Tracing Computations Of Functional Programs

Olaf Chitil

University of Kent, Canterbury, United Kingdom
Why Trace & Debug Functional Programs Differently?

Haskell, OCaml, ML, Scheme, Lisp, . . .

Functional Programs have specific features.

Hence

- Conventional methods are ill-suited for functional languages.
- New, more powerful methods can take advantage of features of functional languages.
- In the future these methods may be transferred to other languages (like garbage collection, generic types, lambdas).
No Mutable Variables and Statements —
Pure Functions and Expressions

All example code in Haskell.

```
max :: Char -> Char -> Char
max x y = if x > y then x else y
```

A computation:
```
max 'a' (max 'b' 'c')
```

*Immutability Changes Everything*
All example code in Haskell.

\[
\text{max} :: \text{Char} \to \text{Char} \to \text{Char}
\]
\[
\text{max } x \ y = \text{if } x > y \text{ then } x \text{ else } y
\]

A computation:
\[
\text{max } 'a' \ (\text{max } 'b' \ 'c')
\]
\[
\equiv \text{max } 'a' \ (\text{if } 'b' > 'c' \text{ then } 'b' \text{ else } 'c')
\]
No Mutable Variables and Statements —
Pure Functions and Expressions

All example code in Haskell.

```haskell
max :: Char -> Char -> Char
max x y = if x > y then x else y
```

A computation:

```haskell
max 'a' (max 'b' 'c')
⇝ max 'a' (if 'b' > 'c' then 'b' else 'c')
⇝ max 'a' (if False then 'b' else 'c')
⇝ 'c'
```

Immutability Changes Everything
No Mutable Variables and Statements —
Pure Functions and Expressions

All example code in Haskell.

```
max :: Char -> Char -> Char
max x y = if x > y then x else y
```

A computation:

```
max 'a' (max 'b' 'c')
⇝ max 'a' (if 'b' > 'c' then 'b' else 'c')
⇝ max 'a' (if False then 'b' else 'c')
⇝ max 'a' 'c'
```

*Immutability Changes Everything*
No Mutable Variables and Statements —
Pure Functions and Expressions

All example code in Haskell.

\[
\text{max} :: \text{Char} \to \text{Char} \to \text{Char}
\]

\[
\text{max}\ x\ y = \text{if } x > y \text{ then } x \text{ else } y
\]

A computation:

\[
\text{max } 'a'\ (\text{max } 'b'\ 'c')
\]

\[
\Rightarrow \text{max } 'a'\ (\text{if } 'b' > 'c' \text{ then } 'b' \text{ else } 'c')
\]

\[
\Rightarrow \text{max } 'a'\ (\text{if } \text{False} \text{ then } 'b' \text{ else } 'c')
\]

\[
\Rightarrow \text{max } 'a'\ 'c'
\]

\[
\Rightarrow \text{if } 'a' > 'c' \text{ then } 'a' \text{ else } 'c'
\]

*Immutability Changes Everything*
No Mutable Variables and Statements —
Pure Functions and Expressions

All example code in Haskell.

```
max :: Char -> Char -> Char
max x y = if x > y then x else y
```

A computation:
```
max 'a' (max 'b' 'c')
⇝ max 'a' (if 'b' > 'c' then 'b' else 'c')
⇝ max 'a' (if False then 'b' else 'c')
⇝ max 'a' 'c'
⇝ if 'a' > 'c' then 'a' else 'c'
⇝ if False then 'a' else 'c'
```

*Immutability Changes Everything*
No Mutable Variables and Statements — Pure Functions and Expressions

All example code in Haskell.

```
max :: Char -> Char -> Char
max x y = if x > y then x else y
```

A computation:
```
max 'a' (max 'b' 'c')
  ⇝ max 'a' (if 'b' > 'c' then 'b' else 'c')
  ⇝ max 'a' (if False then 'b' else 'c')
  ⇝ max 'a' 'c'
  ⇝ if 'a' > 'c' then 'a' else 'c'
  ⇝ if False then 'a' else 'c'
  ⇝ 'c'
```

*Immutability Changes Everything*
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
\rightarrow if 3 > 1 then factorial (3-1) * 3 else 1
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
⇒ if 3 > 1 then factorial (3-1) * 3 else 1
⇒ if True then factorial (3-1) * 3 else 1
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
\Rightarrow if 3 > 1 then factorial (3-1) * 3 else 1
\Rightarrow if True then factorial (3-1) * 3 else 1
\Rightarrow factorial (3-1) * 3
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
  ⇝ if 3 > 1 then factorial (3-1) * 3 else 1
  ⇝ if True then factorial (3-1) * 3 else 1
  ⇝ factorial (3-1) * 3
  ⇝ factorial 2 * 3
No Loops — All Iteration By Recursion

factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
⇝ if 3 > 1 then factorial (3-1) * 3 else 1
⇝ if True then factorial (3-1) * 3 else 1
⇝ factorial (3-1) * 3
⇝ factorial 2 * 3
⇝ (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
⇝ 6
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
~~ if 3 > 1 then factorial (3-1) * 3 else 1
~~ if True then factorial (3-1) * 3 else 1
~~ factorial (3-1) * 3
~~ factorial 2 * 3
~~ (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
~~ (if True then factorial (2-1) * 2 else 1) * 3
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
⇝ if 3 > 1 then factorial (3-1) * 3 else 1
⇝ if True then factorial (3-1) * 3 else 1
⇝ factorial (3-1) * 3
⇝ factorial 2 * 3
⇝ (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
⇝ (if True then factorial (2-1) * 2 else 1) * 3
⇝ (factorial (2-1) * 2) * 3
⇝ 2 * 3
⇝ 6
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
⇝ if 3 > 1 then factorial (3-1) * 3 else 1
⇝ if True then factorial (3-1) * 3 else 1
⇝ factorial (3-1) * 3
⇝ factorial 2 * 3
⇝ (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
⇝ (if True then factorial (2-1) * 2 else 1) * 3
⇝ (factorial (2-1) * 2) * 3
⇝ (factorial 1 * 2) * 3
⇝ 6
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
~~> if 3 > 1 then factorial (3-1) * 3 else 1
~~> if True then factorial (3-1) * 3 else 1
~~> factorial (3-1) * 3
~~> factorial 2 * 3
~~> (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
~~> (if True then factorial (2-1) * 2 else 1) * 3
~~> (factorial (2-1) * 2) * 3
~~> (factorial 1 * 2) * 3
~~> ((if 1 > 1 then factorial (1-1) * 1 else 1) * 2) * 3
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
\[ \Rightarrow \text{if } 3 > 1 \text{ then factorial (3-1) * 3 else 1} \]
\[ \Rightarrow \text{if True then factorial (3-1) * 3 else 1} \]
\[ \Rightarrow \text{factorial (3-1) * 3} \]
\[ \Rightarrow \text{factorial 2 * 3} \]
\[ \Rightarrow (\text{if } 2 > 1 \text{ then factorial (2-1) * 2 else 1}) * 3 \]
\[ \Rightarrow (\text{if True then factorial (2-1) * 2 else 1}) * 3 \]
\[ \Rightarrow (\text{factorial (2-1) * 2}) * 3 \]
\[ \Rightarrow (\text{factorial 1 * 2}) * 3 \]
\[ \Rightarrow ((\text{if } 1 > 1 \text{ then factorial (1-1) * 1 else 1}) * 2) * 3 \]
\[ \Rightarrow ((\text{if False then factorial (1-1) * 1 else 1}) * 2) * 3 \]
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
~~ if 3 > 1 then factorial (3-1) * 3 else 1
~~ if True then factorial (3-1) * 3 else 1
~~ factorial (3-1) * 3
~~ factorial 2 * 3
~~ (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
~~ (if True then factorial (2-1) * 2 else 1) * 3
~~ (factorial (2-1) * 2) * 3
~~ (factorial 1 * 2) * 3
~~ ((if 1 > 1 then factorial (1-1) * 1 else 1) * 2) * 3
~~ ((if False then factorial (1-1) * 1 else 1) * 2) * 3
~~ (1 * 2) * 3
~~ 2 * 3
~~ 6
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
⇝ if 3 > 1 then factorial (3-1) * 3 else 1
⇝ if True then factorial (3-1) * 3 else 1
⇝ factorial (3-1) * 3
⇝ factorial 2 * 3
⇝ (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
⇝ (if True then factorial (2-1) * 2 else 1) * 3
⇝ (factorial (2-1) * 2) * 3
⇝ (factorial 1 * 2) * 3
⇝ ((if 1 > 1 then factorial (1-1) * 1 else 1) * 2) * 3
⇝ ((if False then factorial (1-1) * 1 else 1) * 2) * 3
⇝ (1 * 2) * 3
⇝ 2 * 3
⇝ 6
factorial :: Integer -> Integer
factorial n = if n > 1 then factorial (n-1) * n else 1

A computation
factorial 3
⇝ if 3 > 1 then factorial (3-1) * 3 else 1
⇝ if True then factorial (3-1) * 3 else 1
⇝ factorial (3-1) * 3
⇝ factorial 2 * 3
⇝ (if 2 > 1 then factorial (2-1) * 2 else 1) * 3
⇝ (if True then factorial (2-1) * 2 else 1) * 3
⇝ (factorial (2-1) * 2) * 3
⇝ (factorial 1 * 2) * 3
⇝ ((if 1 > 1 then factorial (1-1) * 1 else 1) * 2) * 3
⇝ ((if False then factorial (1-1) * 1 else 1) * 2) * 3
⇝ (1 * 2) * 3
⇝ 2 * 3
⇝ 6
most frequently used type is list: [Integer], [Bool], [Char], ...
empty list is []
list with first element \( M \) and rest list \( MS \) is \( M : MS \)
instead of \( 1 : 2 : 3 : [] \) usually write \([1,2,3]\)
a string is a list of characters; "abc" shorthand for [’a’,’b’,’c’]
define a function by pattern matching and several equations

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

A computation: \text{insert} ’b’ "ac" \( \rightsquigarrow * \) "abc"
Higher-Order Functions: Functions Are Data

Apply a function to all elements of a list:
map :: (a -> b) -> [a] -> [b]

A computation:  map (> 2) [1,2,3] ⇝* [False,False,True]

Combine all elements of a list:
foldr :: (a -> b -> b) -> b -> [a] -> b

product :: [Integer] -> Integer
product = foldr (*) 1

A computation:  product [1,2,3] ⇝* (1 * (2 * (3 * 1))) ⇝* 6
Lazy vs. Eager Evaluation: What Is It?

- **eager evaluation**: arguments of a function are evaluated before function is called
- **lazy evaluation**: function is called with unevaluated arguments; pattern matching and primitive functions force evaluation; duplicated expression is evaluated only once.
  - can define new control structures like `if then else`
    
    ```haskell
    ifPositive :: Integer -> a -> a -> a
    ifPositive n yes no = if n > 0 then yes else no
    ```
  - can define infinite data structures
    ```haskell
    ones :: [Integer]
    ones = 1 : ones
    ```
  - intermediate data structures (lists) do not increase space complexity
    ```haskell
    factorial :: Integer -> Integer
    factorial n = product (enumFromTo 1 n)
    ```

enumFrom :: Integer -> [Integer]
enumFrom b = b : enumFrom (b+1) -- infinite list

take :: Integer -> [Integer] -> [Integer]
take n [] = []
take n (x:xs) = if n > 0 then x : take (n-1) xs else []

data enumFromTo :: Integer -> Integer -> [Integer]  
data enumFromTo b e = take (e-b+1) (enumFrom b)

A computation
enumFromTo 4 6
Lazy vs. Eager Evaluation: Example

```haskell
enumFrom :: Integer -> [Integer]
enumFrom b = b : enumFrom (b+1)  -- infinite list

take :: Integer -> [Integer] -> [Integer]
take n [] = []
take n (x:xs) = if n > 0 then x : take (n-1) xs else []

enumFromTo :: Integer -> Integer -> [Integer]
enumFromTo b e = take (e-b+1) (enumFrom b)

A computation
enumFromTo 4 6
\rightarrow take (6-4+1) (enumFrom 4)
```

Olaf Chitil (University of Kent, UK)
Lazy vs. Eager Evaluation: Example

```haskell
enumFrom :: Integer -> [Integer]
enumFrom b = b : enumFrom (b+1)  -- infinite list

take :: Integer -> [Integer] -> [Integer]
take n [] = []
take n (x:xs) = if n > 0 then x : take (n-1) xs else []

take :: Integer -> [Integer] -> [Integer]
take n (x:xs) = if n > 0 then x : take (n-1) xs else []

enumFromTo :: Integer -> Integer -> [Integer]
enumFromTo b e = take (e-b+1) (enumFrom b)

A computation
enumFromTo 4 6
⇝ take (6-4+1) (enumFrom 4)
⇝ take (6-4+1) (4 : enumFrom (4+1))
```
Lazy vs. Eager Evaluation: Example

```haskell
enumFrom :: Integer -> [Integer]
enumFrom b = b : enumFrom (b+1)  -- infinite list

take :: Integer -> [Integer] -> [Integer]
take n [] = []
take n (x:xs) = if n > 0 then x : take (n-1) xs else []

enumFromTo :: Integer -> Integer -> [Integer]
enumFromTo b e = take (e-b+1) (enumFrom b)

A computation
enumFromTo 4 6
  ⇝ take (6-4+1) (enumFrom 4)
  ⇝ take (6-4+1) (4 : enumFrom (4+1))
  ⇝ if (6-4+1) > 0 then 4 : take ((6-4+1)-1) (enumFrom (4+1)) else []
```

Olaf Chitil (University of Kent, UK)
Lazy vs. Eager Evaluation: Example

\[
\text{enumFrom} :: \text{Integer} \rightarrow [\text{Integer}] \\
\text{enumFrom} \ b = b : \text{enumFrom} \ (b+1) \quad -- \text{infinite list} \\
\text{take} :: \text{Integer} \rightarrow [\text{Integer}] \rightarrow [\text{Integer}] \\
\text{take} \ n \ [] = [] \\
\text{take} \ n \ (x:xs) = \text{if} \ n > 0 \ \text{then} \ x : \text{take} \ (n-1) \ xs \ \text{else} \ [] \\
\text{enumFromTo} :: \text{Integer} \rightarrow \text{Integer} \rightarrow [\text{Integer}] \\
\text{enumFromTo} \ b \ e = \text{take} \ (e-b+1) \ (\text{enumFrom} \ b) \\
\]

A computation

\[
\text{enumFromTo} \ 4 \ 6 \\
\leadsto \text{take} \ (6-4+1) \ (\text{enumFrom} \ 4) \\
\leadsto \text{take} \ (6-4+1) \ (4 : \text{enumFrom} \ (4+1)) \\
\leadsto \text{if} \ (6-4+1) > 0 \ \text{then} \ 4 : \text{take} \ ((6-4+1)-1) \ (\text{enumFrom} \ (4+1)) \ \text{else} \ [] \\
\leadsto \text{if} \ 3 > 0 \ \text{then} \ 4 : \text{take} \ (3-1) \ (\text{enumFrom} \ (4+1)) \ \text{else} \ []
\]
Lazy vs. Eager Evaluation: Example

```
enumFrom :: Integer -> [Integer]
enumFrom b = b : enumFrom (b+1)  -- infinite list

take :: Integer -> [Integer] -> [Integer]
take n []    = []
take n (x:xs) = if n > 0 then x : take (n-1) xs else []

enumFromTo :: Integer -> Integer -> [Integer]
enumFromTo b e = take (e-b+1) (enumFrom b)
```

A computation

```
enumFromTo 4 6
⇝ take (6-4+1) (enumFrom 4)
⇝ take (6-4+1) (4 : enumFrom (4+1))
⇝ if (6-4+1) > 0 then 4 : take ((6-4+1)-1) (enumFrom (4+1)) else []
⇝ if 3 > 0 then 4 : take (3-1) (enumFrom (4+1)) else []
⇝ if True then 4 : take (3-1) (enumFrom (4+1)) else []
```
Lazy vs. Eager Evaluation: Example

```haskell
enumFrom :: Integer -> [Integer]
enumFrom b = b : enumFrom (b+1) -- infinite list

take :: Integer -> [Integer] -> [Integer]
take n [] = []
take n (x:xs) = if n > 0 then x : take (n-1) xs else []

enumFromTo :: Integer -> Integer -> [Integer]
enumFromTo b e = take (e-b+1) (enumFrom b)
```

A computation
```
enumFromTo 4 6
⇒ take (6-4+1) (enumFrom 4)
⇒ take (6-4+1) (4 : enumFrom (4+1))
⇒ if (6-4+1) > 0 then 4 : take ((6-4+1)-1) (enumFrom (4+1)) else []
⇒ if 3 > 0 then 4 : take (3-1) (enumFrom (4+1)) else []
⇒ if True then 4 : take (3-1) (enumFrom (4+1)) else []
⇒ 4 : take (3-1) (enumFrom (4+1))
⇒ ...
```
Conventional Tracing & Debugging Methods

- adding print / logging statements
- using a debugger to step through computation and observe variables
Conventional Tracing & Debugging Methods

- adding print / logging statements
- using a debugger to step through computation and observe variables

These methods assume
- a single computation model
- of sequentially executing statements and
- mutating the computation state of variables.
Functional programmers

- have many computation models (reductions, interpreters with environment, denotations, ...)
- view large data structures and functions as single values
- disregard evaluation order
  \[ f (g \ x) (h \ y) \]

New problems

- Expressions can be huge.
- Lazy functional programming languages have a complex evaluation order, the runtime stack does not reflect function calls.
The Problem with Printing and Lazy Evaluation

Impure function \texttt{traceShow :: String \to [Int] \to [Int]}

\texttt{insert :: Int \to [Int] \to [Int]}
\begin{align*}
\text{insert } x \; [] & \; = \; [x] \\
\text{insert } x \; (y:ys) & = \\
& \quad \text{if } x > y \; \text{then } y : (\text{traceShow } "\textgreater" \; (\text{insert } x \; ys)) \\
& \quad \text{else } x : ys
\end{align*}

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

Output:

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

- output mixed up
- non-termination \Rightarrow observation changes behaviour
Aside: How Do Functional Programers Debug Today?

- Use conventional methods. Still work at least for eager evaluation.
- Use assertions / contracts to ensure properties of input and output. Example contract:
  \[
  \text{(define/contract sqrt}
  \quad \text{(bigger-than-zero? \rightarrow bigger-than-zero?)}
  \quad (...)\text{)}
  \]
  Popular for Scheme-dialect Racket.
- Use random testing of properties. Example property:
  \[
  \text{prop_rev xs = reverse (reverse xs) == xs}
  \]
  Popular for Haskell.
Outline

1 Features of Functional Programs ✓
2 Views of Computations
   - Observation of Values
   - Algorithmic Debugging
   - Following Redex Trails
3 Non-Tracing
4 Tracing Methods
   - Andy Gill’s Event Sequence for Observation
   - Maarten Faddegon’s Algorithmic Debugging Based on Event Sequence
   - The Augmented Redex Trail, Obtainable From More Events
5 Open Challenges
6 Summary
Part II

Views of Computations
Separating Trace Generation and Viewing

Thus independent from time arrow of computation.

Example Program: Faulty Insertion Sort

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

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

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

Unexpected output:

```
os
```
Example Program: Faulty Higher-Order Insertion Sort

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

sort :: [Char] -> [Char]
sort = foldr insert []

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

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

Unexpected output:

os
```
Observation of an Expression

- Observe values that a marked expression denotes during the computation.
- Several values per expression, because evaluated several times.

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

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

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

Olaf Chitil (University of Kent, UK)
Observation of a Function

- An observed value can be a function.
- A function is a finite map from inputs to outputs.
- Inputs together with their outputs provide more information.

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

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

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

```haskell
{'s' "o" -> "os"}
{'o' "r" -> "o"}
{'r' "t" -> "r"}
{'t' "" -> "t"}
```
Observation of a Higher-Order Function

An observed value can be a higher-order function.

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

sort :: [Char] -> [Char]
sort = foldr insert []

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

insert :: Char -> [Char] -> [Char]
insert x [] = [x]
insert x (y:ys) = if x > y then y:(insert x ys) else x:ys
```
Observation of Values

- Printing / logging for the functional programmer
  - can observe values of any type (functions, trees, ...)
  - works for lazy functional languages
- Invented by Andy Gill: HOOD (ACM Workshop on Haskell, 2000)
- Later Haskell tracer Hat also provides this view: HAT-OBSERVE.

HAT-OBSERVE allows easy observation of top-level functions:

```haskell
insert 's' "o" = "os"
insert 's' "" = "s"
insert 'o' "r" = "o"
insert 'r' "t" = "r"
insert 't' "" = "t"
```
main = putStrLn "os" ?

Bug identified:
"Insert.hs":8-9:
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys
Algorithmic Debugging

main = putStrLn "os" ? n
Algorithmic Debugging

main = putStrLn "os" 
sort "sort" = "os" 

Bug identified:
"Insert.hs":8-9:
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys

Systematic traversal of a Computation Tree. Each tree node relates to (part of) a function definition.

Olaf Chitil (University of Kent, UK)
Algorithmic Debugging

main = putStrLn "os" $ n
sort "sort" = "os" $ 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

Systematic traversal of a Computation Tree.
Each tree node relates to (part of) a function definition.
Algorithmic Debugging

main = putStrLn "os" ? \nsort "sort" = "os" ? \ninset 's' "o" = "os" ?

Bug identified:
"Insert.hs":8-9:
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys

Systematic traversal of a Computation Tree.
Each tree node relates to (part of) a function definition.
main = putStrLn "os" ? n
sort "sort" = "os" ? n
insert 's' "o" = "os" ? y
Algorithmic Debugging

main = putStrLn "os" ? n
sort "sort" = "os" ? n
insert 's' "o" = "os" ? y
sort "ort" = "o" ?
main = putStrLn "os" ? n
sort "sort" = "os" ? n
insert 's' "o" = "os" ? y
sort "ort" = "o" ? n
main = putStrLn "os" ? n
sort "sort" = "os" ? n
insert 's' "o" = "os" ? y
sort "ort" = "o" ? n
insert 'o' "r" = "o" ?
Algorithmic Debugging

main = putStrLn "os" ? n
sort "sort" = "os" ? n
insert 's' "o" = "os" ? y
sort "ort" = "o" ? n
insert 'o' "r" = "o" ? n
main = putStrLn "os" ? n
sort "sort" = "os" ? n
insert 's' "o" = "os" ? y
sort "ort" = "o" ? n
insert 'o' "r" = "o" ? n
'o' > 'r' = False ?

Bug identified:
"Insert.hs":8-9:
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x : ys

Systematic traversal of a Computation Tree.
Each tree node relates to (part of) a function definition.

Olaf Chitil (University of Kent, UK)
Algorithmic Debugging

main = putStrLn "os" ?  n
sort "sort" = "os" ?  n
insert 's' "o" = "os" ?  y
sort "ort" = "o" ?  n
insert 'o' "r" = "o" ?  n
'o' > 'r' = False ?  y
main = putStrLn "os" ? n
sort "sort" = "os" ? n
insert 's' "o" = "os" ? y
sort "ort" = "o" ? n
insert 'o' "r" = "o" ? n
'o' > 'r' = False ? y

Bug identified:
"Insert.hs":8-9:
insert x [] = [x]
insert x (y:ys) = if x > y then y:(insert x ys) else x:ys

- Systematic traversal of a Computation Tree.
- Each tree node relates to (part of) a function definition.
main = putStrLn "os"

sort "sort" = "os"

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
main = putStrLn "os"

sort "sort" = "os"

sort "ort" = "o"

insert 's' "o" = "os"

's' > 'o' = True

insert 'o' "r" = "o"

sort "rt" = "r"

insert 'r' "t" = "r"

'o' > 'r' = False

sort "t" = "t"

insert 't' "" = "t"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"

'o' > 'r' = False

sort "rt" = "r"

insert 's' "o" = "os"
Algorithmic Debugging: Computation Tree

main = putStrLn "os"

sort "sort" = "os"

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 "" = ""

insert 't' "" = "t"

'"' > 't' = False
main = putStrLn "os"

sort "sort" = "os"

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

insert 't' "" = "t"

'r' > 't' = False

sort "" = ""
main = putStrLn "os"

sort "sort" = "os"

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"

'o' > 'r' = False
main = putStrLn "os"

sort "sort" = "os"

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"

'o' > 'r' = False

sort "" = ""
main = putStrLn "os"

sort "sort" = "os"

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"

'0' > 'r' = False

sort "t" = "t"

insert 't' "" = "t"

'0' > 't' = False

sort """ = ""

insert 't' "" = "t"
Recall: Faulty Higher-Order Insertion Sort

main = putStrLn (sort "sort")

sort :: [Char] -> [Char]
sort = foldr insert []

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

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

Unexpected output:

os
Higher-Order Insertion Sort: Evaluation Dependence Tree

main = putStrLn "os"

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert ’s’ "o" = "os"

insert ’s’ "" = "s"

’s’ > ’o’ = True

insert ’o’ "r" = "o"

insert ’o’ "r" = "o"

’o’ > ’r’ = False

insert ’r’ "t" = "r"

’r’ > ’t’ = False

insert ’t’ "" = "t"
main = putStrLn "os"

foldr insert [] "sort" = "os"

sort = foldr insert []

foldr insert [] "ort" = "o"

insert 's' "o" = "os"

's' > 'o' = True

insert 's' "" = "s"

foldr insert [] "rt" = "r"

insert 'o' "r" = "o"

'o' > 'r' = False

foldr insert [] "t" = "t"

insert 'r' "t" = "r"

'o' > 'r' = False

foldr insert [] "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (University of Kent, UK)
main = putStrLn "os"

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

's' > 'o' = True

insert 's' "" = "s"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

'o' > 'r' = False

insert 't' "" = "t"

'r' > 't' = False
main = putStrLn "os"

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

insert 's' "o" = "os"

's' > 'o' = True

insert 'o' "r" = "o"

insert 'o' "r" = "o"

'o' > 'r' = False

insert 'r' "t" = "r"

insert 'r' "t" = "r"

'o' > 'r' = False

insert 't' "" = "t"

' r' > ' t' = False

're' > 't' = False
main = putStrLn "os"
foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

insert 's' "" = "s"

' s' > ' o' = True

insert 'o' "r" = "o"

insert 'o' "" = ""

' o' > ' r' = False

insert 'r' "t" = "r"

insert 't' "" = ""
Higher-Order Insertion Sort: Evaluation Dependence Tree

```
main = putStrLn "os"

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

' s' > ' o' = True

insert 's' "" = "s"

insert 'o' "r" = "o"

insert 'o' "r" = "o"

' o' > ' r' = False

insert 'r' "t" = "r"

' r' > ' t' = False

insert 't' "" = "t"

' r' > ' t' = False
```
Higher-Order Insertion Sort: Evaluation Dependence Tree

main = putStrLn "os"

foldr insert [] "sort" = "os"

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert []

insert 's' "o" = "os"

insert 's' "" = "s"

insert 'o' "r" = "o"

insert 'o' "r" = "o"

insert 'r' "t" = "r"

'\' > 'o' = True

'o' > 'r' = False

'r' > 't' = False

'\' > 'o' = True

insert 't' "" = "t"
Function Dependence Tree with Functions as Finite Maps

main = putStrLn "os"

sort = {"sort" -> "os"}

foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "sort" = "os"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "ort" = "o"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "rt" = "r"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "t" = "t"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "" = ""

insert 's' "o" = "os" insert 'o' "r" = "o" insert 'r' "t" = "r" insert 't' "" = "t"

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

Olaf Chitil (University of Kent, UK) Tracing Functional Programs
Function Dependence Tree with Functions as Finite Maps

main = putStrLn "os"

sort = {"sort" -> "os"}

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "sort" = "os"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "ort" = "o"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "rt" = "r"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "t" = "t"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "" = ""

insert 's' "o" = "os" insert 'o' "r" = "o" insert 'r' "t" = "r" insert 't' "" = "t"

's'>'o' = True insert 's' "" = "s" 'o'>'r' = False 'r'>'t' = False
Function Dependence Tree with Functions as Finite Maps

main = putStrLn "os"

sort = \"sort\" -> \"os\"

foldr \{ 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" \} [] \"sort\" = \"os\"

foldr \{ 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" \} [] \"ort\" = \"o\"

foldr \{ 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" \} [] \"rt\" = \"r\"

foldr \{ 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" \} [] \"t\" = \"t\"

foldr \{ 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t" \} [] \"\" = \"

insert 's' "o" = "os"
insert 'o' "r" = "o"
insert 'r' "t" = "r"
insert 't' "" = "t"

's'>'o' = True
insert 's' "" = "s"
'o'>'r' = False
'r'>'t' = False
Function Dependence Tree with Functions as Finite Maps

```haskell
main = putStrLn "os"
sort = {"sort" -> "os"}
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "sort" = "os"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "ort" = "o"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "rt" = "r"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "t" = "t"
foldr { 's' "o"->"os", 'o' "r"->"o", 'r' "t"->"r", 't' ""->"t"} [] "" = ""

insert 's' "o" = "os"  insert 'o' "r" = "o"  insert 'r' "t" = "r"  insert 't' "" = "t"

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

Olaf Chitil (University of Kent, UK)
Tracing Functional Programs
Function Dependence Tree with Functions as Finite Maps

```
main = putStrLn "os"

sort = {"sort" -> "os"}

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "sort" = "os"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "ort" = "o"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "rt" = "r"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "t" = "t"

foldr { 's' "o" -> "os", 'o' "r" -> "o", 'r' "t" -> "r", 't' "" -> "t"} [] "" = ""

insert 's' "o" = "os" insert 'o' "r" = "o" insert 'r' "t" = "r" insert 't' "" = "t"

's'>'o' = True insert 's' "" = "s" 'o'>'r' = False 'r'>'t' = False
```
main = putStrLn "os"

sort = \"sort\" -> "os\"

foldr \{'s' 'o'->"os", 'o' 'r'->"o", 'r' 't'->"r", 't' ""->"t"\} [] "sort" = "os"

foldr \{'s' 'o'->"os", 'o' 'r'->"o", 'r' 't'->"r", 't' ""->"t"\} [] "ort" = "o"

foldr \{'s' 'o'->"os", 'o' 'r'->"o", 'r' 't'->"r", 't' ""->"t"\} [] "rt" = "r"

foldr \{'s' 'o'->"os", 'o' 'r'->"o", 'r' 't'->"r", 't' ""->"t"\} [] "t" = "t"

foldr \{'s' 'o'->"os", 'o' 'r'->"o", 'r' 't'->"r", 't' ""->"t"\} [] "" = ""

insert 's' "o" = "os"
insert 'o' "r" = "o"
inert 'r' "t" = "r"
inert 't' "" = "t"

's'>'o' = True
insert 's' "" = "s"
'o'>'r' = False
'r'>'t' = False
1. `main = putStrLn "os"`
2. `sort "sort" = "os"`
3. `sort "ort" = "o"

---- Insert.hs ---- lines 3 to 8 --------------------------

```haskell
sort :: [Char] -> [Char]
sort [] = []
sort (x:xs) = insert x (sort xs)
```

```haskell
insert :: Char -> [Char] -> [Char]
insert x [] = [x]
```

Reminds of stepping debugger, but freely going forwards and backwards.
Following a Redex Trail

Output: -----------------------------------------------
os

Trail: ------- Insert.hs ------------------------------------
Output: ---------------
  os

Trail: ------ Insert.hs ------------------------------
<- putStrLn "os"
<- putStrLn "os"
Following a Redex Trail

Output: 
-----------------------------------------------
os

Trail: ------- Insert.hs -----------------------------------------------
<- putStrLn "os"
<- insert 's' "o" | if True
Following a Redex Trail

Output: os
Trail: ------- Insert.hs ------------------------------
<- putStrLn "os"
<- insert 's' "o" | if True
<- insert 'o' "r" | if False
Following a Redex Trail

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

os

Trail: ------- Insert.hs -----------------------------------------------

<- putStrLn "os"
<- insert 's' "o" | if True
<- insert 'o' "r" | if False
<- insert 'r' "t" | if False
Output: 

```
os
```

Trail: ------- Insert.hs -----------------------------------------------

```
<- putStrLn "os"
<- insert 's' "o" | if True
<- insert 'o' "r" | if False
<- insert 'r' "t" | if False
<- insert 't' []
```
Following a Redex Trail

Output: -----------------------------------------------
os
Trail: ------ Insert.hs ----------------------------------
<- 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?
- A redex is the smallest expression describing a computation step.
- Can explore any subexpression.
- More connections than in computation tree.
Part III

Non-Tracing
Non-Tracing

The programmer **does not want to trace** most of the program.

- trusted modules (standard libraries, checked code)
- untrusted modules
  - cannot be traced (language extensions, other languages)
  - information not wanted (test framework, old code, details)

- Viewing unnecessary information detracts.
- Tracing unnecessary information costs time and space.

To avoid problems with untraceable modules and reduce time costs, want only traced modules to be changed by tracing method.
Part IV

Tracing Methods
Trace Generation Methods Used by Different Systems

Freja (1994)  Modified abstract machine
Tracer (1997)  Program transformation
Buddha (1998) Program transformation
HOOD (2000)  Program annotations + library
Hat (2000)    Program transformation
BIO (2007)    Program transformation

All program transformations and modified abstract machine are complex.
**HOOD: To Observe Values, Generate an Event Sequence**

```haskell
import Observe

main = putStrLn (sort "so")

sort :: [Char] -> [Char]
sort [] = []
sort (x:xs) = insert x (observe "list" (sort xs))

Event sequence:

1 Root Observe "list"
2 ...
3 Root Observe "list"
4 ...
5 Parent 3 Cons 0 " []"
6 Parent 1 Cons 2 ":"  \[ \text{Observe "list"} \]
7 ...
8 Parent 6 Left Cons 0 ":o"
9 ...
10 Parent 6 Right Cons 0 " []"

\[ \text{Cons 0 " :]" \] \]
\[ \text{Cons 0 ":o"} \]
\[ \text{Cons 0 " []"} \]
\[ \text{'o': []} \]
\[ [] \]

Olaf Chitil (University of Kent, UK)
observe :: Observable a => String -> a -> a

observe label orig = unsafePerform0bs $ do
  eventNo <- sendEvent Root (Observe label)
  observer (Parent eventNo) orig

instance Observable a => Observable [a] where
  observer parent (x:xs) = do
    eventNo <- sendEvent parent (Cons 2 "":"")
    return ((observer_ (Parent eventNo Left) x) :
      (observer_ (Parent eventNo Right) xs))
  observer parent [] = do
    sendEvent parent (Cons 0 "[]")
    return []

observer_ parent orig =
  unsafePerform0bs (observer parent orig)
Reconstructing Computation Tree Nodes

Observe all suspected top-level functions as follows:

```plaintext
insert = observe "insert" insert'
insert' x [] = [x]
insert' x (y:ys) = if x > y then y : (insert x ys) else x:ys
```

HOOD gives

```plaintext
insert
  { 'o' "r" -> "o"
   , 'r' "t" -> "r"
   , 's' "o" -> "os"
   , 's' [] -> "s"
   , 't' [] -> "t" }
```

So we get the nodes of the computation tree:

```plaintext
insert 'o' "r" = "o"
...
```
Reconstructing Computation Tree Edges: Event Brackets

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1 | Root | Observe "list"
| 2 | Parent 1 | Request
| 3 | Root | Observe "list"
| 4 | Parent 3 | Request
| 5 | Parent 3 | Cons 0 "[]"
| 6 | Parent 1 | Cons 2 ":"
| 7 | Parent 6 Left | Request
| 8 | Parent 6 Left | Cons 0 ", o"
| 9 | Parent 6 Right | Request
| 10 | Parent 6 Right | Cons 0 "[]"

- Every non-Root event is preceded by a request event.
- Request + response event are like brackets in event sequence:
  - either in sequence (one pair after another)
  - or nested
Event brackets for results are directly nested.

Brackets for any argument make a gap in surrounding nesting.

\[
\text{sort ('t':[])} = 't':[] \\
\text{sort [] = []} \\
\text{insert 't' [] = 't':[]} 
\]
A Universal Trace: The Augmented Redex Trail (ART)

ART contains wealth of information for numerous views.
The ART is a graph of nodes and three types of edges.

```
main = putStrLn (sort ['t'])

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
```
main = putStrLn (sort ['t'])

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
main = putStrLn (sort ['t'])

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
Structure of the Augmented Redex Trail (ART)

main = putStrLn (sort ['t'])

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
main = putStrLn (sort ['t'])

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
Structure of the Augmented Redex Trail (ART)

- New nodes for right-hand-side, connected via redex edge
- Only add to graph, never remove
- Sharing ensures compact representation
- Every node has pointer back to its parent redex
HatLight: New Events and Event-Combinators for ART

\[ \text{sort :: [Char] -> [Char]} \]
\[ \text{sort [] = con "[]" 0 []} \]
\[ \text{sort (x:xs) =} \]
\[ \quad \text{app2 (var "insert" insert) (lamVar [R,L,R] x)} \]
\[ \quad \quad \text{(app (var "sort" sort) (lamVar [R,R] xs))} \]

\[ \text{insert :: Char -> [Char] -> [Char]} \]
\[ \text{insert x [] =} \]
\[ \quad \text{app2 (con ":" 2 (:)) (lamVar [L,R] x) (con "[]" 0 [])} \]
\[ \text{insert x (y:ys) = app3 (var "if" ifThenElse)} \]
\[ \quad \text{(app2 (var ">" (>)) (lamVar [L,R] x) (lamVar [R,L,R] y))} \]
\[ \quad \quad \text{(app2 (con "(:)" 2 (:)) (lamVar [R,L,R] y)} \]
\[ \quad \quad \quad \text{(app2 (var "insert" insert)} \]
\[ \quad \quad \quad \quad \quad \text{(lamVar [L,R] x) (lamVar [R,R] ys))} \]
\[ \quad \quad \text{(app2 (con "(:)" 2 (:))} \]
\[ \quad \quad \quad \quad \quad \text{(lamVar [L,R] x) (lamVar [R,R] ys))} \]

More invasive transformation, but all types are unchanged.
Part V

Open Challenges
Open Challenges

- Non-tracing: Make ART generation via events work with unmodified modules like for algorithmic debugging.
- Combine algorithmic debugging with following redex trails; user says which subexpression is wrong. Cf. rational debugging by Pereira.
- Prove that ART generation via events is correct; develop theory.
- Develop sound and useful mixture of evaluation and function dependence tree (limite size of finite maps).
- Develop tracing and debugging for abstract data types.
- Develop tracing and debugging of input/output.
- Develop tracing and debugging of effectful computations, e.g. state monad with references.
Summary

- Functional programmers compose expressions that denote values.

Two-phase tracing liberates from time arrow of computation.
- There exist many useful different views of a computation.
- Events recorded during computation can provide a detailed trace.
- Not tracing most of a program is key in practice.
- There is still a lot to do!

Big Thanks to Maarten Faddegon and Colin Runciman!
Part VIII

Appendix
Lazy Evaluation of an expression

Program

```
elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)
```

Computation

```
 elem 42 [1..]
```

Here reduction steps for `map` and `or` are skipped.
Lazy Evaluation of an expression

Program

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

Computation

elem 42 [1..]
⇒ or (map (== 42) [1..])
Lazy Evaluation of an expression

Program

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

Computation

elem 42 [1..]
  ↦ or (map (== 42) [1..])
  ↦ or (map (== 42) (1:[2..])))
Lazy Evaluation of an expression

Program

\[\text{elem} :: Int \rightarrow \text{[Int]} \rightarrow \text{Bool}\]
\[\text{elem } x \text{ } \text{xs} = \text{or } (\text{map } (==x) \text{ } \text{xs})\]

Computation

\[
\begin{align*}
\text{elem } 42 \text{ } \text{[1..]} \\
\sim & \text{or } (\text{map } (== 42) \text{ } \text{[1..]}) \\
\sim & \text{or } (\text{map } (== 42) (1:\text{[2..]})) \\
\sim & \text{or } (\text{False } : \text{map } (== 42) \text{ } \text{[2..]})
\end{align*}
\]
Lazy Evaluation of an expression

Program

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

Computation

elem 42 [1..]
\rightarrow or (map (== 42) [1..])
\rightarrow or (map (== 42) (1:[2..]))
\rightarrow or (False : map (== 42) [2..])
\rightarrow or (map (== 42) [2..])

Here reduction steps for map and or are skipped.
Lazy Evaluation of an expression

Program

```haskell
elem :: Int -> [Int] -> Bool
elem x xs = or (map (==x) xs)
```

Computation

```
elem 42 [1..]
    ⇝ or (map (== 42) [1..])
    ⇝ or (map (== 42) (1:[2..]))
    ⇝ or (False : map (== 42) [2..])
    ⇝ or (map (== 42) [2..])
    ⇝ or (map (== 42) (2:[3..]))
    ...  
    ⇝ True
```

Here reduction steps for `map` and `or` are skipped.
Lazy Evaluation of an expression

Program

\[
\text{elem :: } \text{Int} \rightarrow \text{[Int]} \rightarrow \text{Bool} \\
\text{elem } x \text{ xs } = \text{or } (\text{map } (==x) \text{ xs})
\]

Computation

\[
\text{elem 42 [1..]} \\
\leadsto \text{or } (\text{map } (== 42) [1..]) \\
\leadsto \text{or } (\text{map } (== 42) (1:[2..])) \\
\leadsto \text{or } (\text{False : map } (== 42) [2..]) \\
\leadsto \text{or } (\text{map } (== 42) [2..]) \\
\leadsto \text{or } (\text{map } (== 42) (2:[3..])) \\
\leadsto \text{or } (\text{False : map } (== 42) [3..])
\]
Lazy Evaluation of an expression

Program

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

Computation

elem 42 [1..]
\[\mapsto\] or (map (== 42) [1..])
\[\mapsto\] or (map (== 42) (1:[2..]))
\[\mapsto\] or (False : map (== 42) [2..])
\[\mapsto\] or (map (== 42) [2..])
\[\mapsto\] or (map (== 42) (2:[3..]))
\[\mapsto\] or (False : map (== 42) [3..])
\[\mapsto\] or (map (== 42) [3..])
\[\mapsto\] True
Lazy Evaluation of an expression

Program

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

Computation

elem 42 [1..]
\rightarrow or (map (== 42) [1..])
\rightarrow or (map (== 42) (1:[2..]))
\rightarrow or (False : map (== 42) [2..])
\rightarrow or (map (== 42) [2..])
\rightarrow or (map (== 42) (2:[3..]))
\rightarrow or (False : map (== 42) [3..])
\rightarrow or (map (== 42) [3..])
\vdots
\vdots
Lazy Evaluation of an expression

Program

\[
\text{elem} :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{Bool}
\]
\[
\text{elem} \ x \ \text{xs} = \text{or} \ (\text{map} \ (==x) \ \text{xs})
\]

Computation

\[
\text{elem} \ 42 \ \text{[1..]}
\]
\[
\leadsto \text{or} \ (\text{map} \ (== \ 42) \ \text{[1..]})
\]
\[
\leadsto \text{or} \ (\text{map} \ (== \ 42) \ (1:\text{[2..]}))
\]
\[
\leadsto \text{or} \ (\text{False} : \text{map} \ (== \ 42) \ \text{[2..]})
\]
\[
\leadsto \text{or} \ (\text{map} \ (== \ 42) \ \text{[2..]})
\]
\[
\leadsto \text{or} \ (\text{map} \ (== \ 42) \ (2:\text{[3..]}))
\]
\[
\leadsto \text{or} \ (\text{False} : \text{map} \ (== \ 42) \ \text{[3..]})
\]
\[
\leadsto \text{or} \ (\text{map} \ (== \ 42) \ \text{[3..]})
\]
\[
\vdots \ \vdots
\]
\[
\leadsto \text{True}
\]

Here reduction steps for \text{map} and \text{or} are skipped.
Augment every expression with a pointer to its description in the trace:

```
data R a = R a RefExp
```

All data types are transformed. E.g. \([a] \) becomes:

```
data List a = Nil | Cons (T.R a) (T.R (List a))
```

Every function needs to know about its parent redex (caller):

```
newtype Fun a b = Fun (RefExp -> R a -> R b)
```

E.g. the function type

\([a] \to [a] \to [a] \)

becomes

```
T.Fun (T.List a) (T.Fun (T.List a) (T.List a))
```
Hat-Trans: An Example

\texttt{rev :: [a] -> [a] -> [a]}
\texttt{rev [] ys = ys}
\texttt{rev (x:xs) ys = rev xs (x:ys)}

is transformed into

\texttt{grev :: T.RefSrcPos -> T.RefExp ->}
\texttt{T.R (T.Fun (T.List a) (T.Fun (T.List a) (T.List a)))}
\texttt{grev prev p = T.fun2 arev prev p hrev}

\texttt{hrev :: T.R (T.List a) -> T.R (T.List a) -> T.RefExp -> T.R (T.List a)}
\texttt{hrev (T.R T.Nil _) fys p = T.projection p5v13v5v14 p fys}
\texttt{hrev (T.R (T.Cons fx fxs) _) fys p =}
\texttt{ T.app2 p6v17v6v28 p6v17v6v19 p arev hrev fxs}
\texttt{ (T.con2 p6v25v6v28 p T.Cons T.aCons fx fys)}

\texttt{tMain = T.mkModule "Main" "Test.hs" Prelude.True}
\texttt{arev = T.mkVariable tMain 50001 60028 3 2 "rev" Prelude.False}
\texttt{p5v13v5v14 = T.mkSrcPos tMain 50013 50014}
\texttt{p6v17v6v28 = T.mkSrcPos tMain 60017 60028}
\texttt{p6v17v6v19 = T.mkSrcPos tMain 60017 60019}
Idea: Use Hood’s Instrumentation Method

Instrument code with side effects that write an event sequence.

```haskell
sendEvent :: String -> IO a

tr :: String -> a -> a
tr name exp = unsafePerformIO $ do
    sendEvent name
    return exp
```

Instrumentation: True \(\Rightarrow\) tr "True" True
myId True = True
myId False = False
myNot True = myId False
myNot False = myId True
z = myNot (myNot True)
Essential to ART Structure: Chains of Reductions

myId True = True
myId False = False
myNot True = myId False
myNot False = myId True

z = myNot (myNot True)
myId True = True
myId False = False
myNot True = myId False
myNot False = myId True
z = myNot (myNot True)
myId True = True
myId False = False
myNot True = myId False
myNot False = myId True
z = myNot (myNot True)
Essential to ART Structure: Chains of Reductions

myId True = True
myId False = False
myNot True = myId False
myNot False = myId True
z = myNot (myNot True)

There are two chains of reductions, one nested within the other.
Idea: Delimit Chains in Event Sequence

A tracing combinator produces event for beginning and end of a chain. Wrap every expression with this combinator to mark chain of that expression.

$$\text{ev} :: a \rightarrow a$$

$$\text{ev} \ x = \text{unsafePerformIO} \ \text{do}$$

sendEvent "Begin chain"

$$x \ 'seq' \ \text{sendEvent} \ "End chain"$$

return \ x

An event sequence:

- Begin chain
- App myNot
- App myNot
- App myId
- False
- End chain
- App myId
- True
- End chain
HatLight’s Events and Tracing Combinators

```
data Event =
    Var String
  | Con String Int
  | App
  | Enter EventId Branch
  | Value

var :: String -> a -> a
var name v = unsafePerformIO $ do
    sendEvent (Var name)
    return v

con :: String -> Int -> a -> a
con name arity c = ...

app :: (a -> b) -> a -> b
app f x = unsafePerformIO $ do
    eventNum <- sendEvent App
    return ((eval eventNum L f)
            (eval eventNum R x))

eval :: EventId -> Branch -> a -> a
eval eId b x = unsafePerformIO $ do
    sendEvent (Enter eId b)
    x `seq` sendEvent Value
    return x
```
Instrumented Example Program

myId :: Bool -> Bool
myId True = con "True" 0 True
myId False = con "False" 0 False

myNot :: Bool -> Bool
myNot True = app (var "myId" myId) (con "False" 0 False)
myNot False = app (var "myId" myId) (con "True" 0 True)

z :: Bool
z = app (var "myNot" myNot)
   (app (var "myNot" myNot) (con "True" 0 True))
Translation from Event Sequence to ART

```
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0
```

Stack:
```
<table>
<thead>
<tr>
<th>Chain 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>
```

Olaf Chitil (University of Kent, UK)

Tracing Functional Programs
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack
Context 9 L
Chain 9

Olaf Chitil (University of Kent, UK)
Tracing Functional Programs
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack
Chain 11
Chain 9

myNot
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
28: Con "False" 0
29: Value
36: Value
37: Con "True" 0

Stack
Chain 9

Olaf Chitil (University of Kent, UK)
Tracing Functional Programs
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Context 9 R
Chain 9
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 14
Chain 9

Olaf Chitil (University of Kent, UK)

Tracing Functional Programs
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Context 14 L
Chain 14
Chain 9

V ➔ myNot ➔ 8 ➔ 9 ➔ 11 ➔ 14 ➔ 16 ➔ 15 ➔ 14 ➔ 13 ➔ 10 ➔ 8
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack
Chain 16
Chain 14
Chain 9

8
9

myNot

myNot
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Context 14 R
Chain 14
Chain 9

myNot

myNot

v
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 19
Chain 14
Chain 9
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

- Chain 14
- Chain 9
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 21
Chain 9

Olaf Chitil (University of Kent, UK)
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Context 21 L
Chain 21
Chain 9
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 23
Chain 21
Chain 9
Translation from Event Sequence to ART

8: Var "z"  
9: App  
10: Enter 9 L  
11: Var "myNot"  
12: Value  
13: Enter 9 R  
14: App  
15: Enter 14 L  
16: Var "myNot"  
17: Value  
18: Enter 14 R  
19: Con "True" 0  
20: Value  
21: App  
22: Enter 21 L  
23: Var "myId"  
24: Value  
25: Enter 21 R  
26: Con "False" 0  
27: Value  
28: Con "False" 0  
29: Value  
30: App  
31: Enter 30 L  
32: Var "myId"  
33: Value  
34: Enter 30 R  
35: Con "True" 0  
36: Value  
37: Con "True" 0

Stack

Chain 21
Chain 9

myNot

myNot

True

myId

v

8

9
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Context 21 R
Chain 21
Chain 9
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 26
Chain 21
Chain 9

Olaf Chitil (University of Kent, UK)
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 21
Chain 9
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 28
Chain 9

myNot
myNot
True
myId
False
False
True
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 9

---

Olaf Chitil (University of Kent, UK)  Tracing Functional Programs
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 30

V

myNot

myNot

myId

False

True

False
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack
Context 30 L
Chain 30

Diagram:

- Var "z"
- App
- Enter 9 L
- Var "myNot"
- Value
- Enter 9 R
- App
- Enter 14 L
- Var "myNot"
- Value
- Enter 14 R
- Con "True" 0
- Value
- App
- Enter 21 L
- Var "myId"
- Value
- Enter 21 R
- Con "False" 0
- Value
- App
- Enter 30 L
- Var "myId"
- Value
- Enter 30 R
- Con "True" 0
- Value
- Con "True" 0

Olaf Chitil (University of Kent, UK)
Tracing Functional Programs
53 / 57
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 32
Chain 30

Olaf Chitil (University of Kent, UK)
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 30

Olaf Chitil (University of Kent, UK)
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack
Context 30 R
Chain 30
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 35
Chain 30

myNot
myNot
myId
myId
False
False
True
True

Olaf Chitil (University of Kent, UK)
Tracing Functional Programs
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 30
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack

Chain 37

\[ \text{myNot} \quad \text{True} \]
\[ \text{myNot} \quad \text{False} \]
\[ \text{myId} \quad \text{False} \]
\[ \text{myId} \quad \text{True} \]
Translation from Event Sequence to ART

8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value
18: Enter 14 R
19: Con "True" 0
20: Value
21: App
22: Enter 21 L
23: Var "myId"
24: Value
25: Enter 21 R
26: Con "False" 0
27: Value
28: Con "False" 0
29: Value
30: App
31: Enter 30 L
32: Var "myId"
33: Value
34: Enter 30 R
35: Con "True" 0
36: Value
37: Con "True" 0

Stack
Chain 37

Olaf Chitil (University of Kent, UK)
sort \, (x:xs) \, = \, \text{insert} \, x \, (\text{sort} \, xs)

No ART nodes for parameter variables, but component edges point backwards, to share.
A list of branches, \( x: [R,L,R] \) and \( xs: [R,R] \), locates the variable / computation in the left-hand side / ART redex.

\[
sort (x:xs) = \text{insert } x (\text{sort } xs)
\]
Additions to Tracing for Parameter ($\lambda$-bound) Variables

Additional event:

```haskell
data Events = ... | LamVar [Branch]
```

Additional tracing combinator:

```haskell
lamVar :: [Branch] -> a -> a
lamVar pos var = unsafePerformIO $ do
  var 'seq' sendEvent (LamVar pos)
  return var
```

Use `seq` to record variable computation before variable event itself.

Instrument equation of `sort`:

```haskell
sort (x:xs) =
  app2 (var "insert" insert) (lamVar [R,L,R] x)
  (app (var "sort" sort) (lamVar [R,R] xs))
```
Non-Instrumented Code Remains a Challenge

⊕ Instrumented and non-instrumented code have the same type. 
⇒ They can be combined.

⊖ Resulting event sequence does not yield an ART.
  • Values produced by non-instrumented code are not recorded; these may include functional values.
  • Calls from non-instrumented to instrumented code lead to several unconnected ART parts.

Future plan
Combine with our lightweight computation tree tracing (PLDI 2016).
  • Represent functional values as finite maps.
  • Use nesting of Enter-Value events.