# 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

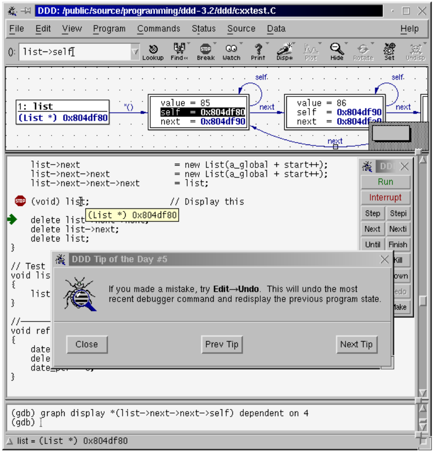
strong type system $\Longrightarrow$ cannot corrupt run-time system

### 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)



# Why Debug Functional Programs Differently?

- 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.

```
map f . map g = 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.

```
pExp = pChar '(' >> pExp >> pChar '+' >> pExp >> pChar ')'
```

```
factorial n = 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)
```

$\underline{\texttt{elem 42 [1..]}}$
$\rightsquigarrow$ `or (map (== 42)` $\underline{\texttt{[1..]}}$`)`

# Evaluation of an expression

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

$\underline{\texttt{elem 42 [1..]}}$
$\leadsto$ `or (map (== 42)` $\underline{\texttt{[1..]}}$`)`
$\leadsto$ `or (`$\underline{\texttt{map (== 42) (1:[2..])}}$`)`

# 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..]
 ⇝ or (map (== 42) [1..])
 ⇝ or (map (== 42) (1:[2..]))
 ⇝ or (False : map (== 42) [2..])
 ⇝ or (map (== 42) [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..])
 ⇝ or (map (== 42) [2..])
 ⇝ or (map (== 42) (2:[3..]))
```

# Evaluation of an expression

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

$$
\begin{array}{ll}
& \underline{\texttt{elem 42 [1..]}} \\
\rightsquigarrow & \texttt{or (map (== 42) }\underline{\texttt{[1..]}}\texttt{)} \\
\rightsquigarrow & \texttt{or (}\underline{\texttt{map (== 42) (1:[2..])}}\texttt{)} \\
\rightsquigarrow & \underline{\texttt{or (False : map (== 42) [2..])}} \\
\rightsquigarrow & \texttt{or (map (== 42) }\underline{\texttt{[2..]}}\texttt{)} \\
\rightsquigarrow & \texttt{or (}\underline{\texttt{map (== 42) (2:[3..])}}\texttt{)} \\
\rightsquigarrow & \underline{\texttt{or (False : map (== 42) [3..])}}
\end{array}
$$

# 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..])
```

# 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..])
 ⋮   ⋮
```

# 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 `traceShow :: String -> [Int] -> [Int]`

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys) =
  if x > y then y : (traceShow ">" (insert x ys))
           else x:y:ys
main = print (take 5 (insert 4 [1..]))
```

Output:

```
[1>[2>[3>[4,4,5,6,7,8,9,10,11,...
```

- output mixed up
- non-termination ⇒ 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)`

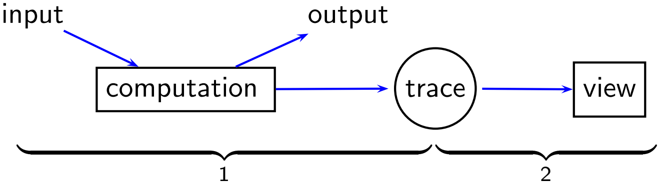
## Advantages for Debugging

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

# Outline

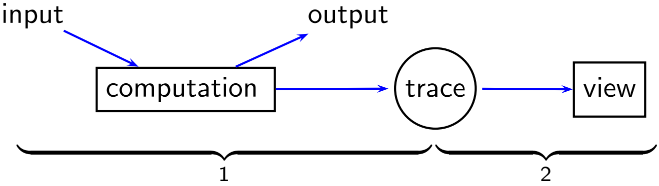

1. Two-Phase Tracing
2. Views of Computation
    - Observation of Functions
    - Algorithmic Debugging
    - Source-based Free Navigation
    - Program Slicing
    - Call Stack
    - Redex Trails
    - Animation
    - ...
    - Trusting
    - New Views
3. A Theory of Tracing
4. Summary

# Two-Phase Tracing




Liberates from time arrow of computation.

# Two-Phase Tracing

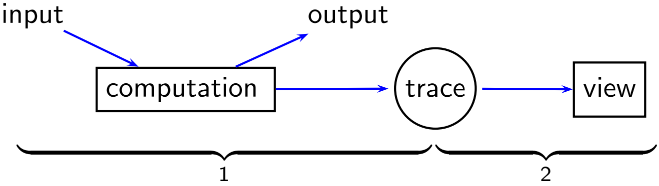



Liberates from time arrow of computation.

Trace stored in

- Memory.
- File.
- Generated on demand by reexecution.

# Two-Phase Tracing




Liberates from time arrow of computation.

### Trace stored in

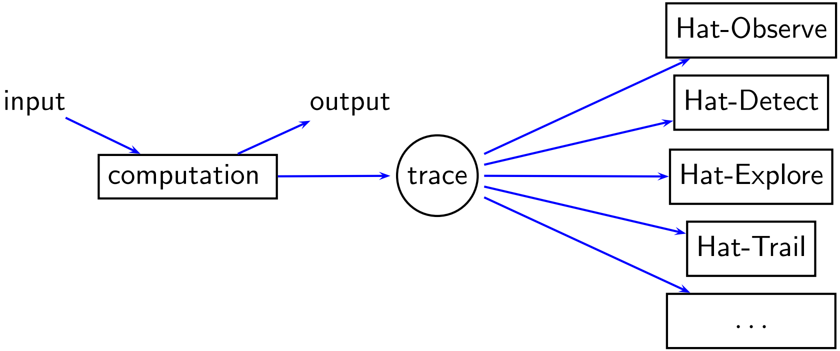
- Memory.
- File.
- Generated on demand by reexecution.

### Trace Generation

- Program annotations + library.
- Program transformation.
- Modified abstract machine.

# Hat

- 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 Expressions and Functions

# Observation of Expressions and Functions

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"
```

# Observation of Expressions and Functions

- 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.

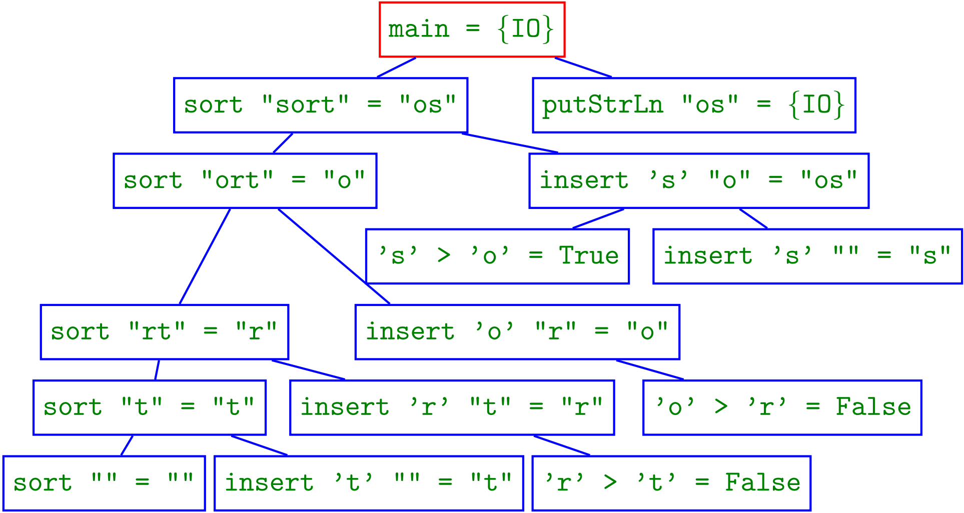
```
insert 3 (1:2:3:4:_) = 1:2:3:4:_
insert 3 (2:3:4:_) = 2:3:4:_
insert 3 (3:4:_) = 3:4:_
```

# Algorithmic Debugging

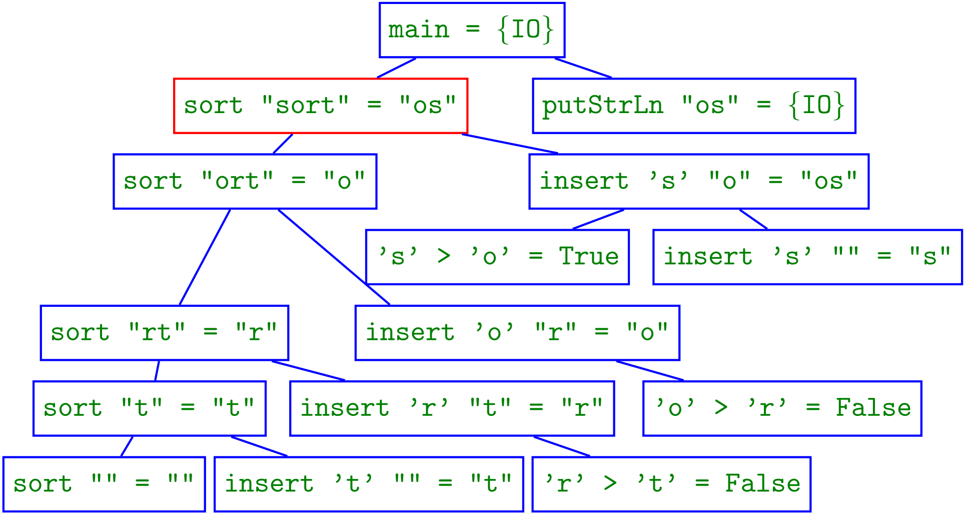
# Algorithmic Debugging

```
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
```

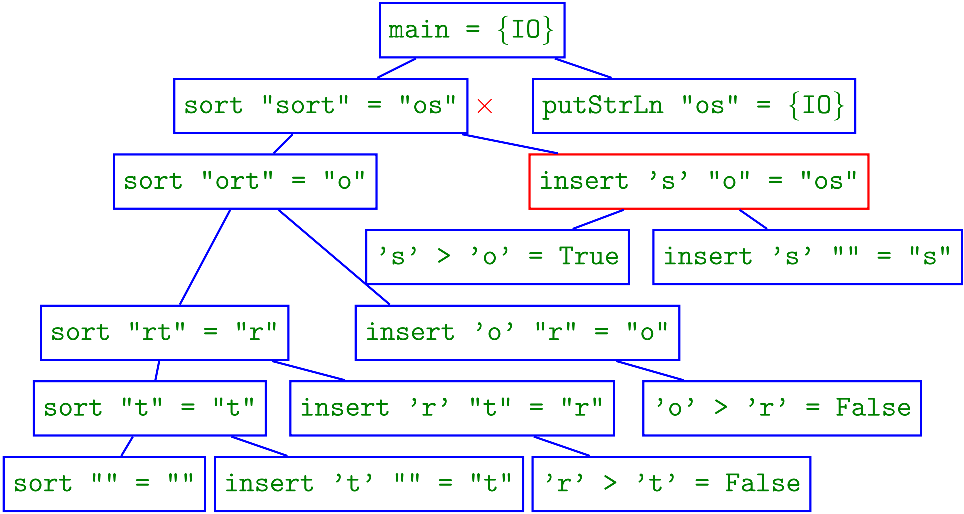
# The Computation Tree



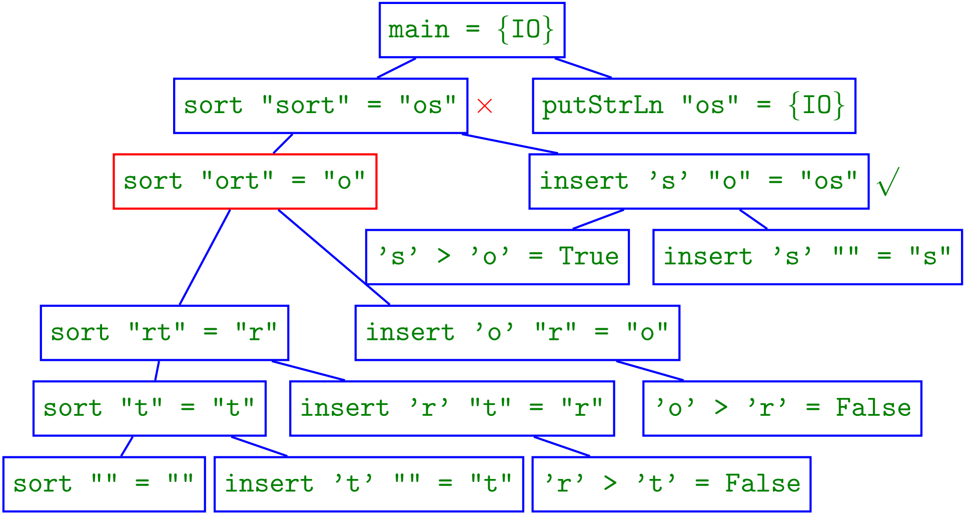
# The Computation Tree



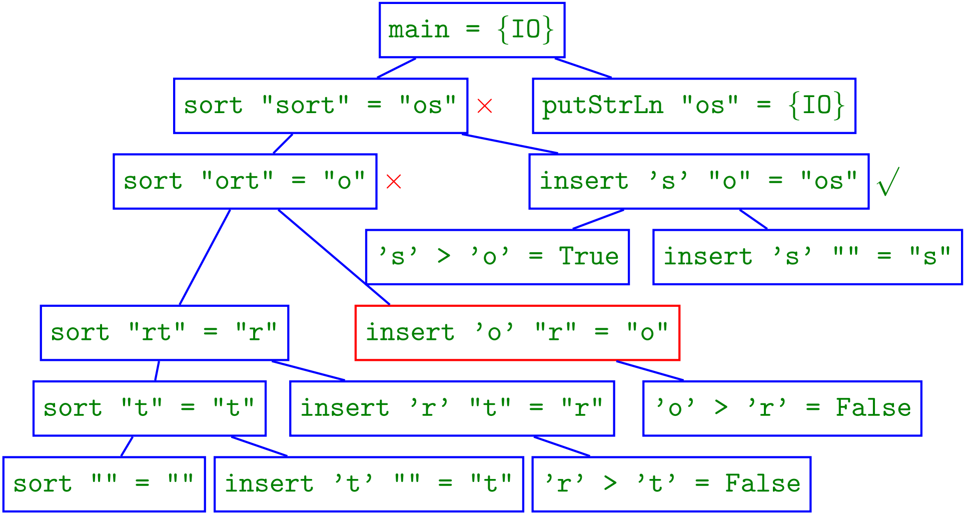
# The Computation Tree



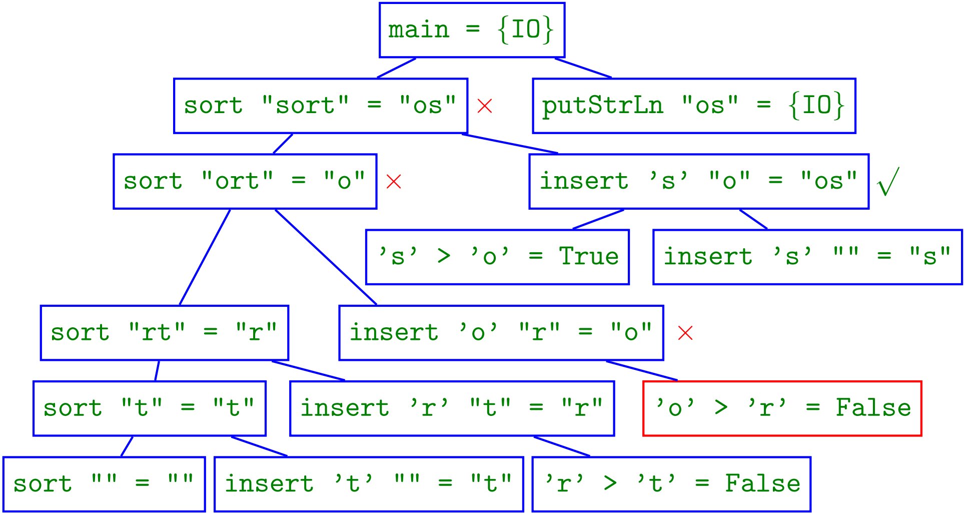
# The Computation Tree



# The Computation Tree



# The Computation Tree



# 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"`

# Algorithmic Debugging

- Shapiro for Prolog, 1983.
- Henrik Nilsson's Freija for lazy functional language, 1998.
- Bernie Pope's Buddha for Haskell, 2003.

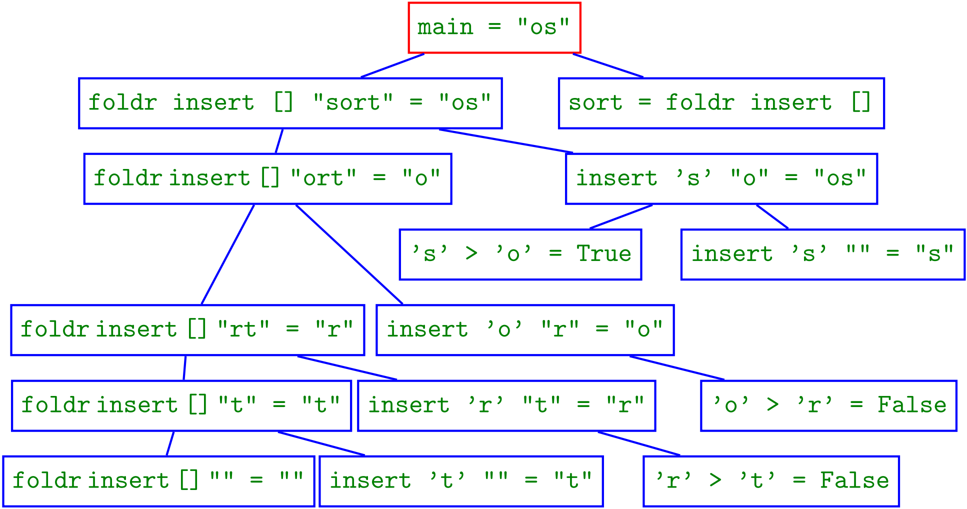
- Correctness of tree node according to intended semantics.
- Incorrect node whose children are all correct is faulty.
- Each node relates to (part of) a function definition.

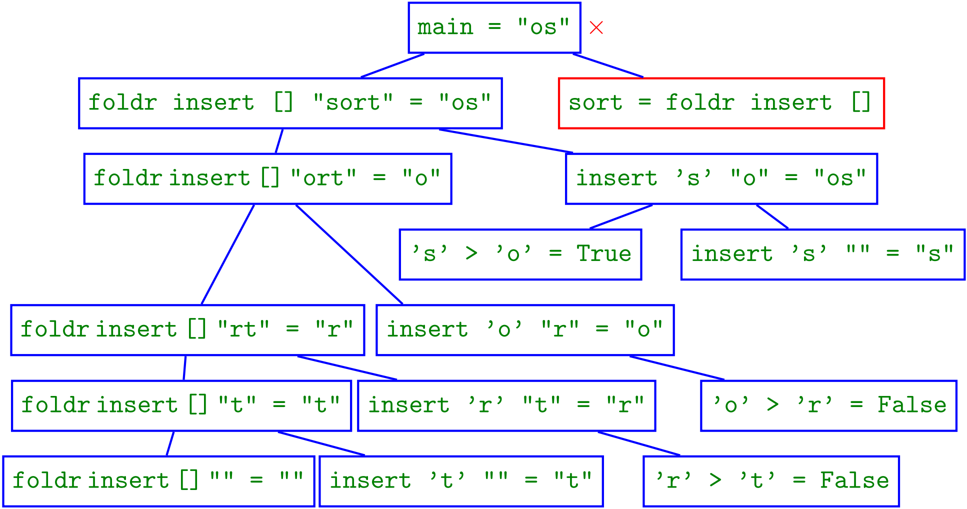
- Relates to natural, big-step semantics.

# Higher-Order Insertion Sort

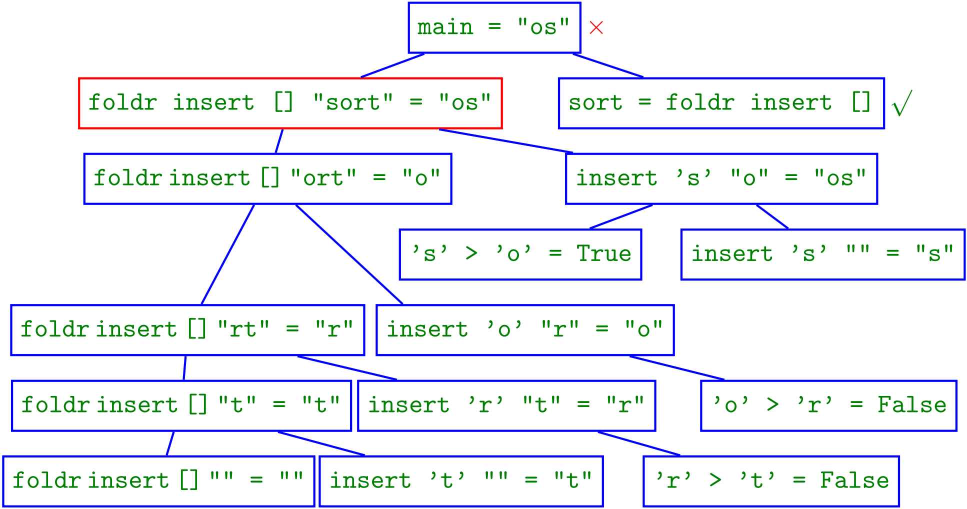
```
main :: String
main = sort "sort"

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
```

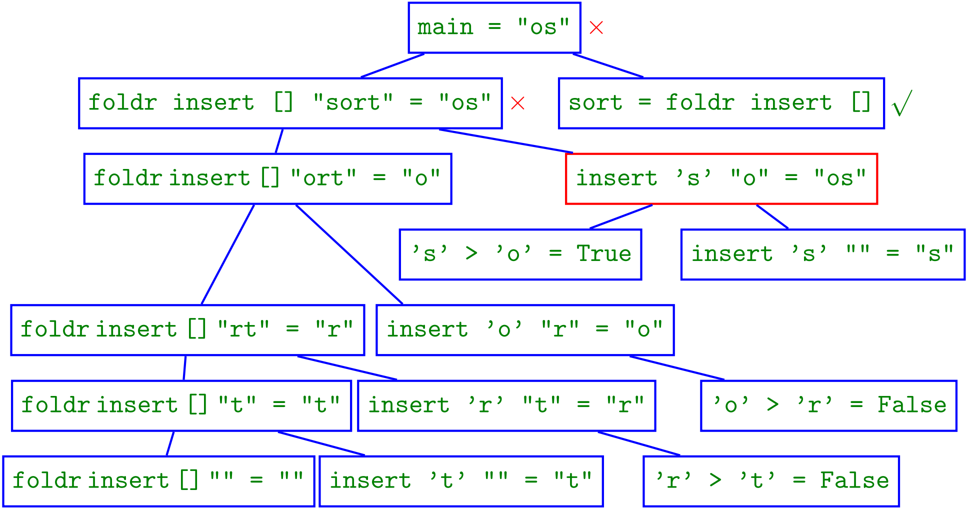
# Higher-Order Algorithmic Debugging



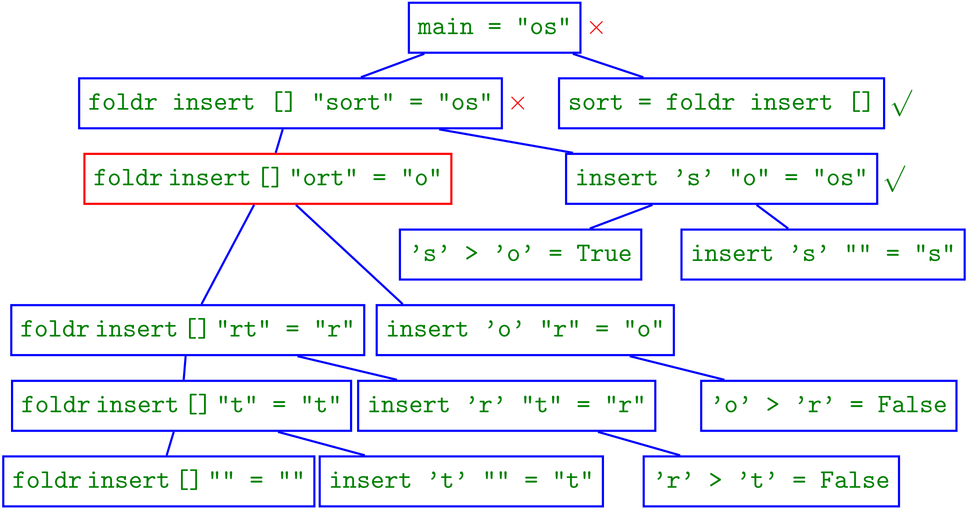

main = "os"
×
foldr insert [] "sort" = "os"
sort = foldr insert []
foldr insert [] "ort" = "o"
insert 's' "o" = "os"
's' > 'o' = True
insert 's' "" = "s"
foldr insert [] "rt" = "r"
insert 'o' "r" = "o"
foldr insert [] "t" = "t"
insert 'r' "t" = "r"
'o' > 'r' = False
foldr insert [] "" = ""
insert 't' "" = "t"
'r' > 't' = False

# Higher-Order Algorithmic Debugging

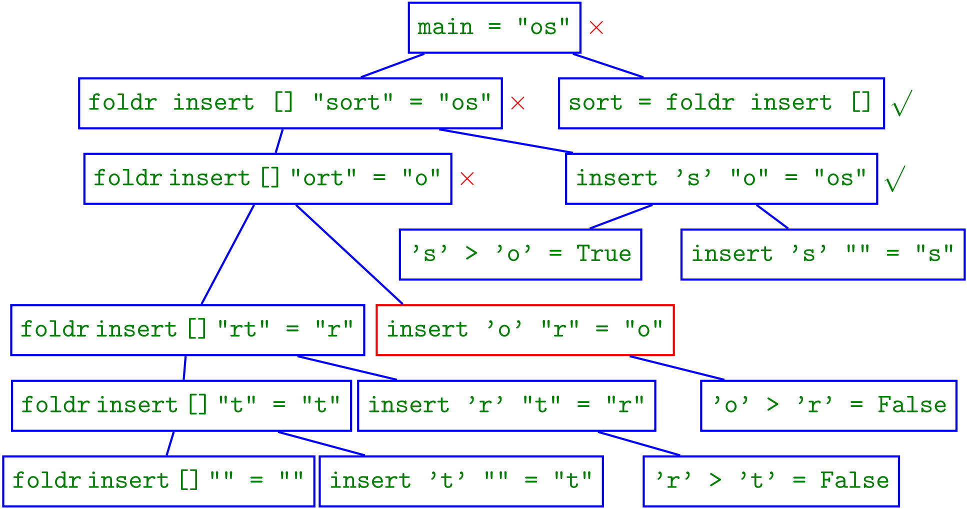



main = "os" ×
foldr insert [] "sort" = "os" ×
sort = foldr insert [] √
foldr insert [] "ort" = "o"
insert 's' "o" = "os"
's' > 'o' = True
insert 's' "" = "s"
foldr insert [] "rt" = "r"
insert 'o' "r" = "o"
foldr insert [] "t" = "t"
insert 'r' "t" = "r"
'o' > 'r' = False
foldr insert [] "" = ""
insert 't' "" = "t"
'r' > 't' = False

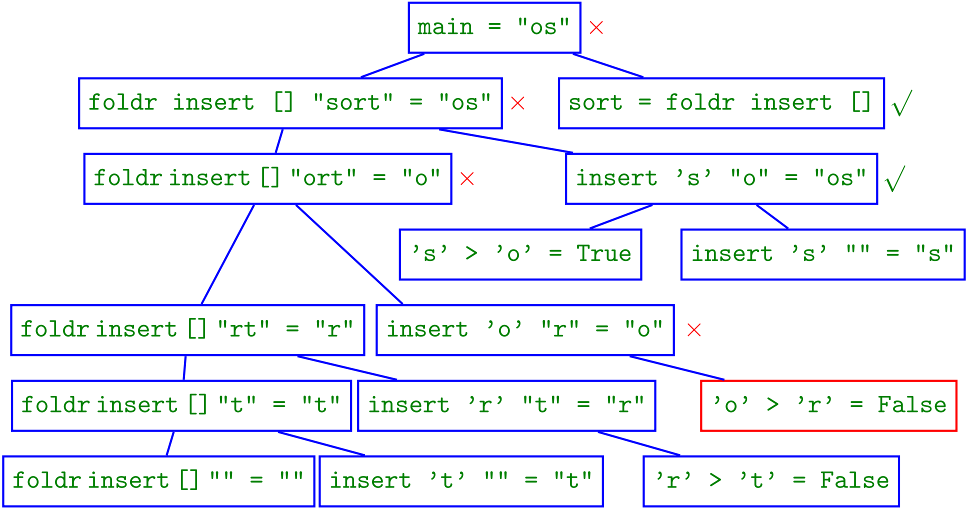
# Higher-Order Algorithmic Debugging



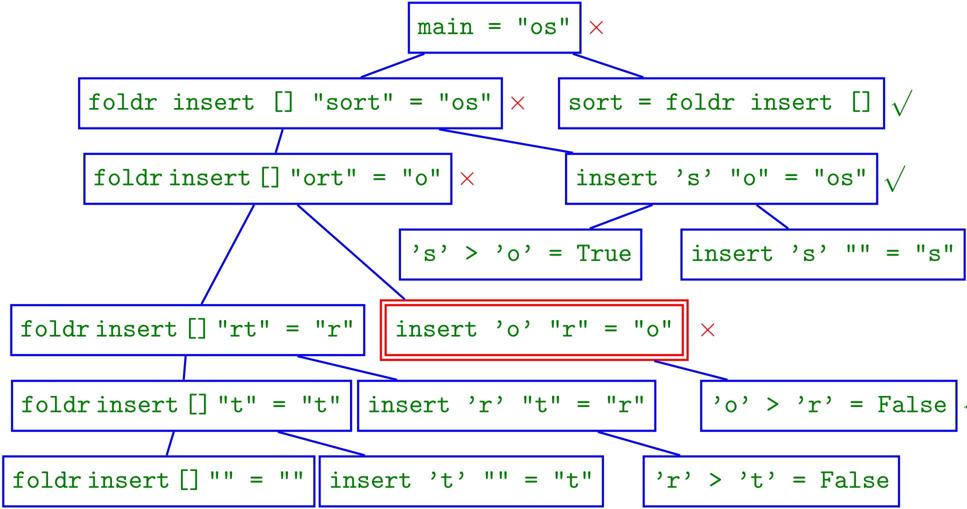
# Higher-Order Algorithmic Debugging



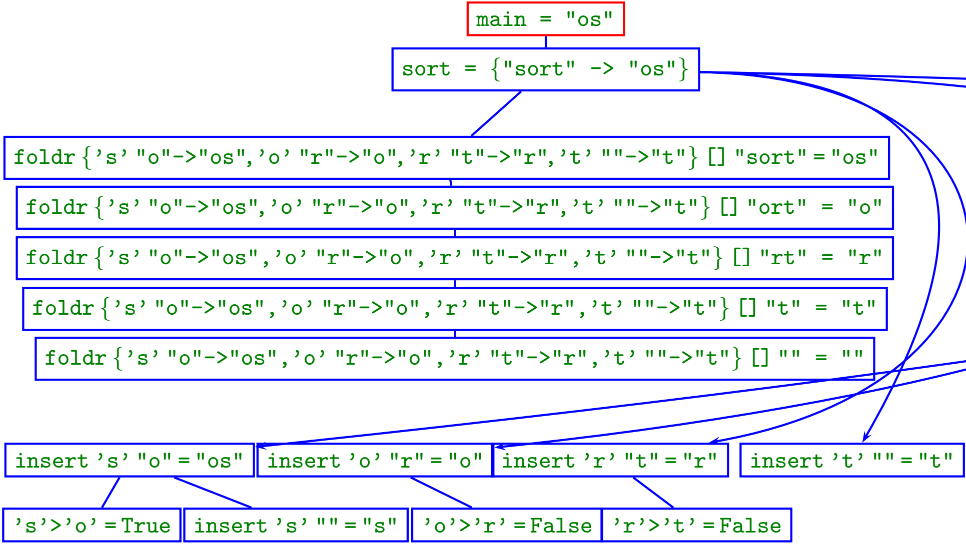
# Higher-Order Algorithmic Debugging



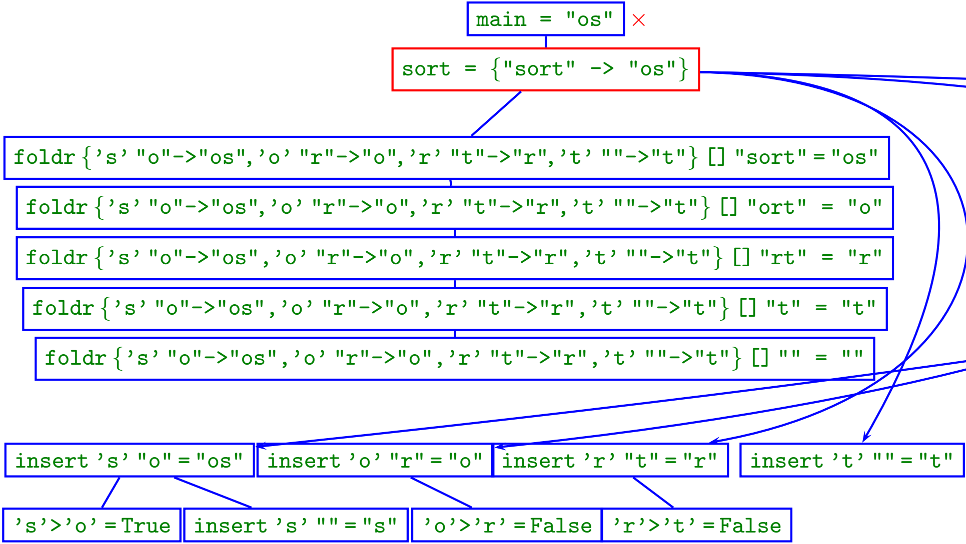
# Higher-Order Algorithmic Debugging



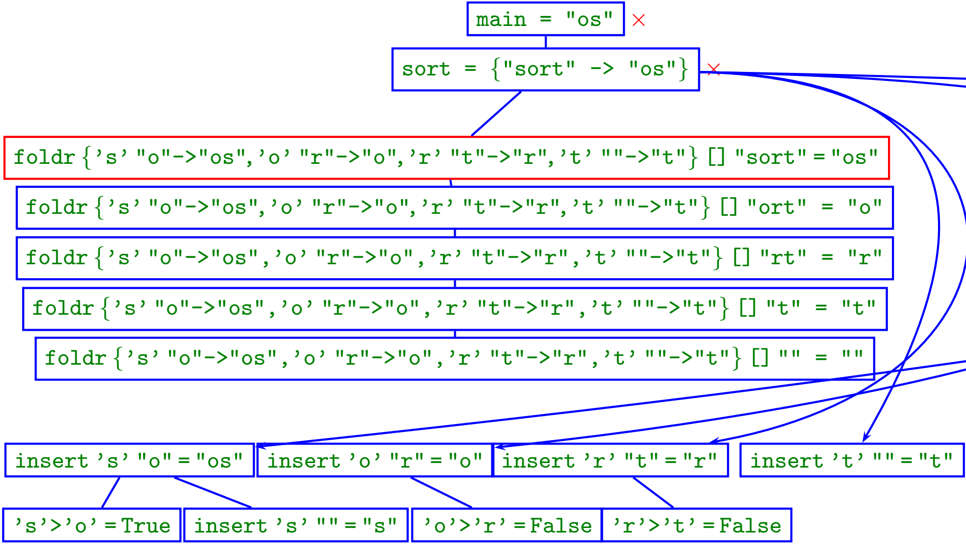
# Higher-Order Algorithmic Debugging II



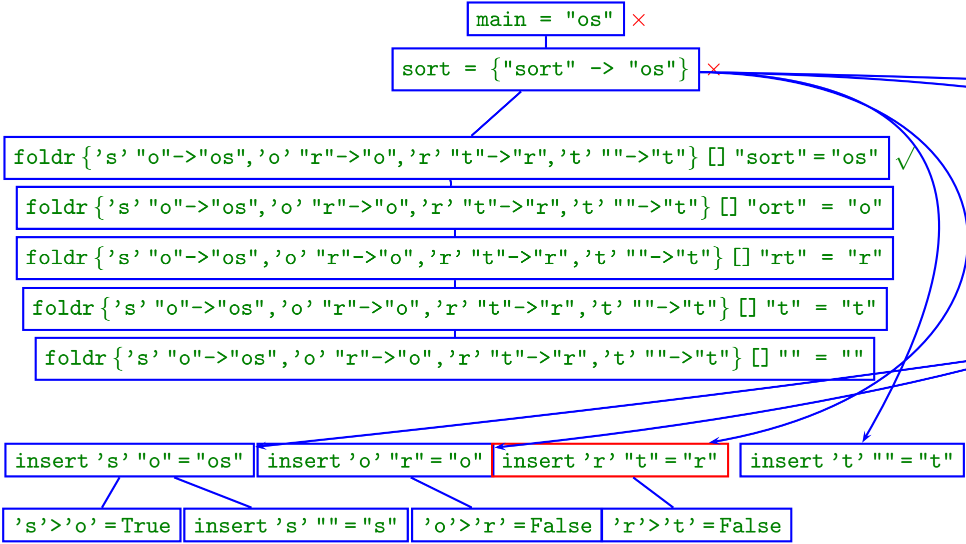
# Higher-Order Algorithmic Debugging II



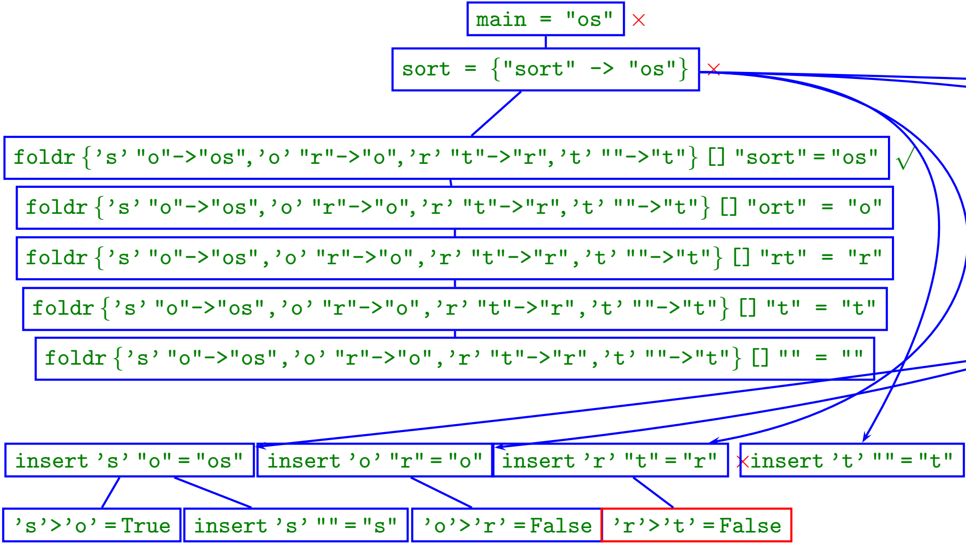
# Higher-Order Algorithmic Debugging II



# Higher-Order Algorithmic Debugging II



# Higher-Order Algorithmic Debugging II



# 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   √
```

# Source-based Free Navigation and Program Slicing

# 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 --------------------------
   if x > y then y : insert x ys
           else x : ys

sort :: [Char] -> [Char]
sort [] = []
sort (x:xs) = insert x ( sort xs )
```

# Call Stack

```
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

# Redex Trails

```
Output: -----------------------------------------
os\n

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 []
```

# Redex Trails

- Colin Runciman and Jan Sparud, 1997.

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

- Based on graph rewriting semantics of abstract machine.

# Animation of Lazy Evaluation

```
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"
```

# Trusting

Trust a module: Do not trace functions in module.

- Smaller trace file.
- Avoid viewing distracting details.

  `4 + 7 = 11`

# 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:

```
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:

```
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 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.

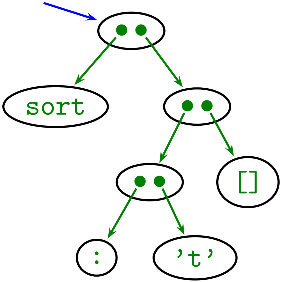
# What is a Good Trace?

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

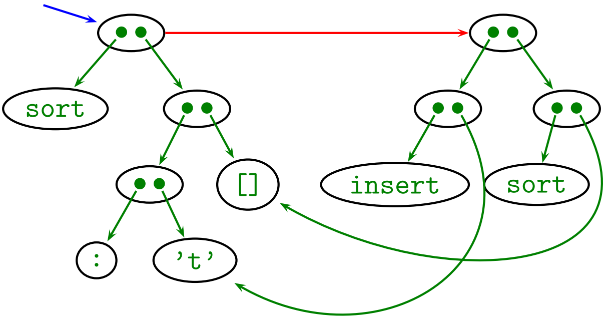
What is a computation? Semantics answers:

- Term rewriting: A sequence of expressions.
  $t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4 \longrightarrow t_5 \longrightarrow \ldots \longrightarrow t_n$
- Natural semantics: A proof tree.

# The Trace: Simple Graph Rewriting

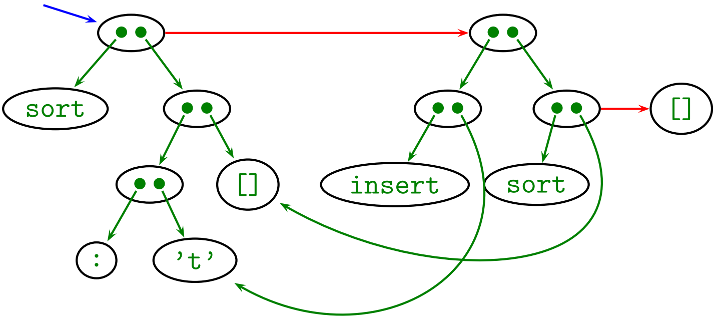



Start with expression `sort ('t':[])`



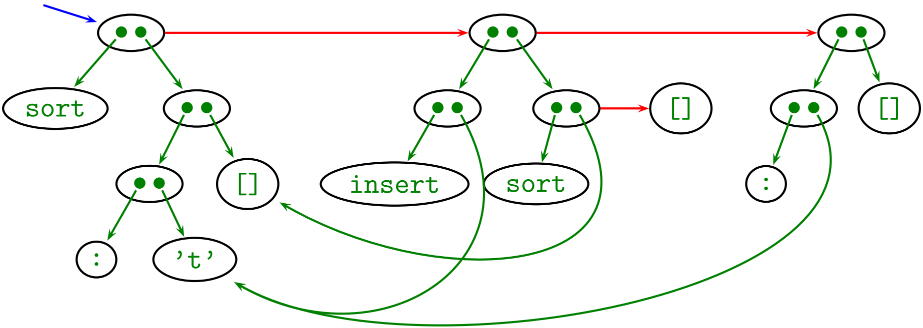

```
sort [] = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys
```

# The Trace: Simple Graph Rewriting



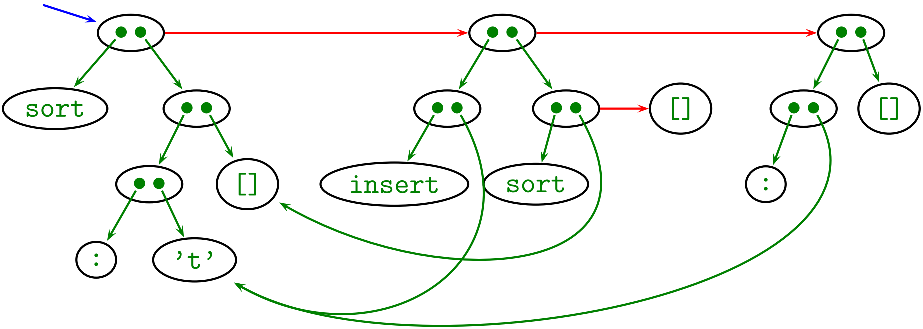

```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

# The Trace: Simple Graph Rewriting



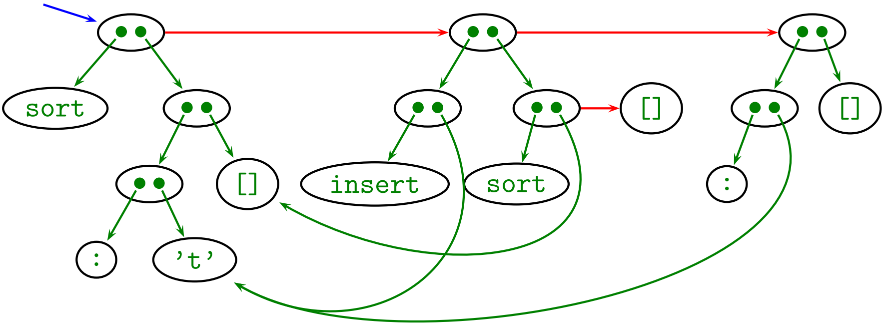

```
sort [] = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys
```

# The Trace: Simple Graph Rewriting




- New nodes for right-hand-side, connected via result pointer.
- Only add to graph, never remove.
- Sharing ensures compact representation.

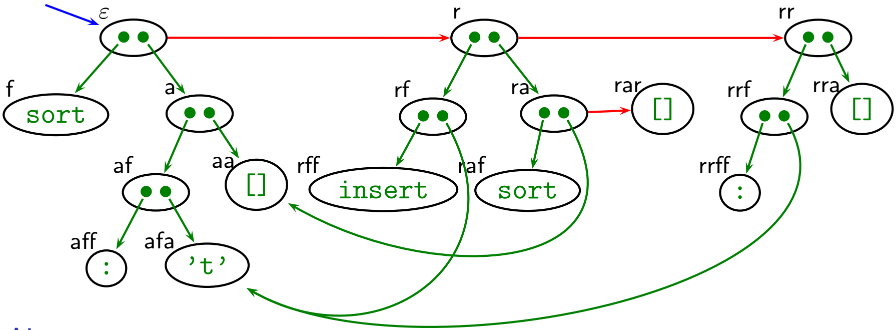
# The Node Naming Scheme




### Aim

- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes

# The Node Naming Scheme



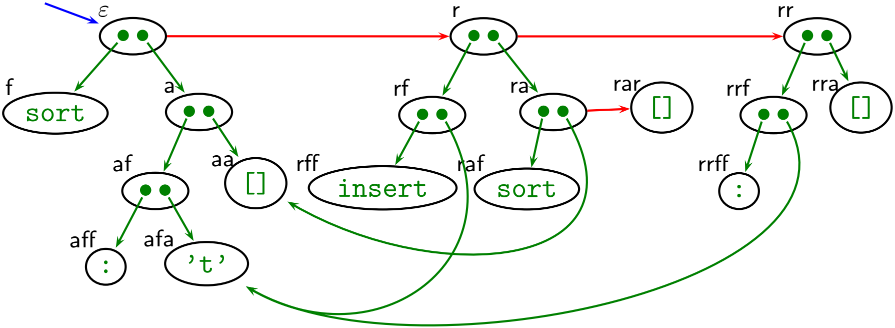

## Aim

- 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



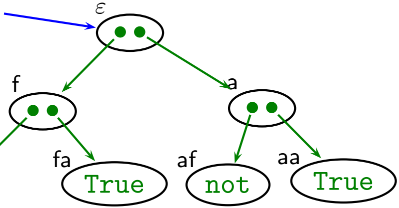

| | | | | |
|---|---|---|---|---|
| node | $n$ | := | $\{\mathrm{f}, \mathrm{a}, \mathrm{r}\}^*$ | |
| label term | $T$ | := | $a$ | atom |
| | | \| | $n\ m$ | application of nodes |
| atom | $a$ | := | $f \mid C \mid 42 \mid \ldots$ | defined variable, data constructor<br>atomic literal, ... |

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)

```
True && x = x
not True = False
```

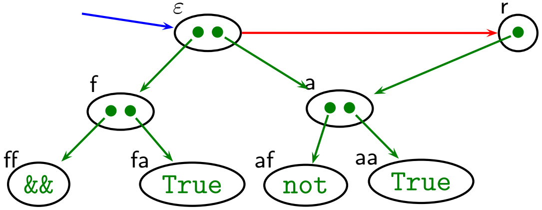


# Projections

- 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
```



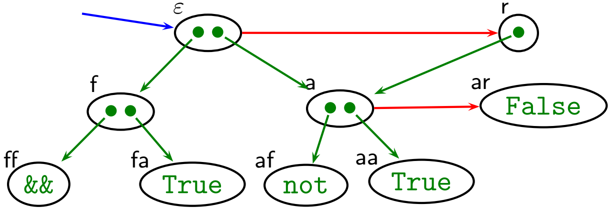

| | | | | |
|---|---|---|---|---|
| label term | $T$ | $:=$ | $a$ | atom |
| | | $\vert$ | $n\ m$ | application of nodes |
| | | $\vert$ | $n$ | indirection |
| atom | $a$ | $:=$ | $x \mid C \mid 42 \mid \ldots$ | variable, data constructor, ... |

# 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)

$\Rightarrow$ A projection requires an indirection as result.

```
True && x = x
not True = 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, ... |

# The Trace: The Augmented Redex Trail (ART)

A trace $\mathcal{G}$ for initial term $M$ and program $P$ is a partial function from nodes to term constructors, $\mathcal{G} : n \mapsto T$, defined by

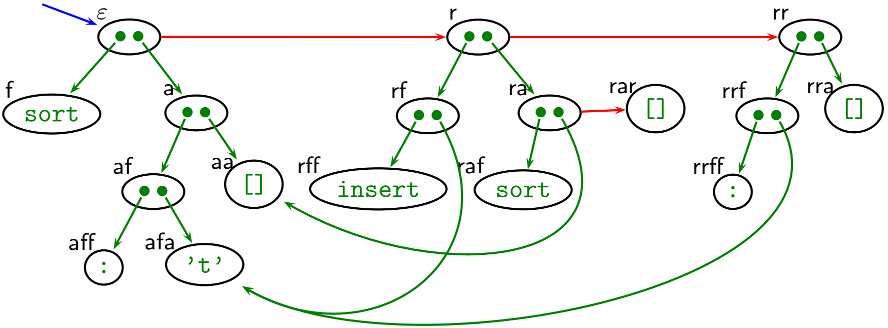
- The unshared graph representation of $M$, $\mathrm{graph}_{\mathcal{G}}(\varepsilon, M)$, is a trace.
- If $\mathcal{G}$ is a trace and
  - $L = R$ an equation of the program $P$,
  - $\sigma$ a substitution replacing argument variables by nodes,
  - $\mathrm{match}_{\mathcal{G}}(n, L\sigma)$,
  -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

  then $\mathcal{G} \cup \mathrm{graph}_{\mathcal{G}}(n\mathrm{r}, R\sigma)$ 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.




$$\mathrm{mef}_{\mathcal{G}}(\varepsilon) = \texttt{(:) 't' []}$$

**Definition**

$$n \succ_{\mathcal{G}} m \quad \Leftrightarrow \quad m = n\mathrm{r} \vee \mathcal{G}(n) = m$$

$$\lceil n \rceil_{\mathcal{G}} = m \quad \Leftrightarrow \quad n \succ_{\mathcal{G}}^{*} m \wedge \nexists o.\, m \succ_{\mathcal{G}} o$$

**Definition**

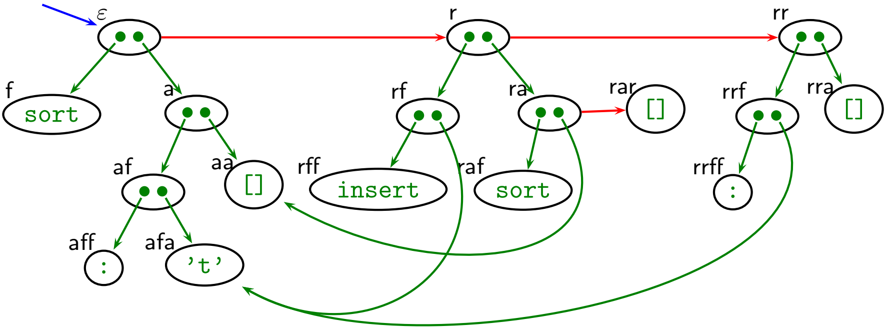
$$\mathrm{mef}_{\mathcal{G}}(n) = \mathrm{mefT}_{\mathcal{G}}(\mathcal{G}(\lceil n \rceil_{\mathcal{G}}))$$

$$\mathrm{mefT}_{\mathcal{G}}(a) = a$$

$$\mathrm{mefT}_{\mathcal{G}}(n) = \mathrm{mef}_{\mathcal{G}}(n)$$

$$\mathrm{mefT}_{\mathcal{G}}(n\, m) = \mathrm{mef}_{\mathcal{G}}(n)\ \mathrm{mef}_{\mathcal{G}}(m)$$

# Redexes and Big-Step Reductions




$\text{redex}_{\mathcal{G}}(\mathsf{r}) = $ `insert 't' []`

$\text{bigstep}_{\mathcal{G}}(\mathsf{r}) = $ `insert 't' [] = (:) 't' []`

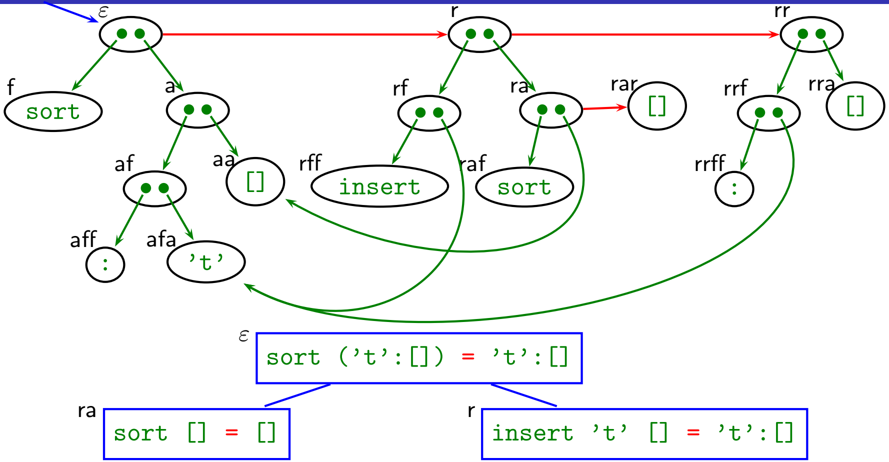
## Definition

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

$$\text{redex}_{\mathcal{G}}(n) = \begin{cases} \text{mef}_{\mathcal{G}}(m)\ \text{mef}_{\mathcal{G}}(o) & \text{, if } \mathcal{G}(n) = m\ o \\ a & \text{, if } \mathcal{G}(n) = a \end{cases}$$

$$\text{bigstep}_{\mathcal{G}}(n) = \text{redex}_{\mathcal{G}}(n) = \text{mef}_{\mathcal{G}}(n)$$

# From Trace to Big-Step Computation Tree

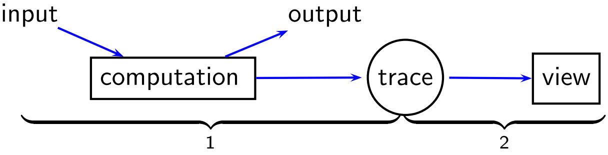



- Every redex node $n$ yields a tree node $n$ labelled $\mathrm{bigstep}_{\mathcal{G}}(n)$.
- Tree node $n$ is child of tree node $\mathrm{parent}(n)$.

$$
\begin{aligned}
\mathrm{parent}(nr) &= n\\
\mathrm{parent}(nf) &= \mathrm{parent}(n)\\
\mathrm{parent}(na) &= \mathrm{parent}(n)\\
\mathrm{parent}(\varepsilon) &= \mathrm{undefined}
\end{aligned}
$$

# Summary

- 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.