Towards a Theory of Tracing for Functional Programs based on Graph Rewriting

Olaf Chitil and Yong Luo

University of Kent, UK
Supported by EPSRC grant EP/C516605/1

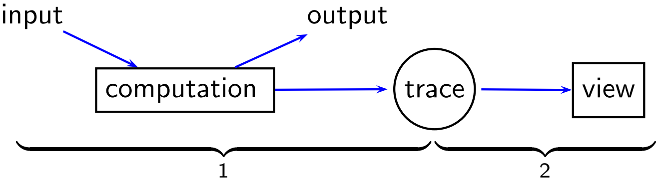
1st April 2006

Tracing Functional Programs

Why Tracing?

- Locate a fault (wrong output, run-time error, non-termination).
- Comprehend a program.

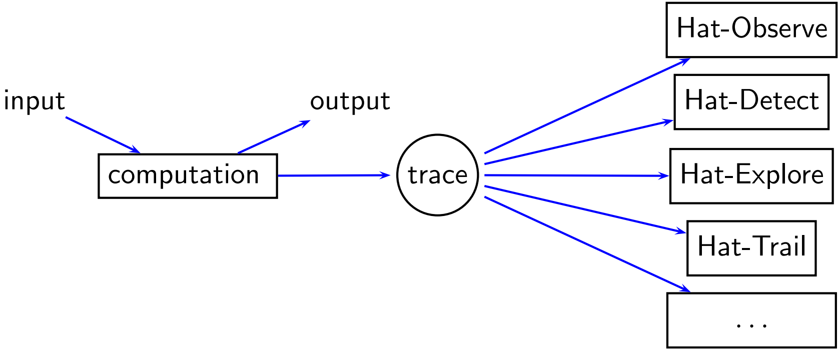
Two-Phase Tracing: A trace as data structure



- Liberates from time arrow of computation.
- Enables views based on different execution models.
  (small-step, big-step, interpreter with environment, denotational)
- Enables compositional views.

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

- Multi-View Tracer



- Trace = Augmented Redex Trail (ART); distilled as unified trace.

Aim: A theoretical model of the ART.

The Programming Language

Launchbury's and related semantics

- Subset of λ-calculus plus `case` for matching.
- Any program can be translated into this core calculus.

For tracing

- Close relationship between trace and original program essential.
- Language has most frequently used features:
    - named functions
    - pattern matching

The Programming Language

Launchbury's and related semantics

- Subset of λ-calculus plus `case` for matching.
- Any program can be translated into this core calculus.

For tracing

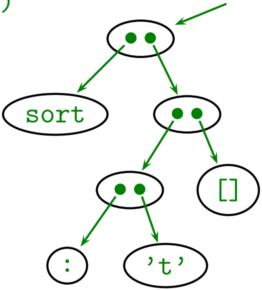
- Close relationship between trace and original program essential.
- Language has most frequently used features:
  - named functions
  - pattern matching

⇒ Higher-order term rewriting system

```
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
```

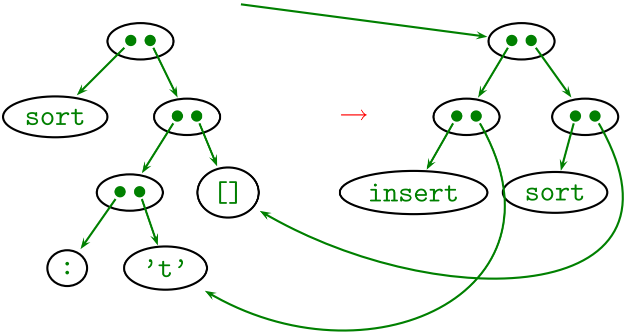
Graph Rewriting I

`sort ('t':[])`



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

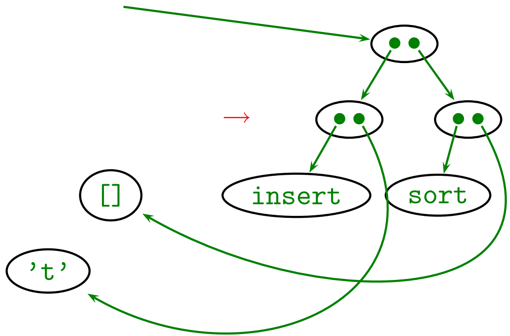
Graph Rewriting I



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

- Create new nodes for right-hand-side.
- Nodes of subexpressions are shared.

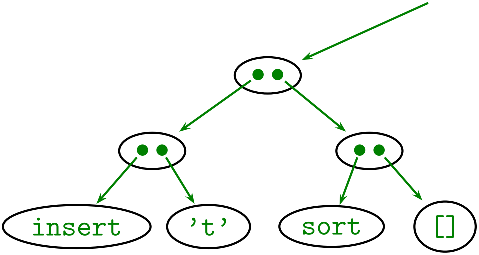
Graph Rewriting I



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

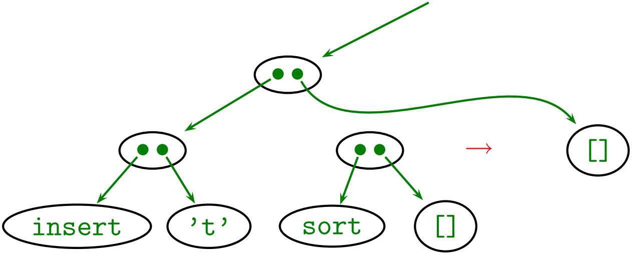
- Create new nodes for right-hand-side.
- Nodes of subexpressions are shared.
- Some old nodes become garbage.

Graph Rewriting II



```
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
```

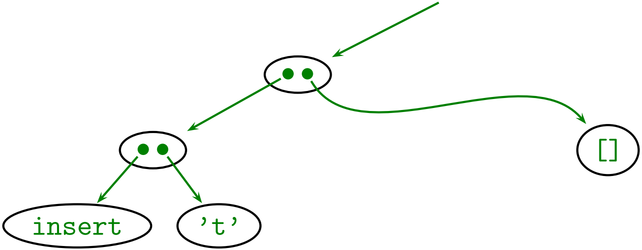
Graph Rewriting II



```
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
```

- Application node of redex replaced by new node.

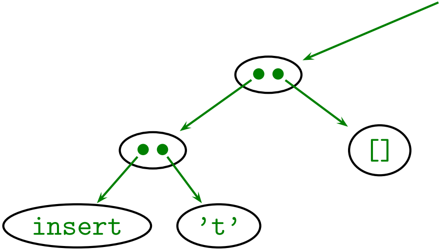
Graph Rewriting II



```
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
```

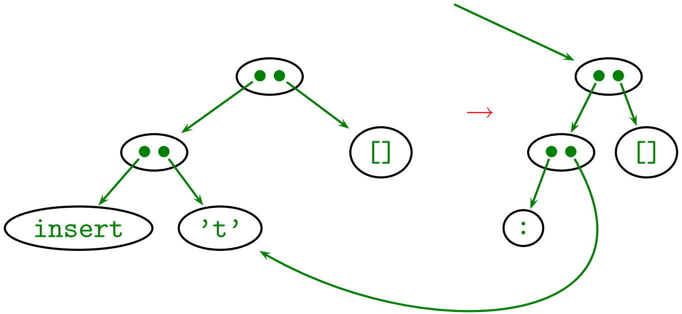
- Application node of redex replaced by new node.

Graph Rewriting III



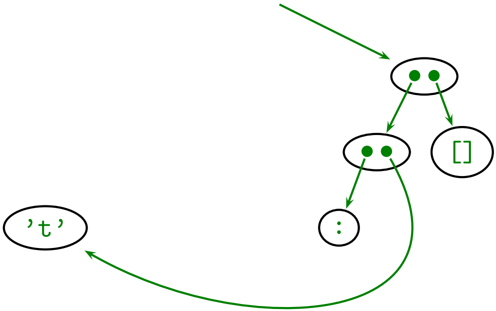
```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

Graph Rewriting III



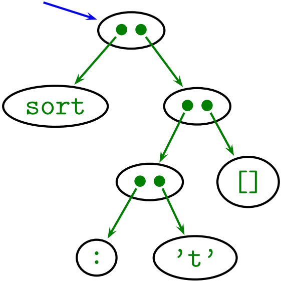
```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

Graph Rewriting III

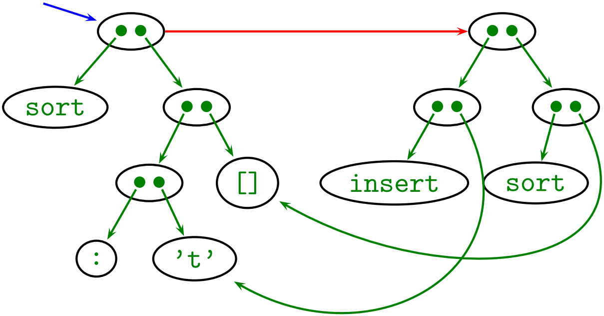


```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

The Trace

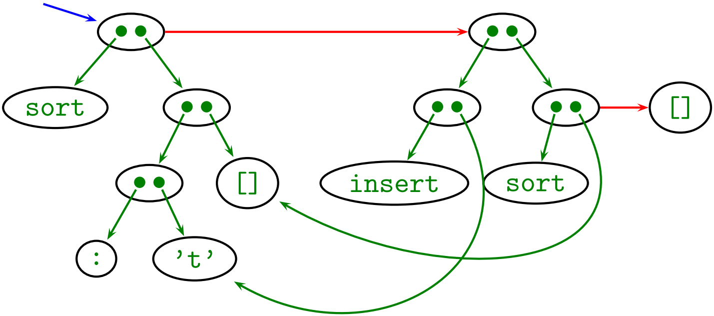


The Trace



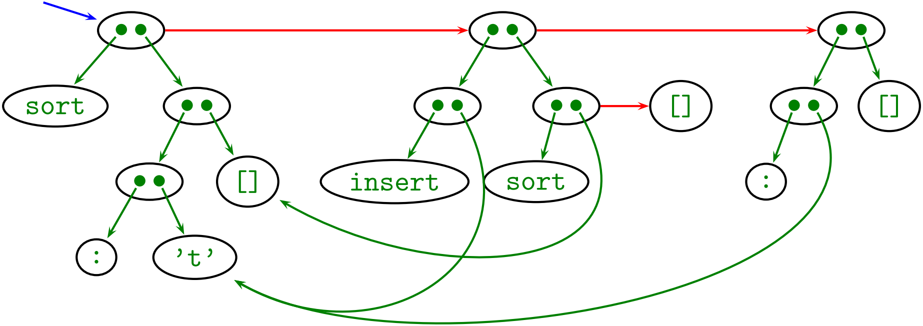
- New nodes for right-hand-side, connected via result pointer.
- Only add to graph, never remove.
- Sharing ensures compact representation.

The Trace



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

The Trace



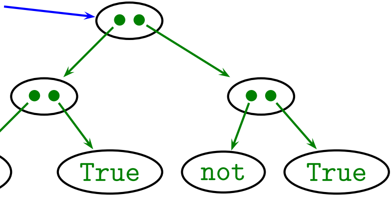
- New nodes for right-hand-side, connected via result pointer.
- Only add to graph, never remove.
- Sharing ensures compact representation.

The Node Labels

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

- pointers instead of edges

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

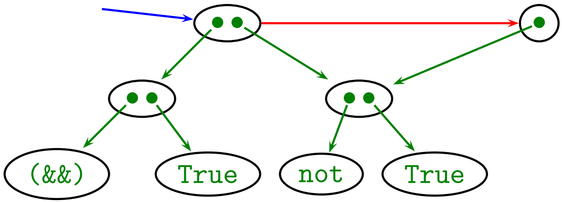


The Node Labels

term constructor	T	$:=$	a	atom
		$\mid$	$n\ m$	application of nodes
		$\mid$	n	indirection
atom	a	$:=$	$x \mid C \mid 42 \mid \ldots$	variable, data constructor atomic literal, …

- pointers instead of edges
- a projection requires an indirection as result

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

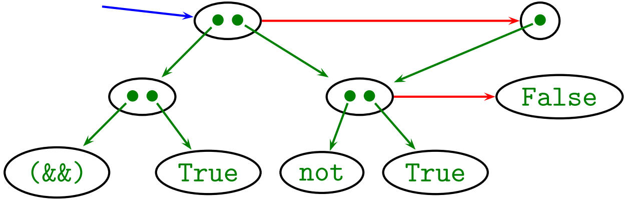


The Node Labels

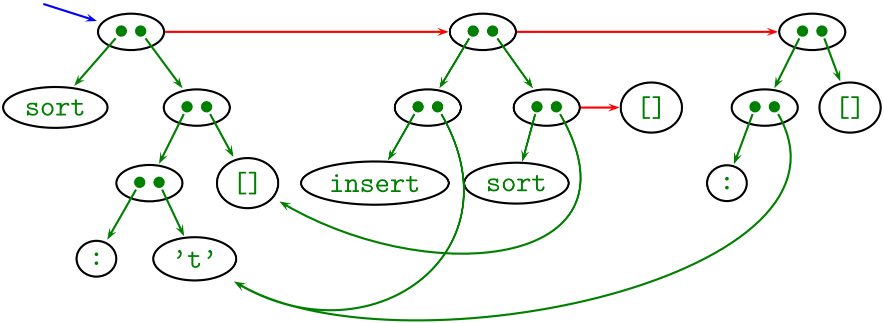
term constructor	T	$:=$	a	atom
		$\mid$	$n\ m$	application of nodes
		$\mid$	n	indirection
atom	a	$:=$	$x \mid C \mid 42 \mid \ldots$	variable, data constructor atomic literal, ...

- pointers instead of edges
- a projection requires an indirection as result

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



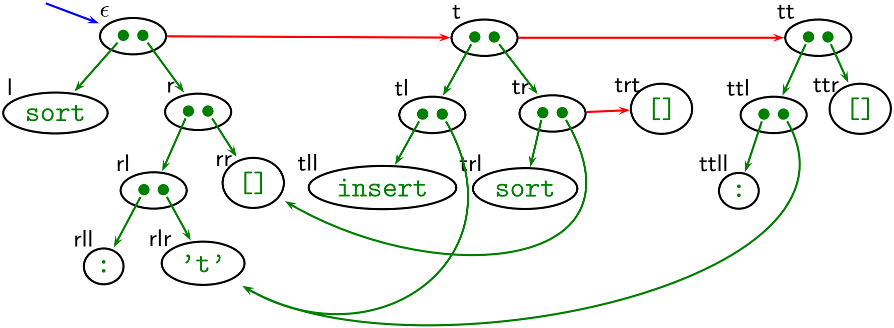
The Node Naming Scheme



Aim

- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes

The Node Naming Scheme



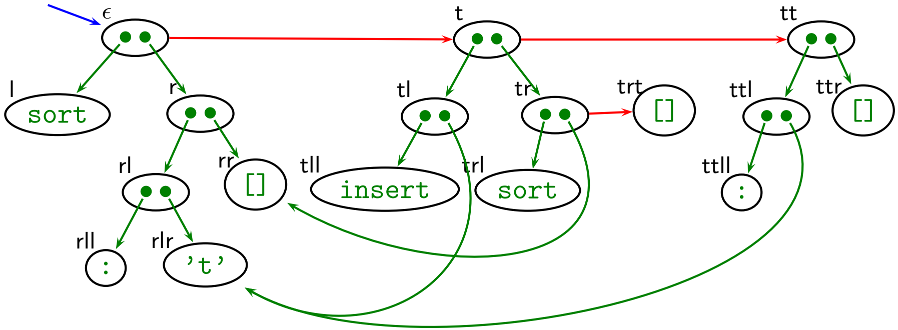
Aim

- 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 Naming Scheme II



- Reduction edge implicitly given through existence of node.
- Node encodes parent; parent = top node of redex causing its creation:

$$\begin{aligned} \mathsf{parent}(n\mathsf{t}) &= n \\ \mathsf{parent}(n\mathsf{l}) &= \mathsf{parent}(n) \\ \mathsf{parent}(n\mathsf{r}) &= \mathsf{parent}(n) \\ \mathsf{parent}(\epsilon) &= \mathsf{undefined} \end{aligned}$$

- Easy to identify right-hand-side of rule: same parent.

The Augmented Redex Trail (ART)

An ART G for start term M, program P and semantics $\cong$ is a partial function from nodes to term constructors, $G : n \mapsto T$, defined by

- The unshared graph representation of M is an ART.
- If G is an ART and
  - $L = R$ an equation of the program P,
  - σ a substitution replacing the variables of the equation by nodes not ending in t,
  - $n \in \mathrm{dom}(G)$ represents $L\sigma$,
  - $n\mathrm{t} \notin \mathrm{dom}(G)$,
  - G' is the unshared graph representation of $R\sigma$,
  - $L\sigma \cong R\sigma$

  then $G \cup G'$ is an ART.

Evaluation order is not fixed.

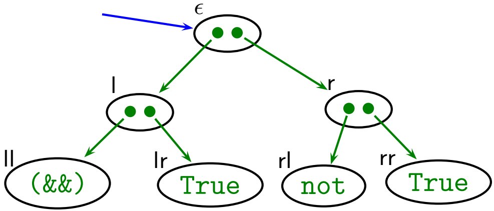
A Reduction Step

If G is an ART and

- $L = R$ an equation of the program P,
- σ a substitution replacing the variables of the equation by nodes not ending in t,
- $n \in \text{dom}(G)$ represents $L\sigma$,
- $n\text{t} \notin \text{dom}(G)$,
- G' is the unshared graph representation of $R\sigma$,
- $L\sigma \cong R\sigma$

then $G \cup G'$ is an ART.

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



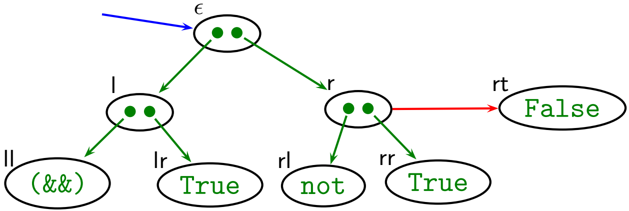
A Reduction Step

If G is an ART and

- $L = R$ an equation of the program P,
- σ a substitution replacing the variables of the equation by nodes not ending in t,
- $n \in \mathrm{dom}(G)$ represents $L\sigma$,
- $n\mathrm{t} \notin \mathrm{dom}(G)$,
- G' is the unshared graph representation of $R\sigma$,
- $L\sigma \cong R\sigma$

then $G \cup G'$ is an ART.

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



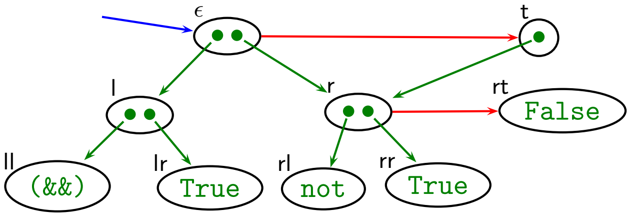
A Reduction Step

If G is an ART and

- $L = R$ an equation of the program P,
- σ a substitution replacing the variables of the equation by nodes not ending in t,
- $n \in \text{dom}(G)$ represents $L\sigma$,
- $n\text{t} \notin \text{dom}(G)$,
- G' is the unshared graph representation of $R\sigma$,
- $L\sigma \cong R\sigma$

then $G \cup G'$ is an ART.

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



Properties of the ART

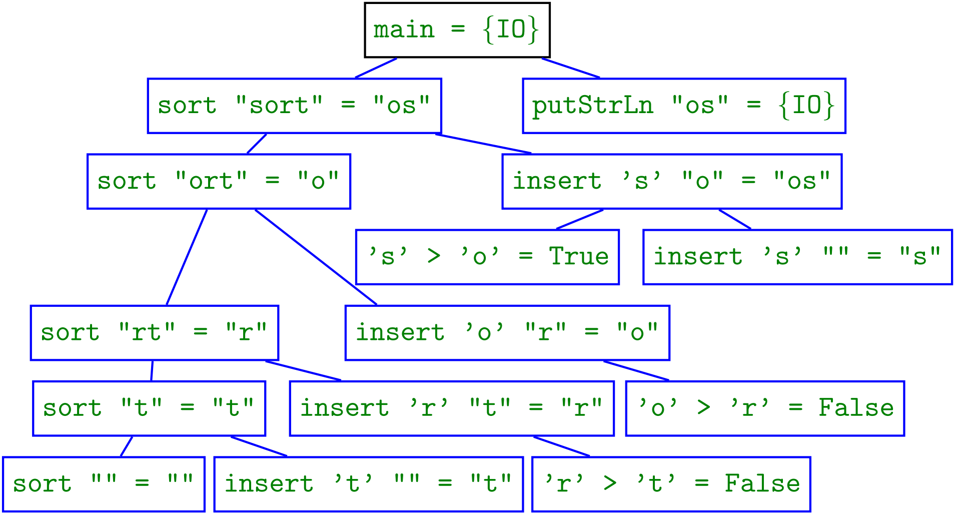
- closed (no dangling nodes)
- domain prefix-closed
- no term constructor contains node ending in t
- only a node ending in t can be an indirection
- if $n\mathsf{l} \in \mathrm{dom}(G)$, then $G(n) = n\mathsf{l}\; m$
- if $n\mathsf{r} \in \mathrm{dom}(G)$, then $G(n) = m\; n\mathsf{r}$
- if $n\mathsf{t} \in \mathrm{dom}(G)$, then n and $n\mathsf{t}$ represent a reduction step
- acyclic
- subcommutative
- ...

Give non-inductive definition of ART based on properties?

Using the ART: 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 Evaluation Dependency 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"`

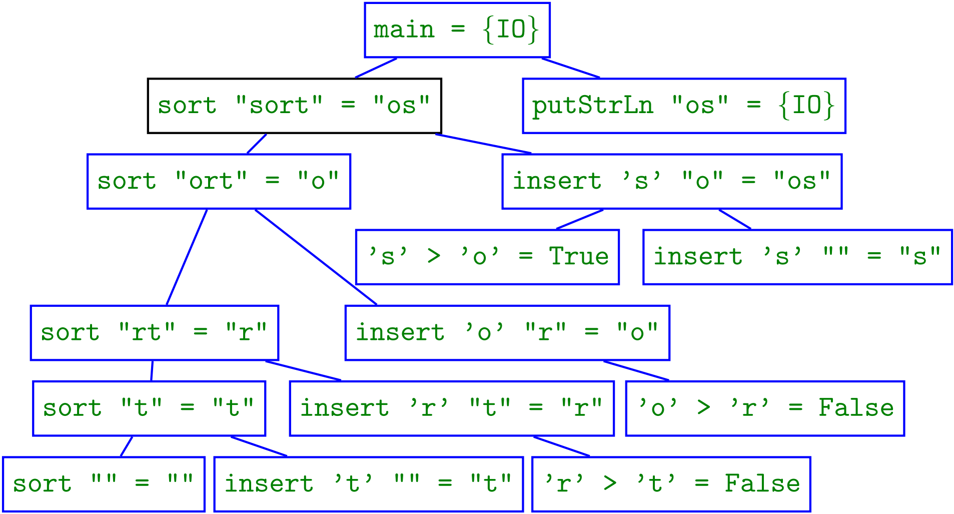
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

The Evaluation Dependency 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"`

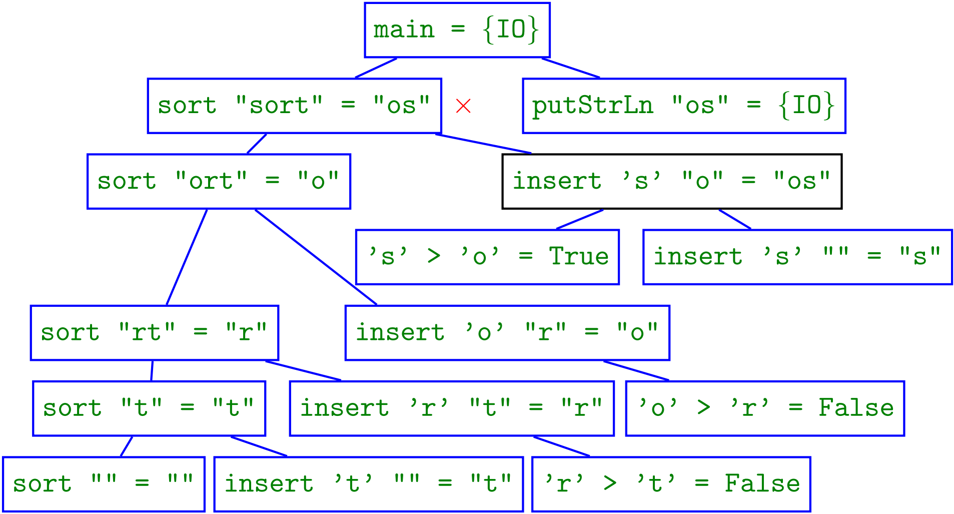
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

The Evaluation Dependency 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"`

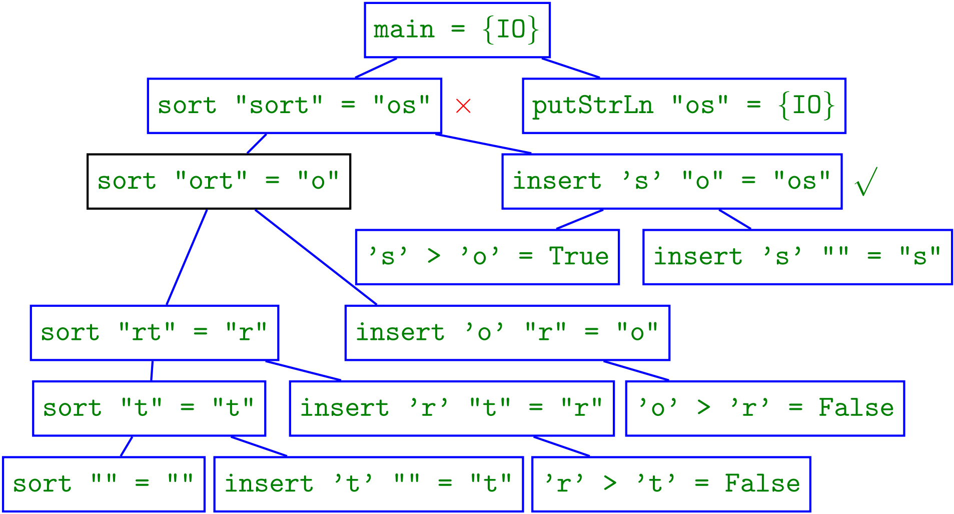
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

The Evaluation Dependency 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"`

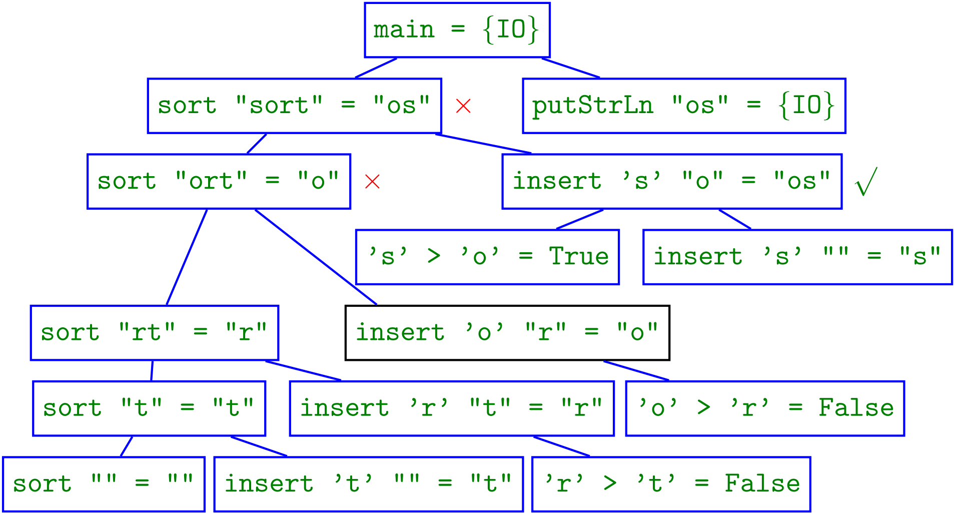
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

The Evaluation Dependency 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"

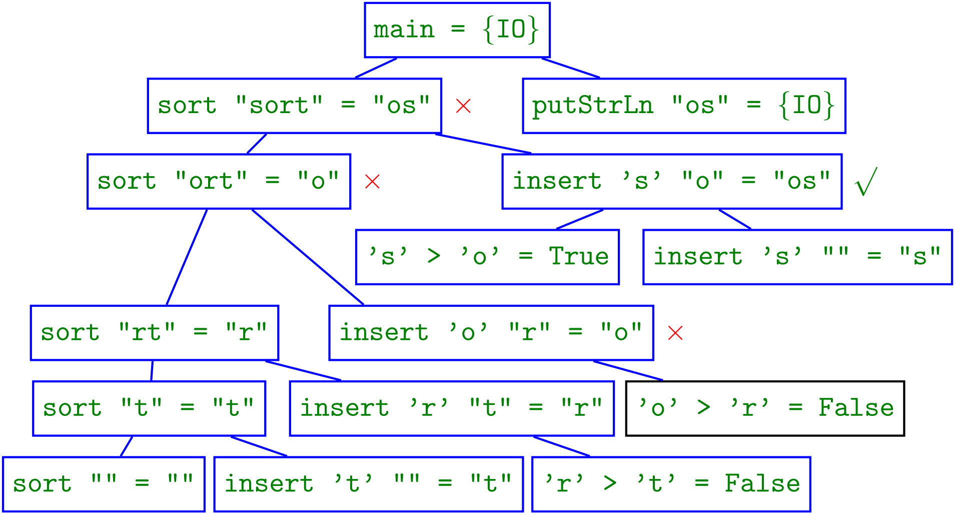
'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

The Evaluation Dependency 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"`

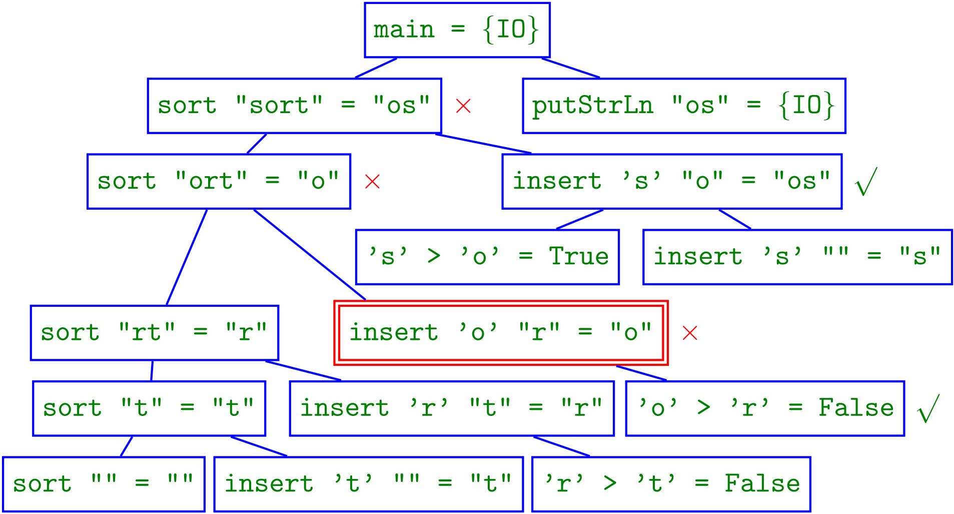
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

The Evaluation Dependency 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"`

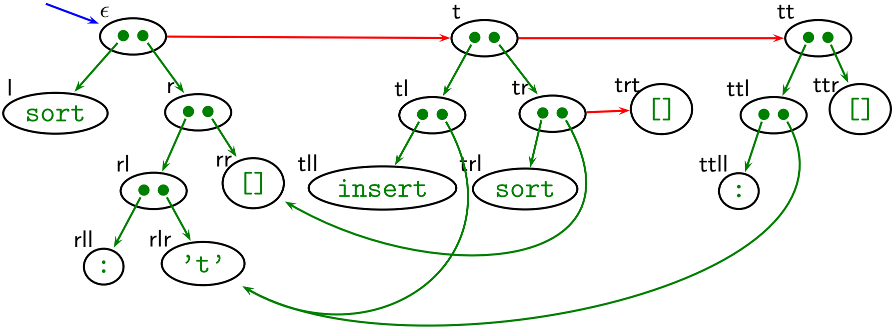
`'o' > 'r' = False` √

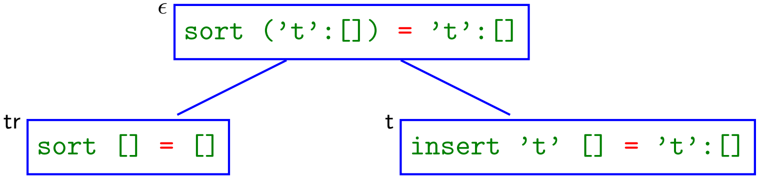
`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

The ART and the Evaluation Dependency Tree





ϵ `sort ('t':[]) = 't':[]`

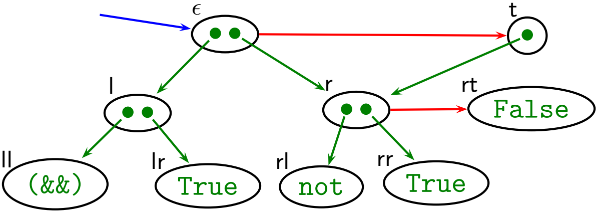
tr `sort [] = []`

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

Conclusions

Summary

- simple model amenable to proof
- contains a wealth of information about computation
- models real-world trace of Haskell tracer Hat
- proved correctness of algorithmic debugging



Future Work

- still play with definitions
- drop non-needed nodes from ART
- model run-time error with error value
- allow local function definitions (⇒ free variables)
- share reductions of constants (⇒ cycles in graph)