Foundations for Tracing Functional Programs and the Correctness of Algorithmic Debugging

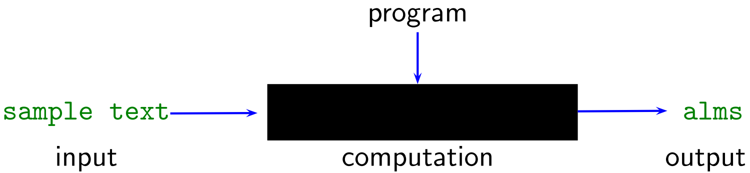
Olaf Chitil and Yong Luo

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

26th April 2006

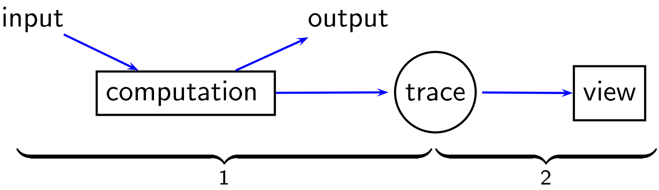
Why Tracing?

```
insert :: Ord a => a -> [a] -> [a]

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

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

main = getLine >>= putStrLn . sort
```



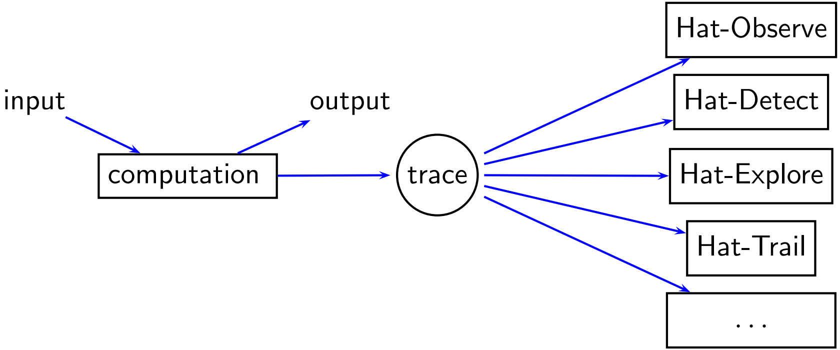
- Locate a fault (wrong output, run-time error, non-termination).
- Comprehend a program.

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.

- Multi-View Tracer



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

Aim: A theoretical model of this trace and its views.

Overview

1. Definition of the Trace through Graph Rewriting
2. Properties of the Trace
3. Views of the Trace
    - Observation of Functions
    - Following Redex Trails
    - Algorithmic Debugging
4. Correctness of Algorithmic Debugging
5. Future Work & Summary

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 must have 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 must have most frequently used features:
  - named functions
  - pattern matching

$\Rightarrow$ Higher-order term rewriting system

```
sort [] = []                              or   sort = foldr insert []
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
```

What is a Good Trace?

Program + input determine every detail 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 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.

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:

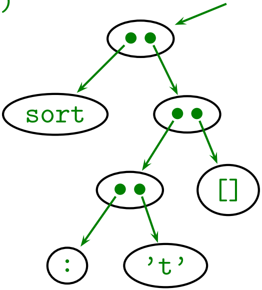
- Term rewriting: A sequence of expressions.
  $t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4 \longrightarrow t_5 \longrightarrow \ldots \longrightarrow t_n$
- Natural semantics: A proof tree.

But

- Lots of redundancy.
- Much structure already lost.

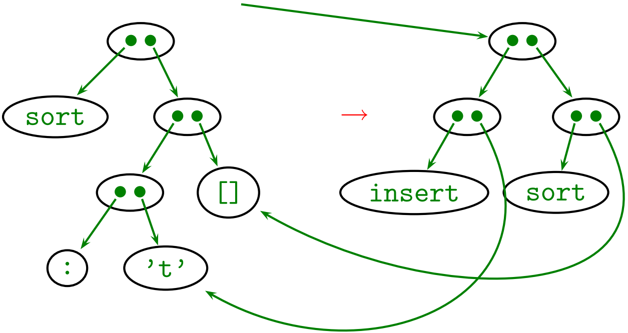
Graph Rewriting I

`sort ('t':[])`




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

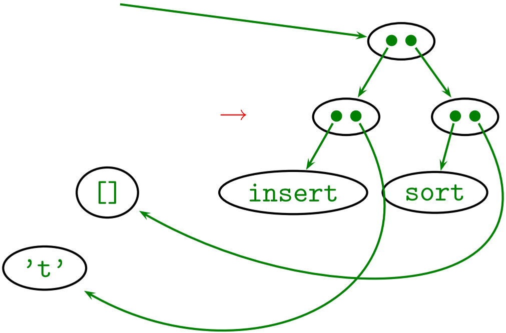
Graph Rewriting I




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

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

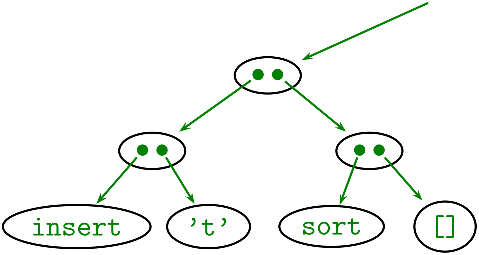
Graph Rewriting I




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

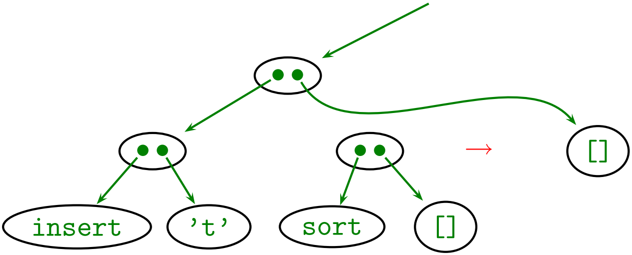
- Create new nodes for right-hand-side.
- Nodes of subexpressions are shared.
- Some old nodes become garbage.

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
```

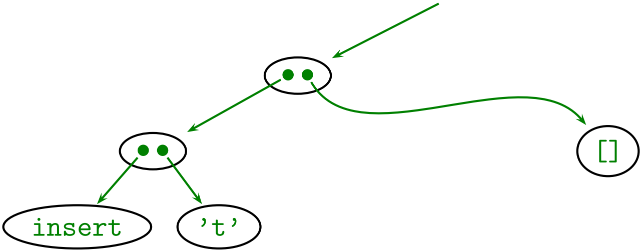
Graph Rewriting II




```
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.

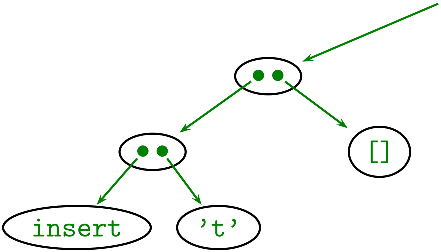
Graph Rewriting II




```
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
```

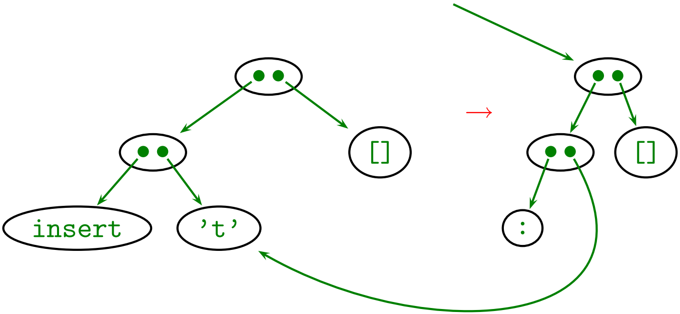
- Application node of redex replaced by new node.

Graph Rewriting III



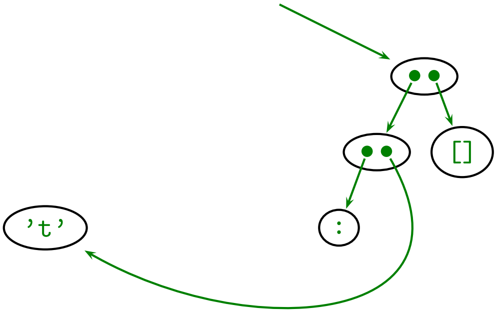

```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

Graph Rewriting III



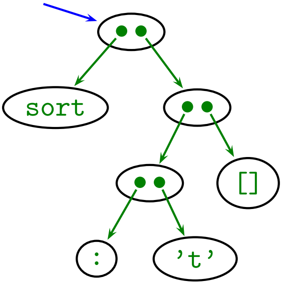

```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

Graph Rewriting III

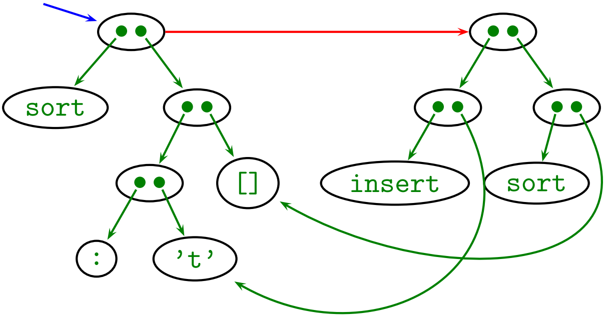



```
sort [] = []
sort (x:xs) = insert x (sort xs)

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

The Trace

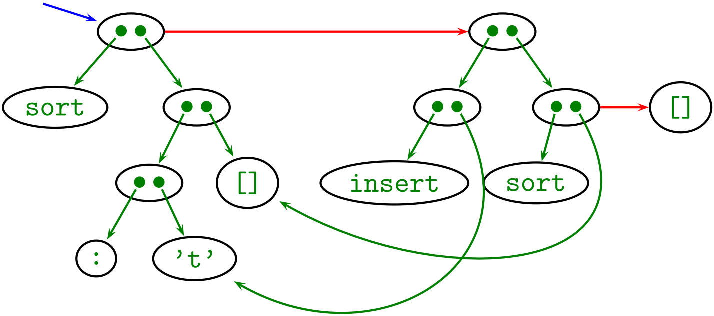


The Trace



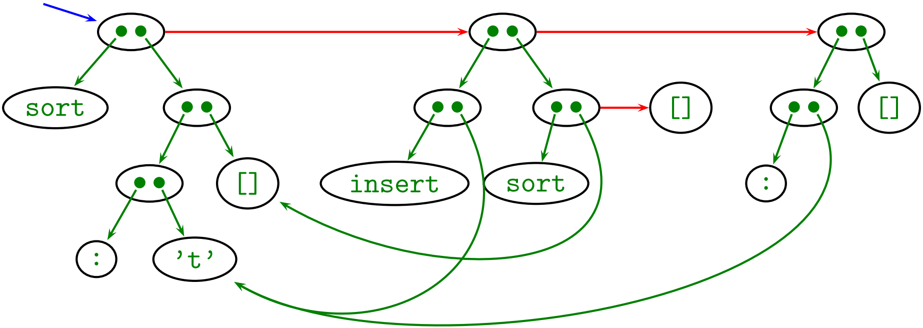
- New nodes for right-hand-side, connected via result pointer.
- Only add to graph, never remove.
- Sharing ensures compact representation.

The Trace



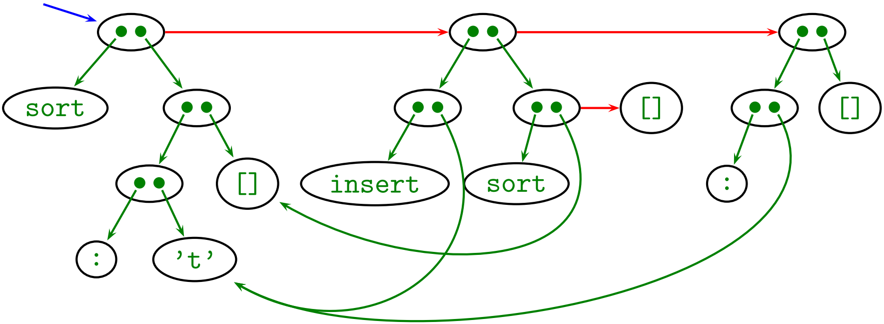
- New nodes for right-hand-side, connected via result pointer.
- Only add to graph, never remove.
- Sharing ensures compact representation.

The Trace



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

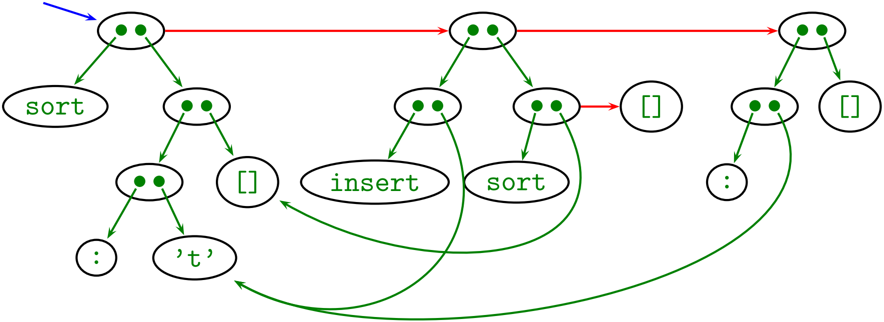
The Node Labels



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

- pointers instead of edges

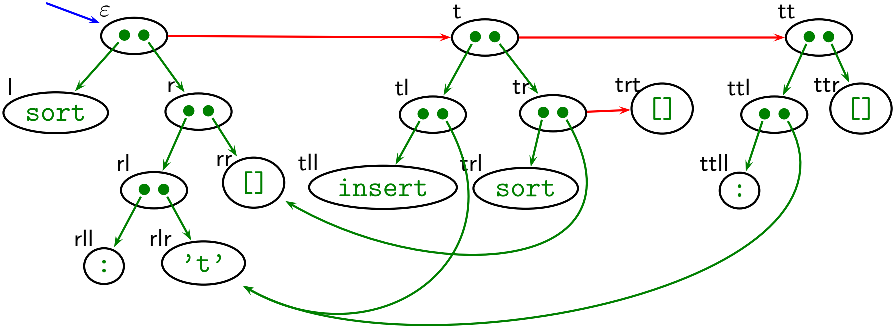
The Node Naming Scheme



Aim

- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes

The Node Naming Scheme



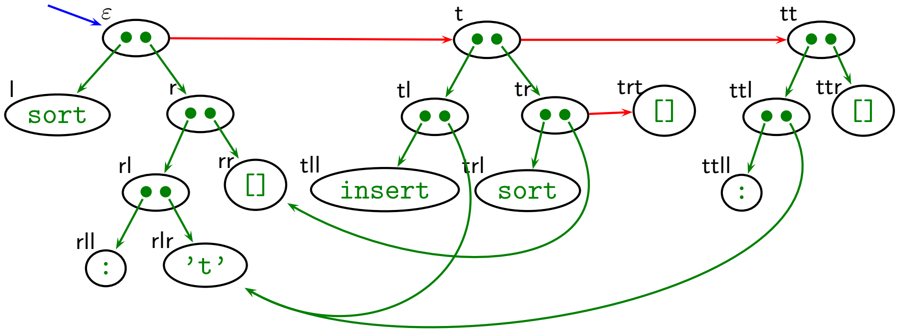
Aim

- 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 = top node of redex causing its creation:

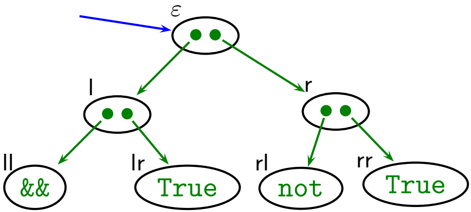
$$\begin{aligned} \text{parent}(n\text{t}) &= n \\ \text{parent}(n\text{l}) &= \text{parent}(n) \\ \text{parent}(n\text{r}) &= \text{parent}(n) \\ \text{parent}(\varepsilon) &= \text{undefined} \end{aligned}$$

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

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
```

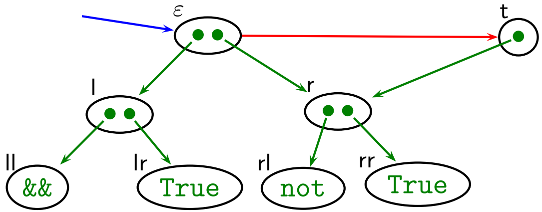


Projections

- 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
```



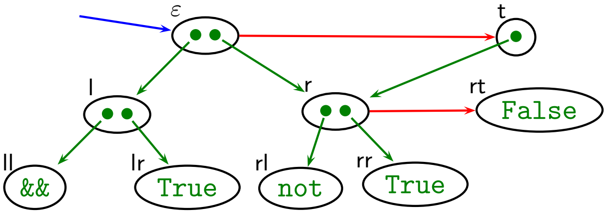
term constructor	T	$:=$	a	atom
		$\mid$	$n\ m$	application of nodes
		$\mid$	n	indirection
atom	a	$:=$	$x \mid C \mid 42 \mid \ldots$	variable, data constructor, ...

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
```



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, ...

The Trace: The Augmented Redex Trail (ART)

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

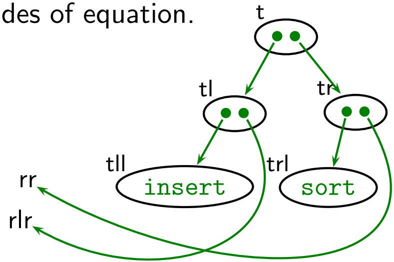
  then $G \cup \mathrm{graph}_G(n\mathrm{t}, R\sigma)$ is a trace.

No evaluation order is fixed.

Unshared Graph Representation

For the initial term and right-hand-sides of equation.

$$\text{graph}(\text{t}, \texttt{insert}\ \text{rlr}\ (\text{sort}\ \text{rr})) =$$



Definition

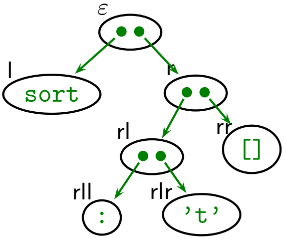
$$\text{graph}(n, a) = \{(n, a)\}$$

$$\text{graph}(n, m) = \{(n, m)\}$$

$$\text{graph}(n, M\,N) = \begin{cases} \{(n, M\,N)\} & \text{, if } M, N \text{ are nodes} \\ \{(n, M\,n\text{r})\} \cup \text{graph}(n\text{r}, N) & \text{, if only } M \text{ is a node} \\ \{(n, n\text{l}\,N)\} \cup \text{graph}(n\text{l}, M) & \text{, if only } N \text{ is a node} \\ \{(n, n\text{l}\,n\text{r})\} \cup \text{graph}(n\text{l}, M) \cup \text{graph}(n\text{r}, N) & \text{, otherwise} \end{cases}$$

Matching

Matching a node with an instance of the left-hand-side of an equation.



$\mathsf{match}_G(\varepsilon, \texttt{sort}\ (\mathsf{rlr}\texttt{:}\mathsf{rr}))$

Definition

$$\mathsf{match}_G(n, M) = \text{if } M \text{ is a node then } n = M \text{ else } \mathsf{matchT}_G(\mathsf{last}_G(n), M)$$

$$\mathsf{matchT}_G(a, M) = (a = M)$$

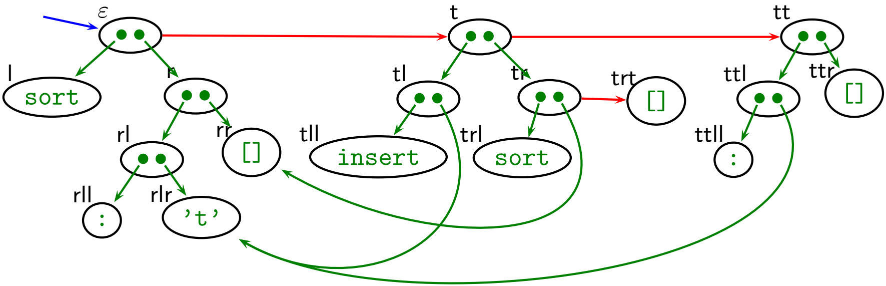
$$\mathsf{matchT}_G(n, M) = \mathsf{matchT}_G(\mathsf{last}_G(n), M)$$

$$\mathsf{matchT}_G(n\, o, M) = \exists N, O.\, (N\, O = M) \wedge \mathsf{match}_G(n, N) \wedge \mathsf{match}_G(o, O)$$

$$\mathsf{last}_G(n) = \text{if } nt \in \mathsf{dom}(G) \text{ then } \mathsf{last}_G(nt) \text{ else } G(n)$$

The Most Evaluated Form of a Node

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



$\text{mef}_G(\text{tr}) = $ `[]`

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

Definition

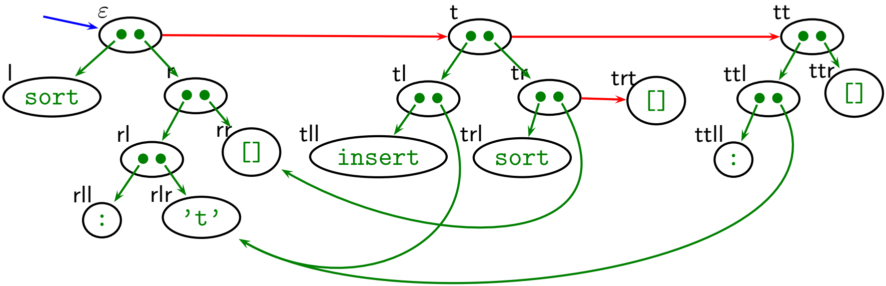
$$\text{mef}_G(n) = \text{mefT}_G(\text{last}_G(n))$$

$$\text{mefT}_G(a) = a$$

$$\text{mefT}_G(n) = \text{mef}_G(n)$$

$$\text{mefT}_G(n\,m) = \text{mef}_G(n)\,\text{mef}_G(m)$$

Redexes and Big-Step Reductions



$$\mathrm{redex}_G(\mathrm{t}) = \texttt{insert 't' []}$$

$$\mathrm{bigstep}_G(\mathrm{t}) = \texttt{insert 't' []} = \texttt{(:) 't' []}$$

Definition

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

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

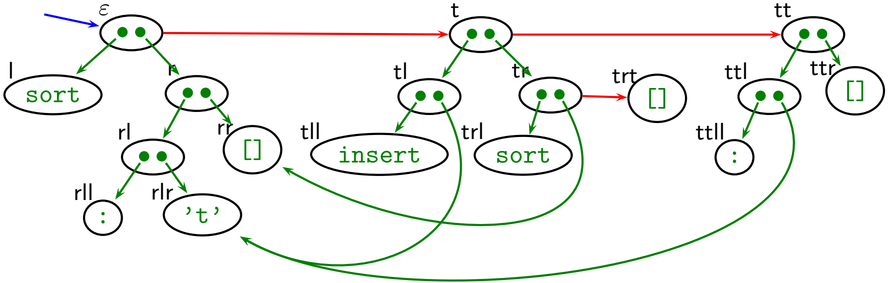
$$\mathrm{bigstep}_G(n) = \mathrm{redex}_G(n) = \mathrm{mef}_G(n)$$

Properties of the ART

- closed (no dangling nodes)
- domain prefix-closed
- acyclic
- strongly confluent
- no application contains a node ending in t
- only a node ending in t can be an indirection
- if $n\mathsf{l} \in \mathsf{dom}(G)$, then $G(n) = n\mathsf{l}\; m$
- if $n\mathsf{r} \in \mathsf{dom}(G)$, then $G(n) = m\; n\mathsf{r}$
- if $n\mathsf{t} \in \mathsf{dom}(G)$, then $\mathsf{redex}_G(n) = L\sigma$ and $\mathsf{reduct}_G(n) = R\sigma$ for some program equation $L = R$ and substitution σ

Give non-inductive definition of ART based on properties?

Reduct of a Small Step Reduction



$reduct_G(\varepsilon)$ = `insert 't' (sort [])`

Definition

$$\mathrm{reduct}_G(n) = \mathrm{reductB}_G(n\mathrm{t})$$

$$\mathrm{reductB}_G(n) = \begin{cases} a & \text{, if } G(n) = a \\ \mathrm{mef}_G(m) & \text{, if } G(n) = m \\ \mathrm{reductB}_G(n\mathrm{l})\,\mathrm{reductB}_G(n\mathrm{r}) & \text{, if } G(n) = n\mathrm{l}\; n\mathrm{r} \\ \mathrm{reductB}_G(n\mathrm{l})\,\mathrm{mef}_G(o) & \text{, if } G(n) = n\mathrm{l}\; o \text{ and } o \neq n\mathrm{r} \\ \mathrm{mef}_G(m)\,\mathrm{reductB}_G(n\mathrm{r}) & \text{, if } G(n) = m\; n\mathrm{r} \text{ and } m \neq n\mathrm{l} \\ \mathrm{mef}_G(m)\,\mathrm{mef}_G(o) & \text{, if } G(n) = m\; o,\ m \neq n\mathrm{l} \text{ and } o \neq n\mathrm{r} \end{cases}$$

Views of the Trace

- Observation of Expressions and Functions
- Following Redex Trails
- Algorithmic Debugging

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"
```

Big step reductions of redex nodes.

Following Redex Trails

Following Redex Trails

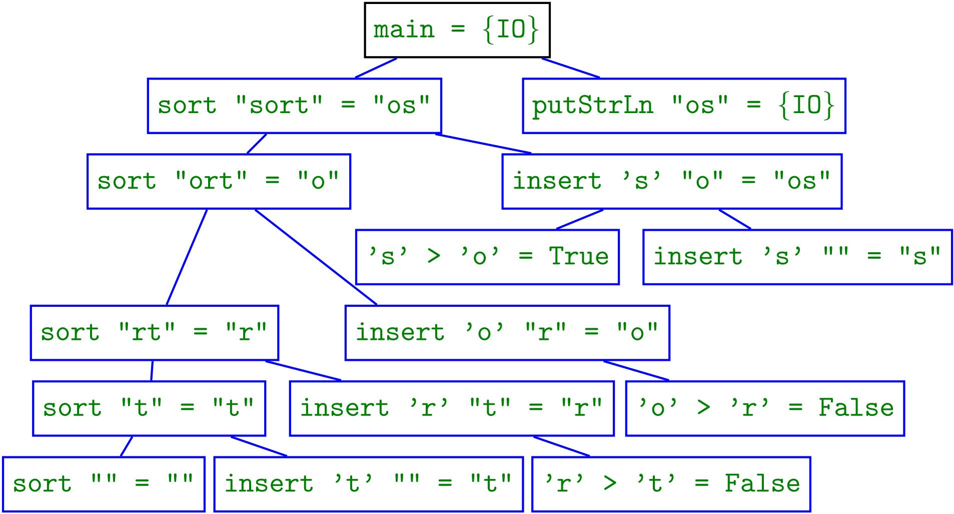
```
Output: ------------------------------------------------
os\n

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 []
```

- Go backwards from observed failure to fault.
- Which redex created this expression?
- To prove: every reduction step reachable from final result.

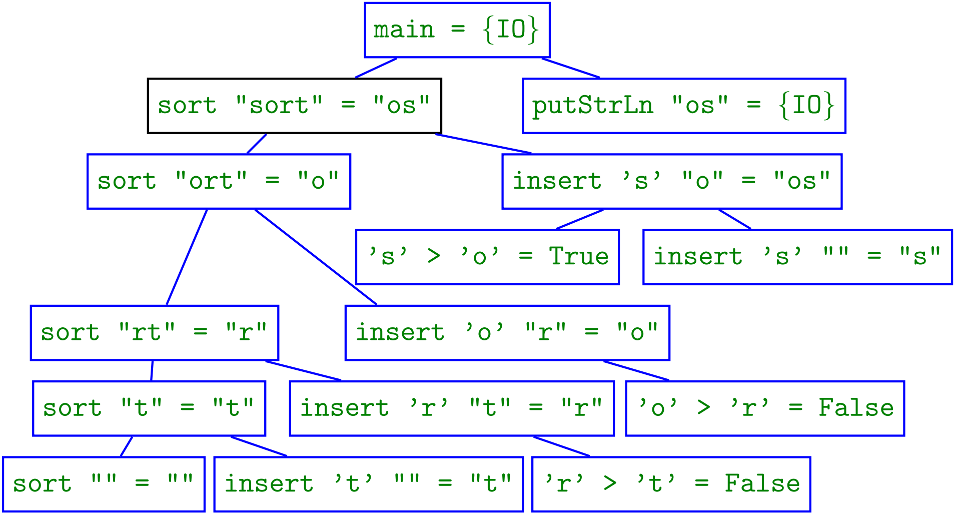
Algorithmic Debugging

```
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
```

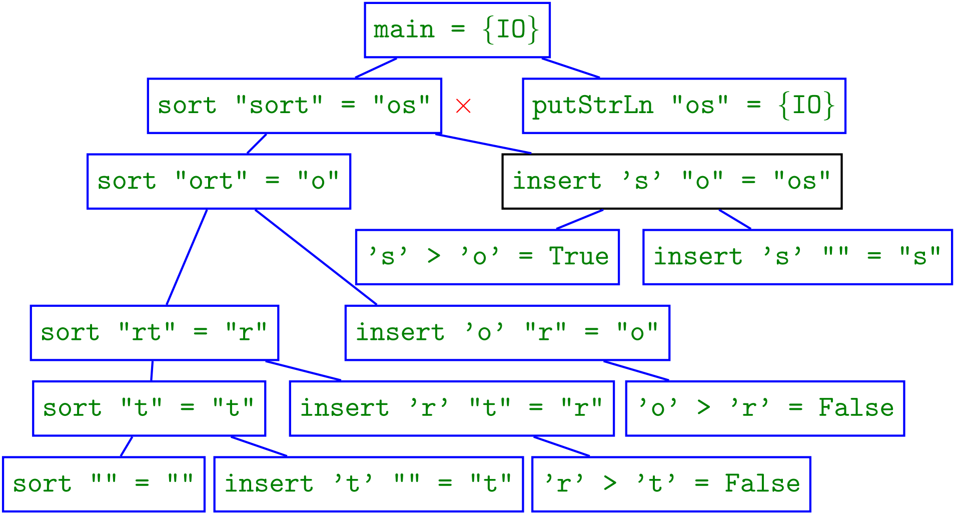
The Evaluation Dependency Tree



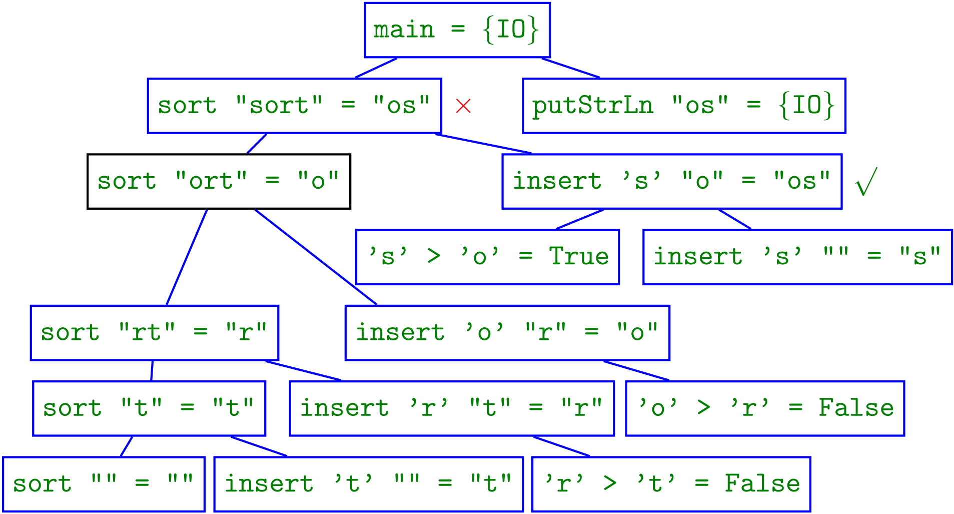
The Evaluation Dependency Tree



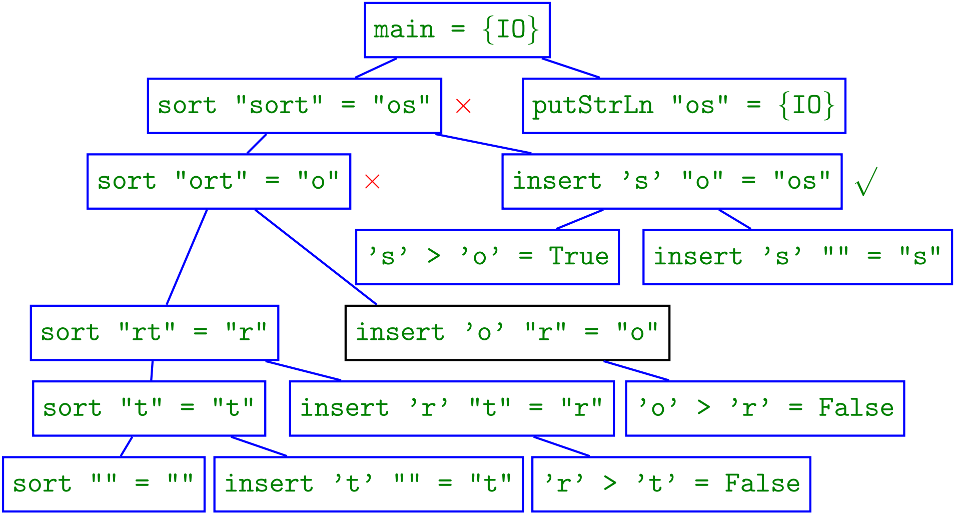
The Evaluation Dependency Tree



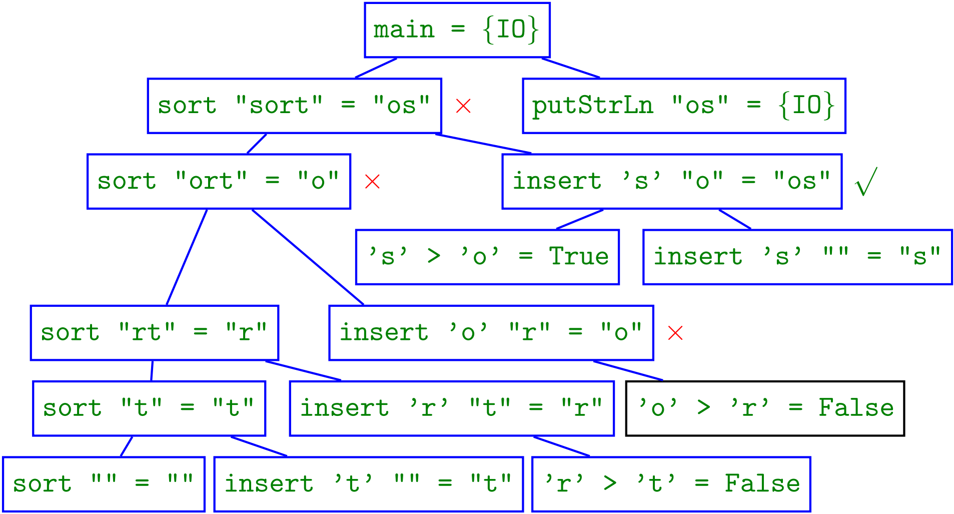
The Evaluation Dependency Tree



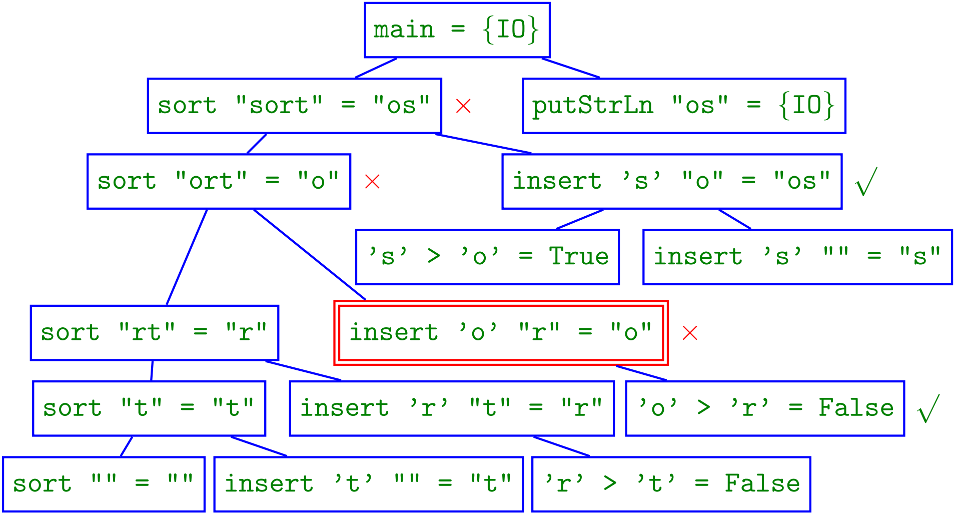
The Evaluation Dependency Tree



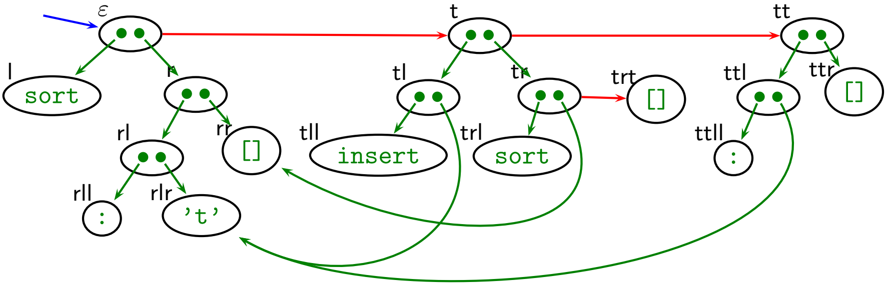
The Evaluation Dependency Tree

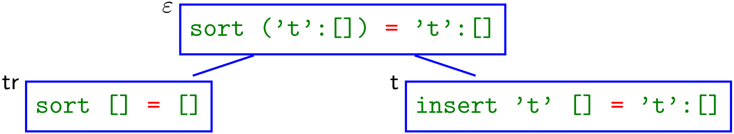


The Evaluation Dependency Tree



The ART and the Evaluation Dependency Tree

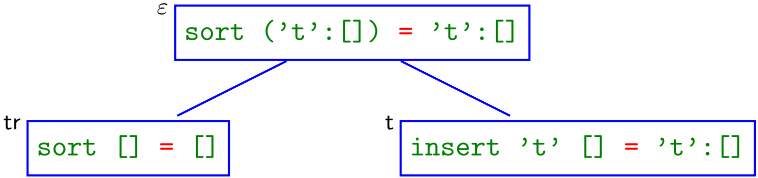




- Every redex node n yields a tree node n labelled $\text{bigstep}_G(n)$.
- Tree node n is child of tree node $\text{parent}(n)$.
- Usually root label $\text{bigstep}_G(\varepsilon) = \texttt{main = ...}$

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.



Definition

Tree node n incorrect $\Leftrightarrow$ $\mathrm{redex}_G(n) \not\cong_I \mathrm{mef}_G(n)$.
Tree node n faulty $\Leftrightarrow$ $\mathrm{redex}_G(n) \not\cong_I \mathrm{reduct}_G(n)$.

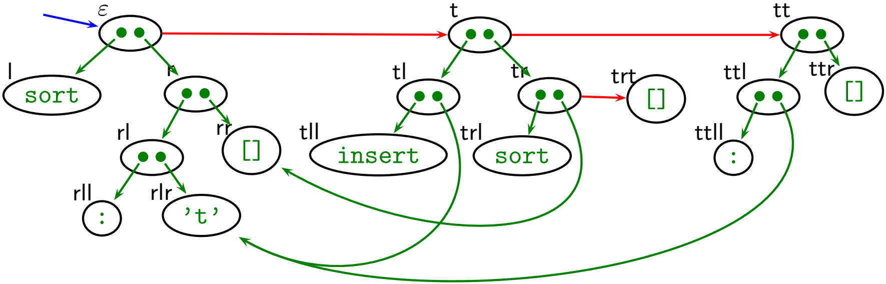
If tree node n faulty, then for its program equation $L = R$ exists substitution σ such that $L\sigma \not\cong_I R\sigma$.

Correctness of Algorithmic Debugging: Main Theorem

Theorem

Let n be a redex node. If for all redex nodes m with $parent(m) = n$ we have $redex_G(m) \cong_I mef_G(m)$, then $reduct_G(n) \cong_I mef_G(n)$.

With $\mathrm{redex}_G(n) \not\cong_I \mathrm{mef}_G(n)$ follows $\mathrm{redex}_G(n) \not\cong_I \mathrm{reduct}_G(n)$.

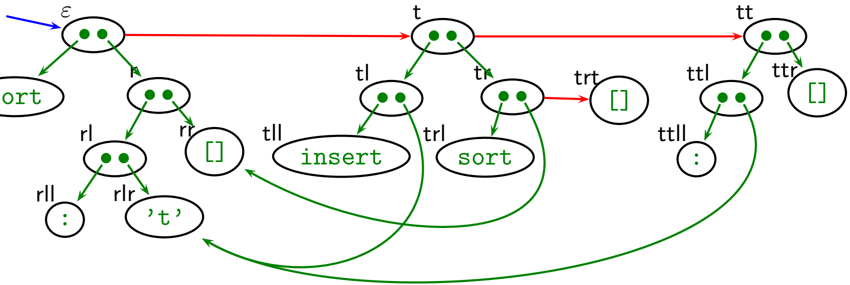


Correctness of Algorithmic Debugging: Proof

Proof.

Generalise property: Let $n \in \mathrm{dom}(G)$. If for all redex nodes m with $\mathrm{parent}(m) = \mathrm{parent}(n)$ we have $\mathrm{redex}_G(m) \cong_\mathrm{I} \mathrm{mef}_G(m)$, then $\mathrm{reductB}_G(n) \cong_\mathrm{I} \mathrm{mef}_G(n)$.

Induction over $\mathrm{hight}_G(n) = \max\{|o| \mid o \in \{\mathrm{l}, \mathrm{r}\}^* \wedge no \in \mathrm{dom}(G)\}$. $\square$



Future Work

- Still play with definitions.
- Extend model further:
  - Drop non-needed nodes from ART (unevaluated expressions).
  - Model run-time error with error value.
  - Allow local function definitions ($\Rightarrow$ free variables).
  - Share reductions of constants ($\Rightarrow$ cycles in graph).
  - Describe strict and mixed semantics.
- Prove further properties.

Conclusions

- 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.