Debugging Functional Programs

Olaf Chitil
Partially supported by EPSRC grant EP/C516605/1

University of Kent
April 2009
A Faulty Haskell Program

```
main = putStrLn (sort "sort")

sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x > y then y:(insert x ys) else x:ys
```

Output: os

**Observable faulty behaviour:**

- **wrong result**
- abortion with run-time error
- non-termination
Conventional Debugging Methods

- The print / logging method: Add print statements to program.
- A stepping debugger such as the Data Display Debugger (DDD)
Conventional methods are ill-suited for non-strict functional languages.

New, more powerful methods can take advantage of properties of purely functional languages.
Haskell: A Non-Strict Purely Functional Programming Language

- **Non-strict** function: it has a well-defined result even when (parts of) arguments are unknown or ill-defined.
- **Purely** functional: an expression only denotes a value, no state transformation.

**Properties:**

- Rich but simple equational program algebra.
  
  \[ \text{map } f \ . \ \text{map } g = \text{map } (f \ . \ g) \]

- Can evaluate function arguments in any order (or not at all).
  
  \[ f (g \ 3 \ 4) \ (h \ 1 \ 2) \ (i \ 5 \ (j \ 3 \ 9) \ (k \ 4)) \]

- Enables programming with recursive values, infinite data structures and efficient data-oriented programming.
  
  \[ \text{pExp} = \text{pChar } '(' \ >> \ \text{pExp} \ >> \ \text{pChar } '+' \ >> \ \text{pExp} \ >> \ \text{pChar } ')')' \]

  \[ \text{factorial } n = \text{product } [1..n] \]
elem :: Int -> [Int] -> Bool

\[
\text{elem } x \text{ } \text{xs} = \text{or} \left( \text{map } (==x) \text{ } \text{xs} \right)
\]

\[
\text{elem 42 [1..]}
\]
Evaluation of an expression

\[
\text{elem} :: \text{Int} \to \left[ \text{Int} \right] \to \text{Bool} \\
\text{elem} \ x \ \text{xs} = \text{or} \ (\text{map} \ (==x) \ \text{xs})
\]

\[
\text{elem} \ 42 \ [1..] \\
\leadsto \text{or} \ (\text{map} \ (== \ 42) \ [1..])
\]
Evaluation of an expression

elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)

\[
\text{elem 42 [1..]}
\]
\[
\leadsto \text{or (map (== 42) [1..])}
\]
\[
\leadsto \text{or (map (== 42) (1:[2..]))}
\]
Evaluation of an expression

elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)

elem 42 [1..]

⇝ or (map (== 42) [1..])
⇝ or (map (== 42) (1:[2..]))
⇝ or (False : map (== 42) [2..])
Evaluation of an expression

elem :: Int -> [Int] -> Bool

elem x xs = or (map (==x) xs)

elem 42 [1..]

\[ \Rightarrow or \ (map \ (== \ 42) \ [1..]) \]

\[ \Rightarrow or \ (map \ (== \ 42) \ (1:2..)) \]

\[ \Rightarrow or \ (False : \ map \ (== \ 42) \ [2..]) \]

\[ \Rightarrow or \ (map \ (== \ 42) \ [2..]) \]
Evaluation of an expression

elem :: Int -> [Int] -> Bool

elem x xs = or (map (==x) xs)

elem 42 [1..]

\[ \Rightarrow \text{or} \left( \text{map} \ (== \ 42) \ [1..] \right) \]

\[ \Rightarrow \text{or} \left( \text{map} \ (== \ 42) \ (1:\[2..]) \right) \]

\[ \Rightarrow \text{or} \left( \text{False} : \text{map} \ (== \ 42) \ [2..] \right) \]

\[ \Rightarrow \text{or} \left( \text{map} \ (== \ 42) \ [2..] \right) \]

\[ \Rightarrow \text{or} \left( \text{map} \ (== \ 42) \ (2:\[3..]) \right) \]
Evaluation of an expression

elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)

elem 42 [1..]
⇝ or (map (== 42) [1..])
⇝ or (map (== 42) (1:[2..]))
⇝ or (False : map (== 42) [2..])
⇝ or (map (== 42) [2..])
⇝ or (map (== 42) (2:[3..]))
⇝ or (False : map (== 42) [3..])
Evaluation of an expression

elem :: Int -> [Int] -> Bool

elem x xs = or (map (==x) xs)

\[
\begin{align*}
\text{elem 42 [1..]} \\
& \Rightarrow \text{or (map (== 42) [1..])} \\
& \Rightarrow \text{or (map (== 42) (1:[2..]))} \\
& \Rightarrow \text{or (False : map (== 42) [2..])} \\
& \Rightarrow \text{or (map (== 42) [2..])} \\
& \Rightarrow \text{or (map (== 42) (2:[3..]))} \\
& \Rightarrow \text{or (False : map (== 42) [3..])} \\
& \Rightarrow \text{or (map (== 42) [3..])}
\end{align*}
\]
Evaluation of an expression

elem :: Int -> [Int] -> Bool

\[
\text{elem } x \text{ } x\text{s } = \text{or (map (==x) } \text{x}\text{s)}
\]

\[
\text{elem } 42 \text{ [1..]}
\Rightarrow \text{or (map (== } 42 \text{ [1..])}
\Rightarrow \text{or (map (== } 42 \text{ (1:[2..]))}
\Rightarrow \text{or (False : map (== } 42 \text{ [2..])}
\Rightarrow \text{or (map (== } 42 \text{ [2..])}
\Rightarrow \text{or (map (== } 42 \text{ (2:[3..]))}
\Rightarrow \text{or (False : map (== } 42 \text{ [3..])}
\Rightarrow \text{or (map (== } 42 \text{ [3..])}
\]
elem :: Int -> [Int] -> Bool

\[\text{elem } x \text{ xs} = \text{or } (\text{map } (==x) \text{ xs})\]

\[
\begin{align*}
\text{elem } 42 \ [1..] \\
\rightarrow & \text{or } (\text{map } (== 42) \ [1..]) \\
\rightarrow & \text{or } (\text{map } (== 42) \ (1:[2..])) \\
\rightarrow & \text{or } (\text{False : map } (== 42) \ [2..]) \\
\rightarrow & \text{or } (\text{map } (== 42) \ [2..]) \\
\rightarrow & \text{or } (\text{map } (== 42) \ (2:[3..])) \\
\rightarrow & \text{or } (\text{False : map } (== 42) \ [3..]) \\
\rightarrow & \text{or } (\text{map } (== 42) \ [3..]) \\
\rightarrow & \text{True}
\end{align*}
\]

Here reduction steps for map and or are skipped.
Why stepping doesn’t work

- No stepping through sequence of statements in source code.
- Complex evaluation order.
- Run-time stack unrelated to static function call structure.
- Unevaluated subexpressions large and hard to read.
Why Printing doesn’t work

Impure function \( \text{traceShow} :: \text{String} \rightarrow \text{[Int]} \rightarrow \text{[Int]} \)

\[ \text{insert} :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{[Int]} \]

\[ \text{insert } x \; [] = [x] \]

\[ \text{insert } x \; (y:ys) = \]

\[ \begin{align*}
& \text{if } x > y \text{ then } y : \text{(traceShow "}" (insert x ys))} \\
& \text{else } x:y:ys
\end{align*} \]

\[ \text{main} = \text{print} \; \text{(take 5 \; (insert 4 \; [1..]))} \]

Output:

\[ [1>2>3>[4,4,5,6,7,8,9,10,11,\ldots] \]

- output mixed up
- non-termination \( \Rightarrow \) observation changes behaviour
Properties of Functional Languages

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics

- An expression denotes only a value
  - independent evaluation of subexpressions
    \[ f (g \ 3 \ 4) (h \ 1 \ 2) (i \ 5) (j \ 3 \ 9 \ 3) \]
Properties of Functional Languages

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics

- An expression denotes only a value
  - independent evaluation of subexpressions
  \[ f(g\ 3\ 4)\ (h\ 1\ 2)\ (i\ 5)\ (j\ 3\ 9\ 3) \]

Advantages for Debugging

- Many semantic models as potential basis.
- Simple and compositional semantics.
- Freedom from sequentiality of computation.
Outline

1 Two-Phase Tracing
2 Views of Computation
   - Observation of Functions
   - Algorithmic Debugging
   - Source-based Free Navigation
   - Program Slicing
   - Call Stack
   - Redex Trails
   - Animation
   - ...
   - Trusting
   - New Views
3 A Theory of Tracing
4 Summary
Liberates from time arrow of computation.
Two-Phase Tracing

Liberates from time arrow of computation.

Trace stored in
- Memory.
- File.
- Generated on demand by reexecution.
Two-Phase Tracing

Liberates from time arrow of computation.

Trace stored in
- Memory.
- File.
- Generated on demand by reexecution.

Trace Generation
- Program annotations + library.
- Program transformation.
- Modified abstract machine.
Multi-View Tracer

For Haskell 98 + some extensions.

Developed by Colin Runciman, Jan Sparud, Malcolm Wallace, Olaf Chitil, Thorsten Brehm, Tom Davie, Tom Shackell, ...
main = putStrLn (sort "sort")

sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys

Output:

os
Observation of function sort:

\begin{align*}
\text{sort } "\text{sort}" &= "\text{os}" \\
\text{sort } "\text{ort}" &= "\text{o}" \\
\text{sort } "\text{rt}" &= "\text{r}" \\
\text{sort } "\text{t}" &= "\text{t}" \\
\text{sort } "" &= "" \\
\end{align*}

Observation of function insert:

\begin{align*}
\text{insert } 's' "\text{o}" &= "\text{os}" \\
\text{insert } 's' "" &= "\text{s}" \\
\text{insert } 'o' "\text{r}" &= "\text{o}" \\
\text{insert } 'r' "\text{t}" &= "\text{r}" \\
\text{insert } 't' "" &= "\text{t}" \\
\end{align*}
Haskell Object Observation Debugger (Hood) by Andy Gill.
- A library.
- Programmer annotates expressions of interest.
- Annotated expressions are traced during computation.
- The print method for the lazy functional programmer.

Observation of functions most useful.

Relates to denotational semantics.

\[
\text{insert 3 (1:2:3:4:_)} = 1:2:3:4: _ \\
\text{insert 3 (2:3:4:_)} = 2:3:4: _ \\
\text{insert 3 (3:4:_)} = 3:4: _
\]
sort "sort" = "os" ?
n
insert 's' "o" = "os" ?
y
sort "ort" = "o" ?
n
insert 'o' "r" = "o" ?
n
Bug identified:

"Insert.hs":8-9:
insert x [] = [x]
insert x (y:ys) = if x > y then y:(insert x ys) else x:ys
main = {IO}

sort "sort" = "os"

putStrLn "os" = {IO}

sort "ort" = "o"

insert 's' "o" = "os"

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

'o' > 'r' = False

insert 'o' "" = "o"

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

insert 't' "" = "t"

sort "" = ""

'" > 't' = False
main = {IO}  

sort "sort" = "os"  

sort "ort" = "o"  

insert 's' "o" = "os"  

's' > 'o' = True  

insert 's' "" = "s"  

sort "rt" = "r"  

insert 'o' "r" = "o"  

'o' > 'r' = False  

sort "t" = "t"  

insert 'r' "t" = "r"  

' o' > ' r' = False  

sort "" = ""  

insert 't' "" = "t"  

'r' > 't' = False  

putStrLn "os" = {IO}
The Computation Tree

main = {IO}

sort "sort" = "os" × putStrLn "os" = {IO}

sort "ort" = "o"

insert 's' "o" = "os"

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

'o' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

sort "" = ""
The Computation Tree

main = \{\text{IO}\}

\text{sort} "sort" = "os"

\text{sort} "ort" = "o"

\text{sort} "rt" = "r"

\text{sort} "t" = "t"

\text{sort} "" = ""

\text{putStrLn} "os" = \{\text{IO}\}

insert 's' "o" = "os" √

's' > 'o' = True

insert 's' "" = "s"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

'\text{insert 'o'}' > 'r' = \text{False}

'\text{insert 'r'}' > 't' = \text{False}

'\text{insert 't'}' > 't' = \text{False}

\text{insert 't'} "" = "t"
The Computation Tree

main = \{IO\}

sort "sort" = "os" × putStrLn "os" = \{IO\}

sort "ort" = "o" × insert 's' "o" = "os" √

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

sort "t" = "t"

insert 'r' "t" = "r"

'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (University of Kent)
The Computation Tree

main = \{IO\}

sort "sort" = "os" \times

putStrLn "os" = \{IO\}

sort "ort" = "o" \times

insert 's' "o" = "os" \checkmark

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o" \times

insert 'r' "t" = "r"

'o' > 'r' = False

sort "t" = "t"

insert 't' "" = "t"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"
Fault located!

main = putStrLn (sort "sort")

sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x:ys

Faulty computation: insert 'o' "r" = "o"
- Shapiro for Prolog, 1983.

- Correctness of tree node according to intended semantics.
- Incorrect node whose children are all correct is faulty.
- Each node relates to (part of) a function definition.

- Relates to natural, big-step semantics.
Higher-Order Insertion Sort

main :: String
main = sort "sort"

sort :: Ord a => [a] -> [a]
sort = foldr insert []

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x:ys
main = "os"

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

insert 't' "" = "t"

's' > 'o' = True

'o' > 'r' = False

'r' > 't' = False

insert 's' "" = "s"
Higher-Order Algorithmic Debugging

main = "os" ×

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

insert 's' "" = "s"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

insert 't' "" = "t"

's' > 'o' = True

'o' > 'r' = False

'r' > 't' = False
Higher-Order Algorithmic Debugging

main = "os"

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

insert 't' "" = "t"

insert 's' "" = "s"

's' > 'o' = True

'o' > 'r' = False

'r' > 't' = False

Olaf Chitil (University of Kent) Debugging Functional Programs April 2009 33 / 60
Higher-Order Algorithmic Debugging

\[ \text{main} = "os" \times \]

\[ \text{foldr insert [] } "\text{sort}" = "os" \times \]

\[ \text{sort} = \text{foldr insert []} \checkmark \]

\[ \text{foldr insert [] } "\text{ort}" = "o" \]

\[ \text{insert 's' } "\text{o}" = "os" \]

\[ 's' > 'o' = \text{True} \]

\[ \text{insert 's' ''} = "s" \]

\[ \text{foldr insert [] } "\text{rt}" = "r" \]

\[ 'o' > 'r' = \text{False} \]

\[ \text{foldr insert [] } "\text{t}" = "t" \]

\[ \text{insert 'r' } "\text{t}" = "r" \]

\[ 'r' > 't' = \text{False} \]

\[ \text{foldr insert [] } "\text{t}" = "t" \]

\[ \text{insert 't' ''} = "t" \]

\[ \text{foldr insert [] } "\text{t}" = "t" \]

\[ \text{insert 't' ''} = "t" \]
Higher-Order Algorithmic Debugging

main = "os" ×

foldr insert [] "sort" = "os" ×
sort = foldr insert [] √

foldr insert [] "ort" = "o"

insert 's' "o" = "os" √

's' > 'o' = True

insert 's' "" = "s"

foldr insert [] "rt" = "r"

insert 'o' "r" = "o"

foldr insert [] "t" = "t"

insert 'r' "t" = "r"

'o' > 'r' = False

foldr insert [] "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (University of Kent)  Debugging Functional Programs  April 2009  33 / 60
main = "os" ×

foldr insert [] "sort" = "os" ×
sort = foldr insert [] ✓

foldr insert [] "ort" = "o" ×
insert 's' "o" = "os" ✓

's' > 'o' = True
insert 's' "" = "s"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""
inser1t 't' "" = "t"

foldr insert [] "t" = "t"
inser1t 'r' "t" = "r"

'o' > 'r' = False

foldr insert [] "" = ""
inser1t 't' "" = "t"

'r' > 't' = False
Higher-Order Algorithmic Debugging

- $\text{main} = "os" \times$
- $\text{foldr insert[]} "sort" = "os" \times$
- $\text{sort} = \text{foldr insert[]} \checkmark$
- $\text{foldr insert[]} "ort" = "o" \times$
- $\text{insert 's' "o"} = "os" \checkmark$
- $'s' > 'o' = \text{True}$
- $\text{insert 's' ""} = "s"$
- $\text{foldr insert[]} "rt" = "r"$
- $\text{insert 'o' "r"} = "o" \times$
- $\text{foldr insert[]} "t" = "t"
- $\text{insert 'r' "t"} = "r"$
- $'o' > 'r' = \text{False}$
- $\text{foldr insert[]} "" = ""
- $\text{insert 't' ""} = "t"
- $'r' > 't' = \text{False}$

Olaf Chitil (University of Kent)
Debugging Functional Programs
April 2009 33 / 60
Higher-Order Algorithmic Debugging

main = "os" ×

foldr insert [] "sort" = "os" ×
sort = foldr insert [] √

foldr insert [] "ort" = "o" ×
insert 's' "o" = "os" √

's' > 'o' = True
insert 's' "" = "s"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"
insert 'r' "t" = "r"

foldr insert [] "" = ""
insert 't' "" = "t"

foldr insert [] "" = ""

'o' > 'r' = False
'r' > 't' = False
Higher-Order Algorithmic Debugging II

main = "os"

sort = {"sort" -> "os"}

foldr { ‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "sort" = "os"
foldr { ‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "ort" = "o"
foldr { ‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "rt" = "r"
foldr { ‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "t" = "t"
foldr { ‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "" = ""

insert ‘s’ "o" = "os"
insert ‘o’ "r" = "o"
insert ‘r’ "t" = "r"
insert ‘t’ "" = "t"

’s’>’o’ = True
insert ‘s’ "" = "s"
’o’>’r’ = False
’r’>’t’ = False
Higher-Order Algorithmic Debugging II

main = "os"

sort = {"sort" -> "os"}

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" } [] "sort" = "os"

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" } [] "ort" = "o"

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" } [] "rt" = "r"

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" } [] "t" = "t"

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" } [] "" = ""

insert 's' "o" = "os"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

insert 't' "" = "t"

's'>'o' = True

insert 's' "" = "s"

'o'>'r' = False

'r'>'t' = False

Olaf Chitil (University of Kent)

Debugging Functional Programs

April 2009 35 / 60
Higher-Order Algorithmic Debugging II

main = "os"

sort = {"sort" -> "os"}

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "sort" = "os"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "ort" = "o"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "rt" = "r"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "t" = "t"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "" = ""

insert 's' "o" = "os"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

insert 't' "" = "t"

's'>'o' = True

insert 's' "" = "s"

'o'>'r' = False

'r'>'t' = False

Olaf Chitil (University of Kent)
Debugging Functional Programs
April 2009 35 / 60
Higher-Order Algorithmic Debugging II

main = "os" ×
sort = {"sort" -> "os"}

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "sort" = "os" √

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "ort" = "o"

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "rt" = "r"

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "t" = "t"

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "" = ""

insert 's' "o" = "os" insert 'o' "r" = "o" insert 'r' "t" = "r" insert 't' "" = "t"

's'>'o' = True insert 's' "" = "s" 'o'>'r' = False 'r'>'t' = False
null
main = "os" ×

sort = {"sort" -> "os"}

foldr {‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "sort" = "os" ✓

foldr {‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "ort" = "o"

foldr {‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "rt" = "r"

foldr {‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "t" = "t"

foldr {‘s’ "o"->"os", ‘o’ "r"->"o", ‘r’ "t"->"r", ‘t’ ""->"t"} [] "" = ""

insert ‘s’ "o" = "os" ✓

insert ‘o’ "r" = "o"

insert ‘r’ "t" = "r" ✓

insert ‘t’ "" = "t"

‘s’>’o’ = True ✓

insert ‘s’ "" = "s"

‘o’>’r’ = False

‘r’>’t’ = False ✓
Source-based Free Navigation and Program Slicing

==== Hat-Explore 2.00 ==== Call 2/2 =======================

1. `main = {IO}

2. `sort "sort" = "os"

3. `sort "ort" = "o"

--- Insert.hs ---- lines 5 to 10 -------------------------

```haskell
if x > y then y : insert x ys
else x : ys
```

`sort :: [Char] -> [Char]

`sort [] = []

`sort (x:xs) = insert x (sort xs)"
Program terminated with error:
   No match in pattern.
Virtual stack trace:
(Last.hs:6)    last’ []
(Last.hs:6)    last’ [ ]
(Last.hs:6)    last’ [_,_]  
(Last.hs:4)    last’ [8,_,_]
(unknown)      main
Output: ---------------------------------------------------------
os

Trail: ------- Insert.hs line: 10 col: 25 ---------------------
<- putStrLn "os"
<- insert 's' "o" | if True
<- insert 'o' "r" | if False
<- insert 'r' "t" | if False
<- insert 't' []
<- sort []

Go backwards from observed failure to fault.
Which redex created this expression?

Based on graph rewriting semantics of abstract machine.
Output: ----------------------------------------------------

Animation: --------------------------------------------------

-> sort "sort"
-> insert 's' ( sort "ort" )
-> insert 's' ( insert 'o' ( sort "rt" ) )
-> insert 's' ( insert 'o' ( insert 'r' ( sort "t" ) ) )
-> insert 's' ( insert 'o' ( insert 'r' "t" ) )
-> "os"
**Trust a module:** Do not trace functions in module.

- Smaller trace file.
- Avoid viewing distracting details.

\[ 4 + 7 = 11 \]
**T rusting**

**Trust a module:** Do not trace functions in module.

- Smaller trace file.
- Avoid viewing distracting details.

\[ 4 + 7 = 11 \]

A trusted function may call a non-trusted function:

\[
\text{map prime } [2,3,4,5] = [True,True,False,True]
\]
T rusting

**Trust a module:** Do not trace functions in module.
- Smaller trace file.
- Avoid viewing distracting details.

\[ 4 + 7 = 11 \]

A trusted function may call a non-trusted function:

```haskell
map prime [2,3,4,5] = [True,True,False,True]
```

**In future?**
- View-time trusting.
- Trusting of local definitions.
New Views

New Ideas

- Follow a value through computation.
New Views

New Ideas

- Follow a value through computation.

Combining Existing Views

- Can easily switch from one view to another.
- All-in-one tool = egg-laying wool-milk-sow?
- Exploring combination of algorithmic debugging and redex trails.
New Ideas

- Follow a value through computation.

Combining Existing Views

- Can easily switch from one view to another.
- All-in-one tool = egg-laying wool-milk-sow?
- Exploring combination of algorithmic debugging and redex trails.

Refining Existing Views

Algorithmic Debugging:

- Different Tree-Traversal Strategies.
- Heuristics.
Why a Theory of Tracing?

- Implementations of tracing tools ahead of theoretical results.
- Correctness of tools?
- Clear methodology for using them?
- Development of advanced features?
What is a Good Trace?

Program + input determine every detail of computation.
What is a Good Trace?

Program + input determine every detail of computation.
⇒ Trace gives efficient access to certain details of computation.
What is a Good Trace?

Program + input determine every detail of computation.
⇒ Trace gives efficient access to certain details of computation.

What is a computation? Semantics answers:

- Term rewriting: A sequence of expressions.
  \[ t_1 \to t_2 \to t_3 \to t_4 \to t_5 \to \ldots \to t_n \]
- Natural semantics: A proof tree.
Start with expression `sort ('t':[])`
sort [] = []
sort (x:xs) = insert x (sort xs)

insert x [] = [x]
insert x (y:ys) = if x > y then y:(insert x ys) else x:ys
sort [] = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys
The Trace: Simple Graph Rewriting

\[
\begin{align*}
\text{sort } [] &= [] \\
\text{sort } (x:xs) &= \text{insert } x \ (\text{sort } xs) \\
\text{insert } x \ [] &= [x] \\
\text{insert } x \ (y:ys) &= \text{if } x > y \text{ then } y : (\text{insert } x \ ys) \text{ else } x : ys
\end{align*}
\]
New nodes for right-hand-side, connected via result pointer.
Only add to graph, never remove.
Sharing ensures compact representation.
The Node Naming Scheme

Aim
- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes
The Node Naming Scheme

Aim
- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes

Solution
- standard representation with node describing path from root
- path at creation time (sharing later)
- path independent of evaluation order
The Node Labels

node \( n \) := \{f, a, r\}^* 

label term \( T \) := \( a \) \hspace{2cm} \text{atom} 
\hspace{1cm} \text{|} \hspace{1cm} \text{nm} \hspace{1cm} \text{application of nodes} 
atom \( a \) := f | C | 42 \hspace{1cm} \text{defined variable, data constructor} 
\hspace{1cm} \ldots \hspace{1cm} \text{atomic literal, \ldots} 

Reduction edge implicitly given through existence of node.
Projections

- Reduction edge implicitly given through existence of node.
- Every redex should be parent of at least one node.
  (otherwise reduction unreachable from computation result)

\[
\begin{align*}
\text{True} \land x &= x \\
\text{not True} &= \text{False}
\end{align*}
\]
Projections

- Reduction edge implicitly given through existence of node.
- Every redex should be parent of at least one node.
  (otherwise reduction unreachable from computation result)

⇒ A projection requires an **indirection** as result.

True && x = x
not True = False

![Diagram of projection and indirection]

**Label term**

\[ T ::= a \mid nm \mid n \]

- **Atom**
  \[ a ::= x \mid C \mid 42 \mid \ldots \]

variable, data constructor, \ldots
Projections

- Reduction edge implicitly given through existence of node.
- Every redex should be parent of at least one node.
  (otherwise reduction unreachable from computation result)

⇒ A projection requires an **indirection** as result.

\[ \text{True} \land x = x \]

\[ \text{not True} = \text{False} \]

label term \( T := a \) atom

\[ \mid n \ m \] application of nodes

\[ \mid n \] indirection

atom \( a := x \mid C \mid 42 \mid \ldots \) variable, data constructor, \ldots
A trace $G$ for initial term $M$ and program $P$ is a partial function from nodes to term constructors, $G : n \mapsto T$, defined by

- The unshared graph representation of $M$, $\text{graph}_G(\varepsilon, M)$, is a trace.
- If $G$ is a trace and
  - $L = R$ an equation of the program $P$,
  - $\sigma$ a substitution replacing argument variables by nodes,
  - $\text{match}_G(n, L\sigma)$,
  - $nr \notin \text{dom}(G)$,
then $G \cup \text{graph}_G(nr, R\sigma)$ is a trace.

No evaluation order is fixed.
The Most Evaluated Form of a Node

A node represents many terms, in particular a most evaluated one.

**Definition**

\[ \text{mef}_G(\varepsilon) = (: \) 't' [] \]

\[ \text{mef}_G(n) = \text{mefT}_G(G([n]_G)) \]

\[ \text{mefT}_G(a) = a \]

\[ \text{mefT}_G(n) = \text{mef}_G(n) \]

\[ \text{mefT}_G(n m) = \text{mef}_G(n) \text{mef}_G(m) \]
Redexes and Big-Step Reductions

\[ \text{redex}_G(n) = \begin{cases} \text{mef}_G(m) \text{mef}_G(o) & \text{if } G(n) = m \circ o \\ a & \text{if } G(n) = a \end{cases} \]

\[ \text{bigstep}_G(n) = \text{redex}_G(n) = \text{mef}_G(n) \]
Every redex node $n$ yields a tree node $n$ labelled $\text{bigstep}_G(n)$.

Tree node $n$ is child of tree node $\text{parent}(n)$.

- $\text{parent}(nr) = n$
- $\text{parent}(nf) = \text{parent}(n)$
- $\text{parent}(na) = \text{parent}(n)$
- $\text{parent}(\varepsilon) = \text{undefined}$
Two-Phase Tracing.

Liberates from time arrow of computation.

There exist many useful different views of a computation.
- Observation of Functions
- Algorithmic Debugging
- Source-based Free Navigation
- Redex Trails
- ...

Semantics.
- Inspire views.
- Enable formulation and proof of properties.
- But do not answer all questions.

Still much to explore.