Practical aspects of Declarative Debugging in Haskell 98

Bernard Pope
The University of Melbourne
Parkville, Australia
bjpop@cs.mu.oz.au

Lee Naish
The University of Melbourne
Parkville, Australia
lee@cs.mu.oz.au

ABSTRACT
Non-strict purely functional languages pose many challenges to the designers of debugging tools. Declarative debugging has long been considered a suitable candidate for the task due to its abstraction over the evaluation order of the program, although the provision of practical implementations has been lagging. In this paper we discuss the solutions used in our declarative debugger for Haskell to tackle the problems of printing values, memory usage and I/O. The debugger is based on program transformation, although much leverage is gained by interfacing with the runtime environment of the language implementation through a foreign function interface.

1. INTRODUCTION
Stepwise tracing debuggers, such as those typically used for imperative languages, are not suitable for non-strict functional languages because the evaluation order of a program and its textual definition are highly decoupled. Declarative debugging is a promising alternative because it permits one to reason about program evaluation in a manner that reflects the structure of the source code, thus abstracting away the difficult issue of evaluation order. In this paper we consider the application of declarative debugging to Haskell 98[1]. In particular we discuss practical issues in the implementation of a debugger that are necessary to make it a useful tool.

A declarative debugger constructs a tree that gives a high-level semantics to the evaluation of a program. We call this an Evaluation Dependence Tree (EDT)[2]. For a functional language, the nodes in the EDT represent function applications and constants that were evaluated for a given execution of the program. The links between nodes are determined by the static call graph of the program (or the dynamic call graph when higher-order code is used). The EDT can be used for browsing or to search for causes of program errors. The key feature of the EDT is that its structure is easily reconciled with that of the program text. Consider the program fragment below:

[1] [2]. Hereafter we drop the *98* from the name.
[2] The EDT terminology is derived from Nilsson and Sparud [12].

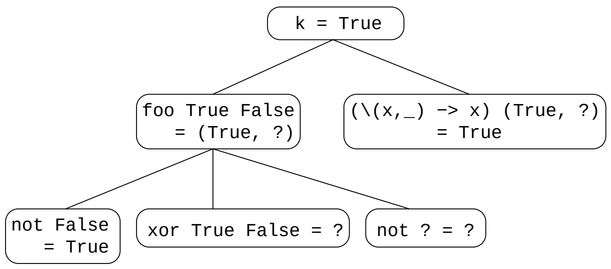

Figure 1: An example EDT.

```
k = (\(x,_) -> x) (foo True False)
foo x y = (not y, not (xor x y))
not True = False
not False = True
xor True x = not x
xor False x = x
```

Figure 1 illustrates the EDT corresponding to the evaluation of the constant `k`. The node for `k` has two children, one for the inner call to `foo`, and one for the outer call to the lambda abstraction '`\(x, _) -> x`', which extracts the first value from a pair. The function `foo` has three children corresponding to the two calls to `not`, and the one call to `xor`. The lambda abstraction, `not` and `xor` form the terminals of the tree because they do not call any other functions. Unevaluated expressions, such as the second component of the tuple, are shown as '`?`'.

There are two strategies proposed in the literature for constructing the EDT. The first strategy is by instrumentation of the language runtime environment and code generator, such that the EDT is produced as a side effect of computation (for example Freya for Haskell [9]). The strength of this approach is that the debugger has intimate knowledge of and access to the runtime environment, including the representation of values on the heap and stack, and the garbage collector. The weaknesses are implementation complexity, and lack of portability (as a result Freya does not support full Haskell and only works on one type of machine architecture). The second strategy is by source-to-source transformation: the original program text is transformed into a new program which computes both the value of the original program and a EDT describing that computation. The advantage of this approach is portability and reduced effort in implementation - one can take advantage of existing compiler and runtime technologies. The main disadvantages are constraints imposed by the static requirements of the source language (most notably in Haskell type correctness) and that the debugger must deal with the runtime environment at arm's length.

We have implemented a declarative debugger for Haskell.[3] There are two parts to the debugger: program transformation and EDT traversal. The transformation [18] is applied to every module in the program (including the standard libraries). It takes a Haskell program as input and produces a Haskell program as output. The transformation is written in Haskell and should work with any compliant implementation of the language. Execution of the transformed program has the effect of computing the original program and building an EDT. Traversal of the EDT is written in a mixture of Haskell and C and is reliant on certain facilities provided by the runtime environment of the language implementation (for reasons that we will describe in the rest of the paper). At present (the EDT traversal part) of our debugger only works with the Glasgow Haskell Compiler (GHC)[4].

In section 2 we show how the EDT is implemented and consider the task of printing information in the nodes. As the debugger traverses the EDT it must print information contained in each node, including the name of the function that was applied, its arguments and eventual result. Writing such a printer in Haskell is difficult because it must be polymorphic, it must respect the laziness and sharing in the underlying value, and it must be able to give a meaningful representation to functions. In section 3 we consider the space usage of the transformed program. It is well known that naively generating an EDT node for every function application causes a prohibitive space leak in the debugger [8, 10]. We reduce the space cost of the EDT by statically pruning it for trusted functions and by re-computing parts of the tree on demand. In section 4 we consider debugging programs that perform I/O and the difficulty of incorporating I/O into the declarative debugging paradigm. In section 5 we apply our debugger to a moderate sized program to see how well it performs in practice, and we also consider how the debugger deals with uncaught exceptions. In section 6 we consider other debugging systems for Haskell and in section 7 we conclude. Familiarity with non-strict purely functional languages is assumed.

2. THE EDT AND PRINTING VALUES

The EDT is implemented by the following type:

```
data EDT = MkEDT Exp [Val] Val [EDT]
```

The `MkEDT` constructor builds a node in the EDT, it has four arguments: the name (or representation) of the function involved in an application; a list of references to the arguments of an application; a reference to the result of the application; and a list of children nodes. Additional information can be kept in the nodes, such as type annotations, and source code references, however we do not show them here.

[3]www.cs.mu.oz.au/~bjpop/buddha

[4]www.haskell.org/ghc

The type `Val` is used as a reference to the arguments and result of function applications. We want to allow references to any number of different types in the same EDT. To do this we need a polymorphic constructor that does not reveal the type of its argument. This can be achieved with existential types, which are not part of the Haskell standard, but are widely supported:

```
data Val = forall a . V a
```

The explicit `forall` occurs in a negative context, which is the same as existentially quantifying the type variable '`a`'.

The name of a function stored in a node is either an identifier or a lambda abstraction. We represent both of these with the type `Exp`:

```
data Exp = ExpId Ident
         | ExpAp Exp Exp
         | ExpLambda [Pat] Exp
         | ExpLet [Equation] Exp
         | ExpCase Exp [Alt]
         | ExpVal Val
```

This type is an abstract syntax for Haskell expressions, though we have truncated the definition for brevity. Identifiers are formed by '`ExpId` *i*', where *i* is a string naming the identifier. Lambda abstractions are formed by '`ExpLambda` *ps e*', where *ps* are bound patterns and *e* is the right-hand-side of the expression. When a node from the EDT is displayed the name of the function is pretty-printed, for example '`ExpId "not"`' would be printed as '*not*'. Applications, let and case expressions are formed by the `ExpAp`, `ExpLet`, and `ExpCase` constructors. We need a full abstract syntax, including patterns, alternatives and equations because we want to be able to show a complete lambda abstraction which may contain all of these syntactic constructs. Free variables in an expression are represented simply as a reference to the value of the variable by the `ExpVal` constructor (notice its argument is of type `Val`).

Each function in the original program is transformed to compute its normal value *and* an EDT node representing an application of the function. At the top of each Haskell program is the constant '`main :: IO ()`'. After transformation this becomes '`main' :: (IO (), EDT)`', and is no longer the top of the program. When we traverse the EDT we want the arguments and result of each function application to be in their final state of evaluation. To ensure this we do not traverse the EDT until the value of the original program has been computed. To avoid some complexity in later parts of the paper we will give names to the two important phases in the evaluation of a transformed program. *Phase one* occurs when we demand the value of the original program to be computed. This amounts to forcing the first component of the tuple returned by `main'` to be fully evaluated. Phase one respects the declarative semantics of the original program, but not the operational semantics (it may consume more memory and take more reductions). *Phase two* occurs when we traverse the EDT which is the second component of the tuple returned by `main'`. A new top-level `main` is introduced after transformation to administer the evaluation of both phases.

When a node in the tree is displayed the values of the arguments and result must be printed. However, printing can interact badly with non-strict evaluation. In a non-strict language a value may be only partially computed. To accurately represent the computation of the program printing must not cause any further evaluation of the arguments and results of applications. Further complication arises when we consider printing values which contain cycles. It is not possible to determine if an expression is evaluated or has cycles from within pure Haskell code. Therefore, to print values referred to by the EDT we have to momentarily step outside the confines of pure functional programming.

Printing works as follows. In Haskell we unpack the value wrapped in the `V` constructor and pass a pointer to the value to C code via the (now standard) Foreign Function Interface (FFI) [1]. The C code traverses the GHC heap representation of the value and prints a representation of the value using Haskell syntax (extended with tokens to indicate unevaluated expressions and cycles). Values on the GHC heap are represented by a graph [6]. Printing terminates in one of three ways: when a node corresponds to a nullary constructor, when a node corresponds to an unevaluated expression, and when a cycle in the graph is found. Unevaluated expressions are denoted on the GHC heap by special values called *thunks*. When a thunk is encountered a question mark is printed to indicate that the value of this expression was not computed. Cycles are identified by hashing the address of heap nodes as they are traversed. Therefore it is essential that traversal of the heap does not cause any garbage collection to happen, lest the nodes are moved to new addresses. Ordinarily the names of data constructors are not retained in the GHC heap, since they are not needed during normal execution of the program. This is of course a problem for printing. However, when GHC compiles code for profiling it uses a slightly more detailed heap representation which includes the names of all constructors. We require that the transformed program is compiled for profiling so that we can access the names of constructors as we traverse the heap. Note that we do not request any profiling statistics to be gathered, so the overheads of this trick are small (and one occurrence of a name is shared amongst all instances of the constructor).

Figure 2 illustrates the GHC heap representation of a partially evaluated tuple (such as that returned by '`foo True False`' from the example program in section 1). Node `A` corresponds to the tuple constructor, whose name is found at node `B`. Nodes `C` and `E` are the components of the tuple, the first being the constructor `True`, and the second being a thunk corresponding to the unevaluated expression '`not (xor x y)`'. The name of the `True` constructor is found in node `D`. Under normal compilation nodes `B` and `D` would be absent, but are included when the program is compiled for profiling.

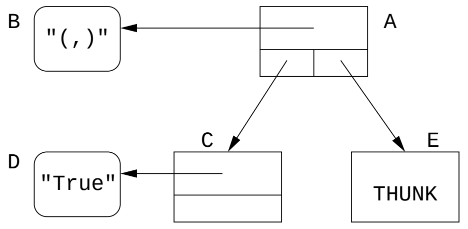

Figure 2: The heap representation of a partially evaluated tuple.

Can we use this technique for printing functional values? Perhaps, but this would at least require the names of all identifiers in the program text to be retained at runtime. In GHC, even with profiling turned on, only the names of top-level functions are retained. Worse still, we would like to represent lambda abstractions in a programmer friendly way (by using source code identifiers and expression syntax). However, in most compilers (including GHC) the compiled form of a function is not easily related back to the source code that produced the function. We use our program transformation to solve this problem by pairing functional values with a representation which is derived directly from the source code. We use the representation whenever we wish to print the function.

To begin with, functional expressions (partial applications or lambda abstractions) are *wrapped up* in the following type:[5]

```
data F a b = MkF (a->b) Exp
```

The constructor `MkF` pairs a function with its representation which is provided by the `Exp` type. When a function is wrapped up in this way, it is said to be *encoded*. So for example the function:

```
\x -> case y of True -> x
```

is represented as:

```
ExpLambda
   [PatId "x"]
   (ExpCase (ExpVal (V y))
      [Alt [PatId "True"] (ExpId "x")])
```

If a variable is free in the function, and that variable is either a let-bound[6] constant or bound by an outer lambda then the variable is embedded into the representation of the function by wrapping it with the `ExpVal` and `V` constructors. Therefore, such variables are represented by their value, not

[5]Here we overlook the fact that functions should produce EDT nodes in the transformed program. That detail makes the presentation unnecessarily complicated. The interested reader should consult [18] for a more thorough treatment of the encoding of functions.
[6]By *let-bound* we mean a variable bound by an `=`. All other variables are considered *lambda-bound*.

their name, which provides more information when the function is printed, and is essential when we come to consider re-evaluation in section 3.

Encoded functions must be unwrapped before they can be applied, this is the job of `apply`:

```
apply :: F a b -> a -> b
apply (MkF f exp) = evalExp exp f
```

The call to `evalExp` evaluates the representation of the function before returning the function itself. Without this the representation of functions would remain as thunks, which would then have to forced before printing. It is much simpler to force the representation at the point of application. Unfortunately this might result in repeated traversals of the function representation for multiple applications of an encoded function - in practice this cost is not too high. The definition of `evalExp` is straightforward, it simply crawls over every branch of the representation, except anything wrapped by `ExpVal` - since these correspond to free variables in the function, and their state of evaluation must not be altered.

Whenever the printer encounters a `MkF` node on the heap it skips immediately to the second argument. However it does not print the representation of the function as it would an ordinary value. Instead the representation is is treated as an abstract syntax tree that should be pretty-printed like source code. If the printer encounters an application of the `V` constructor whilst it is interpreting an encoded function it returns to its normal (literal) mode of printing. Thus the `V` and `MkF` constructors play the role of quotation marks to direct the way values are printed, and the C routines for printing normal heap values and function representations are mutually recursive.

3. RESOURCE CONSUMPTION

Creating an EDT node for every function application is prohibitively expensive. During phase one the construction of EDT nodes is delayed, causing a a thunk to be allocated on the heap for each node. Inside these thunks are pointers to the arguments and result of an application. The space consumed by the tree during phase one is equal to the sum of the sizes of each thunk, plus the size of all the intermediate values that are referred to by the pointers inside the thunks. Consider the naive reverse function:

```
rev :: [a] -> [a]
rev ys = case ys of
            [] -> []
            (x:xs) -> (rev xs) ++ [x]
```

The operator '`++`' is list append in Haskell. If we use this function to reverse some list `[e1, ..., ej, ek]` we end up with the computation:

```
( ... (([] ++ [ek]) ++ [ej]) ++ ... ) ++ [e1]
```

Each inner application of '`++`' produces an intermediate list, which under normal evaluation can be incrementally garbage collected. However, in the transformed program each call to '`++`' results in an EDT node (thunk) which maintains pointers to its arguments and result. It is the presence of these pointers that prohibit the garbage collector from reclaiming the space used by the values pointed to. For long running programs this will almost certainly exhaust the heap in phase one leaving no chance of traversing the EDT.

The obvious solution is to prune the EDT. The simplest way to do this is to pick some functions from the program and tell the transformation not to create EDT nodes for applications of those functions. This is easy to do, and is a good idea for *trusted* functions that are considered correct (such as well-tested library code). However, trusted functions might call un-trusted ones, either statically, or by higher order arguments. In such circumstances we want EDT nodes for the untrusted children of trusted nodes. Again this is easy to do, and leads to a new type of EDT node:

```
data EDT = ...  | MkTrust Val [EDT]
```

The `MkTrust` constructor builds an EDT node that records the result and children of a trusted application. We need to record the result of the application because it is possible that it was not needed (a thunk). In such a case the debugger should not traverse the children of this node because the parent was not needed.[7] Unfortunately, such trusted nodes are not a big space saver: each application of a trusted function results in an EDT node and that node maintains a reference to the result of the application, prohibiting its garbage-collection.

A more radical idea is to allow a sub-tree of the EDT to be discarded (or perhaps not computed at all) during phase one. Then if it happens that nodes in the sub-tree are needed during phase 2 they can be constructed by re-evaluating part of the transformed program. This is a classic space-time tradeoff: during phase one we hope to save the space consumed by (part of) the EDT which we trade for additional computation time in phase 2. Of course we only benefit if the space required to allow re-construction of the sub-tree is less than the size of the sub-tree itself.

We need a way of allowing a sub-tree to be discarded (or not computed at all), whilst providing a means to re-construct it at a later point in time. To understand how this is done we must look more carefully at the program transformation. Below is a sketch of how the function `rev` is transformed to compute a full EDT node:

```
rev :: [a] -> ([a], EDT)
rev ys
 = let (v,c) = ... transformed body ...
       t = MkEDT (ExpId "rev") [V ys] (V v) c
   in (v, t)
```

The new `rev` computes a pair `(v,t)` containing the reversed list and an EDT node. The body of the original function

[7] Although this appears to be a small wrinkle, it is absolutely necessary when the trusted function is recursive and produces a lazily demanded, but possibly infinite result.

is transformed to compute a pair (v,c), where c is a list of nodes corresponding to the children of rev. The node t contains the name of the function, a reference to its argument ys, a reference to its result v and its children c. Space is consumed by the nodes in c, and those nodes contain references to the intermediate values that were used in the construction of v. The presence of these references prohibit the garbage collection of the intermediate values. We can't discard c altogether because we might eventually want to visit those nodes in phase 2. Instead we must disconnect the computation of v and c.

We introduce a third kind of EDT node:

```
data EDT = ... | MkR Exp [Val] Val Val [EDT]
```

It is almost identical to the node constructed by MkEDT, except that it has two copies of the result of the application: each copy denotes the same value but they are obtained from *independent* computations. To construct such a node we modify our transformation slightly:

```
rev :: [a] -> ([a], EDT)
rev ys
 = let (v1,_) = ... transformed body ...
       (v2,c) = ... transformed body ...
       t = MkR (ExpId "rev") [V ys]
                             (V v1) (V v2) c
   in (v1, t)
```

Now there are two copies of the transformed body. The first copy is bound to (v1,_), and it is v1 that is returned at the end of rev along with an EDT node. The underscore in the pattern binding indicates that the children nodes are not needed, so they may be garbage collected during phase one. The second copy is bound to the pair (v2,c). During phase one the value of this computation is not demanded so it remains a thunk. During phase two we might want to traverse the children in c. However, at the end of phase one they refer to an unevaluated computation denoted by v2 - all the arguments and results at each node will be thunks. So v2 must be evaluated before the children can be traversed, and so a reference is kept to v2 in the EDT. In a non-strict language the extent to which an expression is evaluated depends on its context. However, at the point when v2 must be evaluated (during phase two) the original context is lost. To make up for the lack of context, a reference to v1 is kept in the EDT. The state of evaluation of v1 is used to determine the extent to which v2 is evaluated.

EDT traversal for re-evaluation nodes is structured in the following way:

```
trav :: EDT -> IO ()
trav (MkR n as v1 v2 c)
 = do t <- isThunk v1
      case t of
         True  -> return ()
         False -> do ... print the node ...
                     reEval v1 v2
                     travChildren c
```

The code uses the IO monad and *do-notation* of Haskell to allow interaction with the user of the debugger and ensure the correct sequencing of re-evaluation with respect to the traversal of the children. Firstly we check whether v1 is a thunk, supposing the existence of the function 'isThunk :: Val -> IO Bool'. If it is a thunk then the application was not evaluated during phase one so we should not look at the children. If it is not a thunk then we print the node and perhaps interact with the user of the debugger. Before we traverse the children we must make sure that v2 is evaluated to exactly the same extent and v1.

The re-evaluation is performed by:

```
reEval :: Val -> Val -> IO ()
```

We know that the two arguments denote the same value. Our goal is to force the evaluation of the second argument so that it is evaluated to exactly the same extent as the first. It works in the following way. In Haskell we unpack the two arguments to reEval from the V constructors, and pass pointers to their values to C code via the FFI. To assist our presentation we will call the first of the arguments the *old* value and the second the *new* value. We construct a graph on the C heap that mimics the GHC heap representation of the old value. We terminate the C graph in the presence of nullary constructors, thunks and cycles.[8] The new value is initially a thunk. If the C graph is not empty we force evaluation of the thunk to weak head normal (WHNF)[9]. We repeat this process by pairing the children of the C graph with the corresponding children on the GHC heap. We stop forcing branches of the new value when we reach the terminals of the C graph. Why don't we use the GHC heap representation of the old value to direct the evaluation of the new one? The reason is that forcing parts of the new value to WHNF may cause garbage collection. Garbage collection may move nodes around on the GHC heap, which may cause us to lose track of cycles in the graph representation of the old value. When we construct the C graph we ensure that no garbage collection can happen, thus allowing accurate detection of cycles.

The transformation ensures that higher-order values will always be encoded in the result of transformed functions, even when nested inside other data-structures. To re-evaluate an encoded function we demand that the representation of the new encoded function is evaluated to the same extent as the representation of the old encoded function. This includes forcing the free variables in the representation to be evaluated to their previous extent. This detail is rather subtle. Consider the program fragment below:

```
f x = \y -> ... x ...
g z = f (h z)
```

[8]We should also reflect the sharing in the GHC graph, although at present we do not. We anticipate that this is not difficult to implement, and it is intended to be added to future versions of the debugger.
[9]A value is in WHNF if its outermost constructor is evaluated, or it is a function.

Table 1: space (MB) – time (sec) performance of naive reverse

List size	original	empty	full	trust	re-eval	re-eval + orig
500	1 – 0.07	1 – 0.18	38 – 6.3	24 – 4.6	3.5 – 2.5	3.5 – 0.21
1000	1 – 0.34	2.2 – 0.57	210 – 45	103 – 23	3.5 – 10	3.5 – 2.0
1500	1.5 – 0.84	2.2 – 1.3	? – ?	240 – 77	3.5 – 33	3.5 – 2.5
2000	1.5 – 1.6	2.2 – 2.3	? – ?	? – ?	4 – 67	4 – 3.6

The EDT node for `g` has two children corresponding to the calls to `f` and `h`. The result of `f` is a function, and consequently the result of `g` is the same. After transformation this function will be encoded, and the representation inside the encoding will be:

```
ExpLambda [PatId "y"] ... (ExpVal (V x)) ...
```

Note that `x` is a free variable in the function. If we want to re-construct the children of `g` then it is essential that we make the new value of `x` the same as the old one (using the technique described above). Without this `(h z)` would appear to be unevaluated. The re-evaluation of `x` ensures that the node for `h` (under `g`) has its result evaluated to the correct extent. Therefore it would be wrong to use the following representation in the encoded function:

```
ExpLambda [PatId "y"] ... (ExpId "x") ...
```

Doing so would lose information about how much `(h z)` was demanded.

It is essential that the old and new values are obtained by independent computations. Unfortunately there is no way to ensure this within the syntax of Haskell. An optimising compiler might notice that the old and new values are equivalent and arrange for them to share the one computation, thus re-introducing the space leak. To enforce this requirement we have to ask the GHC compiler not to perform common sub-expression elimination when compiling the transformed program. Unfortunately this option is only available on a per-module basis, so we effectively prohibit this optimisation everywhere.

When computing the old value it seems excessive to use the transformed body of the function. Since we are only interested in the value (and not the children nodes), we should be able to compute the value by using the original body of the function, as in:

```
rev :: [a] -> ([a], EDT)
rev ys
 = let v1     = ... original    body ...
       (v2,c) = ... transformed body ...
       t = MkR (ExpId "rev") [V ys]
                             (V v1) (V v2) c
   in (v1, t)
```

This is complicated by the fact that inside the original body there may be calls to functions that are now transformed. To overcome this problem we could keep two copies of every function in the program: the original code to be called from inside the original body of a transformed function (and calls between original functions), and the transformed code. A further complication is caused by the fact that higher-order values are always encoded in the transformed program. This creates an interface problem when we want to pass a higher-order argument from transformed code to original code (the original code will not be expecting an encoded argument). Instead of using original code we could use a light-weight transformation that allows otherwise original code to accept encoded higher-order arguments and produce encoded higher-order results where necessary. Extending the transformation to allow original code with re-evaluation is not implemented in our debugger, but experiments suggest that it is worth the extra complexity.

How effective is re-evaluation? Table 1 lists the maximum space consumption and running time of naive reverse after various types of transformation.[10] The test program constructs a list of a given number of integers, applies `rev` to the list and prints the result. Measurements were made up to the end of phase one (before the EDT is traversed). Memory usages are in megabytes representing the largest amount of space used by the program. Running times are in seconds. Question marks indicate where memory usage was too large to get a result. The left column indicates the number of integers in the list. The remaining columns indicate how the program performed under different styles of transformation:

- *original* - the original program without transformation.
- *empty* - debugging transformation that does not create any EDT nodes for any function applications. This indicates what overheads are introduced by the transformation alone (without the EDT).
- *full* - debugging transformation that creates nodes for every function application.
- *trust* - debugging transformation that creates a trusted node for every application (using the `MkTrust` constructor).
- *re-eval* - debugging transformation that creates a re-evaluation node for each application of `rev` and `++` (we can't re-evaluate I/O code - see section 4).
- *re-eval + original* - debugging transformation that creates a re-evaluation node for each application of `rev` and `++`, but uses original code where possible in the

[10]The tests were performed on a 300MHz AMD x86 using GHC version 5.04.2 with optimisations turned on.

body of the transformed functions, according to the suggested improvement mentioned above.

It is clear that creating a full EDT node for every function application is impractical. Trusted nodes using MkTrust do not produce great savings in space nor time, although this is not particularly surprising. Re-evaluation affords large space savings during phase one. Using untransformed code in combination with re-evaluation does not influence the memory usage, but provides significant improvements in time which are due to reduced garbage collection overheads. We are surprised that this enhancement does not use less memory than the normal re-evaluation transformation, because most of the program execution in the second case is in original code. Further investigation is required to understand why this is so. Of course we should avoid making broad conclusions from one small test case like this, particularly since it does not give any indication of what the costs are when higher-order code is used. In section 5 we will briefly discuss our experiences with more realistic programs.

4. I/O

Naturally, for a debugger to be practical it must support programs that perform I/O. In Haskell, monads have been used to fit I/O quite elegantly into the pure non-strict functional paradigm [14]. Supporting Haskell's I/O presents two problems for our debugger. The first problem concerns the program transformation, and the fact that we must apply the same transformation rules uniformly to the whole program, regardless of which parts perform I/O. Since we have largely avoided discussing the details of the transformation we do not wish to delve too deeply into this issue in the current paper. The interested reader is invited to consult [16] for more information on the nature of the problem and how we solve it. The second problem concerns the relationship between declarative debugging and I/O. The declarative debugging algorithm is directed by the user's intended interpretation of the program: function applications are displayed and the user is asked if the result of the application is correct for the given arguments. This relies on the ability of the user to decide the correctness based on the values of the arguments and results of applications. However, for functions that perform I/O, correctness depends on the sequencing of I/O events and on the environment in which the program is evaluated, which are not easily related to the structure of the EDT. Our solution is to limit the scope of declarative debugging to subtrees of the EDT that do not contain any I/O. In practice this works quite well, and there is scope for alternative debugging techniques to be applied to the part of the tree that concerns I/O activity, although we have not considered this aspect in detail.

In Haskell, the type 'IO t' describes a computation that produces a value of type t and *may* cause side-effects. Such a computation is often called an *action*. Primitive actions are provided by libraries and the runtime environment, and combinators are provided for composition. The top-level main function has an IO type, and so the entire Haskell program is an action (or rather a composition of actions) that modifies the state of the world via input and output interaction with the operating system. Actions can occur as the arguments and results of applications, but how should the

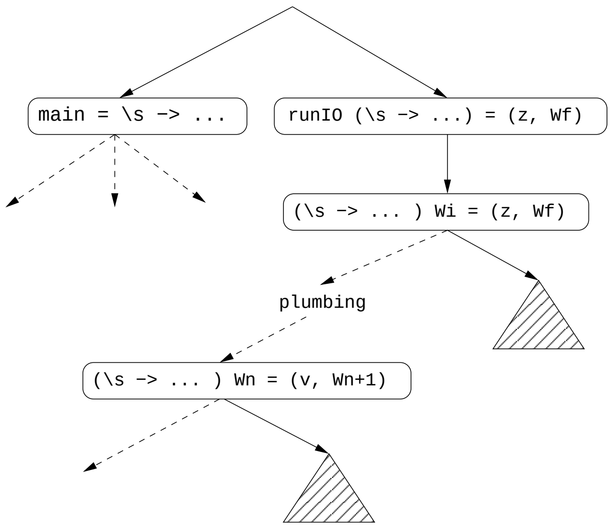
main = \s -> ...
runIO (\s -> ...) = (z, Wf)
(\s -> ...) Wi = (z, Wf)
plumbing
(\s -> ...) Wn = (v, Wn+1)

Figure 3: An EDT with I/O actions.

debugger print such actions? The IO type is abstract in Haskell, although typically it is implemented as a function that threads a token through the program representing the state of the world. This is certainly true in GHC and also in our debugger.[11] It is possible to print this function out, although it is usually a tangle of lambda abstractions and combinators for plumbing the state of the world. What we would like to know is what happens when the function is applied to a state of the world. Unfortunately this application happens outside of main.

Figure 3 illustrates an EDT with nodes that reveal the state-threading of Haskell's I/O monad. As far as the programmer is concerned, main is the top level of the program, under which everything else happens. From an operational point of view, this is not the case. Instead main returns a function. In the debugger's implementation of I/O, that function is applied and evaluated by runIO which is part of the underlying I/O machinery. An initial state of the world token Wi is conjured up by runIO, and at the end of the whole computation the pair (z,Wf) is produced. The value of z is demanded by runIO which causes the evaluation of the entire program, which includes the performance of I/O actions. The final state of the world token is Wf with intermediate tokens Wn and Wn+1 exchanged along the way. The variable v denotes some value in the computation, and the lambda abstractions (\s -> ...) are state threading functions where s is the state parameter. In between the state threading functions are combinators for plumbing the state through the program. The hatched triangles indicate subtrees of the EDT that contain functions that do not perform I/O. It is runIO that causes the program to evaluate, and the bulk the the EDT resides under it (in practice there aren't many nodes underneath main, only those required to construct its functional result).

[11] To facilitate the interface between transformed code and the primitive I/O actions, we re-implement the I/O infrastructure in the debugger on top of that provided by the compiler.

It is not clear how to derive a suitable semantics for I/O actions from the EDT, and thus make them amenable to declarative debugging. For this reason we ignore any EDT nodes that correspond to I/O actions and skip immediately to their children when traversing the tree. In the rare event that an `IO` value appears as an argument to a function that does not produce an `IO` result directly (such as '`map print [1,2,3]`') the `IO` values are printed abstractly as `<IO>`. This strategy means that the user of the debugger only sees the part of the EDT that corresponds to non-`IO` code. Bugs in the I/O part of the program will not be found by our debugger, however the ignored parts of the EDT may be useful for other styles of debugging that are more suitable for I/O, which we will consider in future work.

5. DISCUSSION

How practical is the debugger on a realistic example? This question is difficult to answer because it is impossible to say what a realistic program is, or for that matter a realistic bug. However it is nonetheless useful to get some metrics on the application of the debugger to a non-trivial example. Our test case is a simple ray-tracer that can render three-dimensional scenes consisting of multiple point light sources and multiple spheres. The code for the ray-tracer is available in the examples directory in the source distribution of the debugger, and consists of about 1000 lines of Haskell code (including comments) across nine modules. The program reads a scene description file, parses it, renders each pixel in an n by m image and writes the image to a file. The dimensions of the image (n, m) are provided by command line arguments. For our test case we apply the ray-tracer to two different scenes, the first contains one sphere and one light, the second contains seven spheres and two lights, and we render each scene in an image of 100 by 100 pixels. The machine, compiler and compiler flags are all the same as those used in the previous test cases for naive reverse in section 3. In section 6 we apply a competing Haskell debugging system to the same program for comparison.

Table 2 shows the results of the tests. The left column indicates the scene that was rendered and the remaining columns indicate the space and time performance of the ray-tracer under different kinds of transformation. For the transformed versions of the program the time measurement indicates the duration of executing phase one. At this point the ray-tracing is complete and the debugging may commence. This is how long the user must wait until they can interact with the EDT traversal. The space usage indicates the maximum memory required to execute the program up until the end of phase one. During traversal of the EDT the memory usage may grow further due to the construction of more EDT nodes, though in practice we have found that this is not significant. The numbers immediately underneath each of the transformation types indicate the size of the (statically linked) executable that is produced when the program is compiled. The transformations are as follows:

- *original* - the original program without transformation.
- *empty* - debugging transformation that does not create any EDT nodes for any function applications.
- *re-eval* - debugging transformation that creates a re-evaluation node for each function application (except IO functions).

Transforming the program so that an EDT node is created for every function application (without re-evaluation) requires an unreasonable amount of space (greater than 200MB), so that it is effectively impossible to debug the ray-tracer using that scheme. Therefore, the case for re-evaluation is quite strong in this example. Space consumption is about seven times that of the original, and execution time is about twenty times that of the original. Of course the space savings do not come for free! The tradeoff is that the user of the debugger must wait for parts of the EDT to be reconstructed during EDT traversal. The longest waiting period is approximately equal to the time taken to execute phase one of the transformed program. Although the waiting times will decrease as traversal proceeds further down the tree. In general it is not necessary to transform all of the program for re-evaluation, only the parts that are computationally expensive. Further savings in space and time can be had by transforming trusted parts of the program to produce empty EDTs. For example in the case of the ray-tracer we may not be interested debugging the parser, so we do not need to construct nodes for functions involved in parsing. The necessary transformation style can be easily applied to a whole module, or selectively for each function depending on the requirements of the user.

There is another important practical aspect that we have not mentioned so far in the paper: uncaught exceptions in the debugged program. It is essential that uncaught exceptions that are raised by the program being debugged do not cause the debugger to crash. The potential for such a crash is made possible by the fact that part of the debugging program is the original program (albeit a transformed version of it). Our debugger implements a kind of virtual I/O on top of the I/O facilities provided by the compiler. One advantage of implementing the debugger with GHC is that we can make use of its quite expressive exception handling mechanism: [13]. The interface between our virtual I/O and that of the compiler occurs at the primitive I/O actions (those that implement reading and writing characters for example). When a virtual primitive is called by the program being debugged care is taken to catch any exceptions that are raised by the execution of the real primitive using the exception catching facility provided by GHC. Note that execution of real primitives can cause evaluation of sub-computations in the program, and those computations may raise exceptions - so the evaluation of the primitive effectively causes the exception to be thrown. If an exception is caught then it is reified into the virtual I/O by flagging the exception in the state of the world that is threaded through the I/O actions. A state containing a flagged exception causes the execution of the normal chain of I/O actions to terminate and the state is propagated upwards. If the exception is not handled in the original program then `runIO` will return a state of the world with a flagged exception. The good news is that the the EDT can be created for all function applications up until the exception was thrown, and we may traverse it as usual.

The presence of exceptions provides two additional ways in which the results and arguments of applications may be

Table 2: space (MB) – time (sec) performance of ray-tracer and executable sizes

Scene	original exe = 1190568 bytes	empty exe = 3598926 bytes	re-eval exe = 4559994 bytes
1 sphere	1.5 – 1	4 – 14	13 – 20
7 spheres	2 – 6	4 – 75	14 – 101

printed by the debugger. Under GHC's exception mechanism all types are extended with additional values that represent exceptional computations. Since the debugger must be able to print any (non-bottom) member in a type, it must be able to print those that are exceptional. In GHC a special heap value is used to mark synchronous exceptions, and the type of exceptional event is encoded by an ordinary Haskell data type which is used by the debugger when it prints the exception. Asynchronous exceptions are caused by external events and so cannot be attributed to any particular function application. Expressions that were under evaluation when an asynchronous exception occurs must be able to be resumed after the exception is handled - in general resuming them will not necessarily cause the exception to be raised again. When an asynchronous exception is uncaught in the debugged program the results of function applications that were under evaluation at the time of the exception will be marked by GHC as resumable. Resumable computations are printed by the debugger with an exclamation mark to indicate that their value is not known, but they were under evaluation when an asynchronous exception was thrown. This is to distinguish them from thunks which were not needed at the time of the exception.

6. RELATED WORK

The use of an EDT for declarative debugging is well known: [8, 15, 19, 11, 3]. The main detractor of the earlier approaches being a lack of support for higher-order programming, and the prohibitive cost of the space consumed by the EDT.

The debugger described in this paper stems from earlier research by Naish and Barbour [8] (see also Naish [7]). In recent work [15], a prototype declarative debugger for Haskell was implemented. Only a subset of the language was supported, the most notable omission being arbitrary higher-order code. An attempt to support higher-order code and curried function definitions based on a type-directed program specialisation was proposed in [17]. Unfortunately the specialisation interacts badly with polymorphic recursion and separate compilation. The prohibitive space consumption of the EDT was considered in [8] and also [15], and the basic idea for solving the problem was also formulated in those papers. However, it is not until now that an implementation has been achieved.

Sparud gives a program transformation for declarative debugging of Haskell [19]. Our transformation is similar to his, however we differ in our treatment of higher-order code. For printing values, he suggests the use of type-classes to provide an overloaded function which produces a representation for values in the program. This still requires support from the runtime environment to determine whether an expression is evaluated, or cyclic. Our experience shows that it is more practical to implement all of the printing in C. Unfortunately there appears to be no available implementation of his debugger.

Nilsson [11] uses an instrumented runtime environment to construct an EDT as a side-effect of computation. The advantage of this approach is that it allows for greater access to the runtime representation of values. A technique called *piecemeal tracing* is employed to constrain the memory consumed by the EDT, by placing an upper limit on the size of memory occupied by the EDT at any one time. Recomputation of part of the program is required to generate branches of the tree that do not fit into memory. This works quite well in practice however it requires extensive modification of the runtime environment of the language and would not be easy to do in a compiler like GHC. The disadvantage of this approach is the complexity of implementation. A whole new compiler for a large subset of Haskell was created for the purposes of providing the necessary instrumented runtime environment. Our motivation for employing program transformation is to simplify the implementation of the debugger and to take advantage of existing compiler technology. Unfortunately the compiler does not support enough of the Haskell language to test the debugger with the ray-tracing example discussed in section 5. However, when applied to the task of naive-reversing a list of 2000 integers the debugger takes about 16 megabytes of memory and about 10 seconds to reach the first question. Memory usage remains at about 16 megabytes throughout the debugging session. It appears that development of this tool has stopped for a number of years.

A general framework for tracing, debugging and observing lazy functional computations based on reduction histories (or *Redex Trails*) has been proposed in [21, 20], which is implemented by the *Hat* tool[12]. The trails record a rich amount of information about a computation and various post-processing tools have been developed to view the information in different ways, including declarative debugging [22]. The main cost of recording Redex Trails is the space required to store the trail, the size of which being proportionate to the duration of the computation. To cope with the large space requirement, the trail is serialised and written to a log file rather than being maintained in main memory. Hat is also based on transformation, however the resulting program is not dependent on any particular compiler, so their debugging system is more portable than ours (hat is known to work with two different Haskell compilers). There are two different modes of transformation in Hat. The first (untrusted) mode records reductions of functions in a given module, and the second (trusted) does not record reductions for functions in a given module. The purpose of the trusted mode is predominantly to reduce the size of the log file that is generated - it also reduces the time taken but not by a large degree.

[12]`www.haskell.org/hat`

We applied Hat to the ray-tracing example mentioned in section 5 with mixed results. We used the same compiler, machine and optimisation flags as those used to test our debugger. Firstly we transformed the program so that all modules were transformed as untrusted. The resulting executable after compilation is about 6.5 MB, which is slightly larger than that produced by our debugger. When ray-tracing the first scene containing one sphere and one light, the time taken for the program to complete was 293 seconds (15 times slower than our debugger using re-evaluation), the largest memory usage of the process was 14 MB and the resulting logfile was 112MB. When ray-tracing the second scene containing seven spheres and two lights, the results are less promising. We aborted the execution of the program after 20 minutes, the memory usage at that time was about 18 MB and the logfile was 549 MB. To be fair we limited the untrusted transformation to one module in the program. We chose the core ray-tracing module where most of the work is performed in the program. This reduced the size of the logfile to 23 MB for the scene with one sphere, however memory usage and runtime were the same as before. However, when applied to the scene with seven spheres we did not get a result after 20 minutes of runtime, although at this stage the logfile was only up to 50 MB.

By writing traces of reductions to a log file rather than storing the trace in main memory, Hat is able to keep the main memory requirements of the transformed program proportional to the original program. However, the costs of writing this log file are not small. The time required to execute the transformed program is quite long, and the log files can be very large. Care has to be taken to ensure that the tools that process the log files do not require space proportional to the size of the log file.

A means for displaying the evaluation of expressions in a running Haskell program is provided by HOOD[13] [5]. A type class is used to implement an overloaded function called *observe*. This function has the type of the identity function, and so it can freely be inserted into any expression without changing the type of the program. Calls to observe cause the reduction steps for an expression to be logged as a side-effect. The log faithfully reflects the partial evaluation of data. It can capture the evaluation of functions, which are represented extensionally as sets of input and output values. One advantage of HOOD is that it is easily combined into an existing program and it only relies on a few commonly implemented extensions to Haskell.

Detailed comparisons of HOOD, Freya and Hat are documented by Chitil *et al* [4]. However, it should be noted that the implementation of Hat has changed significantly since those comparisons were made.

7. CONCLUSION

In this paper we have highlighted some impediments to debugging in Haskell, and shown how these issues are handled in our implementation of a declarative debugger. The debugger is based on the construction of a tree that gives a high-level semantics to the evaluation of a program. Nodes in the tree represent function applications, and we use program transformation to generate the tree. Printing values stored in the nodes of the tree is difficult to do in Haskell. We avoid this difficulty be performing the printing in C code which has access to the heap representation of values. This is not quite sufficient for higher-order values for which we encode a printable representation during program transformation. Producing a node in the tree for every function application is prohibitively expensive. We show how space consumed by the tree may be saved at the cost of later re-evaluation. Tests with a moderate sized program indicate that re-evaluation does significantly reduce the space cost of the tree. In practice re-evaluation is necessary in order for debugging to be feasible for non-trivial programs. More gains are anticipated by mixing original code with transformed code, however this feature is yet to be implemented. Our debugger is able to support programs that do arbitrary I/O, and raise uncaught exceptions. We limit the declarative debugging algorithm to those sub-computations that do not do I/O. Further consideration of alternative debugging algorithms for the I/O parts of the program are needed.

An implementation of the debugger, a user's guide and example test cases are available from the web-site:

`www.cs.mu.oz.au/~bjpop/buddha`

[13]Haskell Object Observation Debugger

8. REFERENCES

[1] Haskell 98 Foreign Function Interface. `www.haskell.org/definition`, 2002.

[2] Haskell 98 Language Report. `www.haskell.org/onlinereport`, 2002.

[3] R. Caballero and M. Rodri'guez-Artalejo. A declarative debugging system for lazy functional logic programs. In M. Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier Science Publishers, 2002.

[4] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood - a comparative evaluation of three systems for tracing and debugging lazy functional programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, volume LNCS 2011, pages 176–193, 2001.

[5] A. Gill. Debugging Haskell by observing intermediate data structures. Technical report, University of Nottingham, 2000. In Proceedings of the 4th Haskell Workshop, 2000.

[6] S. P. Jones, S. Marlow, and A. Reid. The STG runtime system (revised). Technical report, Microsoft Research Ltd, Cambridge, England, 2001.

[7] L. Naish. Declarative debugging of lazy functional programs. *Australian Computer Science Communications*, 15(1):287–294, 1993.

[8] L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.

[9] H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Department of Computer and Information Science, Linköpings Universitet, S-581 83, Linköping, Sweden, 1998.

[10] H. Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN international conference on Functional programming*, pages 36–47, Paris, France, 1999. ACM Press.

[11] H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, November 2001.

[12] H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, 1997.

[13] S. Peyton-Jones, A. Reid, F. Henderson, C. A. R. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.

[14] S. Peyton-Jones and P. Wadler. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[15] B. Pope. Buddha: A declarative debugger for Haskell. Technical Report 98/12, The Department of Computer Science and Software Engineering, The University of Melbourne, 1998.

[16] B. Pope and L. Naish. Rolling your own I/O in Haskell 98. `www.cs.mu.oz.au/~bjpop/papers.html`.

[17] B. Pope and L. Naish. Specialisation of higher-order functions for debugging. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 87–100, 2001.

[18] B. Pope and L. Naish. A program transformation for debugging Haskell 98. In M. Oudshoorn, editor, *Twenty Sixth Australasian Computer Science Conference*, volume 16, pages 227–236, 2003.

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

[20] J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. In T. D. C. Clack, K. Hammond, editor, *Selected papers from 9th International Workshop on the Implementation of Functional Languages*, volume LNCS 1467, pages 160–177, 1997.

[21] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *PLILP*, pages 291–308, 1997.

[22] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, 2001.