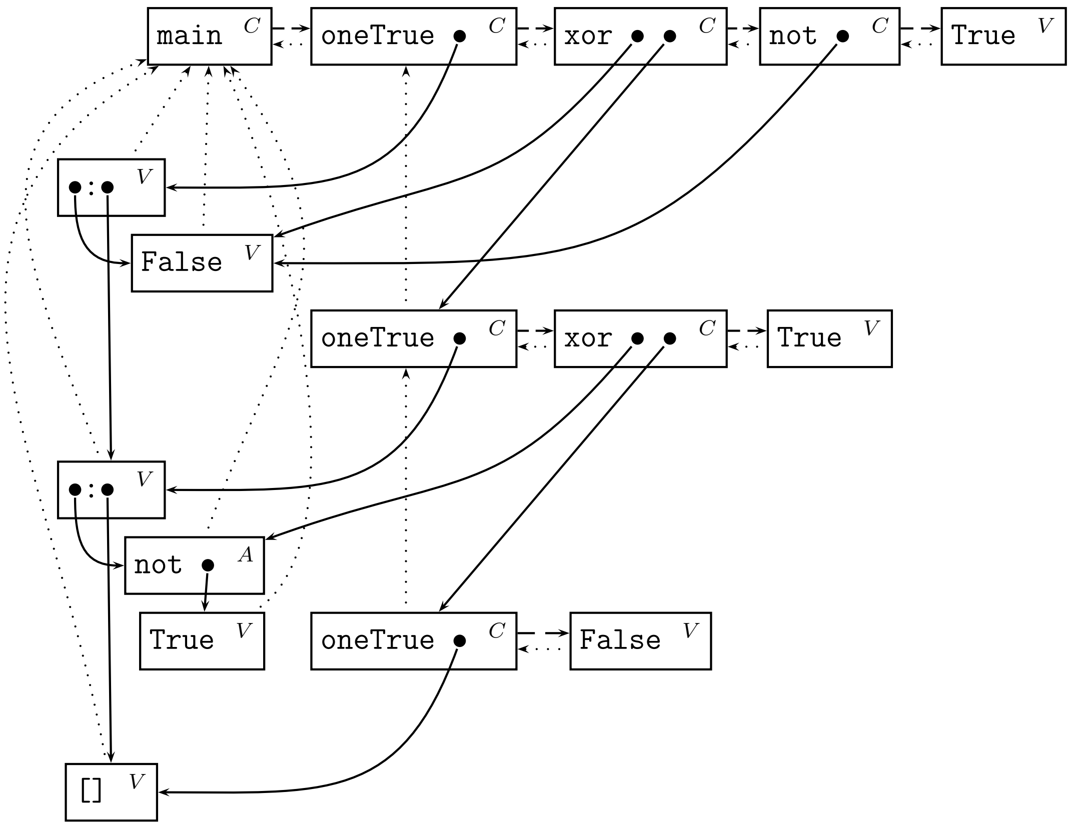


Fig. 8. A Trace

4.6 Other Refinements – Lambda, Do, Case, If, Guard

Not all expressions in Haskell are either a simple value, an application of a constructor to arguments, or an application of a named function to arguments. We also have anonymous functions (lambda expressions), monadic bindings (do statements), and various forms of conditional operations (ifs, cases, and guards). For conditionals, it is important to record in the trace not merely the final result, but how the decision was reached to take a particular branch. We follow much the same treatment for these extra constructs as in the original Redex Trail structure. For lambda expressions, we introduce a new kind of *Atom* to represent any function without a name, and the various kinds of conditional are handled by introducing new kinds of *Expr*s, each of which differs only slightly from the standard *App* kind.

4.7 Projections

There are situations that are slightly tricky to record, due to interesting consequences of the lazy evaluation model. One such is projections. In a definition like *id* $x = x$, the question is, who is the parent of x? In one sense, it is *id*, yet in another sense, *id* is merely passing on the value without touching it, so x's parent is really whatever expression created it, not *id*. Once again, we follow the original Redex Trail structure by introducing another special kind of *Expr*, the *Proj* node, which can be thought of as attaching an additional projective parent to the referenced redex.

4.8 Trusting

Finally, it is sometimes desirable *not* to record all reductions in the trace structure – we *trust* some function definitions, such as those in the Standard Prelude [10]. There are two main reasons for trusting. The first reason is to improve performance. Trace files are very large and quite slow to write. If we know that certain parts of the trace are not of interest, it makes sense to omit them. The second reason is to reduce the amount of information presented to the user of a trace-viewing tool.[6] Traces contain a huge amount of data, so a trace that appears too complex can actually hide the information the user wants. We do not elaborate the details of the trusting mechanism here.

5 Multiple Views from a Single Trace

Having outlined a unifying trace structure, we must now demonstrate that it can satisfy the needs of the Redex Trail, EDT, and Observation views. We have built three separate viewers which mimic the user interface behaviour of the three previous systems (Freja, HOOD, and the old Hat). In this section, we describe how the required information is reconstructed from the new ART trace.

5.1 A Redex Trail View: hat-trail

The original Redex Trail structure can be recovered by following mainly parent pointers. The result pointer chain is used to show a subexpression in its most evaluated form. The original Hat browser has been adapted to use the new ART trace, and is now called `hat-trail`. The viewer starts with program output or an error message, and enables the user to interactively explore a computation backwards from effect to cause by revealing the parent (origin) of any selected subexpression.

5.2 A Static Call Stack: hat-stack

One special-case use of the parent pointers is to show a static call-stack back-trace from any error message. This does not represent the real lazy evaluation stack — often sadly incomprehensible. Instead the backtrace gives the virtual

[6] It would of course be possible to implement a trusting mechanism in the viewing tool itself, rather than omitting the data from the trace altogether.

stack showing how an eager evaluation model would have arrived at the same result. In our system, this tool is `hat-stack`.

5.3 Algorithmic Debugging: hat-detect

The tool `hat-detect` provides Algorithmic Debugging, by extracting a virtual EDT 'by need' from the ART trace structure.[7] It can be seen from Figure 4 that we need three kinds of information from the trace: first, the EDT's root node; second, an EDT node's label, where a label is an equation containing an application and its result; and third, the children of an EDT node.

The root node of the EDT is always the *main* CAF, found at the beginning of the trace file.

Each EDT label is an equation: the LHS is an application or CAF itself, and the RHS is its result in its most fully evaluated form. When an *App* or *Const* has a *status* of *Completed*, we can follow the *result* pointer to determine the eventual value. The immediately referenced node might in turn be *Completed*, so the *result* chain must be followed iteratively until we find a node with an *Applied*, *Blackholed* or *Value* status. An *Applied* node was unevaluated, therefore it cannot have influenced the execution. It is presented to the user as a '_' symbol. A status of *Blackholed* is similarly displayed as $\perp$. Only a *Value* status represents a genuine result, either a simple value or a complex structure, and can be printed as a normal expression.

To determine the children of an EDT node, we must find all fully-evaluated applications on which the evaluation of the current node depended. The first child of a EDT node p, may be found by following p's ART result pointer, but the referenced node q is a child only if its *status* is *Completed* or *Blackholed*. (Only with one of these status annotations does the node describe an application or CAF whose result was actually demanded.) Further children can be found if q is *Completed*, *Blackholed* or a *Value*. In these cases the argument pointers of q are considered. If an argument's ART parent is also p, provided the argument itself is *Completed* or *Blackholed*, it is also a child of p. More children can be found recursively by the same method.

In this way, all the information necessary to define a computation's EDT can be retrieved from an ART trace file.

Only applications of top-level identifiers are considered by `hat-detect`. A locally defined function may depend on the values of free variables bound in an enclosing scope. To decide whether an application appears to be correct, the programmer needs to know the values of the free variables, yet the ART trace does not record any direct link to these variables. Program errors found by our tool therefore always refer to the top-level function; computation within local definitions is attributed upwards to its enclosing top-level definition.

[7] Although we describe the reconstruction of an EDT as if performed in one pass, the implementation need never build the whole structure - it can be constructed and traversed piecemeal.

The dialogue presented by `hat-detect` is straightforward to arrange, following the standard debugging algorithm. It starts at the root of the EDT, the *main* CAF. If the programmer answers that a node label is erroneous, he is asked about the correctness of its children, but children of nodes identified as correct need not be considered. An erroneous node with only correct children, or no children at all, is the location of the bug.

5.4 Observation of Functions: hat-observe

Our tool `hat-observe` displays all function applications of a given identifier within a computation. Unlike HOOD, no annotations are needed in the program's source code. As the observed identifier is chosen independently of the program run, it is easy to make a number of successive observations without modifying or rerunning the computation.

The tool observes a function by searching sequentially through the trace file. First, the identifier itself is found as a *Const* node containing an *Id* atom with the name. (See structures in Figures 7 and 3). Then every application node is checked for a reference to the given *Const* in the function position.

To deal with partial applications we must search the ART trace not only for references to the original *Const* node, but for references to any application which in turn references the *Const*, and so on recursively. If the function involved in an application is a reducible expression (with a function as result) we must follow this expression's forward *result* link, to see whether it is the desired function, or a partial application of it. The cost of such searching from application nodes to determine the associated function turns out to be low, as the relevant expressions are usually found close to the original application node. In particular, an additional file access is very rarely needed, as these expressions are usually within the file's buffer. Linearisation ensures that the function reference in an application node can only refer to an earlier node in the trace[8], so a single linear scan through the trace is sufficient to collect all applications of a specified function.

Not all applications or CAFs have results – they may be unevaluated, or an application may be partial – but where a result is available, the RHS of the equation can be determined as described in Section 5.3. All applications or CAFs with results are displayed as a list of equations.

To avoid redundant output, equivalent or less general applications of the identifier can be omitted in the display. One application of an identifier is considered more general than another if all its arguments are less defined (due to lazy evaluation). To avoid problems with local functions capturing free variables, as described in Section 5.3, we again only permit observations of top-level functions.

Our tool shows all applications of the function throughout the program, whereas HOOD observes a specific function application at one point in the

[8] All *Ref*s in the ART structure, apart from the *result*, refer to earlier nodes.

source code. However, because the source code position is recorded in the ART trace, an equivalent feature could be achieved by a different interface, perhaps a source code browser allowing the user to select expressions to be observed.

6 Trace Generation

The developers of Freja, Hat and HOOD made different choices about the architectural level at which they implemented the creation of the trace. For instance, in HOOD the trace is created by the combinator *observe* defined in a high-level Haskell library, which uses side-effecting I/O to record the information. In Freja the trace is created in the heap by low-level variants of the graph reduction machine instructions [5].

To generate the new ART trace [11] we took the old Redex Trail approach, but adapted to write traces to file instead of constructing them in heap memory. First, the original program is transformed into a new program that computes its trace in addition to its normal result. Second, the transformed program is compiled. Third, the compiled program is run. The computation writes a trace to file in addition to any normal I/O of the original program. Fourth, the trace is viewed.

Currently the program transformation is performed by an early phase of the Haskell compiler nhc98. However, we intend to separate the transformation from the compiler, so that the transformed program can be compiled with all Haskell compilers. The Augmented Redex Trail approach is then potentially as portable as the HOOD implementation, in contrast to Freja. The principle of using an automatic source-to-source transformation, coupled with a library of combinators written in standard Haskell, permits the possibility of using any Haskell compiler system to generate an ART trace.

6.1 The Program Transformation

The transformation wraps every expression of the original program into the R data type, which is defined as follows:

$$\textbf{data}\, R\,\alpha \;=\; R\,\alpha\,\mathit{Ref}$$

The *Ref* is a reference to an *Expr* node of the trace in file. The pairing assures that an expression and its description “travel together” throughout the computation, so that when expressions are plumbed together by application, the corresponding descriptions in the trace can be plumbed together by creating an *App* node at the same time. Trace nodes are written to file by side-effects which are triggered when certain expressions are evaluated. All the plumbing and writing of trace nodes is performed by combinators which are defined in a library.

The program transformation introduces numerous calls of the combinators into the program. For example, here is the original *oneTrue* definition, together with its transformed version.

```
oneTrue          :: [Bool] → Bool
oneTrue []       = False
oneTrue (x : xs) = xor x (oneTrue xs)
```

```
oneTrue :: SrcPos → Ref → R (Ref → (R [Bool]) → R Bool)
oneTrue sr p = fun1 (mkAtomId "oneTrue" 7) oneTrueW sr p
  where
    oneTrueW :: Ref → R [Bool] → R Bool
    oneTrueW p′ (R [] _) =
        con0 (mkSrcPos 2) p′ False (mkAtomId "False" 6)
    oneTrueW p′ (R (x : xs) _) =
        rap2 (mkSrcPos 3) p′ (xor (mkSrcPos 3) p′) x
            (ap1 (mkSrcPos 4) p′ (oneTrue (mkSrcPos 4) p′) xs)
```

In this example the combinators *fun*1, *con*0, *ap*1, *rap*2, *mkAtomId*, and *mkSrcPos* are used. The combinator *fun*1 wraps the function *oneTrueW*, which does the actual work, with *R* constructors. The combinator *con*0 wraps the constructor *False*. The combinators *ap*1 and *rap*2 assure the correct plumbing of applications. The combinators *mkAtomId* and *mkSrcPos* build references to detailed information about the identifier *oneTrue*, the constructor *False* and various source references. Numeric arguments are indexes to tables that contain the detailed information.

Very similar combinators were used in the old Hat system. The most important difference is that the new Hat combinators now record the trace nodes directly to file.

6.2 Writing with Updating

The main technical obstacle is that the trace is a (usually cyclic) graph which is continuously modified during generation. These modifications were no problem in main memory but for efficient writing to file updates have to be minimised.

We assume that writing nodes to file has much better performance if it can be achieved sequentially. However, even a cursory examination of the ART structure tells us that after writing an *Expr* node to file, it is highly likely that we will need to return to it to update the result pointer. Although some expressions remain completely unevaluated throughout the computation, the vast majority of intermediate expressions are indeed entered and evaluated to their reduct.

What is more, in our scheme there are *two* possible updates for each *Expr*, one on entering the expression (*Applied* → *Blackholed*), and another on its completion (*Blackholed* → *Completed*).

However, we observe that a *Blackholed* expression is almost always transient. The only situation in which such an annotation can remain in the final trace is when the program's overall result is undefined, such as an error or interruption. We also observe that the order in which trace expressions are entered and then completed follows a strict stack discipline, mirroring the evaluation stack of the underlying abstract machine. Hence, we do not update *Applied* to *Blackholed* on entry, but only write the remaining stack of 'blackholes' at the end of the computation should it fail.

We also try to avoid interspersing the final update of each *Expr* with the sequential generation of nodes. This is easily achieved by storing a large queue of updates that are performed all at once.

7 Future Work

The practical questions that interest most people are about time and space. How large is the trace? And how long does it take to produce, relative to the original computation?

An ART trace is undoubtedly big, to be measured in megabytes for a computation of any significant size. We estimate about 40–50 bytes are required per reduction. The largest trace we have yet generated is 240Mb in size, for a computation of around 6 million reductions (a chess end-game solver). Traced computations also take about 50 times longer than normal computations.

If Hat is to be used for substantial computations, we have to reduce the slow-down factor for traced computations. The fact that only 10% of traced computation time is spent on actually writing to file demonstrates that the implementation of trace generation can be improved. Since the computation of a transformed program spends most of its time evaluating the combinators, efficient definitions of the combinators are vital. We will also separate the program transformation from nhc98, so that a transformed program can be compiled by an optimising Haskell compiler such as ghc. Not only would this improve absolute runtimes, but aggressive optimisation may also reduce the relative slow-down. Furthermore, the computation of trusted function definitions is not yet much faster than that of untrusted definitions. We intend to investigate how transformed modules can be combined with trusted *untransformed* modules. Such a scheme, not requiring access to the sources of trusted modules, would also aid portability.

Other issues we want to address include:

- Currently I/O actions are traced in a rather ad hoc way that works well only with some views for simple I/O only. We aim to develop a general method for tracing I/O actions.

- We want to solve the problem that none of the views copes well with programs that make substantial use of higher-order combinators, for example in monadic or continuation-passing style.
- We plan to extend `hat-observe` to observe values at any program point. We could also add information about free variables to expressions in the trace, so that `hat-detect` and `hat-observe` can show a fuller trace of local computation. It may even be desirable to switch levels of detail within a view.
- There is scope for new viewing tools. For instance, the evaluation order of the computation is stored implicitly in the sequence in which *Expr* nodes are written to file. Hence, the computation, or selected parts of it, could be shown as an "animation", perhaps in the style of GHood[9]. We could also offer a "stories" view in the style outlined in [1]. A more specialised tool could isolate the circular self-dependency that evokes a "blackhole" error.
- We have begun to integrate `hat-detect` and `hat-observe` into a single tool. Eventually we hope for a full integration allowing the programmer to switch between views at any point during the exploration of a computation.
- How can we evaluate the useability of Hat in practice and gain information to improve it?

More generally, we intend to study the properties of the ART trace further. Is the trace complete with respect to information, such as recorded reductions, intermediate unevaluated expressions and values, and with respect to distinctions and relationships, such as sharing and evaluation order? How conveniently and efficiently can one access the trace to obtain a specific snippet of information? Can we claim some sort of "universality" for the trace structure, in terms of the range of queries it can support? How are all these properties affected by trusting? Does the exclusion of trusted redexes from the trace compromise the reachability of individual trace nodes from designated entry points?

There should be a close relationship between tracing and operational semantics, both of which aim to describe the relationship between a program and the observed actions of a computation of the program. We have begun work to define the ART trace and specific views through conservative extensions of an operational semantics of a program. Different kinds of formal semantics may suggest new views for tracing. For instance, the evaluation dependency tree of algorithmic debugging is closely related to a big-step structured operational semantics; the Redex Trail view was inspired by graph-rewriting machines; the observation of values recalls denotational semantics, especially the view of functional values as (finite) mappings ('minimal function graphs' [4]).

In principle, a semantics defines all the answers a tracer could give for the computation of a particular program with particular input. A tracer

[9] `http://www.cs.ukc.ac.uk/people/staff/cr3/toolbox/haskell/GHood/`

makes this information available. A tracer avoids providing its own semantics but hooks on to a compiler instead. A program transformation provides this "hook" in a portable way. Even more important than the information in a trace is its accessibility.

8 Conclusions

We have presented the new modular architecture of our Haskell tracer Hat. At its heart lies the new Augmented Redex Trail trace structure, designed on the one hand to be written to file while performing the traced computation, and on the other hand to provide data sufficient for multiple views.

As an immediate result, we have widened the applicability of the new Hat considerably. Initial experiences confirm the usefulness of generating a trace only once and viewing it in several different ways.

The new modularity improves the understanding of the tracing process. The new architecture has also prompted us to ask some more general questions, such as those in the Future Work section.

9 Acknowledgements

Many thanks to Amanda Clare, Henrik Nilsson, John O'Donnell, Claus Reinke, and Jan Sparud who each took time to travel to York and play with various different tracing systems. The feedback from these experiments was extremely valuable.

We also thank the referees for their constructive comments.

The work reported in this paper was supported by the UK's Engineering and Physical Sciences Research Council under grant number GR/M81953.

References

[1] Simon P Booth and Simon B Jones. Walk backwards to happiness — debugging by time travel. Technical Report CSM-143, University of Stirling, 1997.

[2] 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.

[3] 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.

[4] N. D. Jones and A. Mycroft. Data Flow Analysis of Applicative Programs using Minimal Function Graphs. In *Proc. 13th Annual Symposium on the Principles of Programming Languages* (POPL'86), pages 296–306, ACM Press, January 1986.

[5] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.

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

[7] Claus Reinke. GHood — Graphical visualisation and animation of Haskell object observations. *Proceedings of ACM Sigplan Haskell Workshop 2001*, September 2001.

[8] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[9] Jan Sparud. *Tracing and Debugging Lazy Functional Computations.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1999.

[10] Jan Sparud and Colin Runciman. Complete and partial redex trails of functional computations. In C. Clack, K. Hammond, and T. Davie, editors, *Selected papers from 9th Intl. Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS Vol. 1467, September 1997.

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