Declarative Debugging with Buddha

5th International Summer School on Advanced Functional Programming Tartu, Estonia

Bernie Pope

bjpop@cs.mu.oz.au

Department of Computer Science & Software Engineering

The University of Melbourne,

Victoria, Australia

Prelude

The buddha project is a collaboration between Lee Naish and myself.

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

Prelude

The buddha project is a collaboration between Lee Naish and myself.

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

There are three parts to this presentation:

Prelude

The buddha project is a collaboration between Lee Naish and myself.

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

There are three parts to this presentation:

- the big picture and user's perspective

Prelude

The buddha project is a collaboration between Lee Naish and myself.

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

There are three parts to this presentation:

- the big picture and user's perspective
- implementation

Prelude

The buddha project is a collaboration between Lee Naish and myself.

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

There are three parts to this presentation:

- the big picture and user's perspective
- implementation
- practicalities

The big picture and user's perspective.

The problem is that it can become very hard to find the small residue of bugs that get past the type checker, because the usual debugging tools (shoving print statements into the program, watching the sequence of events in an animated window, etc.) run into trouble with the restrictions imposed by pure functional languages.

JOHN O'DONNELL

Why not GDB?

- non-strict evaluation

Why not GDB?

- non-strict evaluation
- higher-order functions

Why not GDB?

- non-strict evaluation
- higher-order functions
- doesn’t take advantage of declarative semantics

Why not GDB?

- non-strict evaluation
- higher-order functions
- doesn't take advantage of declarative semantics

That is not to say that low-level step based debuggers are no use to Haskell programmers. Indeed such a debugger has been built on top of the speculative evaluation branch of GHC.

Why not GDB?

- non-strict evaluation
- higher-order functions
- doesn't take advantage of declarative semantics

That is not to say that low-level step based debuggers are no use to Haskell programmers. Indeed such a debugger has been built on top of the speculative evaluation branch of GHC.

However, it seems a pity to promote declarative program development yet fall back to procedural tools for program maintenance.

Declarative Debugging - abstraction

One benefit of declarative languages is that they allow for complex evaluation strategies without undue complexity in the source code.

Declarative Debugging - abstraction

One benefit of declarative languages is that they allow for complex evaluation strategies without undue complexity in the source code.

- backtracking search in Prolog

Declarative Debugging - abstraction

One benefit of declarative languages is that they allow for complex evaluation strategies without undue complexity in the source code.

- backtracking search in Prolog
- lazy evaluation in Haskell

Declarative Debugging - abstraction

One benefit of declarative languages is that they allow for complex evaluation strategies without undue complexity in the source code.

- backtracking search in Prolog
- lazy evaluation in Haskell
- speculative evaluation in Haskell

Declarative Debugging - abstraction

One benefit of declarative languages is that they allow for complex evaluation strategies without undue complexity in the source code.

- backtracking search in Prolog
- lazy evaluation in Haskell
- speculative evaluation in Haskell

A declarative debugger abstracts away the details of evaluation order and constructs a semantics for program evaluation which reflects the logical dependencies in the source code.

Declarative Debugging - EDT

The constructed semantics is a *call graph* annotated with argument and result values. We call this an Evaluation Dependence Tree (EDT).

Declarative Debugging - EDT

The constructed semantics is a *call graph* annotated with argument and result values. We call this an Evaluation Dependence Tree (EDT).

Debugging is an interactive traversal of the EDT in search of function applications (or constants) that produce wrong results but do not *depend* on any other wrong entities.

An example buggy program

```
main = print (digits 341)

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

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

leadingNonZeros :: [Int] -> [Int]
leadingNonZeros = takeWhile (/= 0)

lastDigits :: [Int] -> [Int]
lastDigits = map (10 `mod`)
```

An example buggy program

Expected output: `[3,4,1]`

Actual output: `[1,10,10]`

An example buggy program

Expected output: `[3,4,1]`

Actual output: `[1,10,10]`

Basic assumption: we start with a set of inputs that causes the program to produce an externally observable bug.

An example buggy program

Expected output: `[3,4,1]`

Actual output: `[1,10,10]`

Basic assumption: we start with a set of inputs that causes the program to produce an externally observable bug.

Although . . . in situations where there is no observable bug buddha is also useful as a browsing facility.

Using buddha - program transformation

```
$ 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
```

This produces an executable called `debug`. Execution of this program constitutes a computation “equivalent” to the original program, *plus* a declarative debugging session.

Using buddha - execution

Execution of `debug` does two things.

1) The "original program" is run to completion:

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

2) Where the original would have terminated, debugging begins:

```
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
```

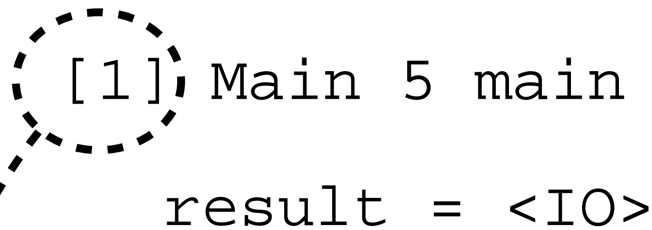
Using buddha - derivations

Each node in the EDT stores information about a function application or constant. We call this information a *derivation*.

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

Using buddha - derivations

Each node in the EDT stores information about a function application or constant. We call this information a *derivation*.



Using buddha - derivations

Each node in the EDT stores information about a function application or constant. We call this information a *derivation*.

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

module name

Using buddha - derivations

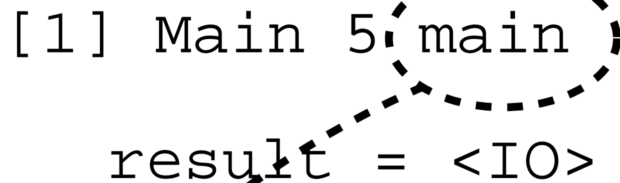
Each node in the EDT stores information about a function application or constant. We call this information a *derivation*.

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

line number

Using buddha - derivations

Each node in the EDT stores information about a function application or constant. We call this information a *derivation*.



function name

Using buddha - derivations

Each node in the EDT stores information about a function application or constant. We call this information a *derivation*.

```
[1] Main 5 main
```

value of the result

Using buddha - starting

- debugging always starts at `main` which is the root of the EDT

Using buddha - starting

- debugging always starts at `main` which is the root of the EDT
- functions have argument values (`main` is a constant)

Using buddha - starting

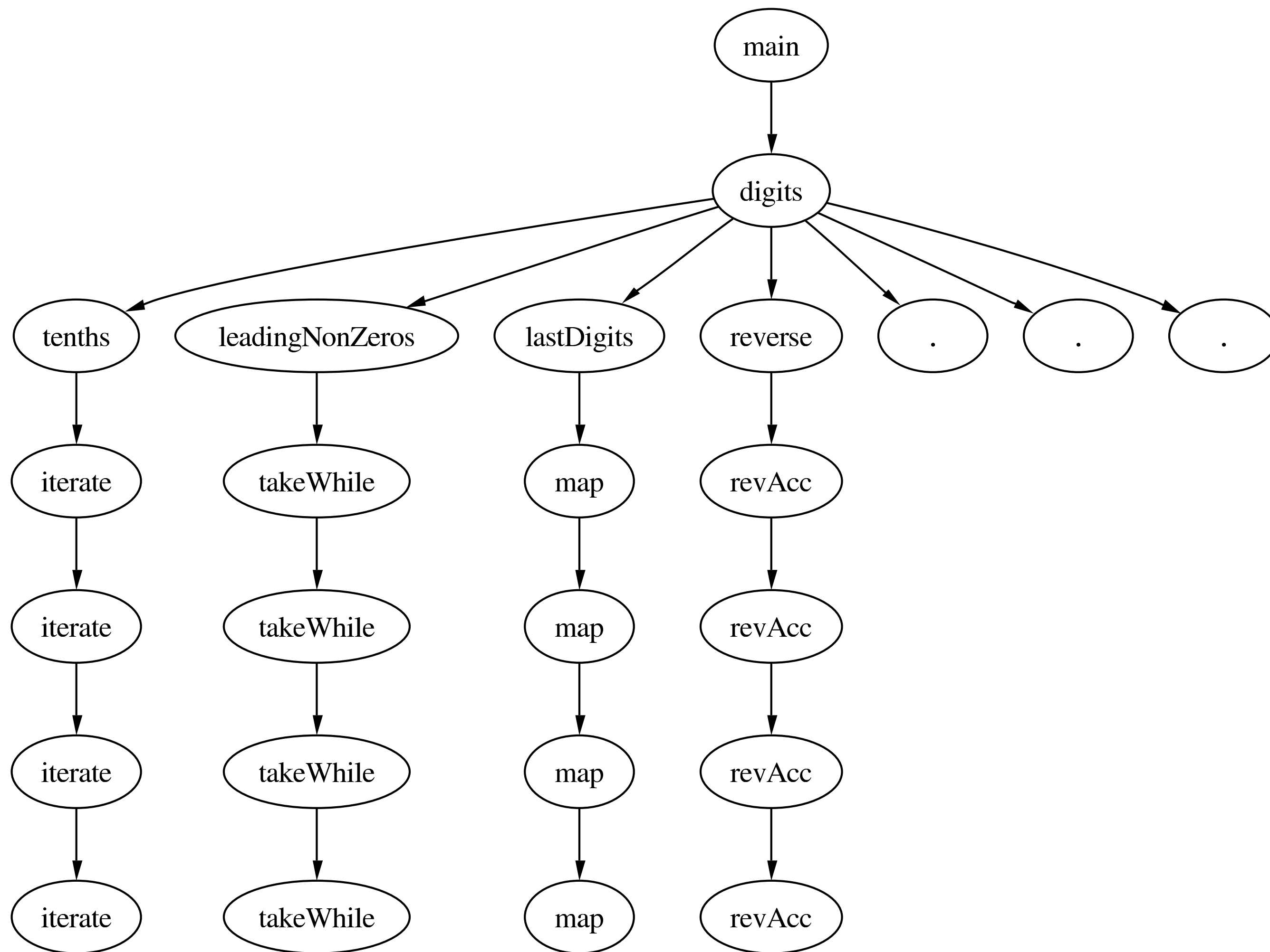
- debugging always starts at `main` which is the root of the EDT
- functions have argument values (`main` is a constant)
- `main` returns an I/O action which is (currently) abstract (this is a limitation of buddha that we would like to remedy)

Using buddha - starting

- debugging always starts at `main` which is the root of the EDT
- functions have argument values (`main` is a constant)
- `main` returns an I/O action which is (currently) abstract (this is a limitation of buddha that we would like to remedy)
- it is difficult to say anything about the correctness of `main` so we have to do some browsing — the unique number of derivations is useful for this

Using buddha - visualise the EDT

```
buddha:  draw edt
```



Using buddha - looking ahead

```
buddha: kids

Children of the current derivation:

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

`main` also calls `print`, but that function is trusted so it is not recorded in the EDT.

Using buddha - jumping and judgement

```
buddha:  jump 2
```

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

Time for a judgement!

```
buddha:  erroneous
```

Either `digits` or one or more of its descendents is buggy.

Debugging is now focused on the subtree rooted at this node.

Using buddha - higher order and laziness

```
[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 are functions
- `..?` indicates an unevaluated list

We can try to defer judgement of difficult derivations:

```
buddha:  defer
```

Using buddha - reverse

We defer two more difficult derivations involving compose, taking us to a call to `reverse`:

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

Clearly this is correct:

```
buddha:  correct
```

Using buddha - lastDigits

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

Clearly this is wrong:

```
buddha:  erroneous
```

At this point buddha narrows the focus of search from the children of `digits` to the children of `lastDigits`.

Using buddha - map

The only child of `lastDigits` 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]
```

This is correct:

```
buddha:  correct
```

Using buddha - diagnosis

Now buddha has found a bug!

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

`lastDigits` is considered *buggy* because it was erroneous but all its children were correct.

Question: What is wrong with the definition of `lastDigits`?

Using buddha - one at a time

- there may be multiple buggy nodes in a given EDT

Using buddha - one at a time

- there may be multiple buggy nodes in a given EDT
- buddha only searches for one bug at a time — it is easy to extend, but it isn't clear whether additional complexity helps or hinders the user

Using buddha - one at a time

- there may be multiple buggy nodes in a given EDT
- buddha only searches for one bug at a time — it is easy to extend, but it isn't clear whether additional complexity helps or hinders the user
- which one is found depends on where the first erroneous node is encountered, which depends on how you browse the EDT

Using buddha - one at a time

- there may be multiple buggy nodes in a given EDT
- buddha only searches for one bug at a time — it is easy to extend, but it isn't clear whether additional complexity helps or hinders the user
- which one is found depends on where the first erroneous node is encountered, which depends on how you browse the EDT
- you might have to run buddha multiple times on the debuggee with the same set of input values

The Oracle

Judgements about correctness are said to be made by an *oracle*.

The Oracle

Judgements about correctness are said to be made by an *oracle*.

The oracle is an entity that knows the intended meaning of each let-bound function and constant in the program.

The Oracle

Judgements about correctness are said to be made by an *oracle*.

The oracle is an entity that knows the intended meaning of each let-bound function and constant in the program.

The oracle is typically a human (the user of the debugger) but it has great potential for automation.

The Oracle

Judgements about correctness are said to be made by an *oracle*.

The oracle is an entity that knows the intended meaning of each let-bound function and constant in the program.

The oracle is typically a human (the user of the debugger) but it has great potential for automation.

Buddha's oracle is a software interface to the user. It maintains a database of already known derivation-judgement pairs. The debugger passes derivations to the oracle and it responds with a judgement.

The Oracle

Judgements about correctness are said to be made by an *oracle*.

The oracle is an entity that knows the intended meaning of each let-bound function and constant in the program.

The oracle is typically a human (the user of the debugger) but it has great potential for automation.

Buddha's oracle is a software interface to the user. It maintains a database of already known derivation-judgement pairs. The debugger passes derivations to the oracle and it responds with a judgement.

If the derivation is found in the database then it is returned immediately. If not, the derivation is printed to the display and the human user must respond.

For Declarative Debugging

- it appeals to a declarative specification of the program

For Declarative Debugging

- it appeals to a declarative specification of the program
- it is simple to use (you don't have to be a guru to use it)

For Declarative Debugging

- it appeals to a declarative specification of the program
- it is simple to use (you don't have to be a guru to use it)
- independent of evaluation strategy

For Declarative Debugging

- it appeals to a declarative specification of the program
- it is simple to use (you don't have to be a guru to use it)
- independent of evaluation strategy
- big potential for automation and smart searching techniques — we can distil a lot of the mundane tasks of debugging into the search strategy and the oracle

For Declarative Debugging

- it appeals to a declarative specification of the program
- it is simple to use (you don't have to be a guru to use it)
- independent of evaluation strategy
- big potential for automation and smart searching techniques — we can distil a lot of the mundane tasks of debugging into the search strategy and the oracle

Programs are complex artifacts, and debugging is often a struggle against complexity. Why should the programmer have to face it alone? Let's make the computer take an active role in helping the programmer deal with complexity.

HENRY LIEBERMAN

Against Declarative Debugging

- resource hungry

Against Declarative Debugging

- resource hungry
- difficult to debug I/O (interactive programs etc)

Against Declarative Debugging

- resource hungry
- difficult to debug I/O (interactive programs etc)
- program is run to completion before debugging begins

Against Declarative Debugging

- resource hungry
- difficult to debug I/O (interactive programs etc)
- program is run to completion before debugging begins
- judgement of derivations can be difficult

Against Declarative Debugging

- resource hungry
- difficult to debug I/O (interactive programs etc)
- program is run to completion before debugging begins
- judgement of derivations can be difficult

There will be times when the bug is associated with the operational semantics of the program (running time, space usage etc). In such circumstances a declarative debugger is not helpful. **Moral:** there is room for more than one (style of) debugger in a given language.