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

Implementation

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.

PHIL WADLER

Issues

how to represent the EDT

Issues

- how to represent the EDT
- how to construct the EDT

Issues

- how to represent the EDT
- how to construct the EDT
- how to print values

Why?

Why?

portability (compiler and machine wise)

Why?

- portability (compiler and machine wise)
- simplicity (the compiler does the hard work)

Why?

- portability (compiler and machine wise)
- simplicity (the compiler does the hard work)

Why?

- portability (compiler and machine wise)
- simplicity (the compiler does the hard work)

Why not?

replication of the front end (parse the code twice)

Why?

- portability (compiler and machine wise)
- simplicity (the compiler does the hard work)

- replication of the front end (parse the code twice)
- (lack of) expressiveness in Haskell (typing restrictions)

Why?

- portability (compiler and machine wise)
- simplicity (the compiler does the hard work)

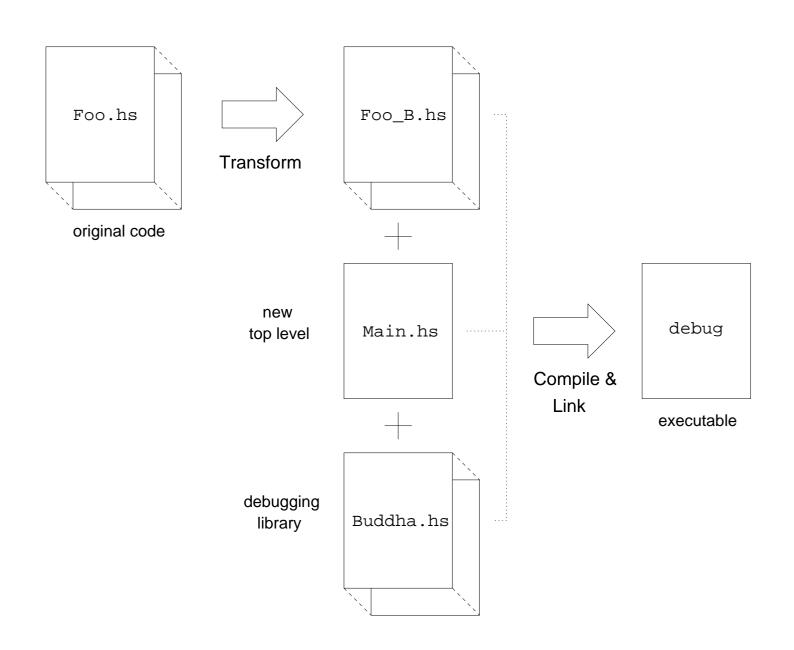
- replication of the front end (parse the code twice)
- (lack of) expressiveness in Haskell (typing restrictions)
- overheads (cost of encoding the EDT as a Haskell data type)

Why?

- portability (compiler and machine wise)
- simplicity (the compiler does the hard work)

- replication of the front end (parse the code twice)
- (lack of) expressiveness in Haskell (typing restrictions)
- overheads (cost of encoding the EDT as a Haskell data type)
- limited control of the runtime (garbage collector etc)

Transformation process



Haskell encoding of EDT

```
data Value = forall a . V a
data Derivation
   = Derivation
      name
           :: String
     , args :: [Value]
     , result :: Value
     , location :: SrcLoc }
data EDT
    EDT
      nodeID
               :: Int
     , derivation :: Derivation
     , children :: [EDT]
```

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

```
reify :: a -> IO Graph
prettyGraph :: Graph -> String
```

reify calls (GHC specific) C code via the FFI

- reify calls (GHC specific) C code via the FFI
- an advantage of using C is that it views all Haskell values in the same way (the typing restrictions at the Haskell level are not present)

- reify calls (GHC specific) C code via the FFI
- an advantage of using C is that it views all Haskell values in the same way (the typing restrictions at the Haskell level are not present)
- unevaluated objects are recognised by the C code and mapped to Thunk

- reify calls (GHC specific) C code via the FFI
- an advantage of using C is that it views all Haskell values in the same way (the typing restrictions at the Haskell level are not present)
- unevaluated objects are recognised by the C code and mapped to Thunk
- cycles are tricky to detect in the presence of garbage collection and tricky to print (do we really want to see them anyway?)

- reify calls (GHC specific) C code via the FFI
- an advantage of using C is that it views all Haskell values in the same way (the typing restrictions at the Haskell level are not present)
- unevaluated objects are recognised by the C code and mapped to Thunk
- cycles are tricky to detect in the presence of garbage collection and tricky to print (do we really want to see them anyway?)
- functions need more support

Question: Should functions be printed by their name or by their behaviour?

Question: Should functions be printed by their name or by their behaviour?

For example:

```
map (plus 1) [1,2] \Rightarrow [2,3]
```

Question: Should functions be printed by their name or by their behaviour?

For example:

```
map (plus 1) [1,2] \Rightarrow [2,3]
```

or:

map
$$\{1 \rightarrow 2, 2 \rightarrow 3\}$$
 $[1,2] \Rightarrow [2,3]$

Question: Should functions be printed by their name or by their behaviour?

For example:

```
map (plus 1) [1,2] => [2,3]
```

or:

map
$$\{1 \rightarrow 2, 2 \rightarrow 3\}$$
 $[1,2] \Rightarrow [2,3]$

Both look reasonable. Our experience is that the second form is more comprehensible in cases where the function is heavily composed or contains lambda abstractions (which don't have a name).

A global table records all applications of functions that we want to print:

```
type FunApplication = (Int, Value, Value)
type FunTable = [FunApplication]
```

Each function is given a unique number which forms the index of the table. A wrapper type is introduced to pair functions with their number:

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

Wrappers are introduced by one of a family of functions:

```
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))
```

The number of the function relates to the arity of the function that it wraps.

When wrapped functions are applied they must be unwrapped and the application must be recorded in the global table.

Updates to the global table are achieved by side-effects!

An example of function wrapping and application recording:

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

map :: F a b -> [a] -> [b]

map f [] = []

map f (x:xs) = apply f x : map f xs
```

The introduction of wrappers necessitates a change in types throughout the program.

reify is applied to wrapped functions just like any other value.

reify is applied to wrapped functions just like any other value.

For example, we might get something like this back:

AppNode 28 "F" [IntNode 4, Function]

reify is applied to wrapped functions just like any other value.

For example, we might get something like this back:

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

Obviously this is not printed as is, rather it is interpreted by the printer.

reify is applied to wrapped functions just like any other value.

For example, we might get something like this back:

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

Obviously this is not printed as is, rather it is interpreted by the printer.

All entries in the global table that correspond to function number 4 are selected and printed accordingly.

EDT construction

We've tried two approaches:

- the purely functional approach
- a tabling approach (relies on side effects)

For comparison we'll show the two styles on this function:

Purely functional EDT construction

Each function and constant is transformed to return a pair containing the original value and an EDT node.

Purely functional EDT construction

- Each function and constant is transformed to return a pair containing the original value and an EDT node.
- Children nodes are collected from function applications that occur in the body.

Purely functional EDT construction

- Each function and constant is transformed to return a pair containing the original value and an EDT node.
- Children nodes are collected from function applications that occur in the body.
- Computation of the original value and the EDT are interwoven.

Purely functional EDT construction

- Each function and constant is transformed to return a pair containing the original value and an EDT node.
- Children nodes are collected from function applications that occur in the body.
- Computation of the original value and the EDT are interwoven.
- This approach is typical in the literature.

Purely functional EDT construction

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

Function applications and constants are uniquely numbered (dynamically).

- Function applications and constants are uniquely numbered (dynamically).
- A global table records each application and constant. The table is indexed by the application number.

- Function applications and constants are uniquely numbered (dynamically).
- A global table records each application and constant. The table is indexed by the application number.
- Each function is transformed to take an extra argument
 the unique number of the parent. Table entries also record this parent number.

- Function applications and constants are uniquely numbered (dynamically).
- A global table records each application and constant. The table is indexed by the application number.
- Each function is transformed to take an extra argument
 the unique number of the parent. Table entries also record this parent number.
- When the debuggee is complete one pass is made over the table to build the EDT (compute the complete parent-child relationship).

- Function applications and constants are uniquely numbered (dynamically).
- A global table records each application and constant. The table is indexed by the application number.
- Each function is transformed to take an extra argument
 the unique number of the parent. Table entries also record this parent number.
- When the debuggee is complete one pass is made over the table to build the EDT (compute the complete parent-child relationship).
- Unique numbers and table updates are done via side effects!

■ The purely functional style can cause more thunks to be created and it tends to hold onto memory longer than we'd like (see the space leak in Lazy State Threads).

- The purely functional style can cause more thunks to be created and it tends to hold onto memory longer than we'd like (see the space leak in Lazy State Threads).
- The tabled implementation decouples the generation of the EDT and the evaluation of the debuggee.

- The purely functional style can cause more thunks to be created and it tends to hold onto memory longer than we'd like (see the space leak in Lazy State Threads).
- The tabled implementation decouples the generation of the EDT and the evaluation of the debuggee.
- This opens the door to incremental construction of the EDT, where program evaluation and debugging are interleaved.

- The purely functional style can cause more thunks to be created and it tends to hold onto memory longer than we'd like (see the space leak in Lazy State Threads).
- The tabled implementation decouples the generation of the EDT and the evaluation of the debuggee.
- This opens the door to incremental construction of the EDT, where program evaluation and debugging are interleaved.
- Decoupling also makes it easier to debug programs that crash (raise uncaught exceptions etc)

- The purely functional style can cause more thunks to be created and it tends to hold onto memory longer than we'd like (see the space leak in Lazy State Threads).
- The tabled implementation decouples the generation of the EDT and the evaluation of the debuggee.
- This opens the door to incremental construction of the EDT, where program evaluation and debugging are interleaved.
- Decoupling also makes it easier to debug programs that crash (raise uncaught exceptions etc)
- The tabled implementation relies on impure side-effects which are difficult to manage in a highly optimising compiler like GHC.