Debugging Functional Programs

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

University of Kent

12th March 2008
Even Functional Programs have Bugs!

Well-typed programs cannot go wrong.

Robin Milner

At least

strong type system $\implies$ cannot corrupt run-time system

But

- 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.
  \[ \text{pExp} = \text{pChar } '(' \gg \text{pExp} \gg \text{pChar } '+' \gg \text{pExp} \gg \text{pChar } ')')' \]
  \[ \text{factorial } n = \text{product } [1..n] \]
Evaluation of an expression

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

elem 42 [1..]
Evaluation of an expression

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

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

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

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

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

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

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

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

\[
\text{elem} 42 \ [1..] \\
\rightarrow \text{or} \ (\text{map} \ (== 42) \ [1..]) \\
\rightarrow \text{or} \ (\text{map} \ (== 42) \ (1:[2..])) \\
\rightarrow \text{or} \ (\text{False} : \text{map} \ (== 42) \ [2..]) \\
\rightarrow \text{or} \ (\text{map} \ (== 42) \ [2..]) \\
\rightarrow \text{or} \ (\text{map} \ (== 42) \ (2:[3..]))
\]
Evaluation of an expression

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

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

\[
\begin{align*}
\text{elem 42 [1..]} & \\
\implies \text{or (map (== 42) [1..])} & \\
\implies \text{or (map (== 42) (1:[2..]))} & \\
\implies \text{or (False : map (== 42) [2..])} & \\
\implies \text{or (map (== 42) [2..])} & \\
\implies \text{or (map (== 42) (2:[3..]))} & \\
\implies \text{or (False : map (== 42) [3..])}
\end{align*}
\]
elem :: Int -> [Int] -> Bool

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

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

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

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

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 \texttt{traceShow} :: \texttt{String} \rightarrow \texttt{[Int]} \rightarrow \texttt{[Int]}

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

\texttt{insert x [] = [x]}
\texttt{insert x (y:ys) =}
\hspace{1em} \texttt{if x > y then y : (traceShow ">" (insert x ys))}
\hspace{1em} \texttt{else x:y:ys}

\texttt{main = print (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.
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
Two-Phase Tracing

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, ...
Faulty Insertion Sort

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:

sort "sort" = "os"
sort "ort" = "o"
sort "rt" = "r"
sort "t" = "t"
sort "" = ""

Observation of function insert:

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

\[
\begin{align*}
\text{insert } 3 \ 1:2:3:4:_{\_} &= 1:2:3:4:_{\_} \\
\text{insert } 3 \ 2:3:4:_{\_} &= 2:3:4:_{\_} \\
\text{insert } 3 \ 3:4:_{\_} &= 3:4:_{\_} \\
\end{align*}
\]
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 = \{\text{IO}\}

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

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

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

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

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

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

\text{\text{insert ’s’ "o" = "os"}}

\text{’s’ > ’o’ = True}

\text{\text{insert ’s’ "" = "s"}}

\text{\text{insert ’o’ "r" = "o"}}

\text{\text{insert ’r’ "t" = "r"}}

\text{\text{insert ’t’ "" = "t"}}

\text{’o’ > ’r’ = False}

\text{’r’ > ’t’ = False}
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"

'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (University of Kent)
The Computation Tree

main = \{\text{IO}\}

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

\text{sort} "\text{ort}" = "o"
\text{insert} 's' "o" = "os"
's' > 'o' = True

\text{sort} "\text{rt}" = "r"
\text{insert} 'o' "r" = "o"
'o' > 'r' = False

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

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

\text{sort} "\text{rt}" = "r"
\text{insert} 'b' "\text{r}" = "\text{r}"

\text{sort} "\text{rt}" = "r"
\text{insert} 'b' "\text{r}" = "\text{r}"

\text{sort} "\text{rt}" = "r"
\text{insert} 'b' "\text{r}" = "\text{r}"
The Computation Tree

main = {IO}

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

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"

' o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

' r' > 't' = False

Olaf Chitil (University of Kent)
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"

'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (University of Kent)

Debugging Functional Programs
The Computation Tree

main = \{\text{IO}\}

\begin{itemize}
  \item sort "sort" = "os" \times
  \item sort "ort" = "o" \times
  \item sort "rt" = "r"
  \item sort "t" = "t"
  \item sort "" = ""
\end{itemize}

\begin{itemize}
  \item putStrLn "os" = \{\text{IO}\}
  \item insert 's' "o" = "os" \checkmark
  \item insert 'o' "r" = "o" \times
  \item insert 'r' "t" = "r"
  \item insert 't' "" = "t"
  \item insert 'r' "" = "t"
\end{itemize}

\begin{itemize}
  \item 's' > 'o' = True
  \item insert 's' "" = "s"
  \item 'o' > 'r' = False
  \item 'r' > 't' = False
\end{itemize}
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
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"

's' > 'o' = True

'o' > 'r' = False

'c' > 't' = False

insert 's' "" = "s"
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"

'" > 'o' = False

foldr insert [] "t" = "t"

insert 'r' "t" = "r"

'o' > 'r' = False

foldr insert [] "" = ""

insert 't' "" = "t"

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

main = "os"  

foldr insert [] "sort" = "os"

correct:

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

foldr insert [] "ort" = "o"

insert 's' "o" = "os"

's' > 'o' = True

foldr insert [] "rt" = "r"

insert 'o' "r" = "o"

'o' > 'r' = False

foldr insert [] "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

foldr insert [] "" = ""

insert 't' "" = "t"

'p' > 't' = False
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"

'0' > 'r' = False

foldr insert [] "t" = "t"

insert 'r' "t" = "r"

'0' > 'r' = False

foldr insert [] "" = ""

insert 't' "" = "t"

'r' > 't' = False

foldr insert [] "" = ""
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  12th March 2008  32 / 59
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
Higher-Order Algorithmic Debugging

main = "os" ×

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

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

's' > 'o' = True
correct

foldr insert [] "rt" = "r"

insert 'o' "r" = "o" ×

foldr insert [] "t" = "t"
insert 'r' "t" = "r"
'o' > 'r' = False

foldr insert [] "" = ""

insert 't' "" = "t"

foldr insert [] "" = ""

'r' > 't' = False

Olaf Chitil (University of Kent)
Debugging Functional Programs
12th March 2008
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
main = "os"

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

foldr {"s" -> "os", "o" -> "o", "r" -> "r", "t" -> "t"} [] "sort" = "os"

foldr {"s" -> "os", "o" -> "o", "r" -> "r", "t" -> "t"} [] "ort" = "o"

foldr {"s" -> "os", "o" -> "o", "r" -> "r", "t" -> "t"} [] "rt" = "r"

foldr {"s" -> "os", "o" -> "o", "r" -> "r", "t" -> "t"} [] "t" = "t"

foldr {"s" -> "os", "o" -> "o", "r" -> "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
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"
```

- `\text{\textprime{}s\textprime{}>'o'} = \text{True}`
- `insert 's' "" = "s"
- `\text{o\textprime{}>'r'} = \text{False}`
- `\text{\textprime{r\textprime{>'t'}} = \text{False}}`
main = "os"

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

code:

\[
\text{foldr}\{\text{'s'}\ "o"\Rightarrow\"os", \text{'o'}\ "r"\Rightarrow\"o", \text{'r'}\ "t"\Rightarrow\"r", \text{'t'}\ ""\Rightarrow\"t\}\\ [] \text{"sort" = "os"}
\]

\[
\text{foldr}\{\text{'s'}\ "o"\Rightarrow\"os", \text{'o'}\ "r"\Rightarrow\"o", \text{'r'}\ "t"\Rightarrow\"r", \text{'t'}\ ""\Rightarrow\"t\}\\ [] \text{"ort" = "o"}
\]

\[
\text{foldr}\{\text{'s'}\ "o"\Rightarrow\"os", \text{'o'}\ "r"\Rightarrow\"o", \text{'r'}\ "t"\Rightarrow\"r", \text{'t'}\ ""\Rightarrow\"t\}\\ [] \text{"rt" = "r"}
\]

\[
\text{foldr}\{\text{'s'}\ "o"\Rightarrow\"os", \text{'o'}\ "r"\Rightarrow\"o", \text{'r'}\ "t"\Rightarrow\"r", \text{'t'}\ ""\Rightarrow\"t\}\\ [] \text{"t" = "t"}
\]

\[
\text{foldr}\{\text{'s'}\ "o"\Rightarrow\"os", \text{'o'}\ "r"\Rightarrow\"o", \text{'r'}\ "t"\Rightarrow\"r", \text{'t'}\ ""\Rightarrow\"t\}\\ [] \text{"" = ""}
\]

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

\[
\text{'s'>'o' = True}
\text{insert 's' "" = "s"}
\text{'o'>'r' = False}
\text{'r'>'t' = False}
\]

\[
\text{valid: }
\]
1. \texttt{main = \{IO\}}

2. \texttt{sort "sort" = "os"}

3. \texttt{sort "ort" = "o"}

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

\begin{verbatim}
if x > y then y : insert x ys 
else x : ys
\end{verbatim}

\texttt{sort :: [Char] -> [Char]}

\texttt{sort [] = []}

\texttt{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
Redex Trails

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:

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

**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:

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

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

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.
  \[ t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5 \rightarrow \ldots \rightarrow t_n \]
- Natural semantics: A proof tree.
Start with expression \texttt{sort (’t’:[])}
The Trace: Simple Graph Rewriting

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

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
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{1cm} \text{atom}
\hspace{1cm} | \hspace{1cm} nm \hspace{1cm} \text{application of nodes}

atom \( a \) := f \mid C \mid 42 \mid \ldots \text{defined variable, data constructor}
\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)

\[
\text{True} \land \land x = x
\]
\[
\text{not True} = \text{False}
\]
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 with nodes and edges labeled with '&&', 'True', 'not', 'True', 'f', 'a', 'ff', 'fa', 'af', 'aa', 'ε', 'r', and '• • • • • r • • • • • • ε']

**Label term**

```
T ::= a
| n m

atom
```

```
atom a ::= x | C | 42 | ... variable, data constructor, ...
```
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.

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

![Diagram of expressions and nodes]

**Label term**

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

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.

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

Definition

\[ n \succ_G m \iff m = nr \lor G(n) = m \]
\[ [n]_G = m \iff n \succ_G^* m \land \nexists o. m \succ_G o \]

Definition

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

Redex $G(r) = \text{insert 't' []}$

Bigstep $G(r) = \text{insert 't' []} = (:) \text{ 't' []}$

**Definition**

For any redex node $n$, i.e., $nr \in \text{dom}(G)$

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

\[
\text{bigstep}_G(n) = \text{redex}_G(n) = \text{mef}_G(n)
\]
From Trace to Big-Step Computation Tree

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

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

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

**Examples**

- $\text{sort} \ ('t'::[]) = 't'::[]$
- $\text{sort} \ [] = []$
- $\text{insert} \ 't' \ [] = 't'::[]$
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.