```
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]
```

GHood

Visualising Observations of Haskell Program Runs

Claus Reinke

Computing Lab, University of Kent at Canterbury

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

.. code as before ..

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


{- compare input tree and program output:

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))

but setting up a sensible testbed isn't always that easy;
we might want to test parts of a program, but in their
original context!
-}
```

GHood

Visualising Observations of Haskell Program Runs

Claus Reinke

Computing Lab, University of Kent at Canterbury

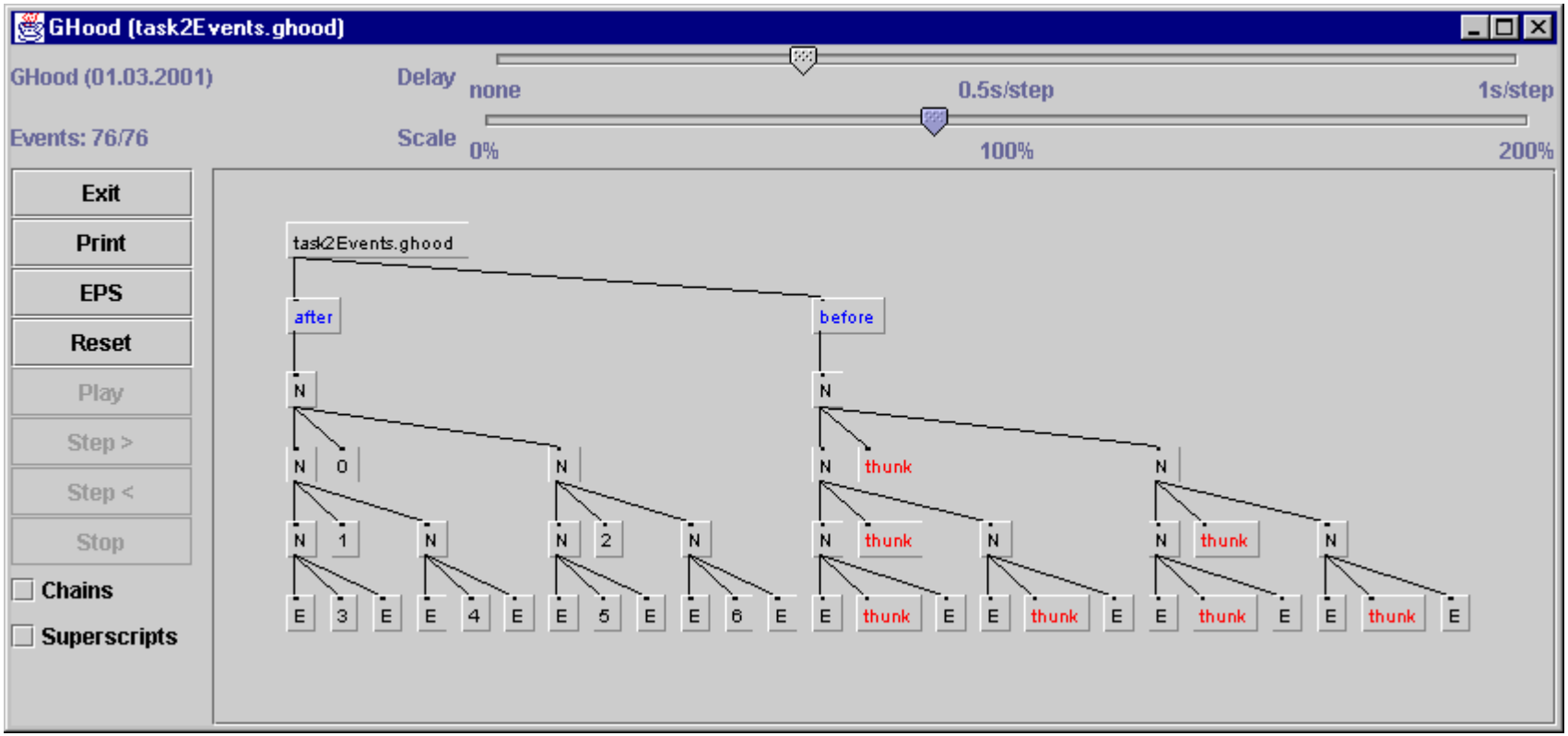
```
import Observe

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

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)

.. code as before ..

main = printO $
  observe "after" $ bfnum $ observe "before" $ someTree
```

GHood

Visualising Observations of Haskell Program Runs

Claus Reinke

Computing Lab, University of Kent at Canterbury



GHood

Visualising Observations of Haskell Program Runs

Claus Reinke

Computing Lab, University of Kent at Canterbury

Declarative versus imperative style

- *declarative programs* tend to *describe* the *"what?"* of computations, leaving the *"how?"* implicit
- *imperative programs* tend to *prescribe* the *"how?"* of computations, leaving the *"what?"* implicit

functional programs* tend to *emphasise a descriptive style* while also *indicating possible ways of computing

- this is a useful compromise, supported by experience:
  - *1. get it right* (only if necessary: *2. make it efficient*)
  - surprisingly often, compilers can do a good job of 2.
  - sometimes, they need a little help..

The implicit "how"

- Functional programmers have tried to leave operational aspects to "clever" compilers, as far as possible..
- ***compilers have become quite good*, but**
  - **they *rarely change your algorithms/data structures***
  - **they *can't guess what you want* (value of resources?)**
- most operational aspects are still handled implicitly:
  0. programs describe algorithms
  1. compilers "derive" code directly from the algorithms in your programs, using some operational semantics
  2. compilers "optimise" within algorithms/data structures
  3. low-level profilers help you to see the resource usage
  4. you change your program and try again..

  $\Rightarrow$ *unknowns in 1&2, declarativeness compromised by 4!*

The "what" of "how" - part I *(specification)*

- *Programs = declarations + operational aspects*
- operational aspects implicit ⇒ compiler chooses defaults
  - one set of defaults is not always adequate
  - should we really try to guess those defaults?

  ⇒ options & pragmas, but do they need to be imperative?
  (who knows appendix E.3 to the Haskell report?)
  - should we really change the declarations to get an indirect handle on the operational aspects?

- ***improved declarative (?) means to describe operational aspects of functional programs seem to be called for, as well as means to instantiate declarative programs with some specific operational properties***

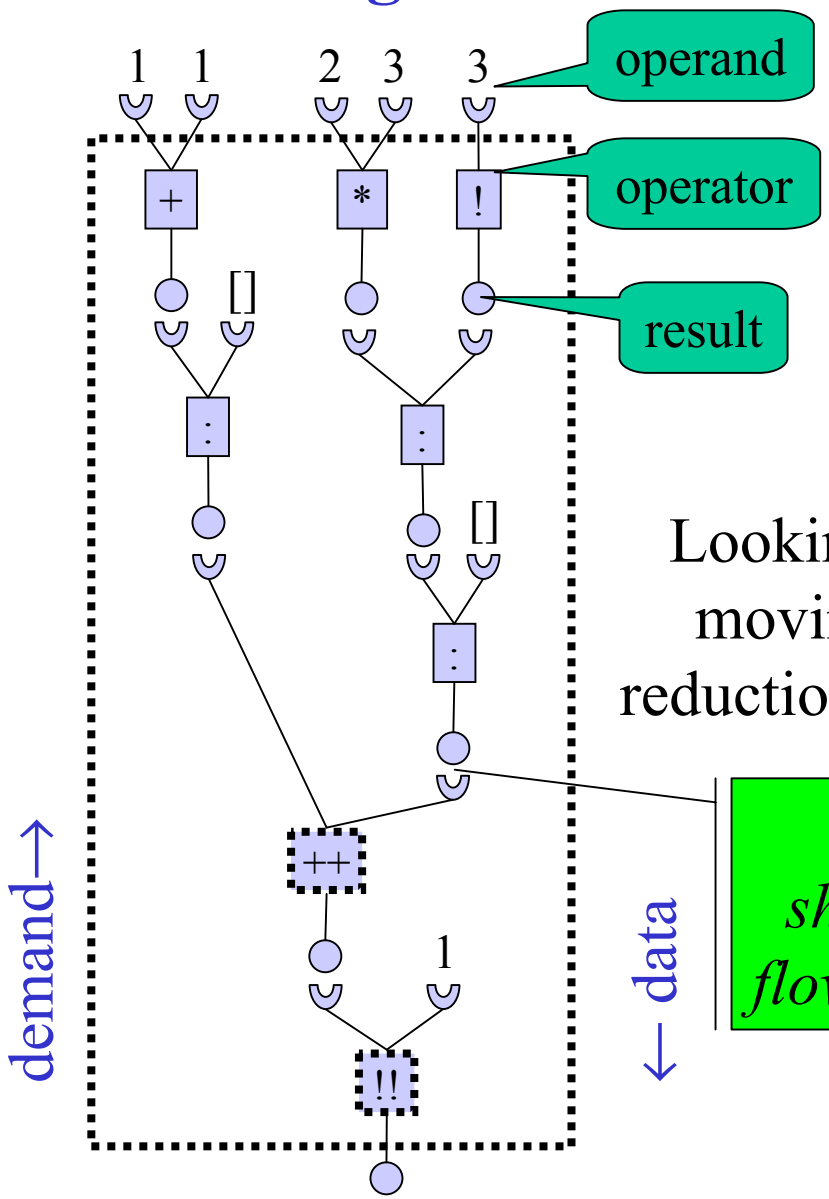
(aspect-oriented programming has some ideas on the latter)

The "what" of "how" – part II *(understanding)*

- *Programs = declarations + operational aspects*
  (similar to logic + proof theory)
- we need a better understanding of program behaviour
  - what are operational aspects? (space/time use during compilation/execution, strictness, dependencies,..)
  - can they be isolated from declarative aspects?
  - *high-level operational semantics* would be useful
  - the problems are dynamic in nature, so *high-level animations of program behaviour* would be a nice way to gain insights
  - ..

$\Rightarrow$ lots of other interesting issues, but this talk is only about the *visualisation of program behaviour*

Program behaviour: flow versus rewriting



redex

([1+1]++[2*3,3!])!!1

↓

reduct

((1+1):([]++[2*3,3!]))!!1

↓

([]++[2*3,3!])!!(1-1)

↓

Looking glass, moving over reduction sequence

([]++[2*3,3!])!!0

↓

[2*3,3!]!!0

↓

2*3

↓

6

← reduction

Probe, showing data flowing through

different views, leading to tools focussing of different aspects

From reduction traces to flow observations, in a few easy steps

- All Haskell implementations support the non-standard

```
trace: String -> a -> a
-- output first parameter, return second parameter
-- a labelled polymorphic identity function?
```

- simple idea of tracing reduction sequences:
  - use `trace` to tag redices with `String` labels
  - runtime system outputs label when redex is reduced
  - the resulting trace of redex labels gives an indication of what reduction sequence was followed

Some problems with `trace`

P1: *where to get sensible trace labels from?*

- compute label from labelled redex (and its context)
- extra code to embed useful information into trace labels
- we're getting closer to debugging by print statements..

P2: *non-strict evaluation of functional programs is quite unlike sequential, in-order execution of imperative code.* Program traces can be very difficult to read.. (e.g., evaluation dependencies can lead to nested labels!)

P3: *replacing constant `String` labels by `String`-valued expressions could change the program under inspection!* As a typical example, `(showTrace s)` changes strictness:

```
showTrace :: Show a => String ->a -> a
showTrace s a = trace (s++(show a)) a
```

Interlude:
Side-effecting pseudo-functions for i/o, again?

- P2 is just an instance of the *general conflict between i/o by side-effect*, as in `trace`, *and non-strict evaluation order*.
- The general solutions *make the sequencing of i/o operations explicit as part of program results*.
- We can't use any of the general solutions here:

we want to change the program under inspection as little as possible!

- i/o and tracing look similar but are actually quite different:
  - functional i/o tries to add i/o to functional programs
  - tracing tries to add i/o to functional language implementations

From `trace` to `observe` (outline)

P1: *where to get sensible labels from?*

- *provide the extra code* for gathering information about parameters *in a library*, *returning labels to a tag-role*
- use type classes to provide versions of *generic code*

P2: *confusing output in evaluation order*

- don't output observations in evaluation order
- use labels to *group related observations* together

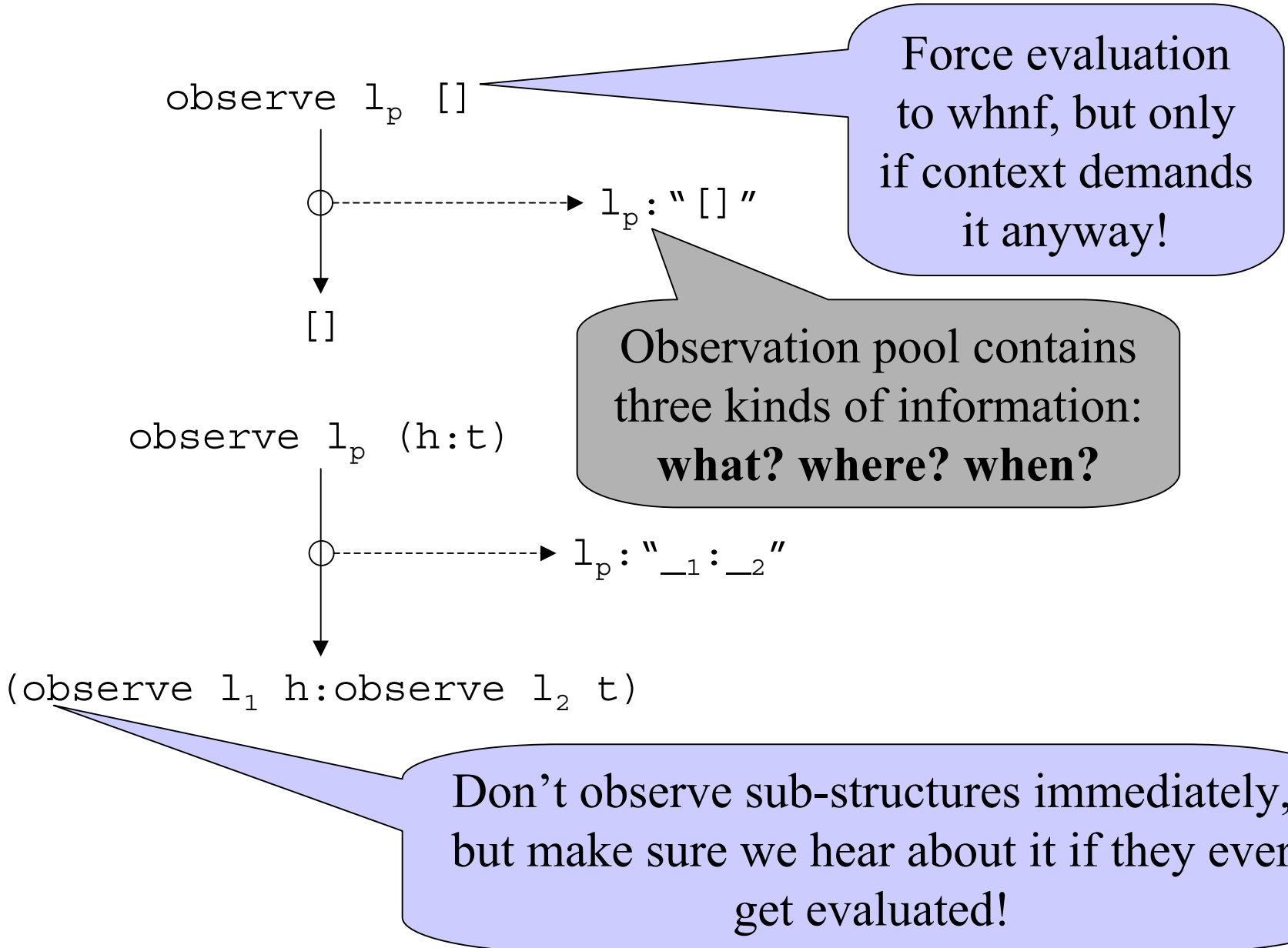
P3: *danger of changing program properties*

- factoring observation code into a library localises possible problems and allows to reuse code that works
- mere inspection of data can introduce new demands for evaluation, so *be careful not to inspect anything unless the original program inspects it as well!*

Behind the scenes

- `unsafePerformIO :: IO a -> a`
- Usually seen as side-effecting pseudo function for dirty work that may need to be done in libraries: use normal IO, then „pretend that nothing has happened“ (prefix `unsafe` indicates proof obligation)
- Just what we need for tracing, observing, and debugging?
  - can implement `trace`:
    ```
    trace l a = unsafePerformIO (putStr l >> return a)
    ```
  - can also maintain state of unique label supply, easing construction of partial data structures from observations
  - could even support user interaction
- Can also be seen as an extension hook in the evaluation mechanism, programmable in Haskell itself!

Observing lists (pseudo-code)

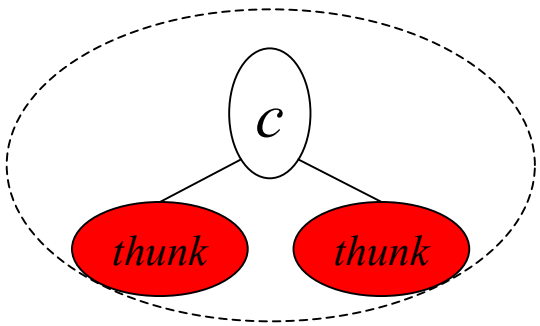


Observations

- Observations form labelled trees, built up incrementally
- At each step, observation trees represent partial knowledge of data structures passing through observation points
- Life-time of a node:

as yet unknown part, not yet under inspection

as yet unknown part, currently under inspection



partially known structure, as yet unknown parts

Hood/Ghood - variations of a theme

- The vanilla version of Hood consists of the library Observe
  - *observations are collected, grouped and pretty-printed*
  - very useful for probing data-flow in your programs, but it only uses the "*what*" and "*where*" information
- Hood in nhc98 comes with a textual observation browser
  - all advantages of vanilla version, plus use of "*when*" information for *animation* (of structure refinement)
  - doesn't work so well when observing bigger structures: lots of scrolling, no scaling, no survey view
  - text wouldn't be visible in smaller scale, and structure gets lost due the form of pretty-printing used
- Ghood is a *graphical observation browser*..

GHood demos

- GHood as a stand-alone browser, started by a Haskell evaluator or from the commandline
- GHood as an applet in webpages
- Current development of GHood focusses on
  - How to capture dynamic info in static snapshots?
  - Visualising evaluation dependencies
  - Replacing scrollbars and scaling-slider with more suitable forms of control (scrollbox, auto-scaling)
  - Upgrading to structure-preserving tree layout algorithm
  - *..lots of other open roads to follow..*
- Some ideas that don‘t quite work: fading colours, scrollbars, life-time maps (visual or textual), ..

Preliminary conclusions

- Animating Hood observations using a Java-based browser allows for portable **visualisation of program behaviour**, **even in online documentation of functional programs!** *[but using Java isn't problem-free]*
- Animation exploits the information represented by the relative ordering of observations, enabling **high-level observations of dynamic program properties**.
- **Graphical visualisation** offers the usual advantages of scale over textual methods (when labels are no longer readable, structure should be recognizable after layout). *[but standard user interface issues come into play]*
- **User feedback has been encouragingly positive so far**.
- Please try it yourselves: http://www.cs.ukc.ac.uk/people/staff/cr3/toolbox/haskell/GHood