Debugging and Tracing Functional Programs

Olaf Chitil

University of Kent, UK

5th December 2005
Why We Need Tracing Tools and Methods

```
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
```

Sample text

Input -> Computation -> Output

Olaf Chitil (University of Kent, UK)
Why We Need Tracing Tools and Methods

```
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
```

- Presence of fault already established:
  - wrong output
  - run-time error
  - non-termination

- Locate fault.
- Comprehend programs.
Conventional Tracing Tools and Methods

A stepping debugger such as DDD

The print method

- Add print statements to program.
Properties of Conventional Tracing Tools and Methods

Show at a point in time in computation a part of computation state.

- Based on one (operational) execution model.
  - program counter
  - state
  - stack
- Computation is a sequence (in time) of states.

Forward stepping of limited value:
  - Fault often only noticed long after executing faulty program part.
Properties of Functional and Logic Programming Languages

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics

- No sequential execution of statements.
  - evaluation of expressions
  - evaluation of subexpressions is independent

\[ f \ (g \ 3 \ 4) \ (h \ 1 \ 2) \ (i \ 5) \ (j \ 3 \ 9 \ 3) \]
Properties of Functional and Logic Programming Languages

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics
- No sequential execution of statements.
  - evaluation of expressions
  - evaluation of subexpressions is independent
    \[ f(g\ 3\ 4)\ (h\ 1\ 2)\ (i\ 5)\ (j\ 3\ 9\ 3) \]

Conclusions for Tracing

- Many semantic models as potential basis for tracing.
- Take advantage of simple and compositional semantics.
- Freedom from sequentiality of computation.
elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)

    elem 42 [1..]
elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)

elem 42 [1..]
\rightarrow or (map (== 42) [1..])
elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)

elem 42 [1..]
  \rightarrow or (map (== 42) [1..])
  \rightarrow or (map (== 42) (1:[2..]))
elem :: Int -> [Int] -> Bool
elem $x \; x$s = or (map (==x) $x$s)

```
> elem 42 [1..]
\rightarrow or (map (== 42) [1..])
\rightarrow or (map (== 42) (1:[2..]))
\rightarrow or (False : map (== 42) [2..])
```
elem :: Int -> [Int] -> Bool

\[
\text{elem } x \text{ } xs = \text{or} \ (\text{map} \ (==x) \ xs)
\]

\[
\text{elem 42} \ [1..]
\]
\[
\rightarrow \text{or} \ (\text{map} \ (== 42) \ [1..])
\]
\[
\rightarrow \text{or} \ (\text{map} \ (== 42) \ (1:[2..]))
\]
\[
\rightarrow \text{or} \ (\text{False} : \text{map} \ (== 42) \ [2..])
\]
\[
\rightarrow \text{or} \ (\text{map} \ (== 42) \ [2..])
\]
elem :: Int -> [Int] -> Bool

\[ \text{elem } x \text{ xs} = \text{or} \ (\text{map} \ (==x) \ \text{xs}) \]

\[
\begin{align*}
\text{elem } 42 \ & \ [1..] \\
\leadsto & \ \text{or} \ (\text{map} \ (== \ 42) \ [1..]) \\
\leadsto & \ \text{or} \ (\text{map} \ (== \ 42) \ (1:[2..])) \\
\leadsto & \ \text{or} \ (\text{False} : \ \text{map} \ (== \ 42) \ [2..]) \\
\leadsto & \ \text{or} \ (\text{map} \ (== \ 42) \ [2..]) \\
\end{align*}
\]

\[
\begin{align*}
\vdots & \ \vdots \\
\leadsto & \ \text{True}
\end{align*}
\]
elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)

elem 42 [1..]
⇒ or (map (== 42) [1..])
⇒ or (map (== 42) (1:[2..]))
⇒ or (False : map (== 42) [2..])
⇒ or (map (== 42) [2..])
⇒ True

Complex execution model:

- Complex evaluation order.
- Unevaluated subexpressions large and hard to read.
- Run-time stack unrelated to static function call structure.
Naive Printing in Haskell

Impure function \( \text{traceShow} :: \text{String} \rightarrow \text{Int} \rightarrow \text{Int} \)

\[
\text{insert} :: \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}]
\]

\[
\text{insert} \ x \ [] = [x]
\]

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

\[
\text{main} = \text{print} \ (\text{take 5} \ (\text{insert} \ 4 \ [1..]))
\]

Output:

\[
[1>[2>[3>[4,4,5,6,7,8,9,10,11,...]
\]

- output mixed up
- non-termination \( \Rightarrow \) observation changes behaviour
Outline

1. Two-Phase Tracing
2. Views of Computation
   - Observation of Functions
   - Algorithmic Debugging
   - Source-based Free Navigation
   - Program Slicing
   - Call Stack
   - Redex Trails
   - Animation
   - ...
   - Trusting
   - New Views
3. A Theory of Tracing
4. Summary
Two-Phase Tracing

Liberates from time arrow of computation.
Two-Phase Tracing

input

computation

output

trace

view

1

2

Liberates from time arrow of computation.

Trace stored in

- Memory.
- File.
- Generated on demand by reexecution.
Two-Phase Tracing

Liberrates from time arrow of computation.

Trace stored in
- Memory.
- File.
- Generated on demand by reexecution.

Trace Generation
- Program annotations + library.
- Program transformation.
- Modified abstract machine.
Multi-View Tracer

For Haskell 98 + some extensions.

Developed by Colin Runciman, Jan Sparud, Malcolm Wallace, Olaf Chitil, Thorsten Brehm, Tom Davie, Tom Shackell, ...
Faulty Insertion Sort

```haskell
main = putStrLn (sort "sort")

sort :: Ord a => [a] -> [a]

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

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

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

Output:

```
os
```
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"
Haskell Object Observation Debugger (Hood) by Andy Gill.
- A library.
- Programmer annotates expressions of interest.
- Annotated expressions are traced during computation.
- The print method for the lazy functional programmer.

Observation of functions most useful.

Relates to denotational semantics.

\[
\text{insert } 3 \ (1:2:3:4:_\_ ) = 1:2:3:4:_\_ \\
\text{insert } 3 \ (2:3:4:_\_ ) = 2:3:4:_\_ \\
\text{insert } 3 \ (3:4:_\_ ) = 3:4:_\_ 
\]
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

insert 'r' "t" = "r"

'o' > 'r' = False

sort "t" = "t"

insert 't' "" = "t"

' r' > 't' = False

sort "" = ""
The Evaluation Dependency Tree

```
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"

'o' > 'r' = False

sort "rt" = "r"

insert 'r' "t" = "r"

'o' > 'r' = False

sort "t" = "t"

insert 't' "" = "t"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (University of Kent, UK)
Debugging and Tracing Functional Programs
5th December 2005 23 / 45
```
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"

'\textasciitilde' > 'r' = False

sort "" = ""

insert 't' "" = "t"

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

main = \{\text{IO}\}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

'\text{r}' > '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"

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

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

Olaf Chitil (University of Kent, UK)  Debugging and Tracing Functional Programs  5th December 2005  23 / 45
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 "" = ""
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 "" = ""
Algorithmic Debugging

- Shapiro for Prolog, 1983.

- Correctness of tree node according to intended semantics.
- Incorrect node whose children are all correct is faulty.
- Each node relates to (part of) a function definition.

- Relates to natural, big-step semantics.
### Hat-Explore 2.00

1. main = {IO}

2. sort "sort" = "os"

3. sort "ort" = "o"

--- Insert.hs --- lines 5 to 10 -------------------------

```haskell
if x > y then y : insert x ys
else x : ys
```

```
sort :: [Char] -> [Char]
sort [] = []
sort (x:xs) = insert x (sort xs)
```
Program terminated with error:
   No match in pattern.
Virtual stack trace:
(Last.hs:6)   last' []
(Last.hs:6)   last' [_]
(Last.hs:6)   last' [_,_]
(Last.hs:4)   last' [8,_,_]
(unknown)     main
Output: ------------------------------------------------------------------
os
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 []
Redex Trails

- Go backwards from observed failure to fault.
- Which redex created this expression?
- Based on graph rewriting semantics of abstract machine.
Animation of Lazy Evaluation

Output: ----------------------------------------------------

Animation: -------------------------------------------------
-> sort "sort"
-> insert 's' ( sort "ort" )
-> insert 's' ( insert 'o' ( sort "rt" ) )
-> insert 's' ( insert 'o' ( insert 'r' ( sort "t" ) ) )
-> insert 's' ( insert 'o' ( insert 'r' "t" ) )
-> "os"
T rusting

Trust a module: Do not trace functions in module.

- Smaller trace file.
- Avoid viewing distracting details.

\[ 4 + 7 = 11 \]
Trust a module: Do not trace functions in module.

- Smaller trace file.
- Avoid viewing distracting details.

\[ 4 + 7 = 11 \]

A trusted function may call a non-trusted function:

\[
\text{map} \ \text{prime} \ [2, 3, 4, 5] = [\text{True}, \text{True}, \text{False}, \text{True}]\
\]
**T rusting**

**Trust a module:** Do not trace functions in module.
- Smaller trace file.
- Avoid viewing distracting details.

4 + 7 = 11

A trusted function may call a non-trusted function:

```plaintext
map prime [2,3,4,5] = [True,True,False,True]
```

**In future?**
- View-time trusting.
- Trusting of local definitions.
New Ideas

- Follow a value through computation.
New Views

New Ideas

- Follow a value through computation.

Combining Existing Views

- Can easily switch from one view to another.
- All-in-one tool = egg-laying wool-milk-sow?
- Exploring combination of algorithmic debugging and redex trails.
New Views

New Ideas

- Follow a value through computation.

Combining Existing Views

- Can easily switch from one view to another.
- All-in-one tool = egg-laying wool-milk-sow?
- Exploring combination of algorithmic debugging and redex trails.

Refining Existing Views

Algorithmic Debugging:

- Different Tree-Traversal Strategies.
- Heuristics.
Implementations of tracing tools ahead of theoretical results.
Correctness of tools?
Clear methodology for using them?
Development of advanced features?
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.
sort ('t':[]) = [\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.
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
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
New nodes for right-hand-side, connected via result pointer.
New nodes for right-hand-side, connected via result pointer.
New nodes for right-hand-side, connected via result pointer.
- New nodes for right-hand-side, connected via result pointer.
- Unique node names
  - Node names independent of evaluation strategy.
  - No graph isomorphism needed.
  - Node name encodes history (parent redex, also reduct).
Summary

- Two-Phase Tracing.

Liberates from time arrow of computation.
- There exist many useful different views of a computation.
  - Observation of Functions
  - Algorithmic Debugging
  - Source-based Free Navigation
  - Redex Trails
  ...
- Semantics.
  - Inspire views.
  - Enable formulation and proof of properties.
  - But do not answer all questions.
- Still much to explore.