Declarative Debugging with Buddha

Bernard Pope

The University of Melbourne, Australia

Abstract. Haskell is a very safe language, particularly because of its type system. However there will always be programs that do the wrong thing. Programmer fallibility, partial or incorrect specifications and typographic errors are but a few of the reasons that make bugs a fact of life. This paper is about the use and implementation of a debugger, called Buddha, which helps Haskell programmers understand why their programs misbehave. Traditional debugging tools that examine the program execution step-by-step are not suitable for Haskell because of its unorthodox evaluation strategy. Instead, a different approach is taken which abstracts away the evaluation order of the program and focuses on its high-level logical meaning.

This style of debugging is called Declarative Debugging, and it has its roots in the Logic Programming community. At the heart of the debugger is a tree which records information about the evaluation of the program in a manner which is easy to relate to the structure of the source code. It resembles a call graph annotated with the arguments and results of function applications, shown in their most evaluated form. Logical relationships between entities in the source are reflected in the links between nodes in the tree. An error diagnosis algorithm is applied to the tree in a top-down fashion in the search for causes of bugs.

1 Introduction

Debugging Haskell is interesting as a topic for research because, quite frankly, it is hard, and conventional debugging technologies do not suit it well.

On the one hand, Haskell is a very safe language. Implicit memory management, pure functions, and static type checking all tend to reduce the kind of bugs that can be encountered. Trivial programming mistakes tend to be caught early on in the development process. To use an old adage, there are fewer ways that a Haskell programmer can "shoot themselves in the foot". On the other hand, no language can stop a programmer from making logical errors. Furthermore, logical errors can be the hardest to find, due to subtle differences between what the programmer intended, and what they actually wrote. Therefore, debugging tools that relate program execution with the source code are vital for finding logical errors that aren't obvious from reading the source code. Typically there is a wide gap between the structure of Haskell code and the way it is evaluated, and this makes the task of debugging hard, in particular it rules out the step-based debugging style widely employed in imperative languages.

Purely functional languages, along with logic languages, are said to be *declarative*. The uniting theme of these languages is that they emphasise *what* a program computes rather than *how* it should do it. Or to put it another way, declarative programs focus on logic rather than evaluation strategy. The declarative style can be adopted in most languages, however the functional and logic languages tend to encourage a declarative mode of thinking, and are usually used most productively in that way. Proponents of declarative programming argue that the style allows programmers to focus on problem solving, and that the resulting programs are concise, and easier to reason about than equivalent imperative implementations. The declarative style allows more freedom in the way that programs are executed because the logic and evaluation strategy are decoupled. This means that declarative languages can take advantage of novel execution mechanisms without adding to the complexity of the source code. The non-strict semantics of Haskell and backtracking search of Prolog are examples of this.

Despite the many advantages of declarative programming, there are situations when the programmer must reason about *how* a program is evaluated. That is, the evaluation strategy is occasionally very important. For example, when performing input and output (I/O) the relative order of side-effects is crucial to the correctness of the program. The inclusion of monads and the statement-based *do notation* in Haskell reflect this, and where necessary one may adopt an imperative style of programming. Also, efficiency considerations sometimes require that the programmer can influence the evaluation strategy — for example strict evaluation may lead to better memory consumption.

Debugging is another task that suffers when the evaluation strategy is unknown to the programmer. The usual approach to debugging is to step through the program evaluation one operation at a time. However, to make any sense, this method requires the programmer to have an accurate mental model of the evaluation strategy — which is the very thing that the declarative style eschews. Forming an accurate mental model of lazy evaluation is quite a challenge. Logical relationships, such as "X depends on Y", which are evident in the source code, may not be apparent in the reduction order.

The other main factor that complicates debugging in Haskell is the tendency for programs to make extensive use of higher-order functions. The difficulty stems from three things: the function type is abstract and more difficult to display than structured data, the relationship between the static and dynamic call graphs is more complicated, and the specification of correctness much more demanding.

A very basic facility of debugging tools is to print out values from a running program. Functions are first-class in Haskell but they have no inherent printable representation. One might suppose that showing the name of a function would suffice, however not all functions in Haskell have a name (*i.e.* lambda abstractions). Also, there is a tendency to construct new functions dynamically by partial application and composition. If the user of the debugger is to have any hope of understanding what their program is doing, functions must be made ob-

servable, and the way they are shown must be easily related to the user's mental model of the program.

Higher-order functions make holes in the static call graph that are only filled in when the program is executed. Curried functions can pick up their arguments in a piecemeal fashion, and there may be a lengthy delay between when the initial application is made and when the function has received enough arguments for a reduction to take place. Whilst the extra flexibility in the call graph is good for abstraction and modularity, the effect for debugging is similar to the problem identified with non-strict evaluation — it is harder to relate the dynamic behaviour of the program with its static description.

Lastly, fundamental questions like: "*Is this function application doing what it should?*" have less obvious answers in the higher-order context. Understanding the meaning of a function is often hard enough, but understanding the meaning of a function that takes another function as its argument, or returns one as its result, exacerbates the problem. A significant challenge in the design of debugging systems for Haskell is how to reduce the cognitive load on the user, especially when there is a large number of higher-order functions involved. This is an issue that has seen little attention in the design of debugging systems for mainstream imperative languages because higher-order code is much less prevalent there.

The debugger we describe in this paper, called Buddha, is based on the philosophy that declarative languages deserve declarative debuggers. Or in other words, an effective way to deal with the problem of non-strict evaluation and higher-order functions is to aim the debugging tool at the declarative level of reasoning. The result is a powerful debugging facility that goes far beyond the capabilities of step-wise debuggers, extending the benefits of the declarative style from program development to program maintenance.

Overview

In this paper we explain how Buddha is used, how it is implemented and the overall philosophy of Declarative Debugging. You, the reader, are assumed to be comfortable with Haskell, or a similar language, though by no means do you have to be an expert.

A number of exercises are sprinkled throughout the text for you to ponder over as you read along. In many cases the questions are open ended, and there may be more than one "right" answer. Some questions might even be research topics on their own! Of course there is no obligation to answer them all, they are merely an aid to help you consolidate the material in between.

The rest of the paper goes as follows:

- Section 2: using Buddha on an example buggy program.
- Section 3: summary of the important parts of Buddha's implementation.
- Section 4: deciding on the correctness of function applications.
- Section 5: controlling Buddha's resource usage.
- Section 6: pointers to related work.
- Section 7: conclusion.

2 An Example Debugging Session

Let's consider a short debugging session. Figure 1 contains a program with a bug.[1] It is supposed to print the digits of `341` as a list, but instead it prints `[1,10,10]`.

This is the intended algorithm:

1. Compute a list of "prefixes" of the number. For example, if the number is `1976`, the prefixes are '`[1976, 197, 19, 1]`'. This is the job of `prefixes`.
2. Take numbers from the front of the above list while they are not zero. The output should be `[341, 34, 3]`. This is the job of the `leadingNonZeros` function.
3. For each number in the above list, obtain the last digit. The output should be `[1,4,3]`. This is the job of the `lastDigits` function.
4. Reverse the above list to give the digits in the desired order.

Normally, functions defined in the standard libraries are trusted by Buddha, making them invisible in the debugging session. To flesh out this example we have re-defined a number of Prelude functions within the module, hence the `hiding` clause in the `import` statement on line 3.

Debugging with buddha takes place in five steps:

1. Program transformation. To make a debugging executable, the source code of the original program (the *debuggee*) is transformed into a new Haskell program. The transformed code is compiled and linked with a declarative debugging library, resulting in a program called `debug`.
2. Program execution. The `debug` program is executed, which causes the debuggee to be executed to completion.
3. Declarative debugging. Once the execution of the debuggee is done, the user is presented with an interactive interface, which takes the form of a dialogue between the debugger and the user. The debugger chooses a function application that was evaluated during the execution of the debuggee and prints it on the screen. The user responds with a judgement about the correctness of the application.
4. Diagnosis. The debugging dialogue continues until either the user terminates the session or the debugger makes a diagnosis.
5. Retry. For a given set of input values there might be more than one cause of an erroneous program execution. To find all the causes the user must repeat the above steps until no more bugs are found.

Each step is outlined below. Boxed text simulates user interaction on an operating system terminal. Italicised text indicates user-typed input, the rest is output. The operating system prompt is indicated by ▷.

[1] Adapted from an example in the HOOD user documentation: `www.haskell.org/hood/documentation.htm`.

```
1   module Main where

    import Prelude hiding (reverse, map, (.), takeWhile, iterate)

5   main = print (digits 341)

    digits :: Int -> [Int]
    digits = reverse . lastDigits . leadingNonZeros . prefixes

10  prefixes :: Int -> [Int]
    prefixes = iterate (`div` 10)

    leadingNonZeros :: [Int] -> [Int]
    leadingNonZeros = takeWhile (/= 0)
15
    lastDigits :: [Int] -> [Int]
    lastDigits = map (10 `mod`)

    reverse :: [a] -> [a]
20  reverse xs
       = revAcc xs []
       where
       revAcc [] acc = acc
       revAcc (x:xs) acc = revAcc xs (x:acc)
25
    map :: (a -> b) -> [a] -> [b]
    map f [] = []
    map f (x:xs) = f x : map f xs

30  takeWhile :: (a -> Bool) -> [a] -> [a]
    takeWhile p [] = []
    takeWhile p (x:xs)
       | p x = x : takeWhile p xs
       | otherwise = []
35
    iterate :: (a -> a) -> a -> [a]
    iterate f x = x : iterate f (f x)

    (.) :: (b -> c) -> (a -> b) -> a -> c
40  (.) f g x = f (g x)
```

Fig. 1. A program with a bug. It is supposed to compute the digits of `341` as a list (`[3,4,1]`), but instead it produces `[1,10,10]`. Line numbers are indicated on the left hand side.

Transformation Buddha is based on program transformation. That is, to make a debugging executable, the source code of the original program (the *debuggee*) is transformed into a new Haskell program. The transformed code is compiled and linked with a declarative debugging library, resulting in a program called `debug`. When `debug` is executed it behaves exactly like the debuggee — it accepts the same command line arguments and performs the same I/O. Where the debuggee would have terminated, `debug` initiates an interactive debugging session. The details of the transformation are dealt with in Section 3.5.

To use Buddha on a program you must first ask it to transform the program source (and compile it *etcetera*). A program called `buddha` is provided for this job.[2]

Suppose that the program resides in a file called `Digits.hs`. The first thing to do is transform it, which goes as follows:

```
▷ buddha Digits.hs
buddha 1.2: initialising
buddha 1.2: transforming: Digits.hs
buddha 1.2: compiling
Chasing modules from: Main.hs
Compiling Main_B           ( ./Main_B.hs, ./Main_B.o )
Compiling Main             ( Main.hs, Main.o )
Linking ...
buddha 1.2: done
```

For each module `X` in the program, `buddha` transforms the code in that module and stores the result in a file called `X_B.hs`. To avoid cluttering the working directory with the new files, `buddha` does all of its work in a sub-directory called `Buddha`, which is created during its initialisation phase. Compilation and linking are done by the Glasgow Haskell Compiler (GHC).

Program execution The debugging executable (`debug`) is stored in the `Buddha` directory, so you must move to that directory and run it:

```
▷ cd Buddha
▷ ./debug
[1,10,10]
```

The first thing done by `debug` is to imitate the behaviour of the debuggee — in this simple example, it just prints the list `[1,10,10]`.

[2] Don't confuse Buddha with `buddha` (the font and capitalisation are significant). Buddha is the name for the whole debugging system, whilst `buddha` is the name of the executable program for performing program transformation. We probably should have called the latter `buddha-trans`, but that would require more typing!

Declarative debugging Where the debuggee would have terminated, `debug` initiates an interactive debugging session:

```
Welcome to buddha, version 1.2
A declarative debugger for Haskell
Copyright (C) 2004, Bernie Pope
http://www.cs.mu.oz.au/~bjpop/buddha

Type h for help, q to quit
```

Buddha introduces itself with the above banner message. Following this is something called a *derivation*, and then the prompt (which is underlined):

```
[1] Main 5 main
   result = <IO>

buddha:
```

A derivation records information about the evaluation of a function application or a constant. In the above case, the derivation reports that the constant `main` evaluated to an I/O action (which is abstract). Each derivation also indicates in which module the entity was defined, and on what line its definition begins. In this case `main` was defined in module `Main` on line 5. If the entity is a function application the derivation will also show representations of its arguments. The number inside square brackets is unique to the derivation, and thus gives it an identity — we'll show how this is useful in a moment.

The derivations are stored in a tree, one per node, called an Evaluation Dependence Tree (EDT).[3] It looks just like a call graph. The root always contains a derivation for `main` because all Haskell programs begin execution there. Debugging is a traversal of this tree, one derivation (or node) at a time.

The evaluation of a function application or constant will often depend on other applications and constants. If you look back at Fig. 1, you will see that the definition of `main` depends on a call to `print` and a call to `digits`. The execution of these calls at runtime forms the *children* of the derivation for `main`, and conversely, `main` is their *parent*.

An important concept is that of the *current derivation*. The current derivation is simply the one that is presently under consideration. In our example the current derivation is the one for `main`. If you ever forget what the current derivation is you can get a reminder by issuing the `refresh` command.

You can ask Buddha to show you the children of the current derivation by issuing the `kids` command:

```
buddha: kids
```

[3] The term *EDT* was coined by Nilsson and Sparud [1].

To save typing, many of Buddha's commands can be abbreviated to one letter, normally the first letter of the long version. For example, `kids` can be abbreviated to `k`. For the remainder of this example we will use the long versions of each command for clarity, but you will probably want to use the short version in practice.

Buddha responds to `kids` as follows:

```
Children of the current derivation:

[2] Main 8 digits
   arg 1  = 341
   result = [1, 10, 10]
```

Surprisingly Buddha says that the derivation for `main` has only one child, namely an application of `digits`. What happened to the derivation for `print`? Since `print` is defined in the Prelude it is trusted to be correct. To reduce the size of the EDT and hopefully save time spent debugging, Buddha does not record derivations for trusted functions.

Note that `kids` does not change the current derivation, it just allows you to look ahead one level in the EDT. At this point in the example the current derivation is still `main`.

Exercise 1. If `print` was not trusted what would its derivation look like in this case?

Buddha can help you visualise the shape of the EDT with the `draw` command:

```
buddha: draw edt
```

This generates a graphical representation of the top few levels of the EDT, using the *Dot* language [2], and saves it to a file called `buddha.dot`. You can use a tool such as *dotty*[4] to view the graph:

```
▷ dotty buddha.dot
```

Figure 2 illustrates the kind of graph produced by the `draw` command. It is worth noting that the resulting graph differs from the EDT in that nodes do not include function arguments or results (or even line numbers *etcetera*).

It is very difficult to say anything about the correctness of the derivation for `main` because its result, an I/O value, is abstract. Therefore, it is a good idea to consider function applications that do not give I/O results.[5]

The child of `main` is suspicious looking, and thus worthy of further scrutiny. Here's where the unique number of the derivation becomes useful. You can

[4] `www.research.att.com/sw/tools/graphviz`

[5] The version of Buddha described in this paper does not support debugging of functions that perform I/O, however future versions should remove this limitation.

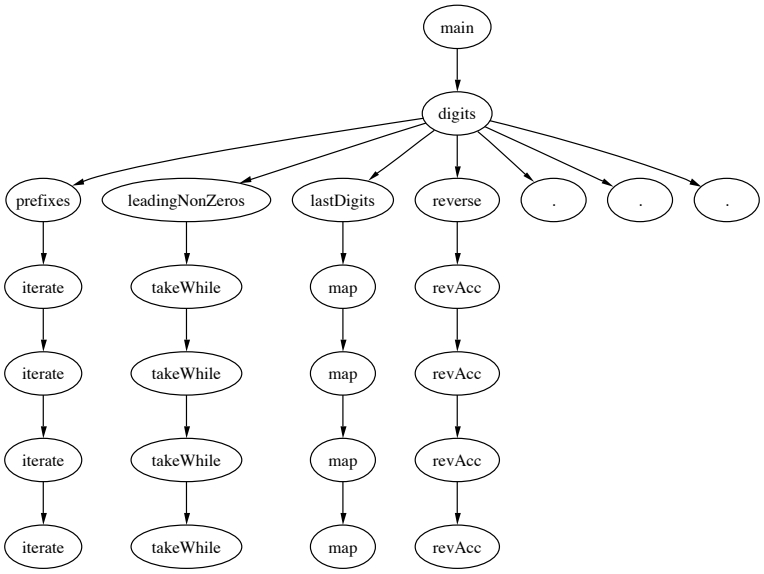


Fig. 2. A representation of the function call graph produced by the `draw` command, which shows the relationship between calls in the EDT. Note that the dot (`.`) is Haskell's name for function composition.

change the current derivation using the `jump` command. The derivation for `digits` has a number of 2, and you can jump to it as follows:

```
buddha: jump 2
```

Although not needed here, it is worth noting that jumps can be undone with the `back` command, which takes you back to the derivation that you jumped from, and allows you to resume debugging from where you left off.

After making the jump Buddha shows the new current derivation:

```
[2] Main 8 digits
   arg 1  = 341
   result = [1, 10, 10]
```

This derivation says that `digits` was applied to `341` and it returned the list `[1, 10, 10]`. This is definitely wrong — it should be `[3, 4, 1]`. When a derivation is found to be wrong you can declare this information by issuing a *judgement*. Function applications or constants that do not agree with your intended interpretation of the program should be judged as erroneous:

```
buddha: erroneous
```

Buddha's objective is to find a *buggy* derivation. A buggy derivation is one that is erroneous, and which either has no children, or all of its children are correct. Such a derivation indicates a function or constant which is improperly defined — it is the cause of at least one bug in the program.

A nice feature of the debugging algorithm is that, if the EDT is finite (and it always is for terminating programs), once you find an erroneous node you are guaranteed to find a buggy node.

Exercise 2. Provide an informal argument for the above proposition. What assumptions (if any) do you have to make, and will they be significant in practice?

We have established that the call to `digits` is erroneous, now buddha must determine if it is buggy. This requires an inspection of its children. In Fig. 2 we see that the application of `digits` has seven children. Each child corresponds to an instance of a function which is mentioned in the body of the definition of `digits`. You might be surprised to see the EDT structured this way, because none of the function applications in the body of `digits` are saturated. In a language that allows partial application of functions, the evaluation contexts in which a function is first mentioned and when it is fully applied can be quite disconnected. For example, the definition of `digits` refers to `prefixes`. However, it is not until `prefixes` is applied (dynamically) in the body of compose that it becomes saturated. Therefore it might also be reasonable to make the call to `prefixes` a child of a call to compose. The parent-child relationship between nodes in the EDT is based on the idea of logical evaluation dependence. The idea is that the correctness of the parent depends, in part, on the correctness of the child. Higher-order programming tends to muddy the waters, and we can see that there is some degree of flexibility as to the positioning of calls to functions which are passed as arguments to other functions. Buddha's approach is to base the parent-child relationship on the dependency which is evident in the static definition of functions. Statically, the definition of `digits` depends on a reference to `prefixes`. Dynamically, a call to `digits` will give rise to a partial application of `prefixes`. All the *full* applications of this particular instance of `prefixes` will become children of the application of `digits`. The same logic is applied to all the other functions which are mentioned in the body of `digits`.

There are two possible scenarios that can lead Buddha to a diagnosis. In the first scenario all the children of `digits` are correct. The conclusion in that case is that `digits` is buggy. In the second scenario one or more of the children of `digits` is incorrect. In that case the erroneous children or one or more of their descendents are buggy. Buddha could collect a *set* of buggy nodes as its diagnosis, but for simplicity it stops as soon as one such node has been identified. The idea is that you find and fix one bug at a time.

In this example Buddha will move to the first child of `digits`. If that child is correct it will move to the next child, and so on. If all the children are found to be correct the diagnosis is complete. If one of the children is erroneous, Buddha will recursively consider the EDT under that child in the search for a bug.

Exercise 3. Will the order that the children are visited affect the diagnosis of the bug? Can you think of any heuristics that might reduce the number of derivations considered in a debugging session?

As it happens, the first child visited by Buddha is a call to compose, which becomes the new current derivation:

```
[3] Main 40 .
   arg 1  = { [341, 34, 3, 0, ..? -> [1, 10, 10] }
   arg 2  = { 341 -> [341, 34, 3, 0, ..? }
   arg 3  = 341
   result = [1, 10, 10]
```

The first two arguments of compose are much more complicated than the examples we have seen before. The complexity comes from two sources. First, this is an instance of *higher order* programming — compose is a function which takes functions as its first two arguments. Printing functions is more difficult than other values, and understanding derivations that contain them can be more taxing on the brain. Second, we have encountered a partially evaluated list — the list that ends with '`..?`'. This arises because Haskell is non-strict, and the value of the tail of the list was never needed. Again, partial values can make derivations harder to comprehend.

Buddha indicates functional values using the notation:

`{` app_1`, ...,` app_n `}`

where each app_i represents a function application of the form:

```
input -> output
```

Typically this is only a partial representation of the function — it only reports the value of the function for the actual values it was applied to. This representation resembles the idea of *minimal function graphs*, which were introduced as part of a data-flow analysis of functional languages in [3]. Nonetheless, this is enough information to proceed with debugging. Thus the first argument to compose is a function that maps '`[341, 34, 3, 0, ..?`' to '`[1, 10, 10]`'.

Exercise 4. Can you think of any other way to show the value of functions that appear as arguments or results in derivations? What benefits/costs do the alternatives have in comparison to the one used by Buddha?

A consequence of the non-strict semantics of Haskell is that some expressions may never reach a normal form throughout the execution of the program. In this derivation we have a list whose prefix was evaluated four elements deep and whose tail, after those four elements, was unevaluated — in functional programming jargon, a *thunk*. Note carefully that Buddha shows all values to the extent that they were evaluated at the end of the execution of the debuggee.

When Buddha encounters a thunk it prints a question mark, although if the thunk appears at the end of a list it uses square bracket notation for the start of the list and '`..?`' to indicate the thunk at the end.

The judgement of derivations involving partial values is considered in more detail in Section 4.2. In this particular case it is not too hard to judge this derivation to be correct, since it is clearly the intention that:

```
(.) {Y -> Z} {X -> Y} X = Z
```

However, I want to take this opportunity to reveal another feature of buddha. It is quite common during declarative debugging to be faced with a very difficult derivation. In many cases it is better to look for simpler derivations that might also lead to a bug diagnosis.

The `defer` command tells Buddha to postpone judgement of the current derivation and move on to another one if possible. Where does it get the next one from? Remember that there were seven children of the derivation for `digits`. We were in the process of checking those children for correctness. In fact we've only looked at one so far, and we found it to be too complicated. Buddha treats the children as a circular queue. Deferral simply moves the current derivation to the end of the queue and makes the next derivation the current one. If we keep deferring we'll eventually get back to the start again.

```
buddha: defer
```

This leads us to two more applications of compose. Again these could be judged correct, but for the point of demonstration we'll defer them both:

```
[4] Main 40 .
   arg 1  = { [341, 34, 3] -> [1, 10, 10] }
   arg 2  = { [341, 34, 3, 0, ..? -> [341, 34, 3] }
   arg 3  = [341, 34, 3, 0, ..?
   result = [1, 10, 10]
```

```
buddha: defer
```

```
[5] Main 40 .
   arg 1  = { [10, 10, 1] -> [1, 10, 10] }
   arg 2  = { [341, 34, 3] -> [10, 10, 1] }
   arg 3  = [341, 34, 3]
   result = [1, 10, 10]
```

```
buddha: defer
```

Finally something which is easy to judge:

```
[6] Main 20 reverse
   arg 1  = [10, 10, 1]
   result = [1, 10, 10]
```

Clearly this application of `reverse` is correct:

```
buddha: correct
```

Exercise 5. The type of `reverse` is:

```
∀ a . [a] -> [a]
```

How could this polymorphic type be used to simplify derivations of `reverse`? **Hint:** `reverse` doesn't care about the value of the items in the list, just their relative order.

A correct child cannot be held responsible for an error identified in its parent. Thus there is no need to consider the subtree under the child, so Buddha moves on to the next of its siblings:

```
[8] Main 17 lastDigits
   arg 1  = [341, 34, 3]
   result = [10, 10, 1]
```

At last we find a child of `digits` which is erroneous (we expect that the last digits of `[341, 34, 3]` to be `[1, 4, 3]`):

```
buddha: erroneous
```

Exercise 6. The definition of `lastDigits` in Fig. 1 is accompanied by a type annotation. The annotation says that `lastDigits` is a function of one argument, however the argument is not mentioned in its definition (`lastDigits` is defined as a constant). How is this possible? Something interesting happens in Buddha if you remove that annotation — the derivation for `lastDigits` changes to the following:

```
[8] Main 16 lastDigits
   result = { [341, 34, 3] -> [10, 10, 1] }
```

Can you explain what has happened here? Why does the type annotation make a difference? Will it influence the diagnosis of the bug?

The discovery of this error causes the focus to shift from the children of `digits` to the sub-tree which is rooted at the derivation for `lastDigits`. The new goal is to decide whether `lastDigits` or one of its descendents is buggy.

As it happens the derivation of `lastDigits` has only one child, which is a call to `map`:

```
[9] Main 27 map
   arg 1  = { 3 -> 1, 34 -> 10, 341 -> 10 }
   arg 2  = [341, 34, 3]
   result = [10, 10, 1]
```

Exercise 7. It would appear from the code in Fig. 1 that `lastDigits` calls two functions. However Buddha only gives it one child. What is the other child, and what happened to it?

Despite the fact that `map`'s first argument is a function it should be pretty clear that this application is correct:

```
buddha: correct
```

Diagnosis This last judgement leads us to a buggy node, which Buddha indicates as follows:

```
Found a buggy node:
[8] Main 17 lastDigits
   arg 1  = [341, 34, 3]
   result = [10, 10, 1]
```

Here is where the debugging session ends. However we haven't yet achieved what we set out to do: find the bug in the program. Buddha has helped us a lot, but we have to do a little bit of thinking on our own.

Exercise 8. Why did Buddha stop here? Trace through the steps in the diagnosis that lead it to this point. Are you convinced it has found a bug? What about those deferred derivations involving compose, is it okay to simply ignore them?

The diagnosis tells us that `lastDigits` returns the wrong result when applied to `[341, 34, 3]`. We also know that every application depended on by `lastDigits` to produce this value is correct.

Exercise 9. What is the bug in the program in Fig. 1? Provide a definition of `lastDigits` that amends the problem.

Retry When we get to this point it is tempting to dust our hands, congratulate ourselves, thank Buddha and move on to something else. However our celebrations may be premature. Buddha only finds one buggy node at a time, but there may be more lurking in the same tree. A diligent bug finder will re-run the program on the same inputs that cause the previous bug, to see whether it has been resolved, or whether there is more debugging to be done. Of course it is prudent to test our programs on a large number and wide variety of inputs as well — the testing suite QuickCheck can be very helpful for this task [4].

2.1 Try it for yourself

Now it's your turn to use buddha to debug a program.

Figure 3 contains a small program for converting numbers written in base ten notation to other bases. It is an extension of the program in Figure 1. It reads two numbers from the user: the number to convert, and the desired base of the output. It prints out the number written in the new base. The intended algorithm goes as follows (all numbers are written in base ten to avoid confusion):

1. Prompt the user to enter a number and a base. Read each as a string, and convert them to integers using the library function `read` (which assumes its argument is in base ten).
2. Compute a list of "prefixes" of the number in the desired base. For example, if the number is `1976`, and the base is `10`, the prefixes are '`[1976, 197, 19, 1]`'. This is the job of `prefixes`.
3. For each number in the above list, obtain the last digit in the desired base. For example if the list is '`[1976, 197, 19, 1]`', the output should be '`[6, 7, 9, 1]`'. This is the job of `lastDigit`.
4. Convert each (numerical) digit into a character. Following the hexadecimal convention, numbers above `9` are mapped to a letter in the alphabet. For example, `10` becomes `'a'`, `11` becomes `'b'` and so on. This is the job of `toDigit`.
5. Reverse the above list to give the digits in the desired order.

Exercise 10. Your job is to test the program to find example input values that cause it to return the wrong result. For each set of inputs that give rise to the wrong behaviour, use buddha to diagnose the cause of the bug. Fix the program, and repeat the process until you are convinced that the program is bug free. To get started, try using the program to convert `1976` to base `10`. The expected output is `1976`, however program produces `:0:`.

3 Implementation

In this section we look at how Buddha is implemented. Space constraints necessitate a fair degree of generalisation, and you should treat it as a sketch, rather than a blueprint.

```
module Main where

main = do putStrLn "Enter a number"
          num <- getLine
          putStrLn "Enter base"
          base <- getLine
          putStrLn (convert (read base) (read num))

convert :: Int -> Int -> String
convert base
   = reverse              .
     map toDigit          .
     map (lastDigit base) .
     prefixes base

toDigit :: Int -> Char
toDigit i
   = index i digitChars
   where
   digitChars = ['0' .. 'z']

prefixes :: Int -> Int -> [Int]
prefixes base n
   | n <= 0 = []
   | otherwise = n : prefixes (n `div` base) base

lastDigit :: Int -> Int -> Int
lastDigit x = \y -> mod x y

index :: Int -> [a] -> a
index n list
   | n == 0 = head list
   | otherwise = index (n - 1) (tail list)
```

Fig. 3. A program for converting numbers in base 10 notation to other bases. The program has a number of bugs.

We begin with a definition of the EDT using Haskell types. Then we look at a useful abstraction called the Oracle, which plays the part of judge in the debugger. After the Oracle, we consider a simple bug diagnosis algorithm over the EDT. Then we discuss the process of turning arbitrary values into textual forms which are suitable for printing on the terminal. Of particular interest is the way that functions are handled. Lastly, we compare two transformation algorithms that introduce code into the debuggee for constructing the EDT.

3.1 The Evaluation Dependence Tree

The Evaluation Dependence Tree (EDT) provides a high-level semantics for the evaluation of a Haskell program, and is at the heart of Buddha. Nodes in the tree contain derivations which show the value of function applications and constants that were needed in the course of the program execution. Edges between the nodes indicate an evaluation dependence which can be related directly to the structure of the source code. Figure 4 illustrates the EDT for the program studied in the previous section. Figure 5 shows Haskell types for describing the EDT.

Each node in the EDT has a unique integer identity, a derivation, and zero or more children nodes. Each derivation names a function or a constant, zero or more argument values, a result value and a source location (constants have zero arguments and functions have at least one).

Perhaps the most interesting type in Fig. 5 is `Value`. It has one constructor, called `V`, which is polymorphic in the type of its argument, however that type is concealed. This means that the EDT can refer to a heterogeneous collection of types without breaking Haskell's typing rules. You might wonder how we retrieve something from its `Value` wrapper. The solution to this problem is discussed later in Section 3.4.

Explicit quantification of type variables are not part of the Haskell 98 standard, however the style used here is widely supported.

3.2 The Oracle

The debugging example in Section 2 shows how Buddha interacts with the user. A basic assumption of the Declarative Debugging algorithm is the existence of someone or something that knows how to judge derivations. It is natural to think of the judge as a person sitting behind a computer terminal. However the role of judge can be automated to a certain degree.

Buddha delegates the task of judgement to an entity called the Oracle. Currently the Oracle is a hybrid of software and human input. The diagnosis algorithm passes derivations to the Oracle which returns a judgement. The goal of the software part is to reduce the number of derivations seen by the user. It keeps a database that records pairs of derivations and judgements, which is populated by prior responses from the user. If a derivation has been seen before the corresponding judgement is retrieved directly from the database. Derivations never seen before are printed on the terminal and judged by the user, and the

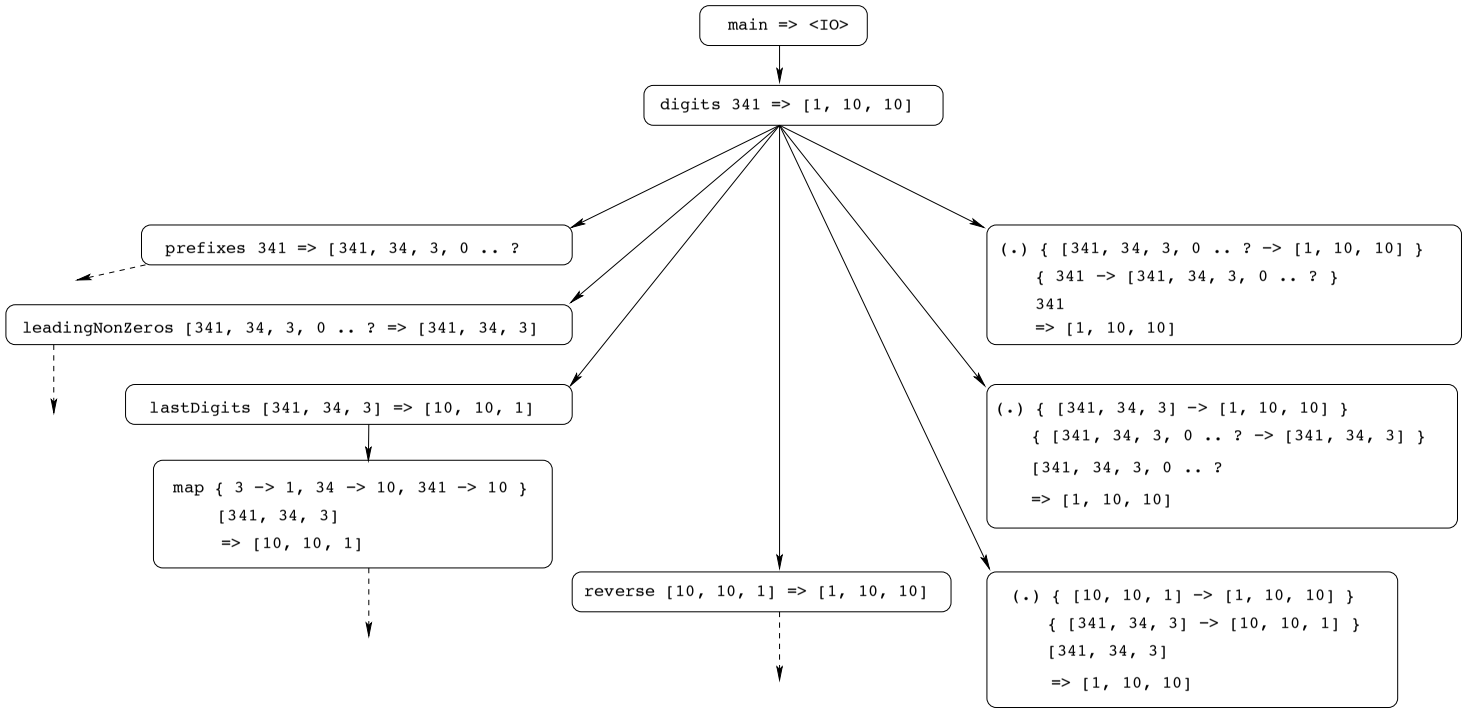


Fig. 4. An EDT for the program in Fig. 1. Dotted edges indicate subtrees which have been truncated for brevity.

```
type Identifier  = String
type LineNumber  = Int
type FileName    = String
type SrcLoc      = (FileName, LineNumber)

-- note the explicit quantifier in this type
data Value = forall a . V a

data Derivation
   = Derivation
     { name     :: Identifier
     , args     :: [Value]
     , result   :: Value
     , location :: SrcLoc }

data EDT
   = EDT
     { nodeID     :: Int
     , derivation :: Derivation
     , children   :: [EDT] }
```

Fig. 5. Haskell types for implementing the EDT.

judgement is saved in the database. Of course there is much room for improvement on the software side. An obvious extension is to allow the database to be saved between debugging sessions. An interesting area of research is in extending the sophistication of the software part of the Oracle.

Exercise 11. Can you think of other features that might be useful in the software part of the Oracle?

3.3 Diagnosis

Figure 6 shows a very simple Declarative Debugging algorithm in Haskell.

Exercise 12. Extend the diagnosis algorithm to collect a set of buggy nodes.

The method for obtaining the EDT depends on the underlying implementation of the debugger. In the above diagnosis algorithm that detail is left abstract. Later, in Section 3.5, we'll see two different ways that have been used in Buddha to produce the tree.

3.4 Observation

Buddha must be able to turn values into text if it is going to print them on the terminal. An important condition is that it must be able to print *any* value. In

```
data Judgement = Correct | Erroneous
data Diagnosis = NoBugs  | Buggy Derivation

debug :: Diagnosis -> [EDT] -> IO Diagnosis
debug diagnosis [] = return diagnosis
debug diagnosis (node:siblings)
   = do let thisDerivation = derivation node
        judgement <- askOracle thisDerivation
        case judgement of
           Correct -> debug diagnosis siblings
           Erroneous
              -> debug (Buggy thisDerivation) (children node)

askOracle :: Derivation -> IO Judgement
askOracle derivation = ... -- abstract

-- the top level of the debugger
main :: IO ()
main
   = do root <- get the root of the EDT
        diagnosis <- debug NoBugs [root]
        case diagnosis of
           NoBugs -> output: no bugs found
           Buggy derivation -> output: this derivation is buggy
```

Fig. 6. Pseudo Haskell code for a Declarative Debugging diagnosis algorithm.

short, we want a universal printer. Unfortunately GHC does not supply one, so Buddha must provide its own.[6]

There are a number of requirements that make the task quite hard:

- It must be possible to observe partial values, and reveal their unevaluated parts without forcing them any further.
- Some values can have cyclic representations. The printer must not generate infinite strings for these.
- It must be able to print functions.

To implement the printer we need reflection, however Haskell is not particularly strong in this regard. For example, there is no facility to determine whether something is a thunk. We work around the restrictions at the Haskell level by interfacing with the runtime environment via the Foreign Function Interface (FFI). That is, we extend GHC's runtime environment with reflective facilities, by the use of C code that observes the representation of values on the GHC heap. The C code constructs a Haskell data structure, of type `Graph`, that mirrors the heap representation, including the presence of thunks and cycles.

`Graph` has the following definition:

```
data Graph
   = AppNode     Word String [Graph]
   | CharNode    Char
   | IntNode     Int
   | IntegerNode Integer
   | FloatNode   Float
   | DoubleNode  Double
   | Cycle       Word
   | Thunk
   | Function
```

`AppNode` represents applications of data constructors. It has three arguments: its address on the heap, the name of the constructor and a list of the arguments to the application (for nullary constructors the list is empty). Specialised `Graph` constructors are provided for the primitive types (`CharNode` *etcetera*). Cycles in the heap representation are encoded with the `Cycle` constructor, its argument is the address of a constructor application — in other words it is a pointer back to some other part of the object. Unevaluated heap objects are mapped to `Thunk`, and all functions are mapped to `Function` (although this apparent limitation will be addressed shortly).

Exercise 13. In Section 2 we saw the partial list `[341,34,3,0,..?`. Recall that the tail of the list indicated by `..?` is a thunk. Provide a `Graph` encoding of that list. You can assume that the numbers are of type `Int`. The memory addresses of constructor applications are not important — just make them up.

[6] Hugs does have a universal printer of sorts, but it has poor support for functions, and regardless, Hugs is too slow to support Buddha.

The interface to the reflection facility is as follows:

```
reifyValue :: Value -> IO Graph
reifyValue (V x) = reify x

reify :: a -> IO Graph
reify x = ... -- call C code via the FFI
```

The function `reifyValue` retrieves an item encapsulated inside a `Value`, and passes it to `reify`, which maps it into a `Graph`. The typing rules of Haskell forbid us from exposing the type of the item. Thus it is crucial that `reify` is polymorphic in its first argument. This is easily satisfied by performing all the `Graph` construction work in C. From the C perspective all values have the same type (a heap object), so there is no limitation on the type of value that can be passed down through `reify`.

Exercise 14. What have we sacrificed by observing values on the heap via the FFI?

The result of `reify` has an `IO` type. This is necessary because multiple applications of `reify` to the same value may give back different `Graphs`. For example, the presence or absence of thunks and cycles in a value depends on *when* it is observed. Buddha ensures that values are observed in their most evaluated form by delaying all calls to `reify` until the evaluation of the debuggee has run to completion — at that point it knows their heap representations will not change.

Cycles. Cyclic values are not uncommon in Haskell. The classic example is the infinite list of ones:

```
ones = 1 : ones
```

The non-strict semantics of Haskell allow programs to operate on this list without necessarily causing non-termination. It is worth pointing out that the language definition does not require this list to be implemented as a cyclic structure, however all of the popular compilers currently do. Here is a `Graph` representation of the list, assuming that it lies at address `12`:

```
AppNode 12 ":" [IntNode 1, Cycle 12]
```

Buddha's default mode of showing cyclic values is rather naive. It prints this list as:

```
[1, <cycle>
```

This indicates the origin of the cycle, but not its destination. Buddha has another mode of printing which uses recursive equations to show cycles. You can turn this mode on with the `set` command:

```
buddha: set cycles True
```

In this mode the list is printed as:

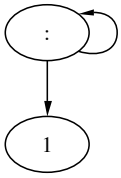
```
let _x1 = [1, _x1 in _x1
```

You might wonder why Buddha doesn't use the second mode by default. Our experience is that for larger values with cycles it can actually make the output harder to comprehend! In any case, it is often preferable to view complex structures as a diagram, so we added this facility by way of the `draw` command. In Section 2 we saw how to use `draw` to produce a diagram of the EDT, using the Dot graph language. You can also use this command for printing values that appear as arguments or results in derivations.

Suppose the current derivation contains `ones`. The following command renders the result of `ones` and saves the output in the file `buddha.dot`:

```
buddha: draw result
```

As before, you can view the diagram with the *dotty* program, the output of which looks like this:



If you want to draw an argument at position 3, for instance, you can issue the command:

```
buddha: draw arg 3
```

Functions. Functions are more troublesome beasts when it comes to printing. The main problem is that, unlike data structures, GHC's internal representation of functions is very hard to reconcile with the source code.

The solution goes as follows. Each function that could possibly be printed is assigned a unique integer. An entry is added to a global table whenever such a function is applied, recording the number of the function, its argument and its result. Table entries have this type:

```
type FunApplication = (Int, Value, Value)
```

Exercise 15. Why is it possible to record only one argument? How do you think multi-argument functions are handled?

Functions and their unique number are "wrapped up" inside a new type called `F`:

```
data F a b = F Int (a->b)
```

This ensures that the function and the number are always together. Wrappers are introduced as part of the program transformation, however the unique numbers are created dynamically.

Wrapped functions are passed to `reify` in the usual way, resulting in a `Graph` value such as:

```
AppNode 28 "F" [IntNode 4, Function]
```

Obviously the printer does not treat this like an ordinary data structure. The presence of the `F` constructor indicates that the `Graph` represents a function. In the above example, the function is identified by the number `4`. The printer scans the global table and collects all records that correspond to this function.

For example, suppose the global table contains these records:

```
[  (1, V True, V False)
,  (2, V 12, V 3)
,  (1, V False, V True)
,  (4, V "ren", V 3)
,  (3, V ('a', 'b'), V 'a')
,  (4, V "stimpy", V 6)
,  (2, V 16, V 4)
]
```

The entries pertaining to function number `4` are set in grey boxes. The arguments and results in the records are `Value`s that must be converted to `Graph`s before they can be printed. In this example function number `4` would be printed in the following way:

```
{ "ren" -> 3, "stimpy" -> 6 }
```

The part of the transformation that deals with function wrapping is quite simple. It consists of three parts. First, wrappers must be introduced where functional values are created (lambda abstractions and partial applications). Second, when a wrapped function is applied it must be unwrapped and the application must be recorded in the global table. Third, function types must be changed to reflect the wrapping.

Let's consider a small example using code from Figure 1. In the body of `lastDigits`, `map` is applied to the expression `(10 `mod`)`. That expression is a function which must be wrapped up. We introduce a family of wrappers, $\texttt{fun}_n$, where n indicates the arity of the function to be wrapped. The first three of those wrappers have the following types:

```
fun1 :: (a -> b) -> F a b
fun2 :: (a -> b -> c) -> F a (F b c)
fun3 :: (a -> b -> c -> d) -> F a (F b (F c d))
```

Exercise 16. Why do we have multiple wrapping functions? Would one suffice?

In our example, the expression `(10 ‘mod‘)` has an arity of one, so the definition of `lastDigits` is transformed like so:

```
lastDigits :: Int -> Int
lastDigits = map (fun1 (10 ‘mod‘))
```

The definition of `map` must also be transformed:

```
map :: F a b -> [a] -> [b]
map f [] = []
map f (x:xs) = apply f x : map f xs
```

The first parameter, called `f`, will be bound to a wrapped function when `map` is applied. This necessitates two changes. First, the type is changed to indicate the wrapping: `(a->b)` becomes `F a b`. Second, where `f` is applied to `x`, it must be unwrapped, and the application must be recorded. The job of unwrapping and recording is done by `apply`, which has this definition:

```
apply :: F a b -> a -> b
apply (F unique f) x
   = let result = f x in updateTable unique x result

updateTable :: Int -> a -> b -> b
```

Updates to the global table are made by `updateTable`, by means of (gasp) impure side-effecting code!

A curious user can view the contents of the global table using the `dump` command as follows:

```
buddha: dump funs
```

Use this command with caution: the table can get quite big.

Exercise 17. The transformation of `map` shows that some function type annotations must be changed to accommodate wrapping. Where else in the program will types have to be changed?

Exercise 18. We have multiple wrapping functions, one for each level of function arity, yet we only have one `apply`. How is this possible?

3.5 Transformation

The purpose of the transformation is to introduce code into the debuggee for constructing the EDT. In the development of Buddha we have experimented with two different styles of transformation. The first style builds the tree in a purely functional way, whilst the second style builds the tree in a table by means of impure side-effects. The current version of Buddha employs the second style for a combination of efficiency concerns and programming convenience.

To help with the presentation we'll consider both styles of transformation applied to the following code for naive reverse:

```
rev :: [a] -> [a]
rev xs
   = case xs of
        []   -> []
        y:ys -> append (rev ys) [y]

append :: [a] -> [a] -> [a]
append xs ys
   = case xs of
        []   -> ys
        z:zs -> z : append zs ys
```

The purely functional style. In the first style each function definition is transformed to return a pair containing its original value and an EDT node. Applications in the body of the function return nodes which make up its children.

Figure 7 shows the result of transforming `rev` using this style.

```
rev xs
   = case xs of
        []   -> let result   = []
                    children = []
                    node = edt name args result children
                in (result, node)
        y:ys -> let (v1, t1) = rev ys
                    (v2, t2) = append v1 [y]
                    result   = v2
                    children = [t1, t2]
                    node = edt name args result children
                in (result, node)
   where
   name = "rev"
   args = [V xs]
```

Fig. 7. Purely functional style transformation of `rev`.

Note the decomposition of the nested function application from the second branch of the case statement:

before		after
`append (rev ys) [y]`	$\longrightarrow$	`(v1, t1) = rev ys` `(v2, t2) = append v1 [y]`

The variables `v1` and `v2` are bound to the original value of the intermediate applications, and `t1` and `t2` are bound to EDT nodes.

Figure 8 shows the transformation of `append`, which follows the same pattern as that for `rev`.

```
append xs ys
   = case xs of
        []   -> let result   = ys
                    children = []
                    node = edt name args result children
                in (result, node)
        z:zs -> let (v1, t1) = append zs ys
                    result   = z : v1
                    children = [t1]
                    node = edt name args result children
                in (result, node)
   where
   name = "append"
   args = [V xs, V ys]
```

Fig. 8. Purely functional style transformation of `append`.

Upon first inspection the effect of the transformation might appear somewhat daunting. However, it is actually only doing two things: constructing a new EDT node for the application of the function and, in the process of doing that, collecting its children nodes from the applications that appear in the body. To simplify things we make use of a helper function called `edt` which constructs a new EDT node from the function's name, arguments, result and children (we've skipped the source location for simplicity). The apparent complexity is largely due to the fact that the computation of the original value and the EDT node are interwoven.

This style of transformation is quite typical in the literature on Declarative Debugging. Variants have been proposed by [5], [6], and [7], amongst others.

The main attraction of this style is that it is purely functional. However, support for higher-order functions is somewhat challenging. The naive version of the transformation assumes that all function applications result in a value *and* an EDT node, although this is only true for saturated applications. An initial solution to this problem was proposed in [7] and improved upon in [8].

Exercise 19. What are the types of the transformed versions of `rev` and `append`?

Exercise 20. Apply this style of transformation to the definition of `map` from Fig. 1? How will you deal with the application of the higher-order argument `f` in the body of the second equation?

The table-based approach. The second style is based on the concept of application numbering. It introduces code which, at runtime, causes each saturated function application to be uniquely numbered. Also, for each application, the function name, its arguments and its result are stored in a global table, which is indexed by the number of the application. The global table is thus a collection of all the derivations in the EDT. Table entries also record the parent number

of the derivation. Parent numbers are passed down through the call graph to their children by extending each function with a new integer argument. When execution of the debuggee is complete a single pass is made over the table to construct the EDT.

Figure 9 shows the transformation of `rev` in this style.

```
rev :: Int -> [a] -> [a]
rev parent xs
   = addNode parent "rev" [V xs]
        (\n -> case xs of
                  []   -> []
                  y:ys -> append n (rev n ys) [y])
```

Fig. 9. Table-based transformation of `rev`.

Notice the new type of `rev`, in particular its first argument is now an integer which corresponds to the unique number of its parent. The task of assigning new unique numbers for each application of `rev` is performed by the helper function `addNode`, whose type is as follows:

```
addNode :: Int -> String -> [Value] -> (Int -> a) -> a
```

Each call to `addNode` does four things:

1. it generates a new unique number for the current application;
2. it threads that number into the body of the transformed function, by way of its fourth argument;
3. it records the entry for this application in the global table;
4. it returns the value of the application as its result.

Writes to the global table are achieved via impure side effects.

The number for each application of `rev` is passed to the calls in its body through the variable `n`, which is introduced by lambda abstraction around the original function body. The idea is that the new function body — the lambda abstraction — is applied to each new number for `rev` inside `addNode`. Hence the type of `addNode`'s third argument is `(Int -> a)`, where `n` takes the value of the `Int` and `a` matches with the type of the original function's result.

Figure 10 shows the transformation of `append`, which follows the same pattern as that for `rev`.

Exercise 21. Consider the transformation of some constant, such as:

```
goodDay = (12, January, 1976)
```

The discussion of the transformation above suggests that we would give `goodDay` an integer argument to represent its parent number. Can you foresee any problem(s) with this in regards to how often `goodDay` is evaluated? How might you

```
append :: Int -> [a] -> [a] -> [a]
append parent xs ys
   = addNode parent "append" [V xs, V ys]
        (\n -> case xs of
                  []   -> ys
                  z:zs -> z : append n zs ys)
```

Fig. 10. Table-based transformation of `append`.

fix it? **Hint:** Ideally each constant should have at most one entry in the global table. However, you will still want to record each time another entity refers to a constant.

You might have noticed some similarity between the use of a global table in this section to record derivations and the use of a global table to record applications of higher-order functions in Section 3.4. Indeed they share some underlying machinery for performing impure updates of the tables. Just like the function table, you can also see the contents of the derivation table, using the `dump` command:

```
buddha: dump calls
```

Again, for long running programs the table can get quite large, so be careful with this command, it can produce a *lot* of output.

The main advantage of the impure table transformation over the purely functional is that it tends to decouple the generation of the EDT and and evaluation of the debuggee. This means that we can stop the debuggee at any point and still have access to the tree produced up to that point. Declarative diagnosis can often be applied to a partial tree. In the purely functional style this is possible but more complicated. The separation makes the handling of errors easier. It doesn't matter if the debuggee crashes with an exception, the table is still easily accessible to the debugging code. In the purely functional style the evaluation of the debuggee and the production of the EDT are interwoven, which makes it harder to access the EDT if the debuggee crashes.

The main problem with the second transformation style is that it relies on impure side effects to generate new unique numbers and make updates to the global table. It is possible that with some clever programming the side effects could be avoided, but we speculate that this will be at a significant cost to performance. Without more investigation it is difficult to be definitive on that point. The problem with the use of impure side effects is that they do not sit well with the semantics of Haskell. As the old saying goes:

If you lie to the compiler, it will get its revenge.

This is certainly true with GHC, which is a highly optimising compiler. Covert uses of impure facilities tend to interact very badly with the optimisations that

it performs, and one must program very carefully around them. We could turn the optimisations off, however that would come at the cost of efficiency in the debugging executable, which is something we want to avoid.

4 Judgement

The Oracle is assumed to have an internal set of beliefs about the intended meaning of each function and constant in the program. We call this the *Intended Interpretation* of the program. Derivations in the EDT reveal the actual behaviour of the program. Judgement is the process of comparing the actual behaviour of functions and constants with their intended behaviour. Differences between the two are used to guide the search for buggy nodes.

4.1 Partial Functions

In the simple debugging algorithm described in Section 3.3 judgement is a binary decision: a derivation is either correct or erroneous. Our experience is that the binary system does not always suit the intuition of the user and we extend it with two more values: *unknown* and *inadmissible*.

Some derivations are just too complicated to be judged. Perhaps they contain very large values, or lots of higher-order code, or lots of thunks. The best choice is to defer judgement of these derivations. However, deferral might only postpone the inevitable need for a judgement. If deferral does not lead to another path to a bug the Oracle can judge a difficult derivation as unknown. If a buggy node is found which has one or more unknown children, Buddha will report those children in its final diagnosis, and remind the user that their correctness was unknown. The idea is that the true bug may be either due to the buggy node or one or more of its unknown children, or perhaps one of their descendents.

For some tasks it is either convenient or necessary to write partial functions: functions which are only defined on a subset of their domain. Most functional programmers have at some point in their life experienced program crash because they tried to take the head of an empty list. An important question is how to judge derivations where the function is actually applied to arguments for which there is no intended result?

Consider the `merge` function which takes two sorted lists as arguments and returns a sorted list as output containing all the values from the input lists. Due to a bug in some other part of the program it might happen that `merge` is given an unsorted list as one of its arguments. In this case Buddha might ask the Oracle to judge the following derivation:

```
[32] Main 12 merge
   arg 1  = [3,1,2]
   arg 2  = [5,6]
   result = [3,1,2,5,6]
```

Let's assume for the moment that our definition of `merge` is correct for all sorted arguments. Is this derivation correct or erroneous? If we judge it to be erroneous then Buddha will eventually diagnose `merge` as buggy, assuming that `merge` only calls itself. This is a bad diagnosis because the bug is not due to `merge`, rather it is due to which ever function provided the invalid argument. To find the right buggy node we could judge the derivation to be correct. However, it feels counter-intuitive to say that a derivation is correct when it should never have happened in the first place. In such circumstances it is more natural to say that the derivation is inadmissible. It has exactly the same effect as judging the derivation to be correct, yet it is a more accurate expression of the user's beliefs.

4.2 Partial Values

Non-strict evaluation means that not all computations will necessarily reach normal forms in the execution of a program. Consider the following definition of boolean conjunction:

```
(&&) :: Bool -> Bool -> Bool
(&&) False _ = False
(&&) True  x = x
```

In the application '`(&&) False exp`', the value of `exp` is not needed, so it will remain as a thunk. It would not be prudent for the debugger to force the evaluation of `exp` to find its value, first because the computation might be expensive, and second, in the worst case is might trigger divergence (`exp` might be non-terminating or it might raise an exception).

Thunks that remain at the end of the program execution cannot be the cause of any observed bugs, so it ought to be possible to debug without knowing their value. Therefore, thunks are treated by Buddha as unknown entities, and it prints a question mark whenever it encounters one. In the above example, this would lead to the following derivation:

```
[19] Main 74 &&
   arg 1  = False
   arg 2  = ?
   result = False
```

How do you judge derivations that have question marks in them? One approach is to assume that the Oracle knows everything about the program, and by implication it has an intended meaning for functions with all possible combinations of partial arguments and results. This is convenient for us as designers of the debugger, but it is not very helpful for the end user.

Generally it is easier for the Oracle (and thus the poor human user) to model their intended interpretation on complete values. It tends to simplify the task of saying what a function should do if you ignore the effects of computation (*i.e.* strictness and non-termination) and think in terms of abstract mathematical functions. In this sense you can say the intended interpretation of `&&` is merely the relation:

```
(True, True, True),   (True, False, False)
(False, True, False), (False, False, False)
```

A derivation with partial arguments is correct if and only if all possible instantiations of those partial arguments agree with the intended interpretation. The above derivation for `&&` is correct because, according to the relation above, it doesn't matter whether we replace the question mark with `True` or `False`, the result is always `False`.

Of course `&&` is a very simple example, and many interesting functions are defined over large or even infinite domains, where it is not feasible to enumerate all mappings of the function. In such cases the Oracle might have to do some reasoning before it can make a judgement.

Exercise 22. Judge these derivations (the module names, line numbers and derivation numbers are irrelevant).

```
[6] Main 55 length
   arg 1  = ?
   result = 0

[99] Main 55 length
   arg 1  = [?]
   result = 1
```

It is also possible to encounter partial values in the result of a derivation. Consider the swap function:

```
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)
```

It might be the case that the program only needs the first component of the output tuple, while the second component remains a thunk, such as:

```
[19] Boo 7 swap
   arg 1  = (?, 12)
   result = (12, ?)
```

When the output contains a partial value the derivation is correct if and only if for all possible instances of the arguments there *exists* a instance of the result which agrees with the intended interpretation.

Exercise 23. Judge these derivations:

```
[6] Main 22 reverse
   arg 1  = [1,2,3]
   result = ?

[99] Main 22 reverse
   arg 1  = [1,2,?]
   result = [1,?,?]
```

Exercise 24. In the example debugging session in Section 2 we deferred this derivation because it was too hard:

```
[3] Main 40 .
   arg 1  = { [341, 34, 3, 0, ..? -> [1, 10, 10] }
   arg 2  = { 341 -> [341, 34, 3, 0, ..? }
   arg 3  = 341
   result = [1, 10, 10]
```

Can you judge it now? Can you justify your judgement?

5 Resource Usage

For all but the briefest runs of a program it is totally infeasible to consider every function application. In long running programs you will be swamped with derivations, and most likely all of your available heap space will be consumed. The key to reducing the number of derivations (and memory usage at the same time) is to declare what you do and don't want to see. Hopefully you will have some feeling for where the bug is in your program, and also which parts are unlikely to be involved. Unit testing can be quite helpful in this regard. Instead of just testing the program as a whole, one may test smaller pieces of code separately. A failed unit test gives a much narrower scope for the bug, and allows any code not touched by the test to be trusted. One may even debug specialised versions of the program that only execute the top call in a failed test case, thus avoiding the extra cost of executing large amounts of trusted code. This idea is discussed in the context of tracing in [9]. Unit tests are supported in Haskell by QuickCheck [4], and also HUnit[7].

The EDT maintains references to the arguments and results of each function application. This means that such values cannot be garbage collected as they might have been in the evaluation of the original program. By not creating nodes for every function application we allow some values to be garbage collected. The fewer the nodes the less memory is required and generally less questions will be asked.

So how do you reduce the number of nodes in this tree? Good question. Basically you prune it statically, by telling Buddha which functions you want to create nodes for and which ones you don't. For each module in the program you can provide an options file that tells Buddha what to do for each function in the module. If the module's name is `X`, the options file is called called `X.opt`, and it must be stored in the `Buddha` directory. The syntax of the options file is very simple. It has a number of lines and each line specifies what kind of transformation you want for a given function.

Here's what you might write for some program:

```
_ ; Trust
prefixes ; Suspect
convert ; Suspect
```

[7] `http://hunit.sourceforge.net`

Each line contains the name of the function, a semi-colon and then an option as to what kind of EDT node you want. The underscore matches with anything (just like in Haskell patterns), so the default is specified on the first line to be `Trust`. Any function which is not mentioned specifically gets the default option. If there is no such default option in the whole file, then the *default* default is `Suspect`. In the case when no option file is present for a module, every function in the module is transformed with the `Suspect` option.

So what do the options mean?

- `Suspect`: Create a full node for each application of this function. Such a node will record the name of the function, its arguments and result, its source location and will have links to all of its children. However, the children will be transformed with their own options which will not necessarily be `Suspect`. This option tends to make Buddha use a lot of memory, especially for recursive functions, so please use it sparingly.
- `Trust`: Don't create a node for applications of this function, but collect any children that this function has.

An effective way for reducing the size of the EDT at any one time is to make use of re-evaluation. The idea is that only the top few levels of the EDT are produced by the initial execution of the debuggee. Debugging commences with only a partial tree. Eventually the traversal of the EDT might come to a derivation whose children were not created in the initial execution of the debuggee. The debugger can regenerate the children nodes by forcing the re-evaluation of the function application at the parent. In the first execution of the debuggee, the children are pruned from the EDT. The purpose of re-evaluating the call at the parent is to cause the previously pruned children nodes to be re-generated. This allows the EDT to be constructed in a piecemeal fashion, at the cost of some extra computation time during the debugging session — a classic space/time tradeoff. Re-evaluation was implemented in a previous version of Buddha, for the purely functional style of transformation, see [10]. However, it has yet to be incorporated into the latest transformation style.

6 Further Reading

Phil Wadler once wrote:

> *Constructing debuggers and profilers for lazy languages is recognised as difficult. Fortunately, there have been great strides in profiler research, and most implementations of Haskell are now accompanied by usable time and space profiling tools. But the slow rate of progress on debuggers for lazy functional languages makes us researchers look, well lazy.* [11]

While it is true that debugging technology for lazy functional languages hasn't set the world on fire, there has nonetheless been a fair amount of improvement since Wadler made that remark.

Perhaps the most significant tool for debugging Haskell is Hat [12]. Like Buddha, Hat is based on program transformation. However Hat-transformed programs produce a very detailed program trace, called a Redex Trail, which explains the reduction history of each expression in the program. The trace is written to a file rather than main memory. This means that the memory usage of a transformed program stays proportional to the usage of the original program, although at the expense of very large trace files. The best feature of the trace is that it can be viewed in many different ways. Hat provides a handful of browsing tools, that are useful for diagnosing different kinds of bugs. An very helpful tutorial on the use of Hat in combination with with QuickCheck was presented at the Advanced Functional Programming School in 2002 [9]. Much of the technology in Hat is based on earlier work by Sparud [6], who also proposed a program transformation for Declarative Debugging, quite similar to Buddha.

A popular debugging tool for Haskell is Hood [13]. Hood provides a library function called `observe`, which can be used to log the evaluation of expressions in the program. In essence it provides a sophisticated form of the so-called `printf` style of diagnostic debugging for Haskell programs. A particularly nice aspect of Hood is that it can observe functional values and partial data structures, without adversely changing the semantics of the underlying program. A negative aspect of Hood is that the user must modify the source code of their program to insert the observations. Such modifications can be a problem for program maintenance.

Another declarative debugger, called Freya, was implemented by Nilsson, for a large subset of Haskell [14]. The main difference from Buddha is that Freya makes use of a modified runtime environment to construct the EDT, rather than by program transformation. An advantage of the Freya approach is that the debugger has intimate knowledge and influence over the mechanics of program evaluation. The most obvious result of which is a sophisticated re-evaluation scheme which allows the EDT to be created in a piecemeal fashion [15]. The downside of the Freya approach is that it requires a whole new compiler and runtime environment, which is difficult to maintain and port.

Buddha is based predominantly on the pioneering work of Naish and Barbour [16–18, 5, 19]. Their ideas have also influenced the development of a declarative debugger for logic/functional languages such as Toy and Curry [7].

Optimistic evaluation of Haskell can reduce the gap between the structure of the code and the evaluation order by reducing most function arguments eagerly [20]. This is the basis for a step-based debugger called HsDebug [21], built on top of an experimental version of the Glasgow Haskell Compiler. However, to preserve non-strict semantics the evaluator must sometimes suspend one computation path and jump to another. This irregular flow of control is likely to be hard to follow in the step-based debugging style.

Of course no paper on Declarative Debugging would be complete without a reference to the seminal work of Shapiro, who's highly influential thesis introduced Algorithmic Debugging to the Prolog language [22], from which the ideas of Declarative Debugging have emerged.

7 Conclusion

Buddha is by no means a finished product. At the time of writing the latest version is 1.2, which supports all of the Haskell 98 standard. In terms of program coverage there are a number of small improvements that need to be made, including support for the FFI, hierarchical modules, and programs spread over more than one directory. It seems that a fair proportion of people who have expressed an interest in using Buddha cannot do so because their code makes use of non-standard features, like multi-parameter type classes and functional dependencies. The next few development cycles of Buddha will look at supporting the most commonly used language extensions.

The biggest limitation of Buddha is the size of the EDT. If we are going to debug long running programs we need to manage the growth of the EDT more effectively. On the one hand, main memory sizes are ever growing, and are becoming increasingly cheaper. At first this looks good for Buddha because we can fit more EDT nodes into memory. On the other hand, the rapid advances in hardware have also given us faster CPUs, which can fill the memory more quickly. Even if we can fit enormous EDTs into main memory we will have to come up with smarter debugging algorithms, lest we become swamped in an insurmountable number of derivations.

Another important area of research is Buddha's interface with the human user. For example, at the moment, it is nigh impossible to judge a derivation that contains very large values. There are of course many avenues to explore in this regard. In particular, the development of "smarter" software oracles can greatly reduce the cognitive load on the user. Another interesting idea is to allow Buddha to be specialised to particular problem domains. For example, if you are writing digital circuit simulators, you might like custom printing routines that show the values in derivations in a fashion that is closer to your mental picture of an electronic component.

Buddha also has a place in education. Haskell is used extensively in Computer Science courses all over the world. Novice programmers often grapple with seemingly difficult concepts like recursion and higher-order functions, especially when they are explained statically in text or on the white board. Buddha can be quite helpful here, especially as an exploratory device. A few minutes spent browsing the EDT can be very enlightening, even if you are not debugging the program.

Lastly, while Declarative Debugging is a useful technique for locating logical errors in programs, it is not the final word on debugging. For starters, we need good testing tools, such as QuickCheck, to help us identify faulty behaviour in programs. Even better would be to automate the process of finding *small* input values that cause the program to go wrong. That way the debugging sessions are more likely to be within a size that is manageable by a human user. For bugs that relate to the operational behaviour of the program, like I/O, or resource usage, we will need to look elsewhere for help.

The Buddha web page provides the latest stable release of the source code, and online versions of the user documentation: `www.cs.mu.oz.au/~bjpop/buddha`.

References

1. Nilsson, H., Spaurd, J.: The evaluation dependence tree as a basis for lazy functional debugging. Automated Software Engineering **4** (1997) 121–150
2. Gansner, E., Koutsofios, E., North, S.: Drawing graphs with dot. `www.research.att.com/sw/tools/graphviz/dotguide.pdf` (2002)
3. Jones, N., Mycroft, A.: Dataflow analysis of applicative programs using minimal function graphs. In: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, Florida, ACM Press (1986) 296–306
4. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming, ACM Press (2000) 268–279
5. Naish, L., Barbour, T.: Towards a portable lazy functional declarative debugger. Australian Computer Science Communications **18** (1996) 401–408
6. Sparud, J.: Tracing and Debugging Lazy Functional Computations. PhD thesis, Chalmers University of Technology, Sweden (1999)
7. Caballero, R., Rodri'guez-Artalejo, M.: A declarative debugging system for lazy functional logic programs. In Hanus, M., ed.: Electronic Notes in Theoretical Computer Science. Volume 64., Elsevier Science Publishers (2002)
8. Pope, B., Naish, L.: A program transformation for debugging Haskell-98. Australian Computer Science Communications **25** (2003) 227–236 ISBN:0-909925-94-1.
9. Claessen, K., Runciman, C., Chitil, O., Hughes, J., Wallace, M.: Testing and Tracing Lazy Functional Programs using QuickCheck and Hat. In: 4th Summer School in Advanced Functional Programming. Number 2638 in LNCS, Oxford (2003) 59–99
10. Pope, B., Naish, L.: Practical aspects of declarative debugging in Haskell-98. In: Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. (2003) 230–240 ISBN:1-58113-705-2.
11. Wadler, P.: Why no one uses functional languages. SIGPLAN Notices **33** (1998) 23–27
12. Wallace, M., Chitil, O., Brehm, T., Runciman, C.: Multiple-view tracing for Haskell: a new hat. In: Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop. (2001) 151–170
13. Gill, A.: Debugging Haskell by observing intermediate data structures. Technical report, University of Nottingham (2000) In Proceedings of the 4th Haskell Workshop, 2000.
14. Nilsson, H.: Declarative Debugging for Lazy Functional Languages. PhD thesis, Department of Computer and Information Science Linköpings Universitet, S-581 83, Linköping, Sweden (1998)
15. Nilsson, H.: How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. Journal of Functional Programming **11** (2001) 629–671
16. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming **1997** (1997)
17. Naish, L., Barbour, T.: A declarative debugger for a logical-functional language. In Forsyth, G., Ali, M., eds.: Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers. Volume 2., Melbourne, DSTO General Document 51 (1995) 91–99
18. Naish, L.: Declarative debugging of lazy functional programs. Australian Computer Science Communications **15** (1993) 287–294

19. Naish, L.: A three-valued declarative debugging scheme. Australian Computer Science Communications **22** (2000) 166–173
20. Ennals, R., Peyton Jones, S.: Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In: Proceedings of the Eighth ACM SIGPLAN Conference on Functional Programming. (2003) 287–298
21. Ennals, R., Peyton Jones, S.: HsDebug: Debugging lazy programs by not being lazy. In Jeuring, J., ed.: ACM SIGPLAN 2003 Haskell Workshop, ACM Press (2003) 84–87
22. Shapiro, E.: Algorithmic Program Debugging. The MIT Press (1982)