Tracing and Debugging Haskell Programs with Hat

Olaf Chitil

University of York

Hat: Jan Sparud, Colin Runciman, Malcolm Wallace

ROPA, Canon Research: 1996-97 EPSRC: 2000-02

Tracing

Goals

- locate errors causing wrong behaviour
  - wrong output
  - abortion with error message
  - non-termination
- understand how program works (?)
- source level trace
- observable behaviour is preserved

Conventional Tracing

examine at given point of computation
part of current state

print-method: static
debugger: interactive, also shows call stack

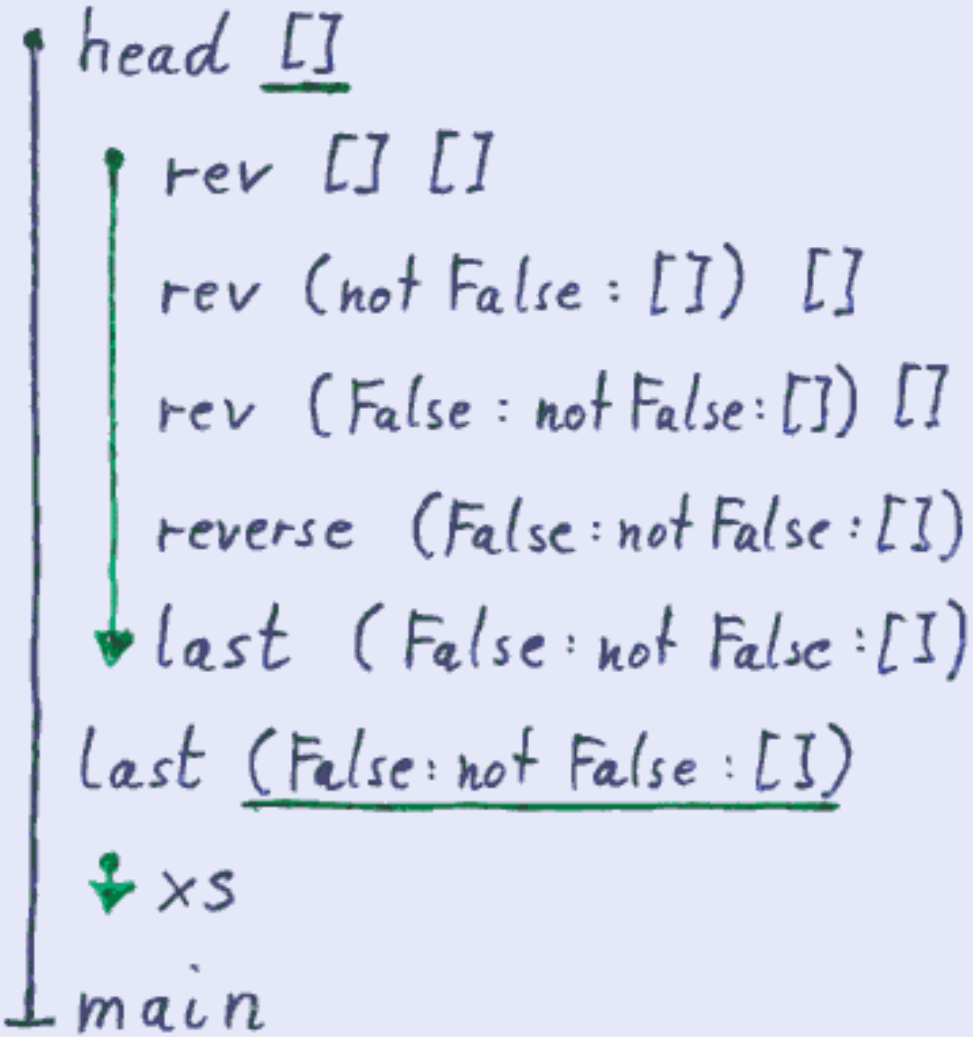
unsuitable for lazy functional languages

- evaluation order confusing (cf. trace)
- unevaluated expressions large & hard to read
- low level (functional view lost)

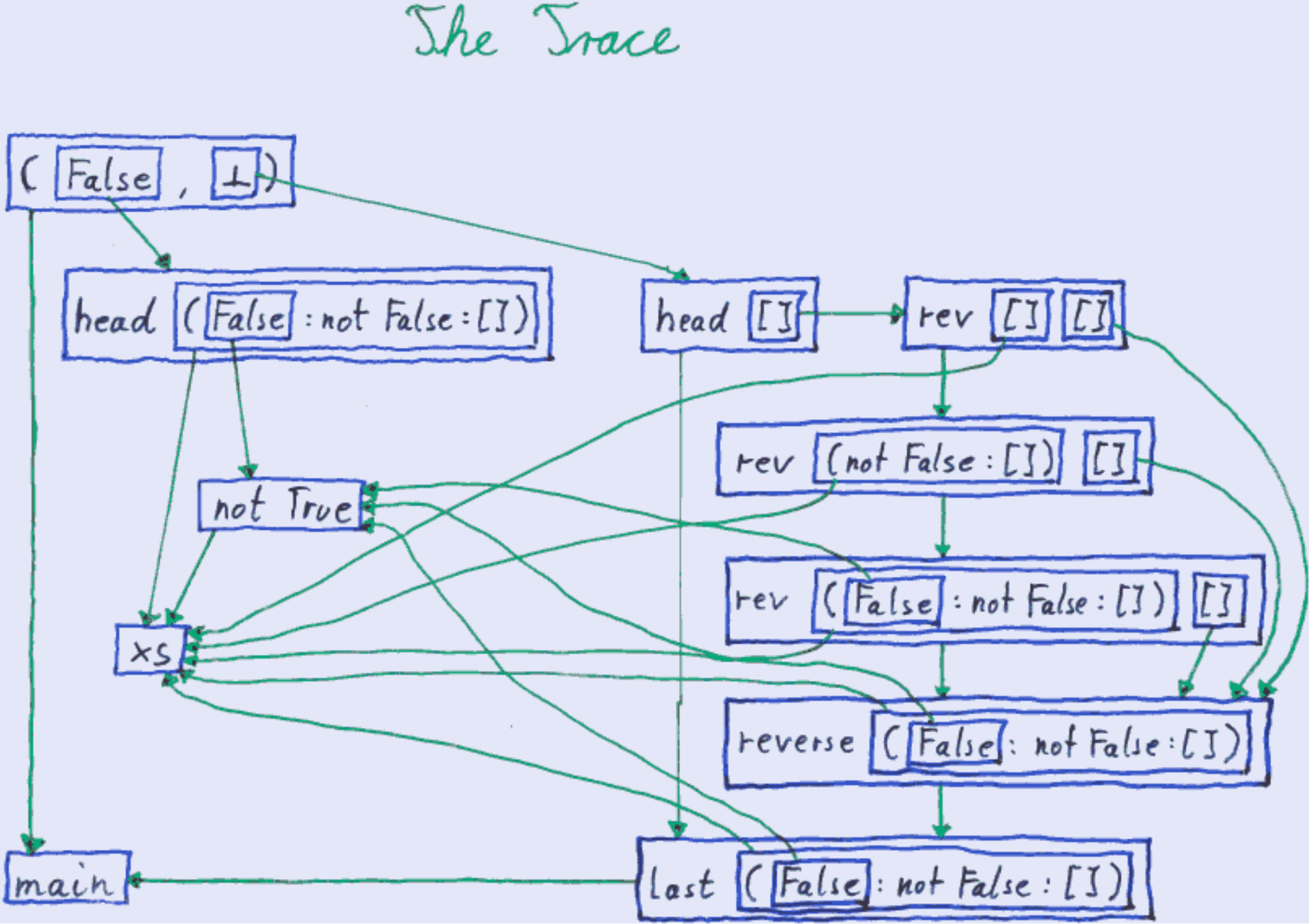
Hat

```
reverse xs = rev xs []

rev [] ys = ys
rev (x:xs) ys = rev xs ys

last xs = head (reverse xs)

main = let xs = [not True, not False]
       in print (head xs, last xs)
```

```
(False,
Error: head of empty list
```

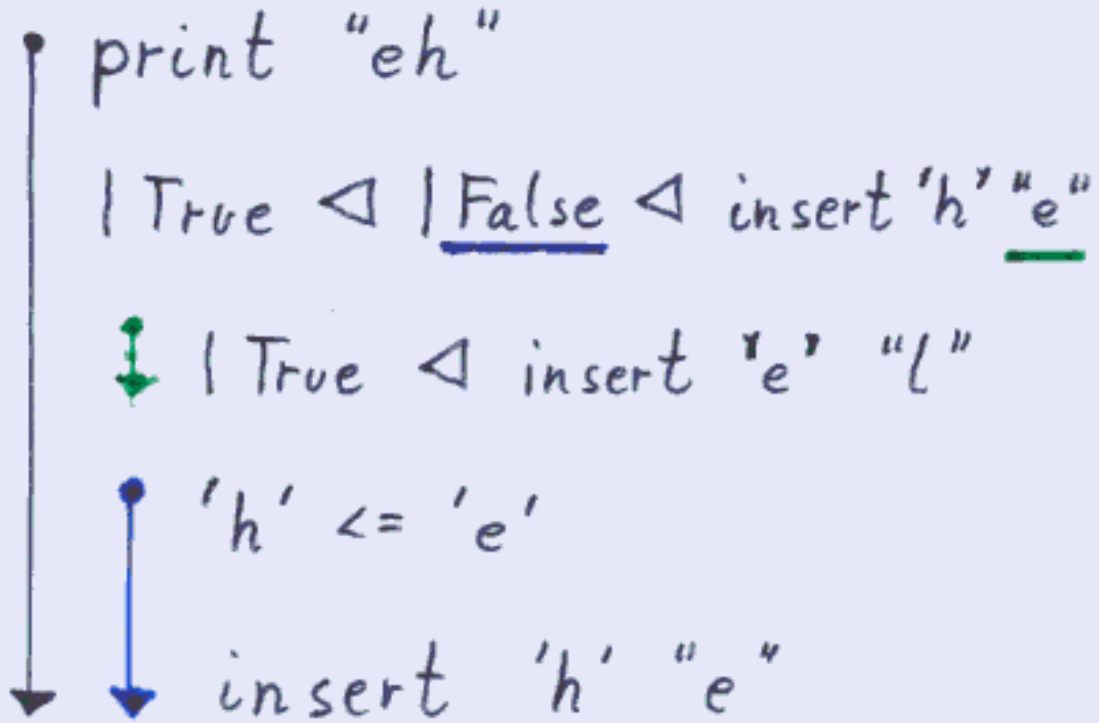


The Trace


(False , ⊥)
head (False : not False : [])
head []
rev [] []
rev (not False : []) []
not True
rev (False : not False : []) []
xs
reverse (False : not False : [])
main
last (False : not False : [])

Guards, Case & If

```
insert x [] = [x]
insert x (y:ys)
  | x <= y  =  x:ys
  | otherwise = y : insert x ys

sort = foldr insert []

main = print (sort "hello")
```

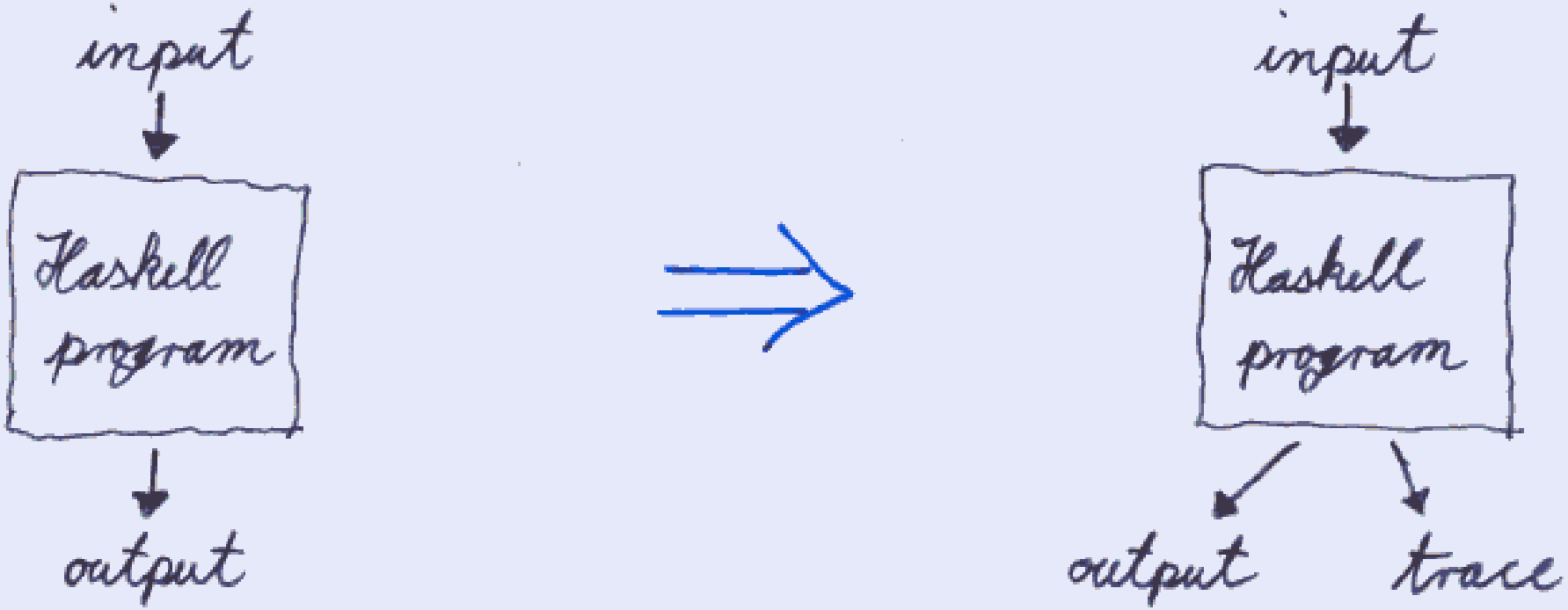


eh

Shared Constants

```
primes = sieve [2..]

firstprimes n = take n primes

main = print (firstprimes 5) >> print primes
```

Implementation by Program Transformation



Expressions wrapped with Traces

```
data Trace = Ap Trace [Trace] | Name Trace String | Root

data R a = R a Trace
```

```
(Bool, Bool)   ⇒   R (R Bool, R Bool)
```

```
(True, False)   ⇒   let t = Name Root "True"
                        f = Name Root "False"
                        p = Ap Root [Name Root "(,)", t, f]
                    in R (R True t, R False f) p
```

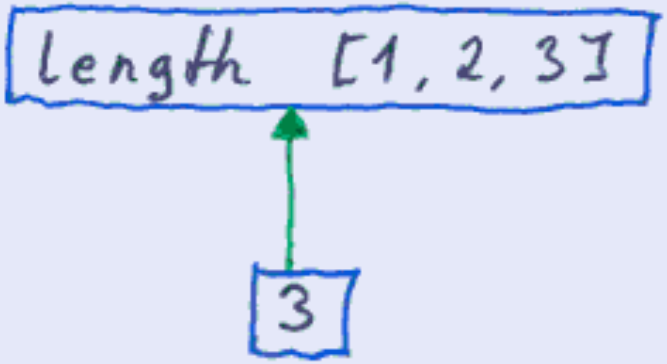
Wrapping of a Redex

```
data Trace = ... | Sat Trace Trace
```

```
not False  ⇒  let n = Name Root "not"
                  f = Name Root "False"
                  R v vt = ap1 (R not n) (R False f)
              in  R v (Sat vt (Ap Root [n, f]))
```

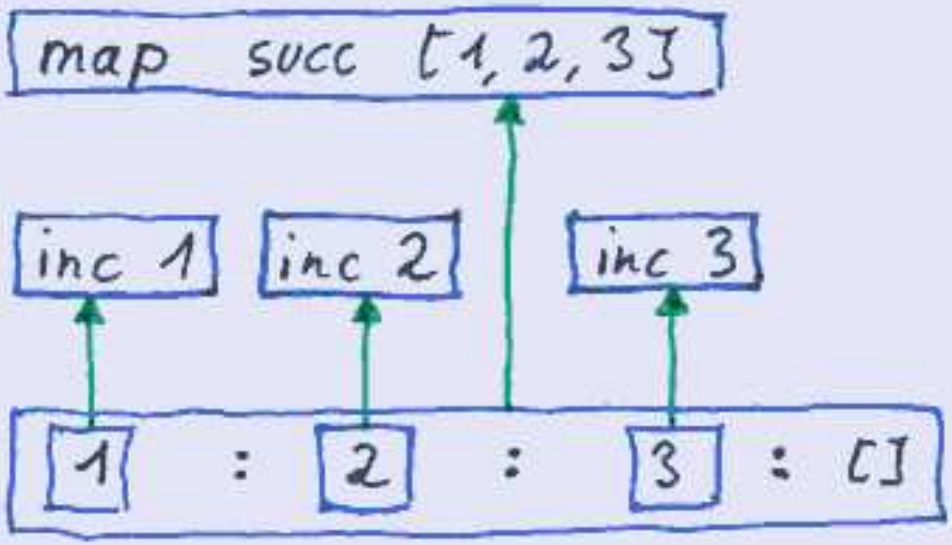
Trusting reduces Trace Size

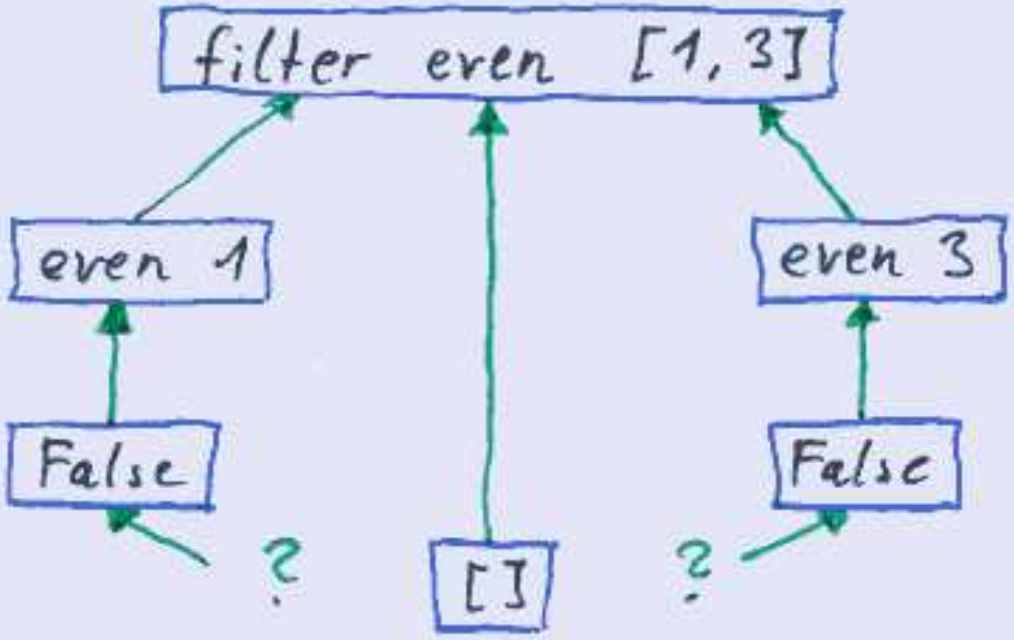
- reduce information
- reduce memory consumption



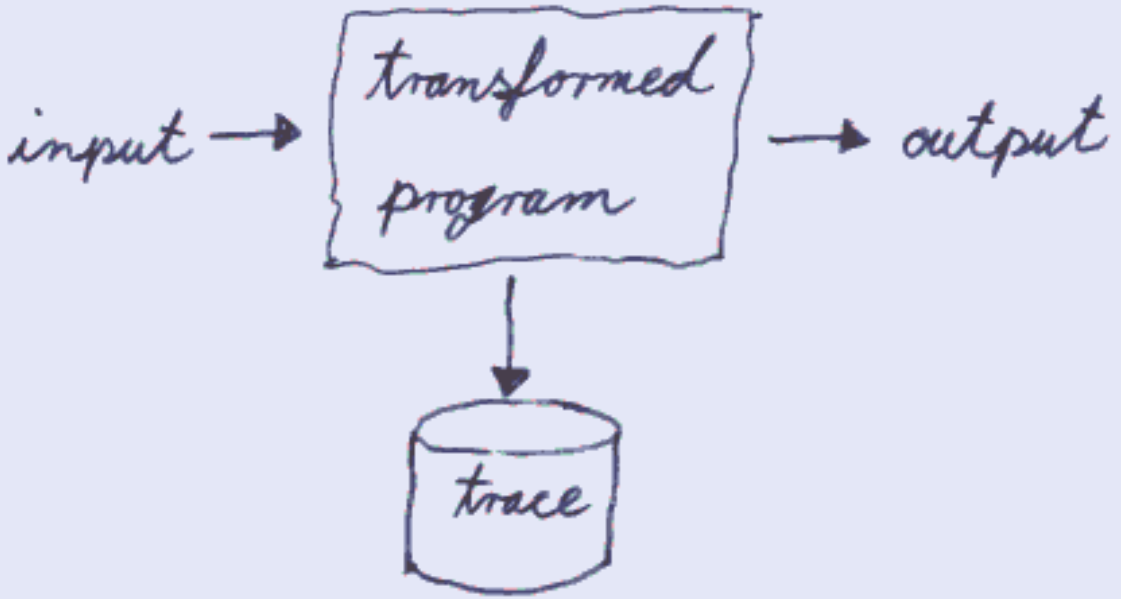
- trust definition of a function
- see only application and result

Trusting Higher-Order Functions





Incremental Archiving of Trace to File




```
data Trace = Ap Trace [Trace]
           | Name Trace String
           | Root
           | Set Trace Trace
```

⟹

```
data Trace = T Filepointer

mkAp
mkName
makeRoot
makeSet
```

Frejа

compiler for a subset of Haskell

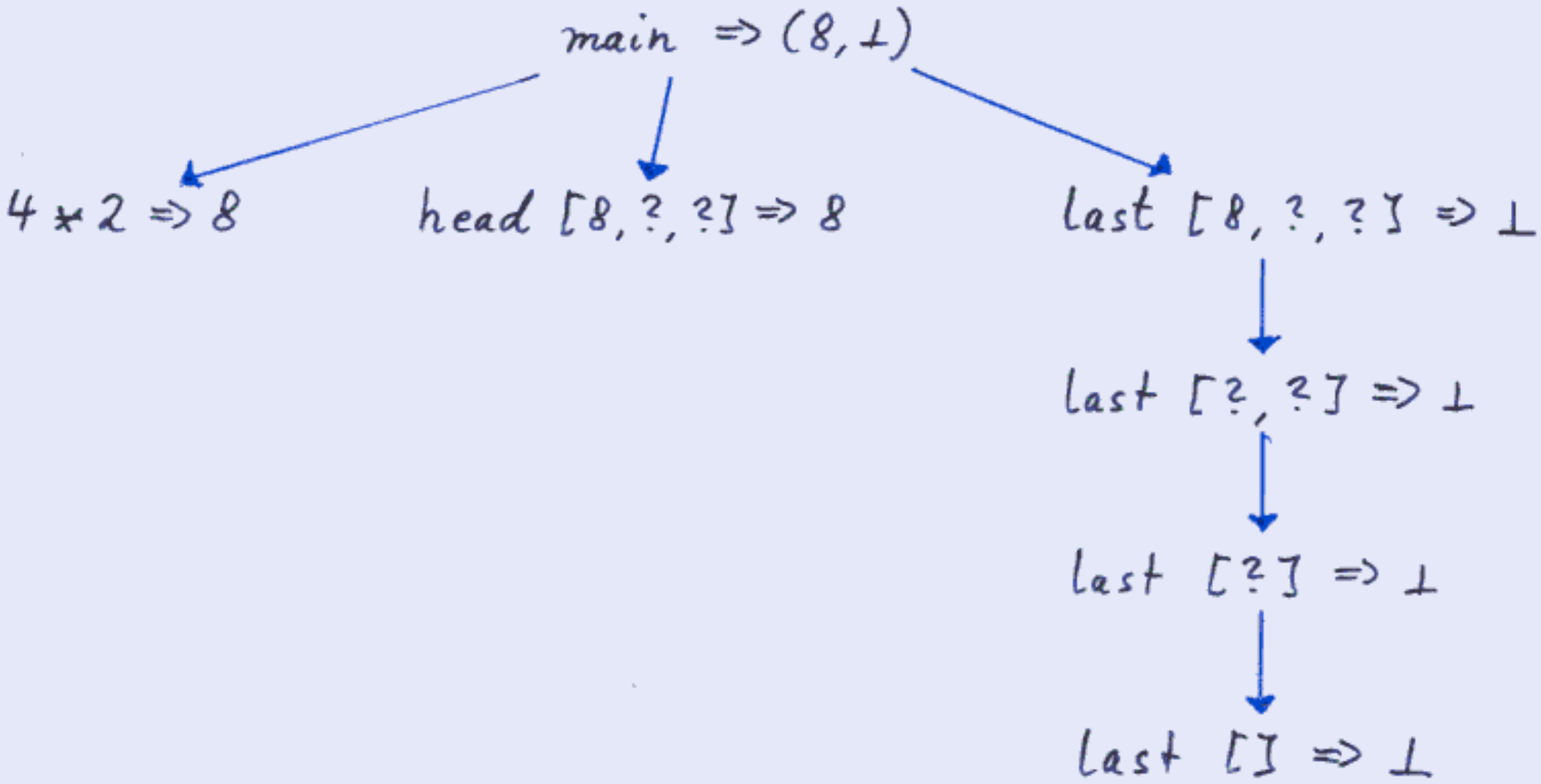
```
main =
   let xs = [4*2, 5/0, 3+6]
   in ( head xs, last xs)

head (x:xs) = x

last (x:xs) = last xs
last [x]    = x
```

⇒ pattern match failure

```
main ⇒ (8, ⊥)                 no
4 * 2 ⇒ 8                     yes
head [8, ?, ?] ⇒ 8            yes
last [8, ?, ?] ⇒ ⊥             no
last [?, ?] ⇒ ⊥                no
Last [?] ⇒ ⊥                   no
last [] ⇒ ⊥                    yes
```

Erroneous reduction: last [?] ⇒ ⊥

Freja's Trace



Wrong Subexpressions

```
translateStatement
  (TableImp
    (newTableFunction
      where
        newIndex = "y",
        newEntry = 2,
        oldTableFunction = newTableFunction
                           where
                             newIndex = "x",
                             newEntry = 1,
                             oldTableFunction = implTableEmpty ))
  ?
  (Assignment "x" (Compound (Variable "x") Minus (Variable "y")))
⇒
([Lod 1, Lod 1, Sb, Sto 1], 4)
```

Hood

Haskell library

```
import Observe

main = print0 $
  let xs = [4*2, 5/0, 3+6]
  in (head xs
     ,(last (observe "arg" xs))

head (x:xs) = x

last = observe "last" last'

last' (x:xs)  = last' xs
last' [x]     = x
```

```
arg:

    _ : _ : _ : []


last:

   { \(_:_:_:[]) -> ⊥
   , \(_:_:[]) -> ⊥
   , \(_:[]) -> ⊥
   , \[] -> ⊥
   ,
   }
```

Modification of a Program for Hood

- instances of Observable for own data types
- observation not just simple annotation
  (function, infix operator)
- new errors may be introduced
+ may be left in program

Free Variables of Locally Defined Functions

Freja:

```
tableRead
  "y"
  (TableImp
    (newTableFunction
      where
        newIndex = "x",
        newEntry = 1
        oldTableFunction = implTableEmpty))
⇒
Just 1
```

Hat:

```
tableRead "y" (TableImp newTableFunction)
↓ tableInsert "x" 1 (TableImp implTableEmpty)
```

Hood:

```
-- moveval
{\8 -> Draw}
{\8 -> Win}
```

Conclusions from Comparison

- all systems useful for debugging
- Erja's trace might be constructed from Hat's trace
- Integration of Erja and Hat feasible
- Hood is rather different

Summary of Hat

- phases
  1. collection of information during computation
  2. browsing of information

- usage
  - start at output / error message / interruption point
  - go backwards through viewing parent redexes
  - inspect arbitrary subexpressions
  - jump to source code positions

- implementation by program transformation
- trusting reduces trace size

Future Work

Implementation

- incrementally archive trace to file
- handle full Haskell 98
- portable variant

General

- how handle I/O, continuation passing style, ...
- strategies for locating errors
- how „understand" a program?

http://www.cs.york.ac.uk/fp/ART