Structure and Properties of Traces for Functional Programs

Olaf Chitil and Yong Luo

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

5th October 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] = [x]
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
Launchbury’s and related semantics

- Subset of $\lambda$-calculus plus \textit{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 $\lambda$-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

$\Rightarrow$ Higher-order term rewriting system

\[
\begin{align*}
\text{sort } \texttt{[]} &= \texttt{[]} \\
\text{sort } (\texttt{x:xs}) &= \texttt{insert x (sort xs)} \\
\text{insert } \texttt{x }\texttt{[]} &= \texttt{[x]} \\
\text{insert } \texttt{x }\texttt{(y:ys)} &= \texttt{if } x > y \texttt{ then } y:(\texttt{insert } x \texttt{ ys}) \texttt{ else } x : \texttt{ys}
\end{align*}
\]
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. 
  \[ t_1 \to t_2 \to t_3 \to t_4 \to t_5 \to \ldots \to t_n \]

- Natural semantics: A proof tree.

But

- Lots of redundancy.
- Much structure already lost.
```
sort ('t':[]) = [t]  

sort [] = []  

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.
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
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
• 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.
• 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 \ |
\quad \quad \quad \quad \quad \quad n m \ ]

atom \[ a \ ::= \ f \ | \ C \ | \ 42 \ | \ldots \ ]

pointers instead of edges
The Node Naming Scheme

Aim
- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes
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)

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

```
\begin{tikzpicture}
  \node (e) at (0,0) {$\varepsilon$};
  \node (ll) at (-2,-1) {&&};
  \node (lr) at (-1,-1) {lr};
  \node (rl) at (1,-1) {rl};
  \node (rr) at (2,-1) {rr};
  \node (true) at (0,-2) {True};
  \node (not) at (1,-2) {not};
  \node (true2) at (2,-2) {True};

  \draw[->] (e) -- (ll);
  \draw[->] (e) -- (lr);
  \draw[->] (e) -- (rl);
  \draw[->] (e) -- (rr);
  \draw[->] (ll) -- (true);
  \draw[->] (lr) -- (true);
  \draw[->] (rl) -- (not);
  \draw[->] (rr) -- (true2);
\end{tikzpicture}
```
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

\[ n m \]

application of nodes

\[ n \]

indirection

atom \( a := \)

\[ x | C | 42 | \ldots \]

variable, data constructor, \ldots
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.

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

```
term constructor  \[ T := a \]
|    \[ n m \]  application of nodes
|    \[ n \]      indirection

atom  \[ a := x \mid C \mid 42 \mid \ldots \]  variable, data constructor, \ldots
```
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 $M$, $\text{graph}(\varepsilon, M)$, 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 $G \cup \text{graph}(nt, R\sigma)$ is a trace.

No evaluation order is fixed.
For the initial term and right-hand-sides of equation.

\[
\text{graph}(t, \text{insert } \text{rlr} (\text{sort } \text{rr})) = \begin{cases}
\text{rr} & , \text{if } M, N \text{ are nodes} \\
\text{(n, M n)} \cup \text{graph(nr, N)} & , \text{if only } M \text{ is a node} \\
\text{(n, nl N)} \cup \text{graph(nl, M)} & , \text{if only } N \text{ is a node} \\
\text{(n, nl nr)} \cup \text{graph(nl, M)} \cup \text{graph(nr, N)} & , \text{otherwise}
\end{cases}
\]
Matching

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

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

Definition

\[
\begin{align*}
[n]_G &= \text{if } nt \in \text{dom}(G) \text{ then } [nt]_G \text{ else if } \exists m. G(n) = m \text{ then } [m]_G \text{ else } n \\
\text{match}_G(o, a) &= (G(o) = a) \\
\text{match}_G(o, M N) &= \exists m, n. (G(o) = m n) \land \\
&\quad (m = M) \lor \text{match}_G([m]_G, M)) \land \\
&\quad (n = N) \lor \text{match}_G([n]_G, N)) \\
\text{match}_G(o, m) &= \text{false}
\end{align*}
\]
Matching: Alternative Definitions

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

**Definition**

\[
\text{match}_G(o, a) = (G(o) = a)
\]

\[
\text{match}_G(o, M N) = \exists m, n. (G(o) = m n) \land
\]

\[
(\text{if } M \text{ is a node then } (m = M) \text{ else } \text{match}_G([m]_G, M)) \land
\]

\[
(\text{if } N \text{ is a node then } (n = N) \text{ else } \text{match}_G([n]_G, N))
\]

\[
\text{match}_G(o, m) = (o = m)
\]
The Most Evaluated Form of a Node

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

\[ \text{mef}_G(tr) = [] \]
\[ \text{mef}_G(\varepsilon) = (:) \quad 't' \quad [] \]

Definition

\[ \text{mef}_G(n) = \text{mefT}_G(G([n]_G)) \]
\[ \text{mefT}_G(a) = a \]
\[ \text{mefT}_G(n \, m) = \text{mef}_G(n) \, \text{mef}_G(m) \]
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

\[
\begin{align*}
\text{if } nl &\in \text{dom}(G), \text{ then } G(n) = nl \cdot m \\
\text{if } nr &\in \text{dom}(G), \text{ then } G(n) = m \cdot nr \\
\text{if } nt &\in \text{dom}(G), \text{ then } \text{redex}_G(n) = L\sigma \text{ and } \text{reduct}_G(n) = R\sigma \\
\end{align*}
\]

for some program equation \( L = R \) and substitution \( \sigma \)

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

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

**Definition**

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

\[
\text{reduct}_P_G(p, n) = \text{if} \ \text{parent}(n) = p \ \text{then} \ \text{reduct}_T_G(p, G(n)) \ \text{else} \ \text{mef}_G(n)
\]

\[
\text{reduct}_T_G(p, a) = a
\]

\[
\text{reduct}_T_G(p, m) = \text{mef}_G(m)
\]

\[
\text{reduct}_T_G(p, n \ o) = \text{reduct}_P_G(p, n) \ \text{reduct}_P_G(p, o)
\]
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 = \{\text{IO}\}

\text{sort} \ "\text{sort}" = \ "\text{os}\"

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

\text{sort} \ "\text{rt}" = \ "\text{r}\"

\text{sort} \ "\text{t}" = \ "\text{t}\"

\text{sort} \ "\"

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

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

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

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

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

\text{insert} 'o' "\text{r}" = "\text{o}\"

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

\text{insert} 'r' "\text{t}" = "\text{r}\"

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

\text{insert} 't' "\" = "\text{t}\"

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

'o' > 'r' = False

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 5th October 2006 35 / 47
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"

'r' > 't' = False

Olaf Chitil and Yong Luo (Kent, UK) Foundations for Tracing 5th October 2006 35 / 47
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"

insert 'r' "t" = "r" 'o' > 'r' = False

insert 't' "" = "t" 'r' > 't' = False

sort "" = "" sort "t" = "t"

sort "" = ""
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

insert 't' "" = "t"

' r' > 't' = False
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 $\text{bigstep}_G(\varepsilon) = \text{main} = \ldots$
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 $\iff$ redex$_G(n) \not\approx_I \text{mef}_G(n)$.
Tree node $n$ faulty $\iff$ redex$_G(n) \not\approx_I \text{reduct}_G(n)$.

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

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

With \( \text{redex}_G(n) \not\sim I \text{mef}_G(n) \) follows \( \text{redex}_G(n) \not\sim I \text{reduct}_G(n) \).
Proof.

Generalise property: Let \( n \in \text{dom}(G) \). If for all redex nodes \( m \) with \( \text{parent}(m) = \text{parent}(n) \) we have \( \text{redex}_G(m) \equiv_1 \text{mef}_G(m) \), then \( \text{reduct}_G(n) \equiv_1 \text{mef}_G(n) \).

Induction over \( \text{height}_G(n) = \max\{|o| \mid o \in \{l, r\}^\ast \land no \in \text{dom}(G)\} \).

\[ \square \]
Conclusions

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