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?

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

```haskell
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
```
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

\[ \text{input} \rightarrow \text{computation} \rightarrow \text{trace} \rightarrow \text{output} \]

- 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
Launchbury’s and related semantics
- Subset of $\lambda$-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
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

⇒ Higher-order term rewriting system

\[
\begin{align*}
\text{sort } [] &= [] \\
\text{sort } (x:xs) &= \text{insert } x \ (\text{sort } xs) \\
\text{insert } x \ [] &= [x] \\
\text{insert } x \ (y:ys) &= \text{if } x > y \text{ then } y:(\text{insert } x \ ys) \text{ else } x:ys
\end{align*}
\]
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.
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.

But
- Lots of redundancy.
- Much structure already lost.
sort (’t’:[]) = [']
sort (x:xs) = insert x (sort xs)
sort [] = []
sort (x:xs) = insert x (sort xs)

- Create new nodes for right-hand-side.
- Nodes of subexpressions are shared.
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.
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
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.
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.
\[
\text{sort} \; [] = [] \\
\text{sort} \; (x:xs) = \text{insert} \; x \; (\text{sort} \; xs)
\]

\[
\text{insert} \; x \; [] = [x] \\
\text{insert} \; x \; (y:ys) = \text{if} \; x > y \; \text{then} \; y:(\text{insert} \; x \; ys) \; \text{else} \; x:ys
\]
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
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

```
sort
   .
   .
   .
   [ ]
  /
)/
\   
  :   't'
```

Olaf Chitil and Yong Luo (Kent, UK)
New nodes for right-hand-side, connected via result pointer.
Only add to graph, never remove.
Sharing ensures compact representation.
New nodes for right-hand-side, connected via result pointer.

Only add to graph, never remove.

Sharing ensures compact representation.
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 \[ T := a \mid nm \] atom

atom \[ a := f \mid C \mid 42 \mid \ldots \] defined variable, data constructor
atomic literal, \ldots

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
Reduction edge implicitly given through existence of node.

Node encodes parent = top node of redex causing its creation:

\[
\begin{align*}
\text{parent}(nt) &= n \\
\text{parent}(nl) &= \text{parent}(n) \\
\text{parent}(nr) &= \text{parent}(n) \\
\text{parent}(\varepsilon) &= \text{undefined}
\end{align*}
\]

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)

\[
\begin{align*}
\text{True} \land x = x \\
\text{not True} = \text{False}
\end{align*}
\]
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 \textit{indirection} as result.

\begin{align*}
\text{True} & \land x = x \\
\text{not True} & = \text{False}
\end{align*}

\begin{math}
\begin{array}{c}
& \varepsilon \\
& \text{ll} \\
&& \text{lr} \\
& \text{ll} \\
& \text{lr} \\
& \text{rr} \\
\end{array}
\end{math}

\begin{align*}
\text{term constructor} & \quad T := a \\
& | n m \\
& | n \\
\text{atom} & \quad a := x | C | 42 | \ldots \quad \text{variable, data constructor, \ldots}
\end{align*}
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.

\[
\begin{align*}
\text{True} & \land x = x \\
\text{not True} & \equiv \text{False}
\end{align*}
\]

\[
\begin{align*}
\text{term constructor} \quad T & := a \\
& \mid n \ m & \text{atom} \\
& \mid n & \text{application of nodes} \\
& \mid \text{indirection}
\end{align*}
\]

\[
\begin{align*}
\text{atom} \quad a & := x \mid C \mid 42 \mid \ldots & \text{variable, data constructor,} \\
& \text{...}
\end{align*}
\]
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,
  - $\text{match}_G(n, L\sigma)$,
  - $nt \notin \text{dom}(G)$,

  then $G \cup \text{graph}_G(nt, R\sigma)$ is a trace.

No evaluation order is fixed.
Unshared Graph Representation

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

\[
\text{graph}(t, \text{insert rlr (sort rr)}) = \text{rr}
\]

Definition

\[
\begin{align*}
\text{graph}(n, a) &= \{(n, a)\} \\
\text{graph}(n, m) &= \{(n, m)\} \\
\text{graph}(n, MN) &= \\
&= \begin{cases} \\
\{(n, MN)\} & , \text{if } M, N \text{ are nodes} \\
\{(n, M nr)\} \cup \text{graph}(nr, N) & , \text{if only } M \text{ is a node} \\
\{(n, nl N)\} \cup \text{graph}(nl, M) & , \text{if only } N \text{ is a node} \\
\{(n, nl nr)\} \cup \text{graph}(nl, M) \cup \text{graph}(nr, N) & , \text{otherwise}
\end{cases}
\end{align*}
\]
Matching a node with an instance of the left-hand-side of an equation.

\[
\text{match}_G(\varepsilon, \text{sort (rlr:rr)})
\]

Definition

\[
\begin{align*}
\text{match}_G(n, M) & = \text{if } M \text{ is a node then } n = M \text{ else } \text{match}_T G(\text{last}_G(n), M) \\
\text{match}_T G(a, M) & = (a = M) \\
\text{match}_T G(n, M) & = \text{match}_T G(\text{last}_G(n), M) \\
\text{match}_T G(n \circ, M) & = \exists N, O. (N \circ O = M) \land \text{match}_G(n, N) \land \text{match}_G(o, O) \\
\text{last}_G(n) & = \text{if } nt \in \text{dom}(G) \text{ then } \text{last}_G(nt) \text{ else } G(n)
\end{align*}
\]
A node represents many terms, in particular a most evaluated one.

\[
mef_G(tr) = []
\]
\[
mef_G(\varepsilon) = (:) \ 't' \ []
\]

**Definition**

\[
mef_G(n) = mefT_G(\text{last}_G(n))
\]
\[
mefT_G(a) = a
\]
\[
mefT_G(n) = mef_G(n)
\]
\[
mefT_G(n \cdot m) = mef_G(n) \cdot mef_G(m)
\]
Redexes and Big-Step Reductions

\[
\text{redex}_G(t) = \text{insert} \ 't' \ [\[] \\
\text{bigstep}_G(t) = \text{insert} \ 't' \ [\[] = (:) \ 't' \ [\[]
\]

**Definition**

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

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

\[
\text{bigstep}_G(n) = \text{redex}_G(n) = \text{mef}_G(n)
\]
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 $nl \in \text{dom}(G)$, then $G(n) = nl m$
- if $nr \in \text{dom}(G)$, then $G(n) = m nr$
- if $nt \in \text{dom}(G)$, then $\text{redex}_G(n) = L\sigma$ and $\text{reduct}_G(n) = R\sigma$ for some program equation $L = R$ and substitution $\sigma$

Give non-inductive definition of ART based on properties?
Reduct of a Small Step Reduction

\[ \text{reduct}_G(\varepsilon) = \text{insert 't' (sort [])} \]

**Definition**

\[ \text{reduct}_G(n) = \text{reductB}_G(nt) \]

\[ \text{reductB}_G(n) = \begin{cases} 
    a, & \text{if } G(n) = a \\
    \text{mef}_G(m), & \text{if } G(n) = m \\
    \text{reductB}_G(nl) \text{reductB}_G(nr), & \text{if } G(n) = nl \text{ nr} \\
    \text{reductB}_G(nl) \text{mef}_G(o), & \text{if } G(n) = nl \text{ o and } o \neq nr \\
    \text{mef}_G(m) \text{reductB}_G(nr), & \text{if } G(n) = m \text{ nr and } m \neq nl \\
    \text{mef}_G(m) \text{mef}_G(o), & \text{if } G(n) = m \text{ o, } m \neq nl \text{ and } o \neq nr 
\end{cases} \]
Views of the Trace

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

Output: ---------------------------------------------
osc

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

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"

'0' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False
The Evaluation Dependency Tree

main = {IO}

sort "sort" = "os"

sort "ort" = "o"

's' > 'o' = True

insert 's' "o" = "os"

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

'o' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

putStrLn "os" = {IO}
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"

'o' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"
The Evaluation Dependency Tree

main = \{\text{IO}\}

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

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

\text{insert 's' "o" = "os"}

\text{\textbackslash 's' > \textbackslash 'o' = True}

\text{insert 's' "" = "s"}

\text{\textbackslash 'o' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash o' > \textbackslash r' = False}

\text{\textbackslash r' > \textbackslash t' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = False}

\text{\textbackslash 'r' > \textbackslash 't' = False}

\text{\textbackslash 't' > \textbackslash 'r' = 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"

insert 'o' "r" = "o"

's' > 'o' = True

insert 's' "" = "s"

sort "t" = "t"

insert 'r' "t" = "r"

'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil and Yong Luo (Kent, UK) Foundations for Tracing 26th April 2006 39 / 46
The Evaluation Dependency Tree

- **main** = \{IO\}
  - **sort** "sort" = "os"
    - **sort** "ort" = "o"
      - **sort** "rt" = "r"
    - **sort** "t" = "t"
  - **sort** "t" = "t"
    - **sort** "" = ""

- **putStrLn** "os" = \{IO\}
  - **insert** 's' "o" = "os"
    - 's' > 'o' = True
      - **insert** 's' "" = "s"
    - **insert** 'r' "t" = "r"
      - 'o' > 'r' = False
      - **insert** 'r' "" = "r"
        - 'r' > 't' = False

Olaf Chitil and Yong Luo (Kent, UK)
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" ×

insert 'r' "t" = "r"

'o' > 'r' = False ✓

sort "t" = "t"

insert 't' "" = "t"

'r' > 't' = False

sort "" = ""

sort "" = ""
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 $\text{bigstep}_G(\varepsilon) = \text{main} = \ldots$

\[\varepsilon\]

\[\text{sort ('t':[])} = 't':[]\]

\[\text{sort []} = []\]

\[\text{insert 't' []} = 't':[]\]
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.

\[ \varepsilon \quad \text{sort ('t':[])} = 't':[] \]

\[ \text{tr} \quad \text{sort [] = []} \quad \text{t} \quad \text{insert 't' [] = 't':[]} \]

**Definition**

Tree node $n$ incorrect $\iff$ \text{redex}_G(n) \not\equiv_1 \text{mef}_G(n)$.

Tree node $n$ faulty $\iff$ \text{redex}_G(n) \not\equiv_1 \text{reduct}_G(n)$.

If tree node $n$ faulty, then for its program equation $L = R$ exists substitution $\sigma$ such that $L\sigma \not\equiv_1 R\sigma$. 
Correctness of Algorithmic Debugging: Main Theorem

**Theorem**

Let $n$ be a redex node. If for all redex nodes $m$ with $\text{parent}(m) = n$ we have $\text{redex}_G(m) \equiv_I \text{mef}_G(m)$, then $\text{reduct}_G(n) \equiv_I \text{mef}_G(n)$.

With $\text{redex}_G(n) \not\equiv_I \text{mef}_G(n)$ follows $\text{redex}_G(n) \not\equiv_I \text{reduct}_G(n)$. 
Correctness of Algorithmic Debugging: Proof

Proof.

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

Induction over $\text{hight}_G(n) = \max\{|o| \mid o \in \{l, r\}^* \land no \in \text{dom}(G)\}$.
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 (⇒ free variables).
- Share reductions of constants (⇒ 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.