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.

input  \[\text{computation}\]  output

1 \[\text{trace}\]  2 \[\text{view}\]
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 $\lambda$-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 has most frequently used features:
  - named functions
  - pattern matching
The Programming Language

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 has most frequently used features:
  - named functions
  - pattern matching

$\implies$ Higher-order term rewriting system

\[
\text{sort } [] = [] \\
\text{sort } (x:xs) = \text{insert } x \ (\text{sort } xs)
\]

\[
\text{insert } x \ [] = [x] \\
\text{insert } x \ (y:ys) = \text{if } x > y \ \text{then } y: (\text{insert } x \ ys) \ \text{else } x:ys
\]
sort ('t':[])
Graph Rewriting I

\[
\text{sort} \; [] = [] \\
\text{sort} \; (x:xs) = \text{insert} \; x \; (\text{sort} \; xs)
\]

- 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
The Trace

sort

[]

: 't'
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 \( T := a \) atom
\[ \mid nm \] application of nodes

atom \( a := x \mid C \mid 42 \mid \ldots \) variable, data constructor
atomic literal, \ldots

- 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, \ldots

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

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

term constructor \( T \) := \( a \) atom
\| \( n m \) application of nodes
\| \( n \) indirection

atom \( a \) := \( x \) | \( C \) | 42 | ... variable, data constructor
atomic literal, ...

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

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

```
(* Diagram *)
```

Olaf Chitil and Yong Luo (Kent, UK)
A Theory of Tracing
1st April 2006 14 / 27
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; parent = top node of redex causing its creation:

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

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

Evaluation order is not fixed.
A Reduction Step

If $G$ is an ART and

- $L = R$ an equation of the program $P$,
- $\sigma$ a substitution replacing the variables of the equation by nodes not ending in $t$,
- $n \in \text{dom}(G)$ represents $L\sigma$,
- $nt \notin \text{dom}(G)$,
- $G'$ 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 $P$,
- $\sigma$ a substitution replacing the variables of the equation by nodes not ending in $t$,
- $n \in \text{dom}(G)$ represents $L_\sigma$,
- $nt \notin \text{dom}(G)$,
- $G'$ 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 $P$,
- $\sigma$ a substitution replacing the variables of the equation by nodes not ending in $t$,
- $n \in \text{dom}(G)$ represents $L\sigma$,
- $nt \not\in \text{dom}(G)$,
- $G'$ 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 $nl \in \text{dom}(G)$, then $G(n) = n\,m$
- if $nr \in \text{dom}(G)$, then $G(n) = m \, nr$
- if $nt \in \text{dom}(G)$, then $n$ and $nt$ represent a reduction step
- acyclic
- subcommutative
- ...

Give non-inductive definition of ART based on properties?
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"

'o' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"
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)

A Theory of Tracing
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)
A Theory of Tracing
1st April 2006 24 / 27
```
main = {IO}

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

insert 's' "o" = "os" √

's' > 'o' = True

insert 's' "" = "s"

sort "ort" = "o"

insert 'o' "r" = "o"

'0' > 'r' = False

sort "rt" = "r"

insert 'r' "t" = "r"

'0' > 'r' = False

sort "t" = "t"

insert 't' "" = "t"

'0' > 't' = False

sort "" = ""

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

main = \{\text{IO}\}

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

\text{sort "ort" = "o"} \times \text{insert 's' "o" = "os"} \checkmark

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

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

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

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

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

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

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

\text{sort "" = ""}

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

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

Olaf Chitil and Yong Luo (Kent, UK)
A Theory of Tracing
1st April 2006
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"

'0' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'0' > '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"  

'o' > 'r' = False  

sort "t" = "t"  

insert 'r' "t" = "r"  

'k' > 't' = False  

sort "" = ""  

insert 't' "" = "t"  

sort "" = ""  

insert 't' "" = "t"  

sort "" = ""  

insert 't' "" = "t"
The ART and the Evaluation Dependency Tree

Olaf Chitil and Yong Luo (Kent, UK)
A Theory of Tracing
1st April 2006 26 / 27
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 ($\Rightarrow$ free variables)
- share reductions of constants ($\Rightarrow$ cycles in graph)