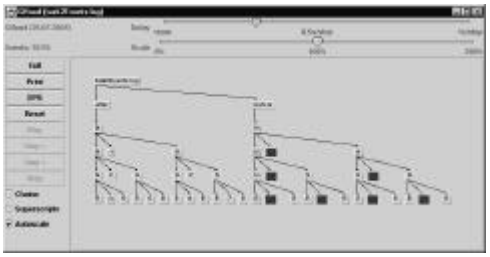


GHood

Visualising Observations of Haskell Program Runs

Claus Reinke

Computing Lab, University of Kent at Canterbury

The problem: program comprehension

```
data Tree a = E | N (Tree a) a (Tree a) deriving (Show)


task n ~rs         E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')


taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')

bfnum t = head $ taskM 1 [t]
```

What does this program do?

The problem: program comprehension

```
data Tree a = E | N (Tree a) a (Tree a) deriving (Show)


task n ~rs         E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')


taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')


bfnum t = head $ taskM 1 [t]
```

hey, this is functional, it should be obvious...

The comments, read the comments! ..

Use the source, Luke!

Tool support for

The problem: program comprehension

```
data Tree a = E | N (Tree a) a (Tree a) deriving (Show)

task :: Integer -> [Tree Integer] -> Tree b
     -> (Integer, [Tree Integer], [Tree b], Tree Integer)
task n ~rs         E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')

taskM :: Integer -> [Tree b] -> [Tree Integer]
taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')

bfnum :: Tree a -> Tree Integer
bfnum t = head $ taskM 1 [t]

someTree = N (N (N E 0 E) 1 (N E 0 E)) 2 (N (N E 0 E) 1 (N E 0 E))

main = print $ bfnum someTree
```

Check the types

Find type inhabitants, apply program, and print what is going on

Tool support for

The problem: program comprehension

```
data Tree a = E | N (Tree a) a (Tree a) deriving (Show)

task :: Integer -> [Tree Integer] -> Tree b
     -> (Integer, [Tree Integer], [Tree b], Tree Integer)
task n ~rs         E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')

taskM :: Integer -> [Tree b] -> [Tree Integer]
taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')

bfnum :: Tree a -> Tree Integer
bfnum t = head $ taskM 1 [t]

someTree = N (N (N E 0 E) 1 (N E 0 E)) 2 (N (N E 0 E) 1 (N E 0 E))

main = print $ bfnum someTree
```

context supplies type inhabitants,might not permit printing

..but what if we need to observe programs in existing contexts?

```
import Observe

instance Observable a => Observable (Tree a) where
  observer E         = send "E" (return E)
  observer (N l x r) = send "N" (return N << l << x << r)

data Tree a = E | N (Tree a) a (Tree a) deriving (Show)

task :: Integer -> [Tree Integer] -> Tree b
     -> (Integer, [Tree Integer], [Tree b], Tree Integer)
task n ~rs         E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')

taskM :: Integer -> [Tree b] -> [Tree Integer]
taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')

bfnum :: Tree a -> Tree Integer
bfnum t = head $ taskM 1 [t]

someTree = N (N (N E 0 E) 1 (N E 0 E)) 2 (N (N E 0 E) 1 (N E 0 E))

main = runO $ print $
       observe "after" $ bfnum $ observe "before" $ someTree
```

.. use Hood and observe

What does `observe` give you?

- with "plain" Hood:
  - static views of intermediate structures in your computation
  - works even where you could not easily print them
  - pre-defined instances of `Observable` for most standard types
  - combinators make it easy to define instances for your own types
  - information is not thrown out "at random" but is collected and pretty-printed at the end of the run
  - observation does not change strictness properties: *you only see what is actually inspected by your program*

Instrumented program output

```
Main> main
N (N (N E 0 E) 1 (N E 0 E)) 2 (N (N E 0 E) 1 (N E 0 E))
N (N (N E 4 E) 2 (N E 5 E)) 1 (N (N E 6 E) 3 (N E 7 E))

-- after
  N (N (N E 4 E) 2 (N E 5 E)) 1 (N (N E 6 E) 3 (N E 7 E))
-- before
  N (N (N E _ E) _ (N E _ E)) _ (N (N E _ E) _ (N E _ E))
```

What does `observe` give you?

- with GHood, you get all the goods of Hood, plus:
  - static and dynamic views of intermediate structures by animated graphical visualisation of observations
  - you can see the order in which observations are made (even differences between Haskell implementations), without losing the structured presentation
  - by animating two observations side-by-side, the relative order of observations can give you insights into data dependencies and strictness-properties
  - *the animations can also be used separately from programs and Haskell implementations, to document and explain program behaviour on web pages (see GHood homepage; URL in paper)*

Hood/GHood demo (introduction)

How does it all work?

just a quick tour, with two stops:

- from `trace` to `observe`, via `unsafePerformIO`
  - `unsafePerformIO` generalises `trace`
  - `observe` can be seen as a much improved `trace`
  - `unsafePerformIO` as an extension hook
- from Hood to GHood, via more hooks

How does it work?

Problems with `trace`

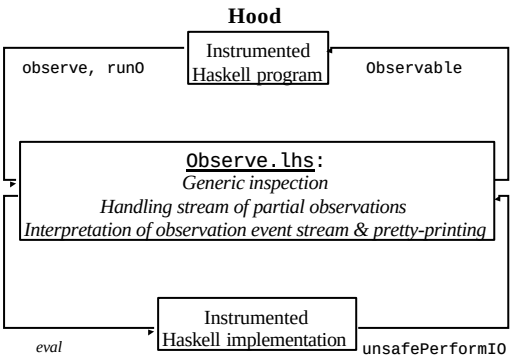
- ```
  trace:: String -> a -> a
  trace s a = unsafePerformIO $
  hPutStr stderr s >> return a
  ```
  ☹ no connection between `s` and `a`, no information apart from `s` (calling context has to supply information via `s`), evaluation of s could force evaluation of `a`, or other traces..
- ```
  traceShow:: Show a => String -> a -> a
  traceShow s a = -- trace (s++show a) a
     unsafePerformIO $ do
           hPutStr stderr (s++show a)
           return a
  ```
  ☹ generic `show` solves first problems, but is strict in `a`!! order of output can still be hard to predict and messed up..

How does it work?

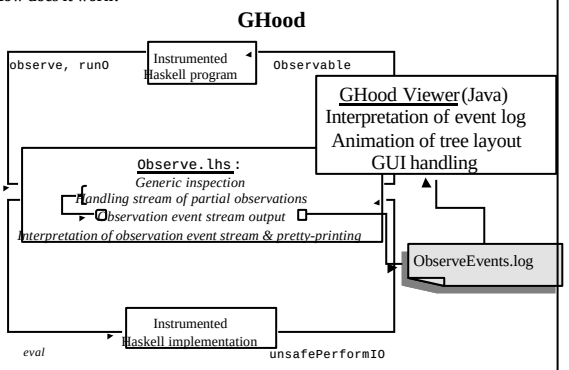
Pseudo-code for `observe`

- ```
  class Observable a where
      observer:: a -> Parent -> a
  ```
  merge generic inspection into observation, so that inspection can be limited to preserve strictness, then replace `show` functionality with less intrusive code
- ```
  observe::Observable a => String-> a-> a
  observe s a = unsafePerformIO $ do
      parent <- init s a
      return $ observer a parent
  ```
  store partial observations out of the way, and preserve links between fragments; observe initialises, details are handled in an observation monad, and
- ```
  observer [] = send "[]" (return [])
  ```
  `observer` definitions guide the process ☺
  ```

How does it work?



How does it work?



A closer look at the example or: finding problems with solutions to Okasaki's breadth-first tree numbering exercise

The problem (informally): label the nodes in a binary tree with natural numbers, in breadth-first order

GHood demo: task-based solution, original version

GHood demo: task-based solution, second version
(reversed solution list for nicer code)

GHood demo: task-based solution, third version
(gets structure and label via separate paths)

Task1

```
task n ~[]      E         = (n  ,[]   ,E)
task n ~[l',r'] (N l x r) = (n+1,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = t':rs'
  where
    (n',tp',t') = task n r t
    ts'         = taskM n' (ts++tp')
    (rs',r)     = splitAt (length ts) ts'

bfnum t = head $ taskM 1 [t]
```

Task2

```
task n ~rs         E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')

bfnum t = head $ taskM 1 [t]
```

Task1new

```
task n ~[]      E         = (n  ,[]   ,E)
task n ~[l',r'] (N l x r) = (n+1,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = t':rs'
  where
    (n',tp',t') = task n r t
    ts'         = taskM n' (ts++tp')
    (rs',r)     = splitAt (length ts) ts'

bfnum t = fillIn t $ head $ taskM 1 [t]

fillIn E         ~E           = E
fillIn (N l _ r) ~(N l' x' r') =
      N (fillIn l l') x' (fillIn r r')
```

Evaluation

- *seeing intermediate structures* is useful, as expected
- surprisingly, *seeing what is not used* is just as useful
- new with Ghood: seeing when which parts are observed
  - the good old uses, but with more precise information
  - dynamic data dependencies become visible
  - graphical animation is a win for medium to large sizes
- animating observations is no substitute for animating reductions (in both senses)
  - only indirect insights into program behaviour, but less cluttered visualisation to begin with
  - more control over visualisation, including observation at different levels of abstraction

Conclusions

- Understanding operational program aspects is important
  program = declarative + operational aspects
- Animated visualisation of observations is a useful tool
- Visualising observations ≠ visualising reductions;
  ideally, both should be part of the programmer's toolbox

Other lessons learned:

The advantages of compositionality extend beyond Haskell programs, to Haskell implementations and tools.

Extending implementations with portable tool components depends on standard implementation extension interfaces (aka extension hooks).

- Olde rule of thumb: "get it right before making it faster"
- Declarative/Functional programming:
  - Focus on description of data and algorithms relevant to problem and solution domains
  - Decompose complex problems into simpler ones, and compose complex solutions from simpler ones
- Not optimal in terms of resource usage, but automatically complementing concise declarative specifications with default operational aspects has proven successful
- But what happens after "we got it right"?
  - Resource usage may become important
  - Imperative programmers had to focus on that anyway

⇒ Modular/aspect-oriented programming:

⇒ specify both declarative and operational aspects,

⇒ combine separate aspects to get complete program