Tracing and Debugging Haskell Programs with Hat

Alaf Chitil

University of York

Hat: Ian Sparud, Colin Runciman, Malcolm Wallace
ROPAC, Canon Research: 1996-97      EPSRC: 2000-02
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

evaluate 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

\[
\begin{align*}
\text{reverse } xs &= \text{rev } xs \ [ ] \\
\text{rev } [ ] \ ys &= ys \\
\text{rev } (x:xs) \ ys &= \text{rev } xs \ ys \\
\text{last } xs &= \text{head } (\text{reverse } xs) \\
\text{main} &= \text{let } xs = \lnot \text{True, } \lnot \text{False}\r
\text{ in } \text{print } (\text{head } xs, \text{last } xs) \\
(\text{False,} \\
\text{Error: head of empty list})
\end{align*}
\]
Guards, Case & If

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

print “eh”
| True < False < insert ‘h’ “e”
| True < insert ‘e’ “l”
| ‘h’ <= ‘e’
| insert ‘h’ “e”

sort = foldr insert []
main = print (sort “hello”)

eh
Shared Constants

primes = sieve \{ 2 \ldots 7 \}

first primes \ n = \text{take } n \text{ primes}

main = \text{print} \ (\text{first primes} \ 5) \Rightarrow \text{print primes}
Implementation by Program Transformation

input

Haskell program

output

trace

input

Haskell program

output
Expressions wrapped with Traces

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

data R a = R a Trace

(Bool, Bool) \Rightarrow R (R Bool, R Bool)

(True, False) \Rightarrow

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 = ... 1 Sat Trace Trace

not False \Rightarrow

let n = Name Root "not"
f = Name Root "False"
R \triangleright vt = ap_1 (R not n) (R False f)
in R \triangleright (Sat vt (Ap Root [n, f]))
Trust reduces trace size

- reduce information
- reduce memory consumption

```
length [1, 2, 3]
```

- trust definition of a function
- see only application and result
Incremental Archiving of Trace to File

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

\[ \Rightarrow \]

data Trace = T Filepointer
  mk Ap
  mk Name
  makeRoot
  makeSat
Freja

compiler for a subset of Haskell

\[
\begin{align*}
\text{main} &= \\
&\text{let } xs = [4 \times 2, 5/0, 3+6] \\
&\text{in (head } xs, \text{ last } xs) \\
\text{head (x:xs)} &= x \\
\text{last (x:xs)} &= \text{last } xs \\
\text{last [x]} &= x \\
\Rightarrow \text{ pattern match failure}
\end{align*}
\]

\[
\begin{align*}
\text{main} &\Rightarrow (8, \bot) \\
4 \times 2 &\Rightarrow 8 \\
\text{head } [8, ?, ?] &\Rightarrow 8 \\
\text{last } [8, ?, ?] &\Rightarrow \bot \\
\text{last } [?, ?] &\Rightarrow \bot \\
\text{last } [?] &\Rightarrow \bot \\
\text{last } [[] &\Rightarrow \bot \\
\text{Erroneous reduction: last } [[] &\Rightarrow \bot
\end{align*}
\]
Freya's Trace

main \Rightarrow (8, \bot)

4 \times 2 \Rightarrow 8

head [8, ?, ?, ?] \Rightarrow 8

last [8, ?, ?, ?] \Rightarrow \bot

last [?, ?, ?] \Rightarrow \bot

last [? ?] \Rightarrow \bot

last [?] \Rightarrow \bot
Wrong Subexpressions

translate Statement
  { Table Imp
    ( new Table Function
      where
      new Index = "y",
      new Entry = 2,
      old Table Function = new Table Function
      where
      new Index = "x",
      new Entry = 1,
      old Table Function = impl( Table Empty ) )

? ( Assignment "x" ( Compound ( Variable "x" ) Minus ( Variable "y" ) ) )

⇒ ( [ Lod 1, Lod 1, Sb, Sto 1 1 1 , 4 )


import Observe

main = printO $ 
  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
Modification of a Program for Flood

- instances of Observable for own data types
- observation not just simple annotation (function, inflé operator)
- new errors may be introduced
+ may be left in program
Free Variables of Locally Defined Functions

Freja:

```hs
tableRead "y"
  (TableImp
    (new Table Function
      where
      newIndex = "x",
      new Entry = 1
    old Table Function = impl(Table Empty)))
=>
  Just 1
```

Flatt:

```hs
(table Read "y" (TableImp new Table Function))
< table Insert "x" 1 (TableImp impl Table Empty)
```

Hool:

```hs
-- more eval
{ \ 8 -> Draw
{ \ 8 -> Win
```
Conclusions from Comparison

- All systems useful for debugging
- Teforej's trace might be constructed from that's trace
- Integration of Teforej and that 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