Foundations for the Debugging of Functional Programs

Olaf Chitil, Yong Luo and Thomas Davie

University of Kent, UK

Supported by EPSRC grant EP/C516605/1

12th February 2008

Programs have Bugs

Even functional programs!

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

but

- wrong result
- abortion with run-time error
- non-termination

Why Debug Functional Programs Differently?

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics
- No sequential execution of statements.
  - evaluation of expressions
  - evaluation of subexpressions is independent

    ```
    f (g 3 4) (h 1 2) (i 5) (j 3 9 3)
    ```

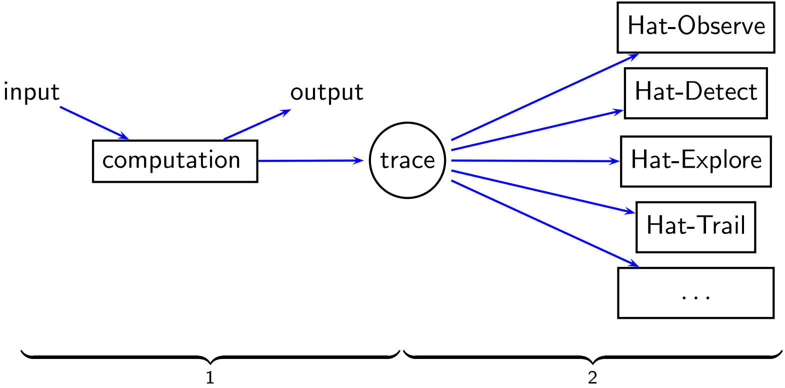
Why Debug Functional Programs Differently?

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics
- No sequential execution of statements.
  - evaluation of expressions
  - evaluation of subexpressions is independent
    `f (g 3 4) (h 1 2) (i 5) (j 3 9 3)`

Conclusions

- Abstract from execution details: views for various semantics models.
- Take advantage of simple and compositional semantics.
- Liberate from sequentiality of computation.

The Haskell Tracer Hat (`www.haskell.org/hat`)



Two-Phase Tracing: The trace as data structure.

- Liberates from the time arrow of computation.
- Enables many different views.

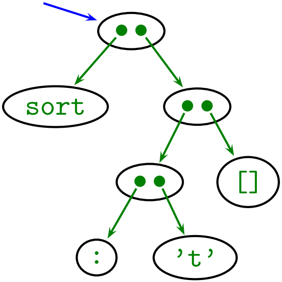
But where are formal definitions you can reason with?

Example: Insertion Sort

```
main :: String
main = 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
```

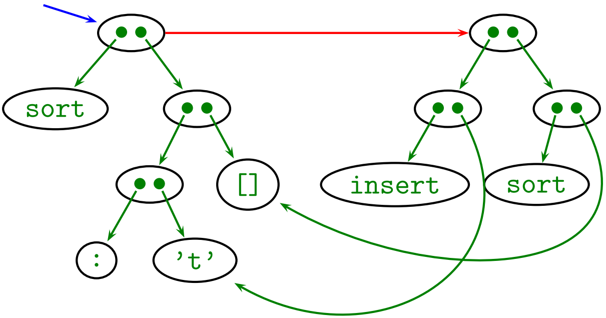
There is a bug: `main = "os"` !

The Trace: Simple Graph Rewriting



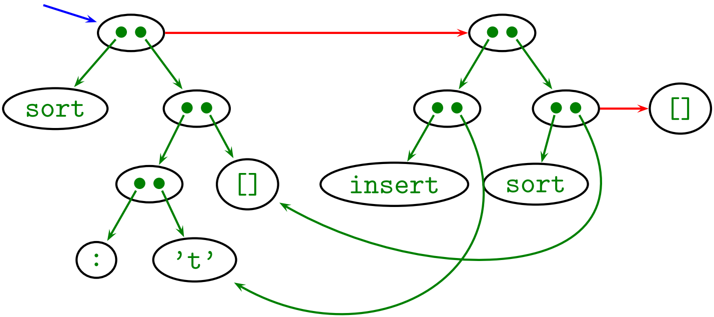
Start with expression `sort ('t':[])`

The Trace: Simple Graph Rewriting



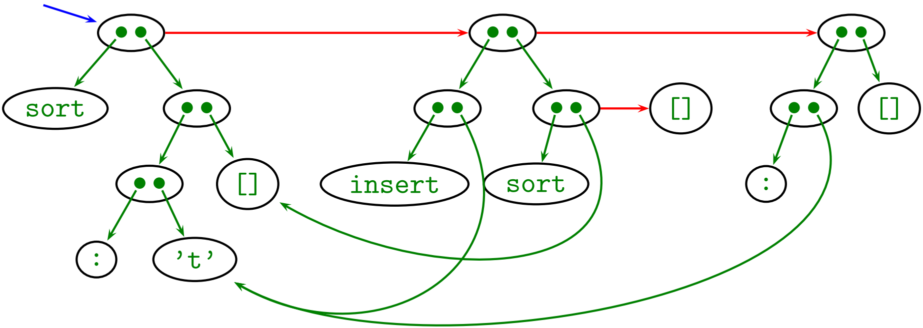

```
sort [] = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys
```

The Trace: Simple Graph Rewriting



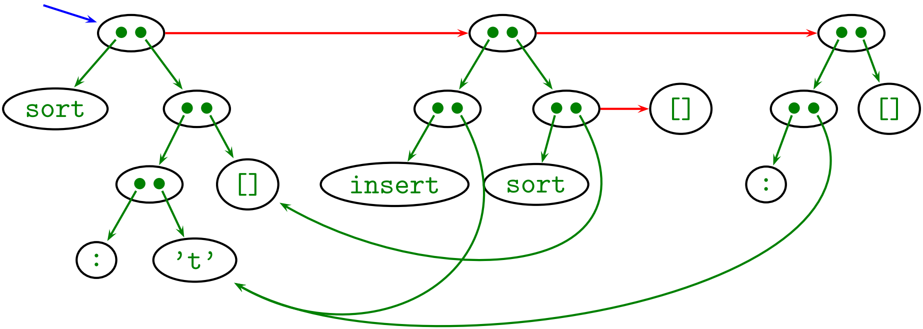

```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

The Trace: Simple Graph Rewriting



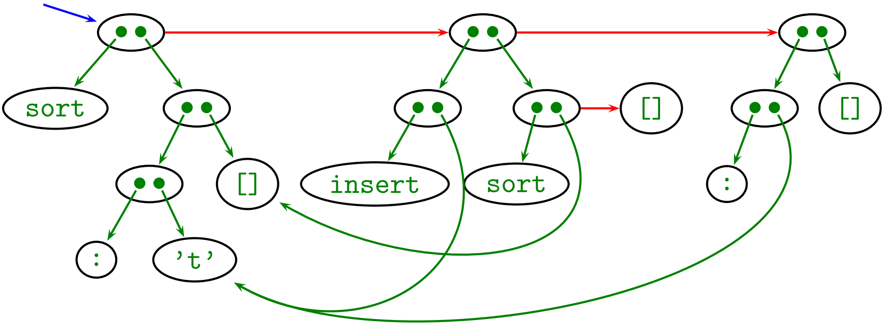

```
sort [] = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys
```

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.

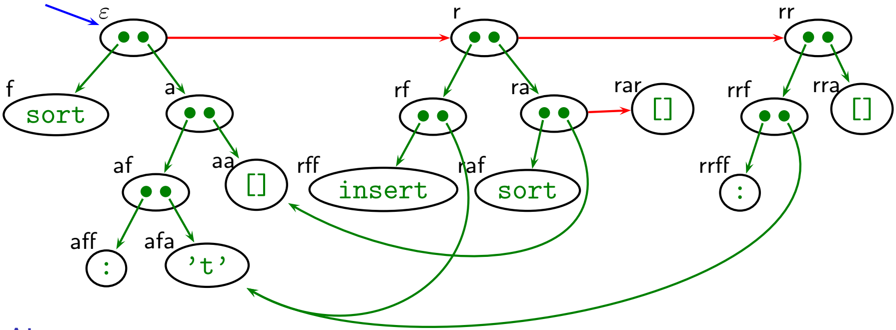
The Node Naming Scheme



Aim

- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes

The Node Naming Scheme



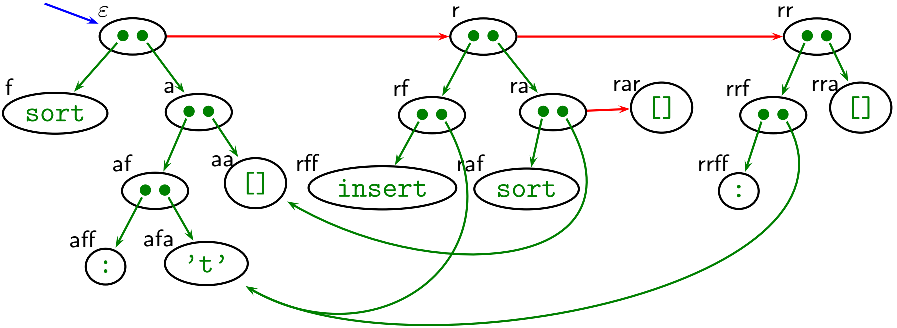
Aim

- 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



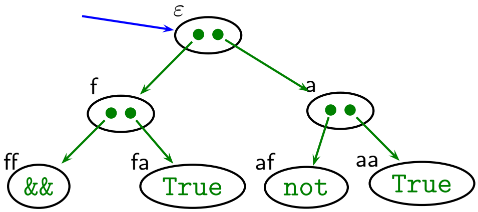
node	n	:=	$\{\mathrm{f}, \mathrm{a}, \mathrm{r}\}^*$	
label term	T	:=	a	atom
		$\mid$	$n\ m$	application of nodes
atom	a	:=	$f \mid C \mid 42 \mid \ldots$	defined variable, data constructor atomic literal, ...

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
```

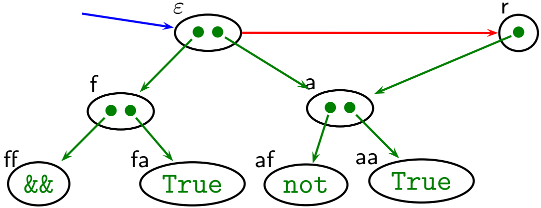


Projections

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

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

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



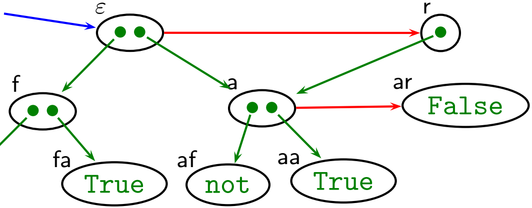
label term	T	$:=$	a	atom
		$\vert$	$n\ m$	application of nodes
		$\vert$	n	indirection
atom	a	$:=$	$x \mid C \mid 42 \mid \ldots$	variable, data constructor, ...

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)

⇒ 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

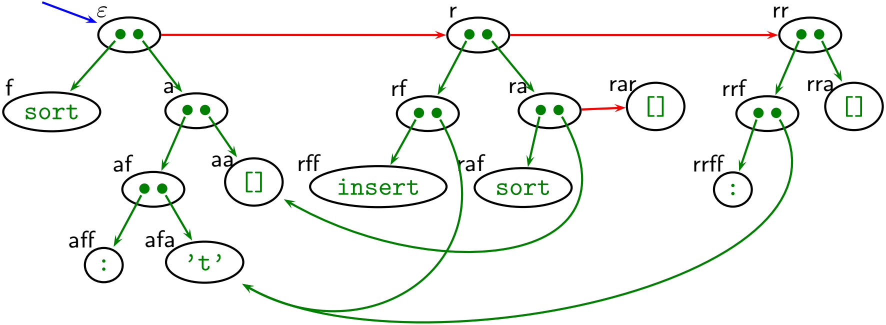
- The unshared graph representation of M, $\text{graph}_{\mathcal{G}}(\varepsilon, M)$, is a trace.
- If $\mathcal{G}$ is a trace and
  - $L = R$ an equation of the program P,
  - σ a substitution replacing argument variables by nodes,
  - $\text{match}_{\mathcal{G}}(n, L\sigma)$,
  - $nr \notin \text{dom}(\mathcal{G})$,

  then $\mathcal{G} \cup \text{graph}_{\mathcal{G}}(nr, R\sigma)$ 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.



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

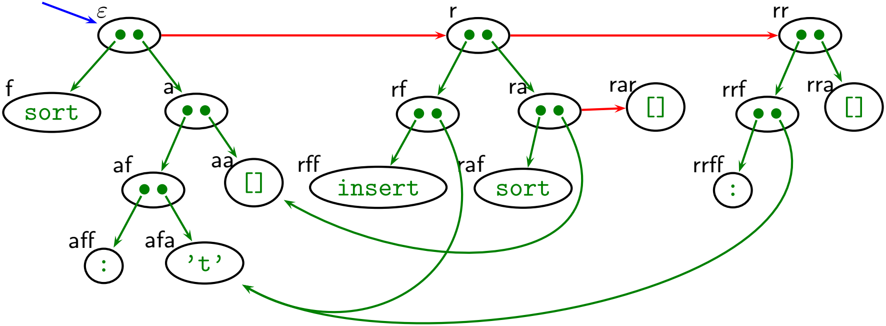
Definition

$$n \succ_{\mathcal{G}} m \quad \Leftrightarrow \quad m = n\mathrm{r} \vee \mathcal{G}(n) = m$$
$$\lceil n \rceil_{\mathcal{G}} = m \quad \Leftrightarrow \quad n \succ^{*}_{\mathcal{G}} m \wedge \nexists o.\, m \succ_{\mathcal{G}} o$$

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



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

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

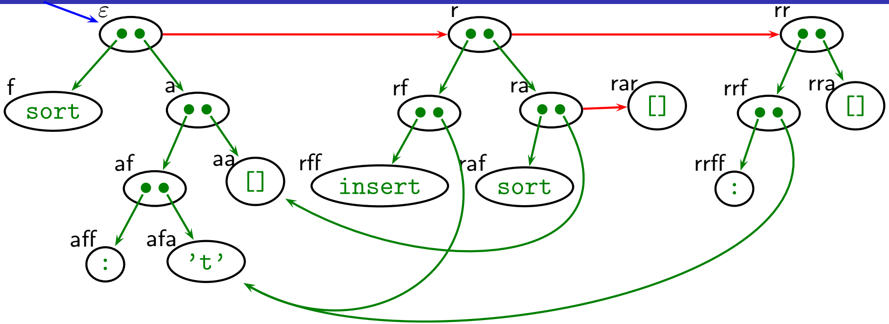
Definition

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



ε `sort ('t':[]) = 't':[]`

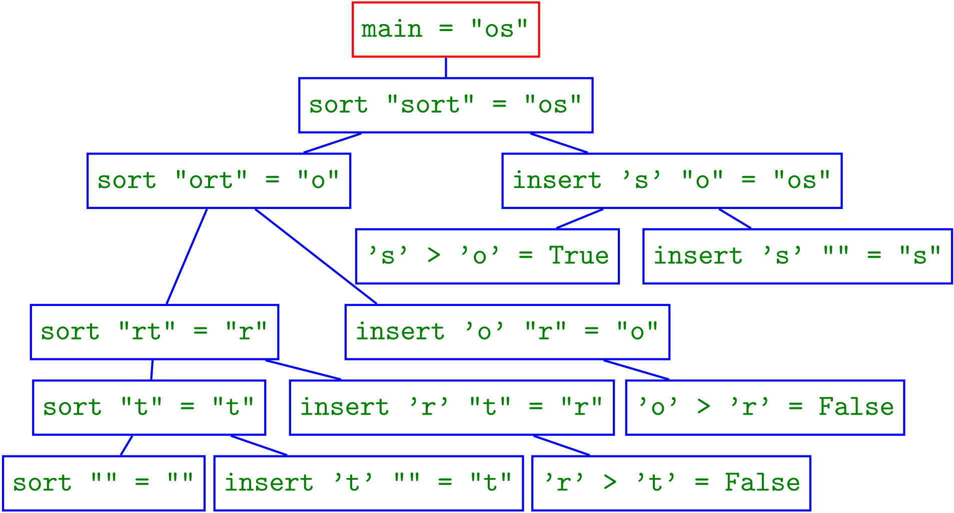
ra `sort [] = []`

r `insert 't' [] = 't':[]`

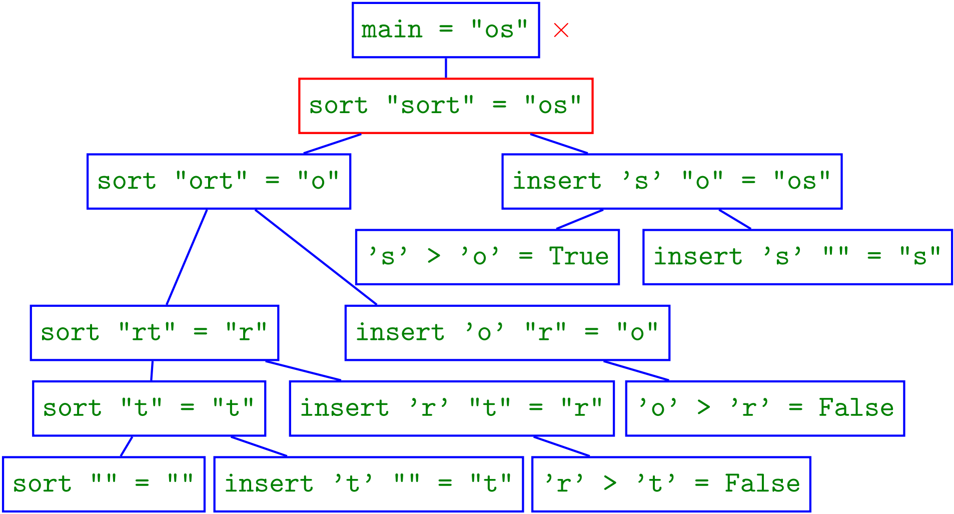
- Every redex node n yields a tree node n labelled $\mathrm{bigstep}_{\mathcal{G}}(n)$.
- Tree node n is child of tree node $\mathrm{parent}(n)$.

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

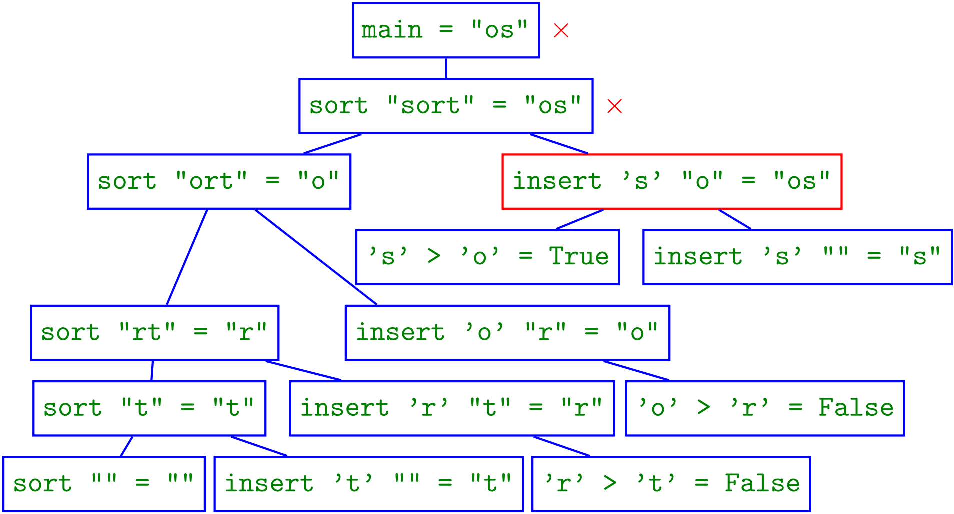
Algorithmic Debugging with the Computation Tree



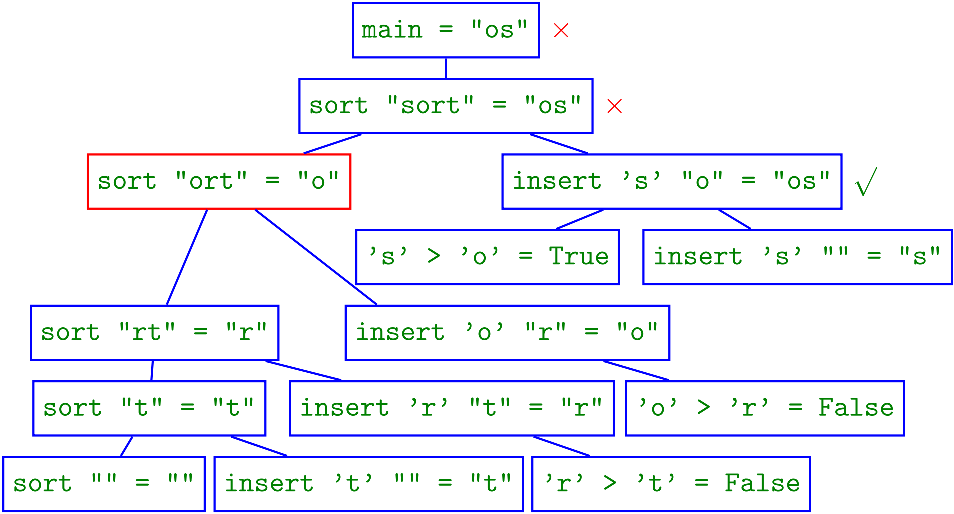
Algorithmic Debugging with the Computation Tree



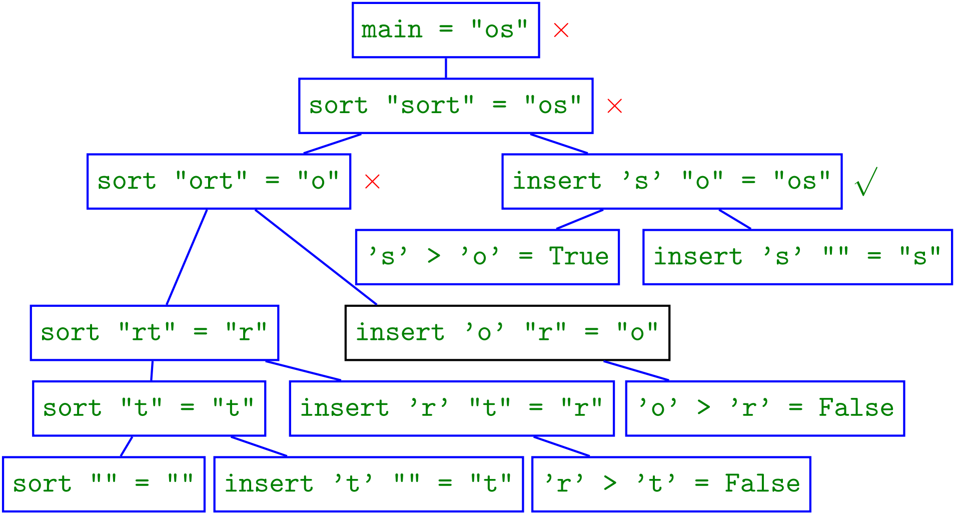
Algorithmic Debugging with the Computation Tree



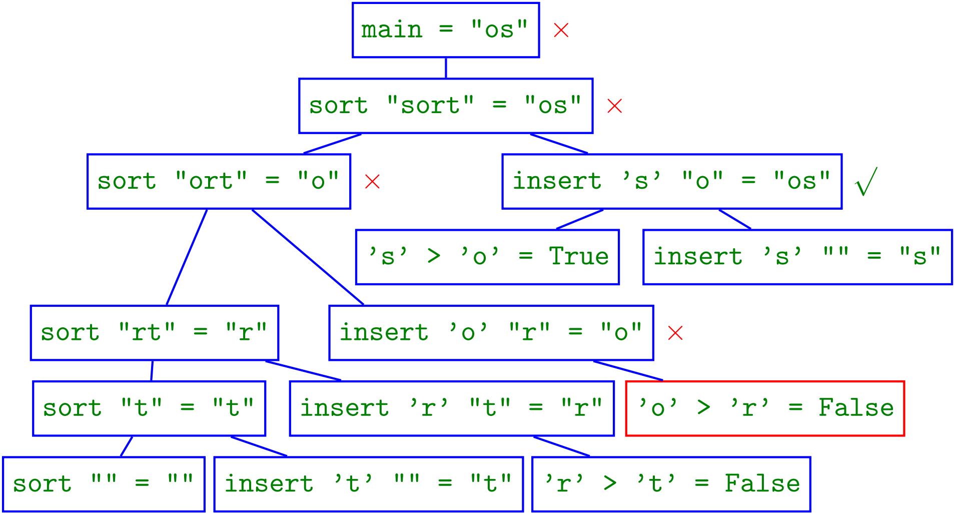
Algorithmic Debugging with the Computation Tree



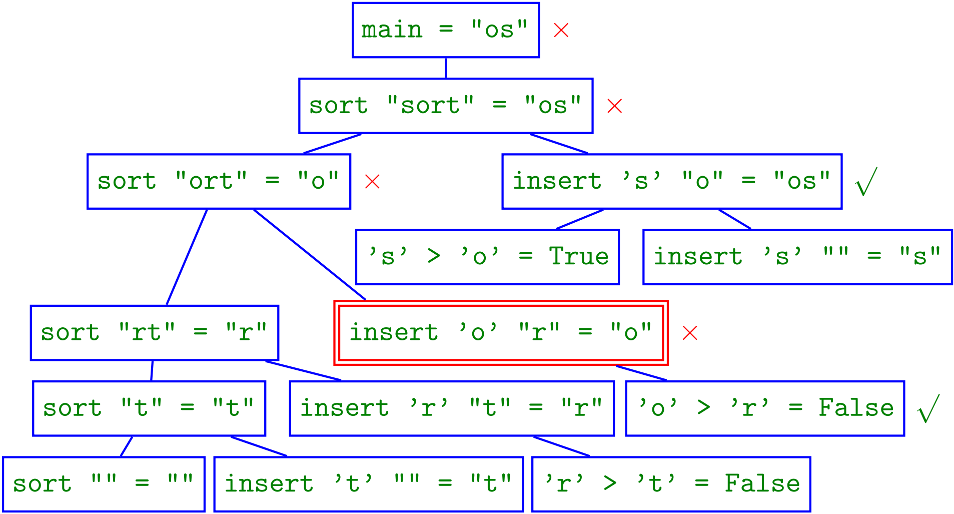
Algorithmic Debugging with the Computation Tree



Algorithmic Debugging with the Computation Tree



Algorithmic Debugging with the Computation Tree



Fault located!

```
main :: String
main = 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"`

Correctness of Algorithmic Debugging: The Property

If node n incorrect and all its children correct, then node n faulty, i.e., its equation is faulty.

ε `sort ('t':[]) = 't':[]`

ra `sort [] = []`

r `insert 't' [] = 't':[]`

Definition

Tree node n incorrect $\Leftrightarrow$ $\mathrm{redex}_{\mathcal{G}}(n) \not\cong_{\mathrm{I}} \mathrm{mef}_{\mathcal{G}}(n)$.

Tree node n faulty $\Leftrightarrow$ $\mathrm{redex}_{\mathcal{G}}(n) \not\cong_{\mathrm{I}} \mathrm{reduct}_{\mathcal{G}}(n)$.

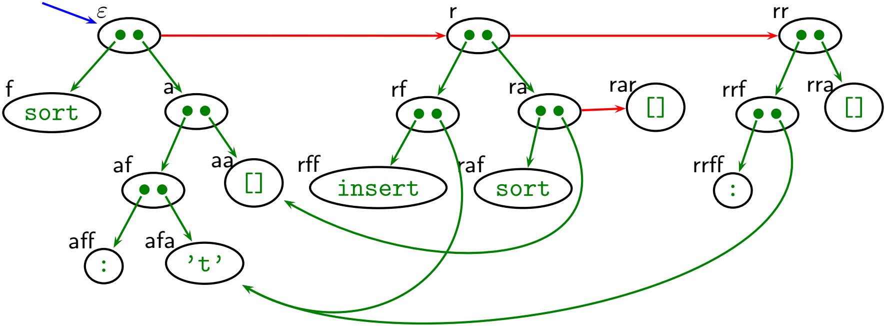
If tree node n faulty, then for its program equation $L = R$ exists substitution σ such that $L\sigma \not\cong_{\mathrm{I}} R\sigma$.

Soundness of Algorithmic Debugging: Main Theorem

Theorem

Let n be a redex node. If for all redex nodes m with $parent(m) = n$ we have $\mathrm{redex}_{\mathcal{G}}(m) \cong_I \mathrm{mef}_{\mathcal{G}}(m)$, then $reduct_{\mathcal{G}}(n) \cong_I \mathrm{mef}_{\mathcal{G}}(n)$.

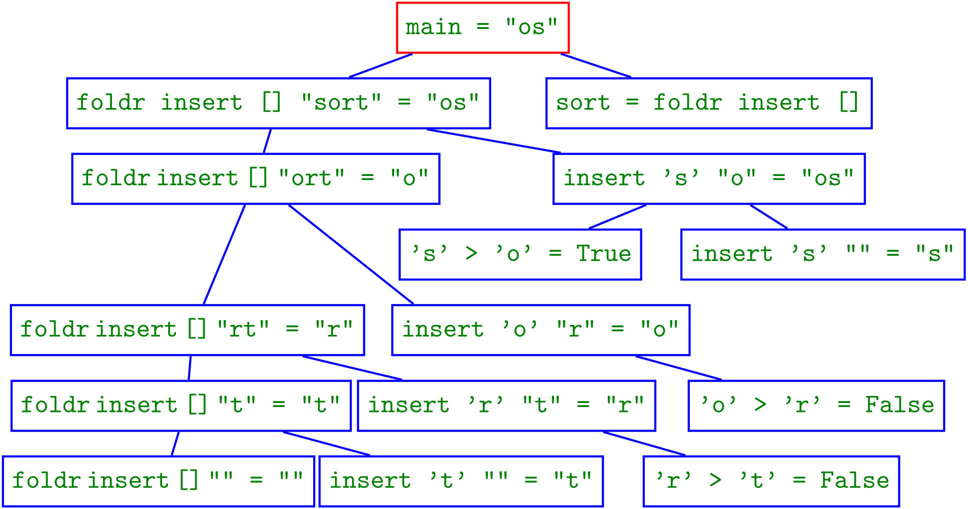
With $\mathrm{redex}_{\mathcal{G}}(n) \ncong_{\mathrm{I}} \mathrm{mef}_{\mathcal{G}}(n)$ follows $\mathrm{redex}_{\mathcal{G}}(n) \ncong_{\mathrm{I}} \mathrm{reduct}_{\mathcal{G}}(n)$.



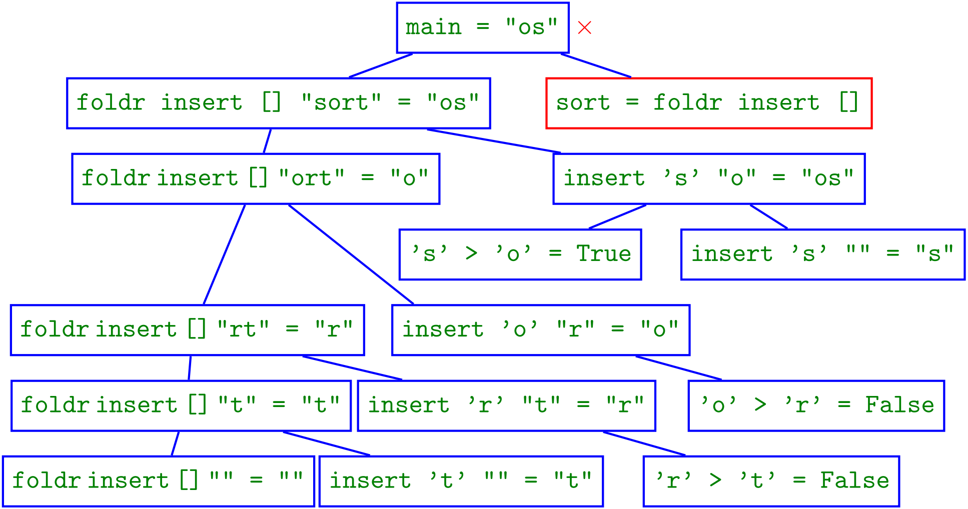
Higher-Order Insertion Sort

```
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
```

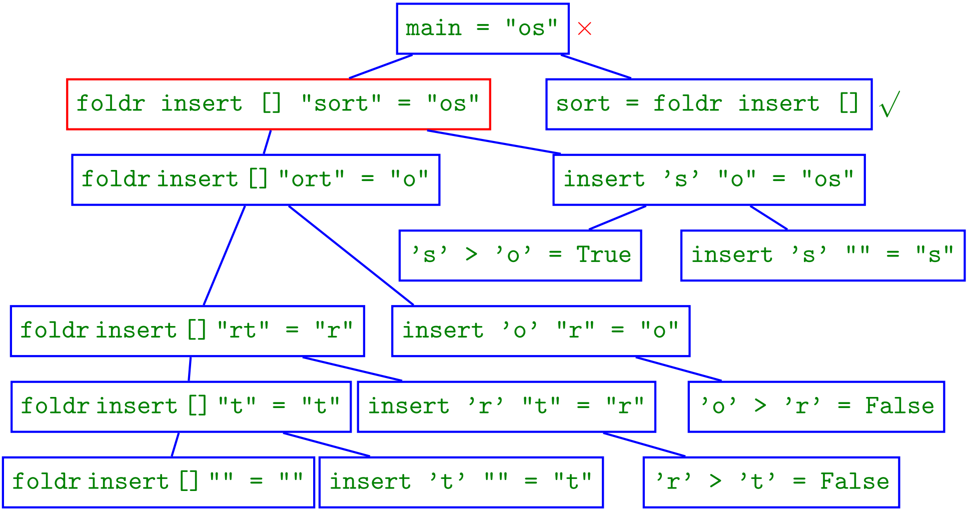
Higher-Order Algorithmic Debugging



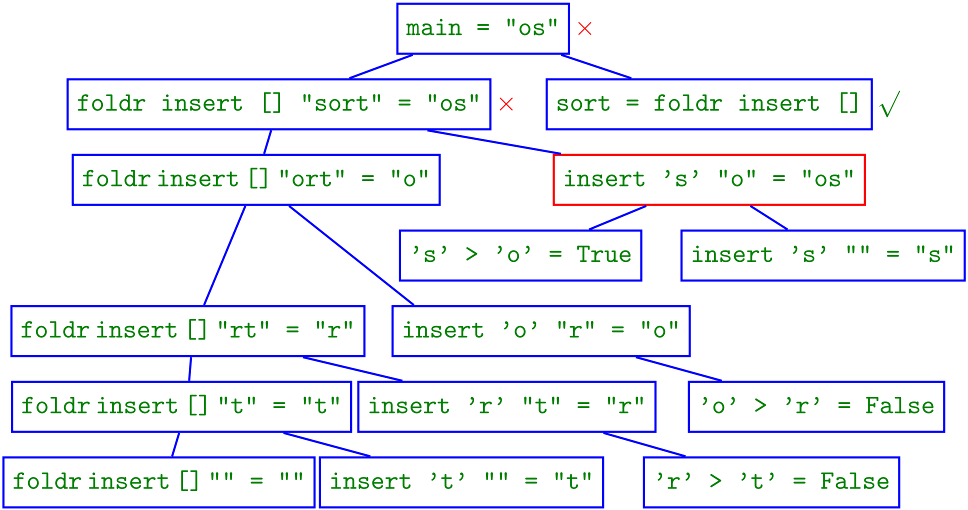
Higher-Order Algorithmic Debugging



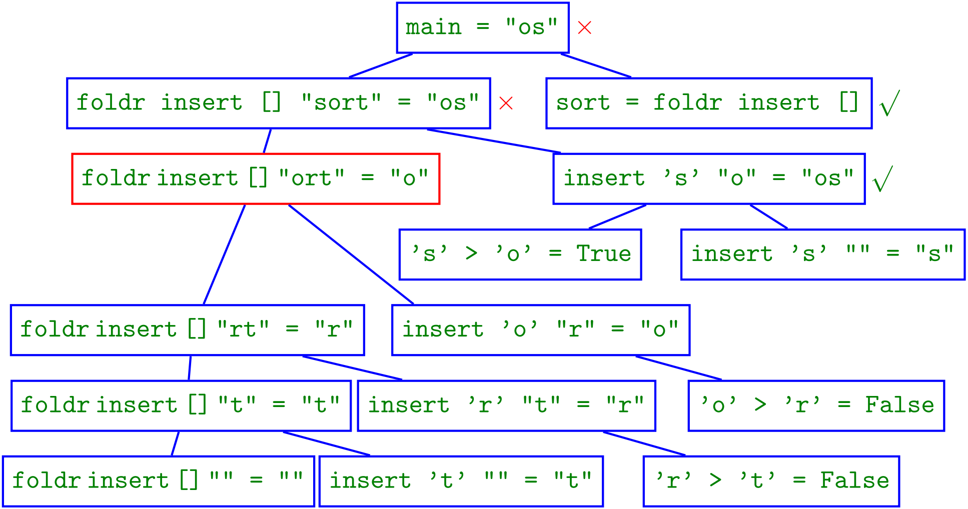
Higher-Order Algorithmic Debugging



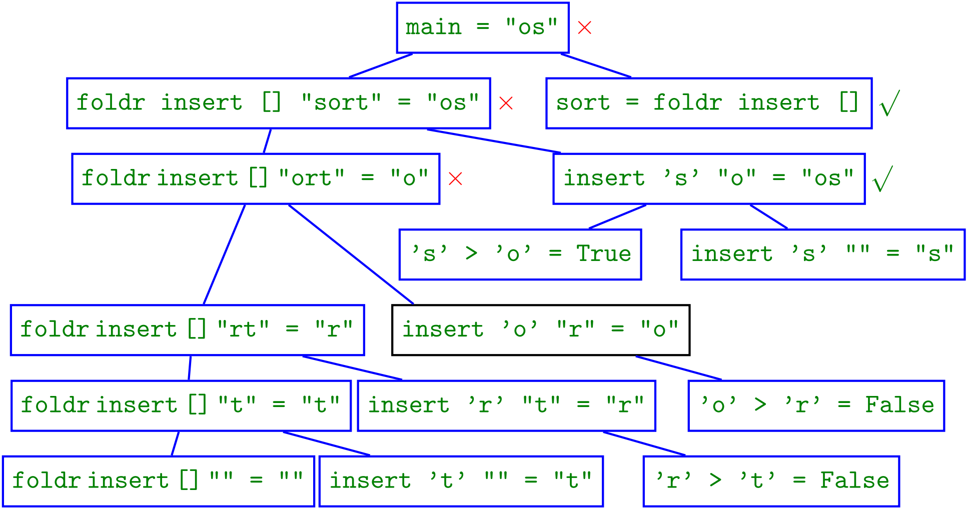
Higher-Order Algorithmic Debugging



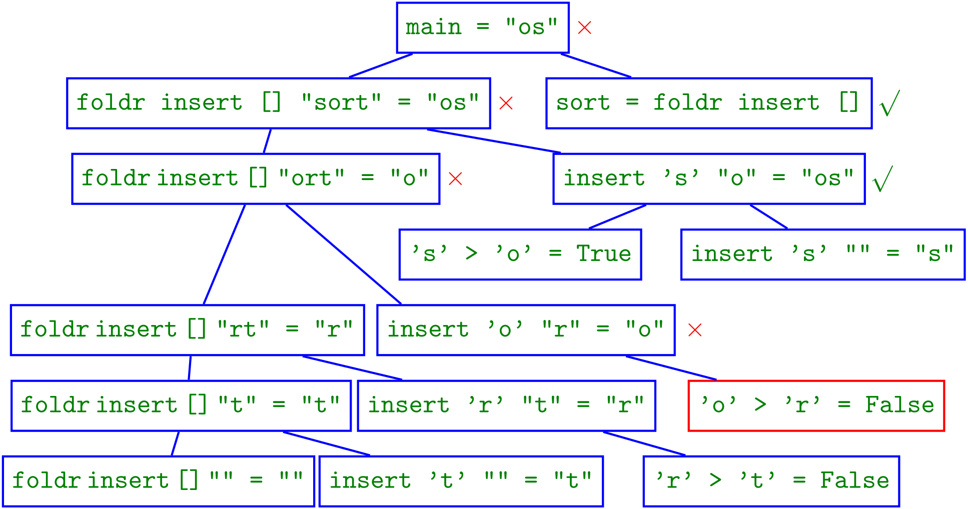
Higher-Order Algorithmic Debugging



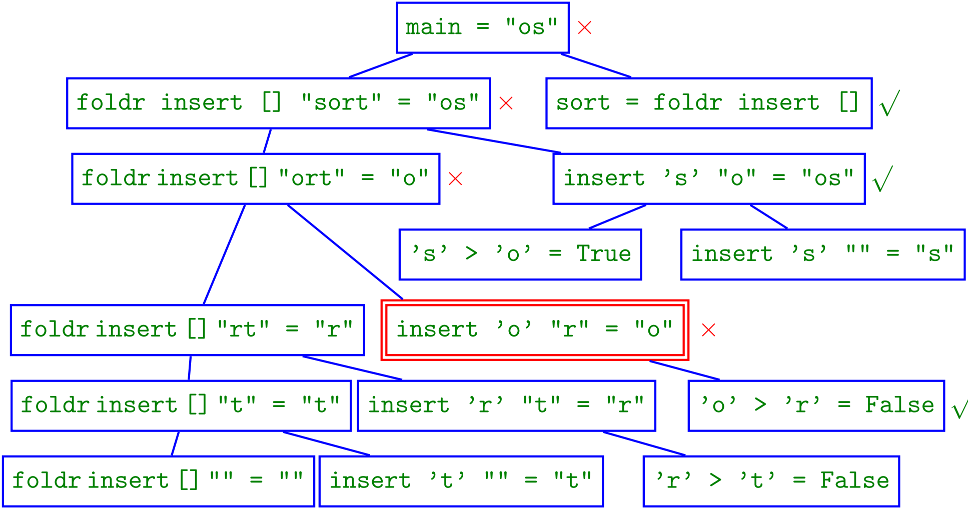
Higher-Order Algorithmic Debugging



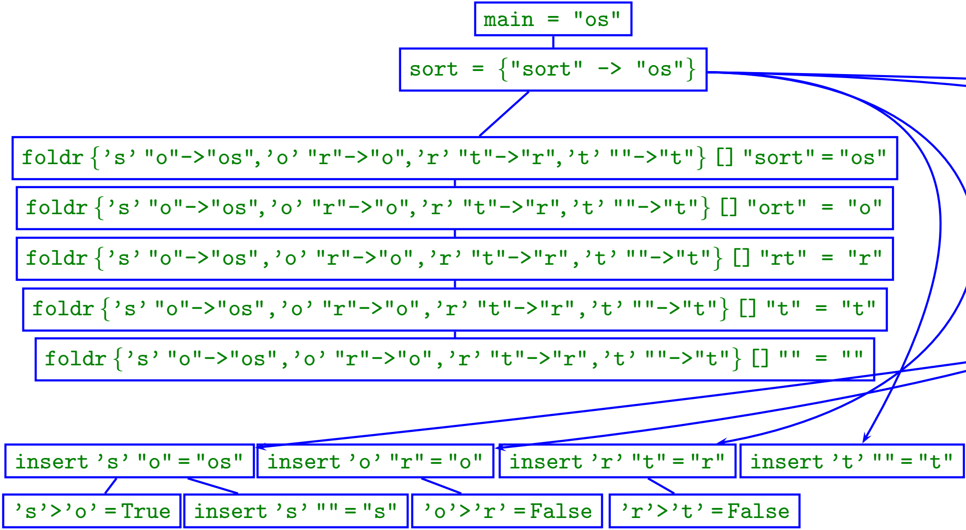
Higher-Order Algorithmic Debugging



Higher-Order Algorithmic Debugging



Higher-Order Algorithmic Debugging II



Modify a Few Definitions I

Definition (Most evaluated form for finite maps)

$$\mathrm{mef}^{\mathrm{M}}_{\mathcal{G}}(n) = \begin{cases} \mathrm{fMap}_{\mathcal{G}}(n) & \text{, if } M = f\ N_1 \ldots N_k \wedge 0 \le k < \mathrm{arity}(f) \\ \{\} & \text{, if } M = f\ N_1 \ldots N_k \wedge k \ge \mathrm{arity}(f) \\ M & \text{, otherwise} \end{cases}$$
$$\text{where } M = \mathrm{mea}_{\mathcal{G}}(n)$$

$$\mathrm{mea}_{\mathcal{G}}(n) = \mathrm{meaT}_{\mathcal{G}}(\mathcal{G}(\lceil n \rceil_{\mathcal{G}}))$$
$$\mathrm{meaT}_{\mathcal{G}}(a) = a$$
$$\mathrm{meaT}_{\mathcal{G}}(m\, n) = \mathrm{mea}_{\mathcal{G}}(m)\ \mathrm{mef}^{\mathrm{M}}_{\mathcal{G}}(n)$$

$$\mathrm{fMap}_{\mathcal{G}}(n) = \{\mathrm{mef}^{\mathrm{M}}_{\mathcal{G}}(o) \mapsto \mathrm{mef}^{\mathrm{M}}_{\mathcal{G}}(m) \mid \mathcal{G}(m) = n'\ o \wedge n' \succ^{*}_{\mathcal{G}} n \wedge \mathrm{mef}^{\mathrm{M}}_{\mathcal{G}}(m) \neq \{\}\}$$

Modify a Few Definitions II

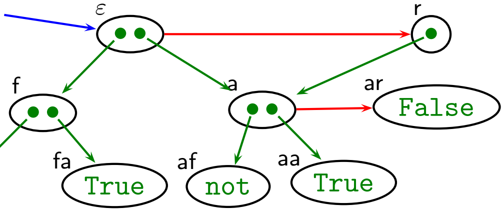
Definition (Parent for finite maps)

$$\mathsf{parentFDT}_{\mathcal{G}} = \mathsf{parent} \cdot \mathsf{fun}_{\mathcal{G}}$$

$$\mathsf{fun}_{\mathcal{G}}(n) = \begin{cases} n & \text{, if } \mathcal{G}(n) = a \\ \mathsf{fun}_{\mathcal{G}}(\lceil m \rceil_{\mathcal{G}}) & \text{, if } \mathcal{G}(n) = m\, o \end{cases}$$

Conclusions

- Simple model amenable to proof.
- Contains a wealth of information about computation.
- Models real-world trace of Haskell tracer Hat.
- Proves soundness of algorithmic debugging.



http://www.haskell.org/hat
http://www.cs.kent.ac.uk/people/staff/oc/traceTheory.html