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

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')
```

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')
```

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.

```
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'
```

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

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
```

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
```

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
```

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
```

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
```

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
```

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
 ⇝ (if True then factorial (2-1) * 2 else 1) * 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
 ⇝ (if True then factorial (2-1) * 2 else 1) * 3
 ⇝ (factorial (2-1) * 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
 ⇝ (if True then factorial (2-1) * 2 else 1) * 3
 ⇝ (factorial (2-1) * 2) * 3
 ⇝ (factorial 1 * 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
 ⇝ (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
```

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
 ⇝ (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
```

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
 ⇝ (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
```

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
 ⇝ (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
```

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
 ⇝ (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
```

Unbound Data Structures and Pattern Matching

- 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

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

A computation: `insert 'b' "ac"` $\leadsto^*$ `"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]` $\rightsquigarrow^*$ `[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]` $\rightsquigarrow^*$ `(1 * (2 * (3 * 1)))` $\rightsquigarrow^*$ `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`

```
ifPositive :: Integer -> a -> a -> a
ifPositive n yes no = if n > 0 then yes else no
```

  - can define infinite data structures

```
ones :: [Integer]
ones = 1 : ones
```

  - intermediate data structures (lists) do not increase space complexity

```
factorial :: Integer -> Integer
factorial n = product (enumFromTo 1 n)
```

Cf John Hughes *Why Functional Programming Matters*, 1989.

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`

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)
```

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))
```

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 []
```

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 []
```

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

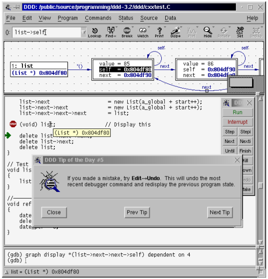
```
enumFrom :: Integer -> [Integer]
enumFrom b = b : enumFrom (b+1)   -- infinite list

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



```
1: list
(List *) 0x804df80

value = 85
self  = 0x804df80
next  = 0x804df90

value = 86
self  = 0x804df90
next  = 0x804df
```

```
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    (void) list;                // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date_
}
```

(List *) 0x804df80

DDD Tip of the Day #5

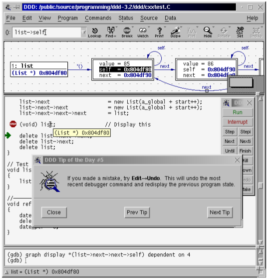
If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

Conventional Tracing & Debugging Methods

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



```
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    (void) list;                // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date
}
```

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

```
list = (List *) 0x804df80
```

These methods assume

- a single computation model
- of sequentially executing statements and
- mutating the computation state of variables.

Why Trace & Debug Functional Programs Differently?

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 `traceShow :: String -> [Int] -> [Int]`

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

main = print (take 5 (insert 4 [1..]))
```

Output:

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

- output mixed up
- non-termination ⇒ 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:

  ```
  (define/contract sqrt
    (bigger-than-zero? |-> bigger-than-zero?)
    (...))
  ```

  Popular for Scheme-dialect Racket.
- Use random testing of properties.
  Example property:

  ```
  prop_rev xs = reverse (reverse xs) == xs
  ```

  Popular for Haskell.

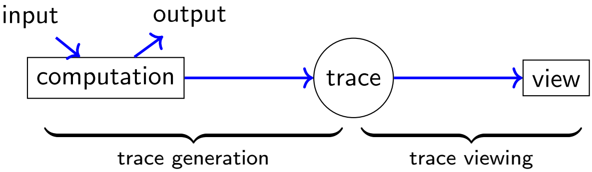
Outline

1. Features of Functional Programs $\surd$
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.

Freja	*Henrik Nilsson, An Algorithmic Debugger, ~1994.*
Tracer	*Sparud & Runciman, Explore Redex Trails Backwards, 1997.*
Buddha	*Bernard Pope, Another Algorithmic Debugger, ~1998.*
HOOD	*Andy Gill, Haskell Object Observation Debugger, 2000.*
Hat	*Runciman, Chitil, Wallace, The Haskell Tracer, 2000.*
BIO	*Braßel et al., Lazy Call-by-Value Evaluation, 2007.*

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

Unexpected output:

```
os
```

Example Program: 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
```

Observation of an Expression

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

```
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
```

```
"o"
"r"
"t"
""
```

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.

```
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
```

```
{'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.

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

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

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

```
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:

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

Algorithmic Debugging

```
main = putStrLn "os" ?
```

Algorithmic Debugging

```
main = putStrLn "os" ?    n
```

Algorithmic Debugging

```
main = putStrLn "os" ?    n
sort "sort" = "os" ?
```

Algorithmic Debugging

```
main = putStrLn "os" ?    n
sort "sort" = "os" ?    n
```

Algorithmic Debugging

```
main = putStrLn "os" ?    n
sort "sort" = "os" ?    n
insert 's' "o" = "os" ?
```

Algorithmic Debugging

```
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" ?
```

Algorithmic Debugging

```
main = putStrLn "os" ?    n
sort "sort" = "os" ?    n
insert 's' "o" = "os" ?    y
sort "ort" = "o" ?    n
```

Algorithmic Debugging

```
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
```

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 ?
```

Algorithmic Debugging

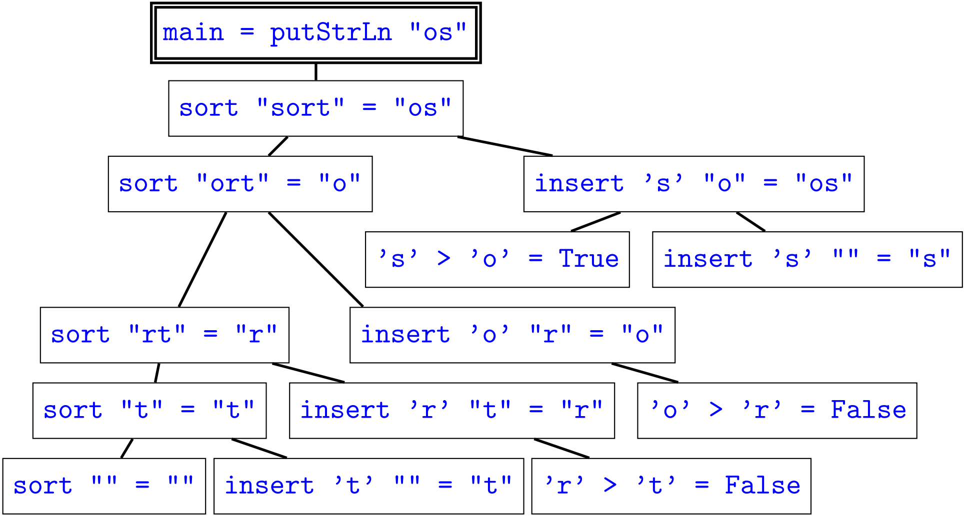
```
main = putStrLn "os" ?    n
sort "sort" = "os" ?    n
insert 's' "o" = "os" ?    y
sort "ort" = "o" ?    n
insert 'o' "r" = "o" ?    n
'o' > 'r' = False ?    y
```

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

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: 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"

sort "t" = "t"

insert ’r’ "t" = "r"

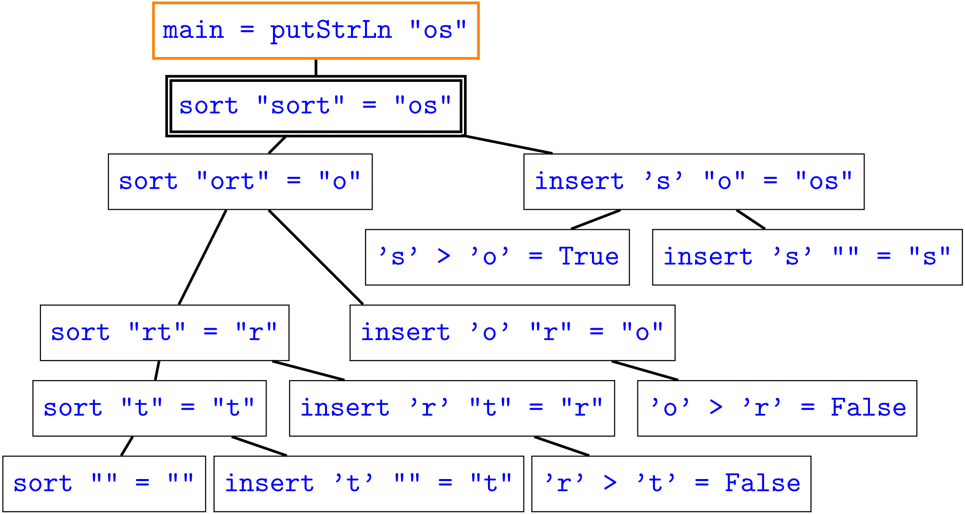
’o’ > ’r’ = False

sort "" = ""

insert ’t’ "" = "t"

’r’ > ’t’ = False

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"

sort "t" = "t"

insert 'r' "t" = "r"

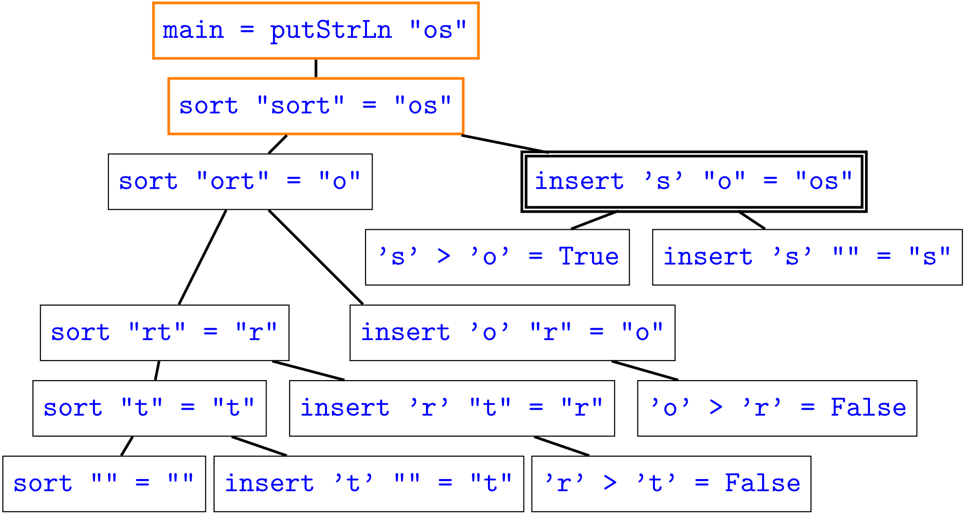
'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

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"`

`sort "t" = "t"`

`insert 'r' "t" = "r"`

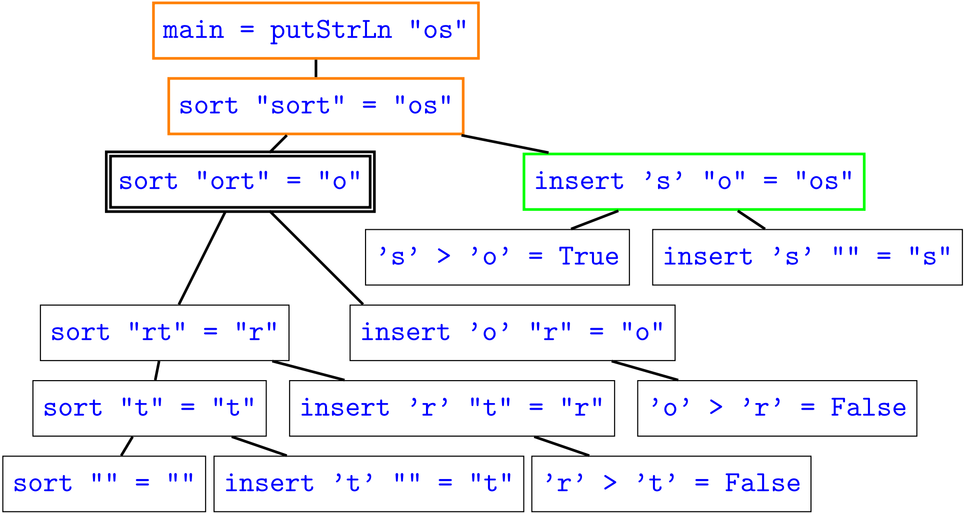
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

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"`

`sort "t" = "t"`

`insert 'r' "t" = "r"`

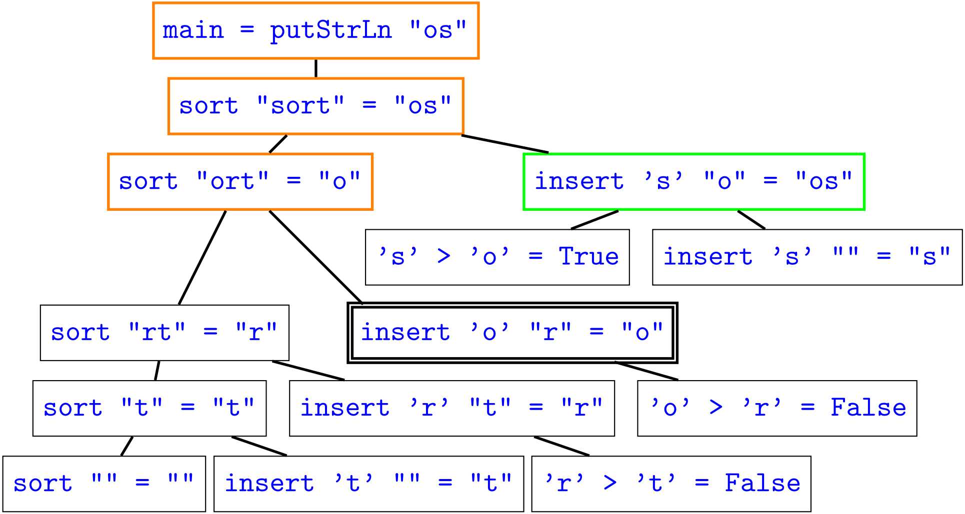
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

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"`

`sort "t" = "t"`

`insert 'r' "t" = "r"`

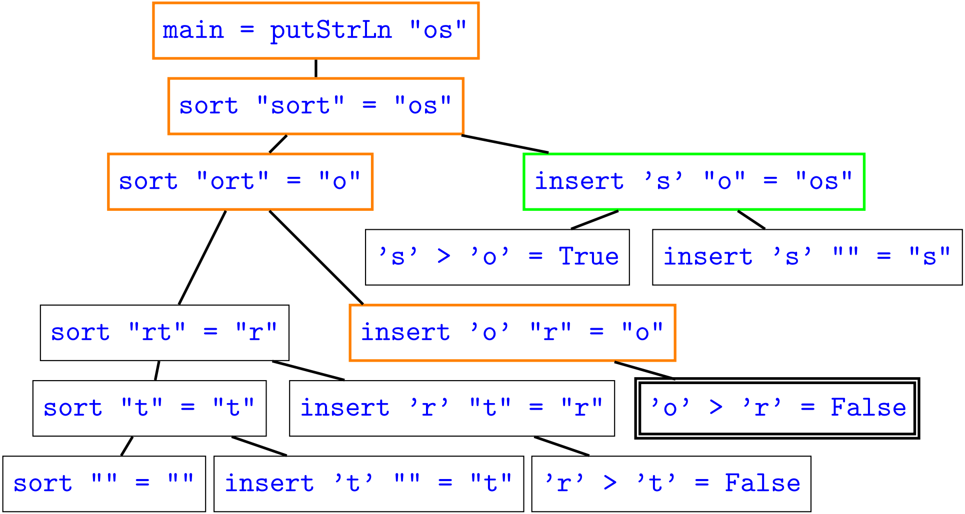
`'o' > 'r' = False`

`sort "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

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"

sort "t" = "t"

insert 'r' "t" = "r"

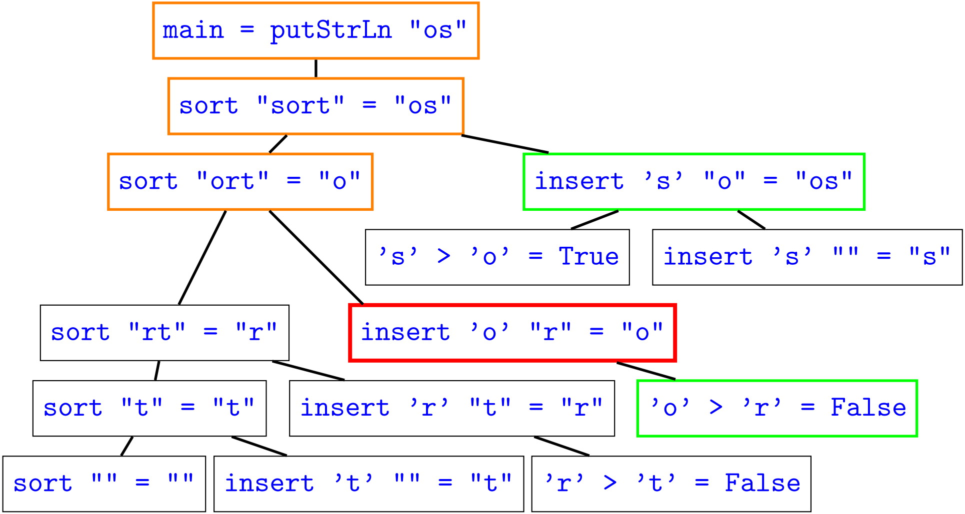
'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

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"
sort "t" = "t"
insert ’r’ "t" = "r"
’o’ > ’r’ = False
sort "" = ""
insert ’t’ "" = "t"
’r’ > ’t’ = False
```

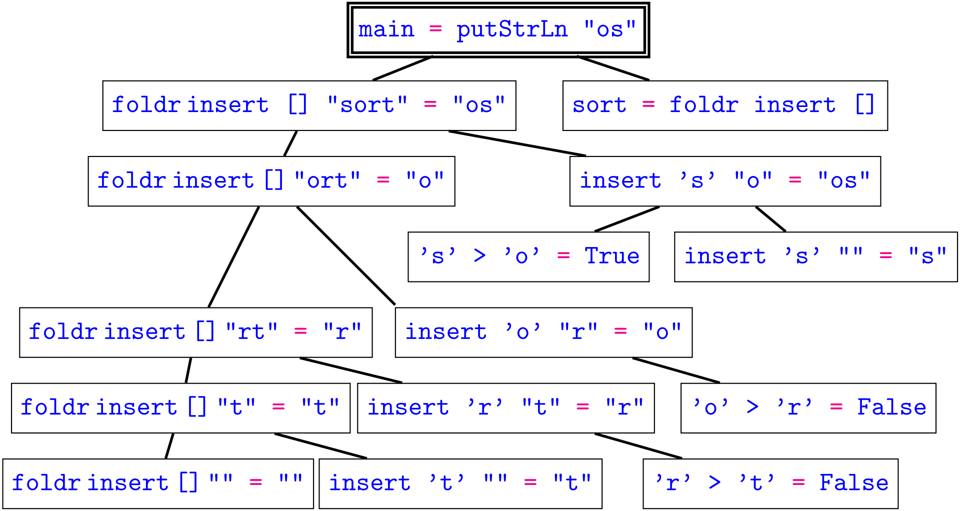
Recall: Faulty Higher-Order Insertion Sort

```
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"`

`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"`

`foldr insert [] "t" = "t"`

`insert 'r' "t" = "r"`

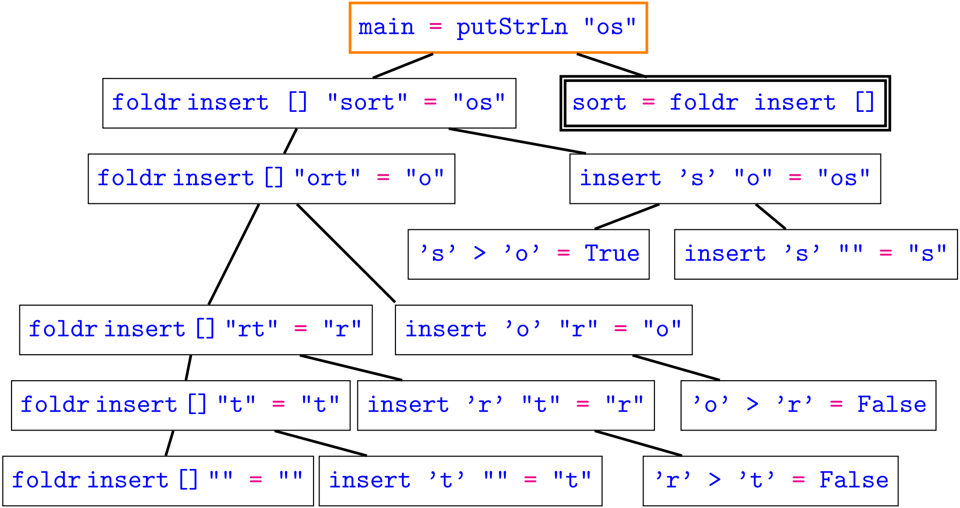
`'o' > 'r' = False`

`foldr insert [] "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

Higher-Order Insertion Sort: Evaluation Dependence Tree



`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"`

`foldr insert [] "t" = "t"`

`insert 'r' "t" = "r"`

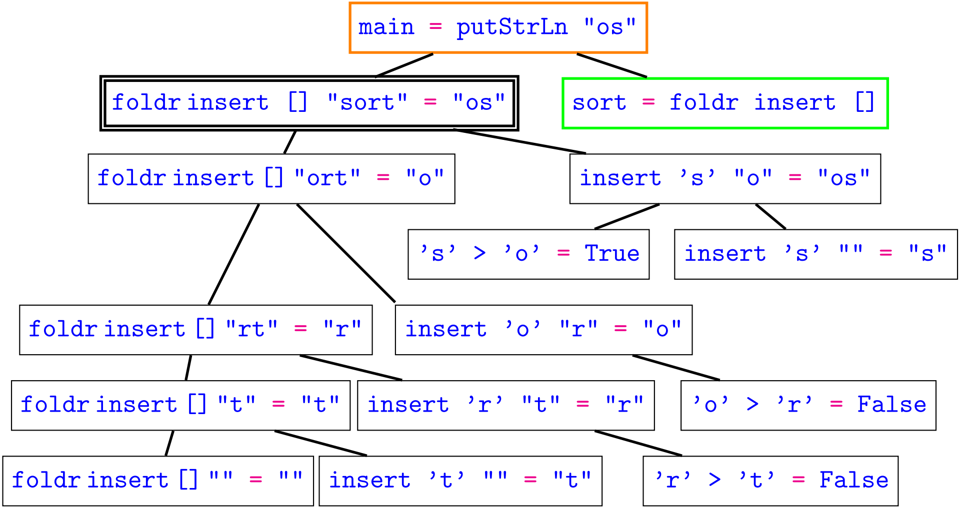
`'o' > 'r' = False`

`foldr insert [] "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

Higher-Order Insertion Sort: Evaluation Dependence Tree



`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"`

`foldr insert [] "t" = "t"`

`insert 'r' "t" = "r"`

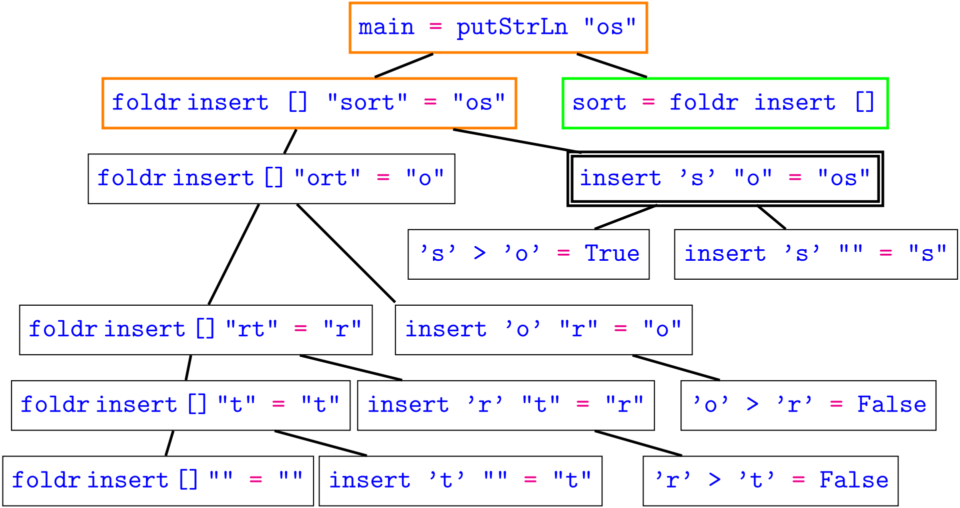
`'o' > 'r' = False`

`foldr insert [] "" = ""`

`insert 't' "" = "t"`

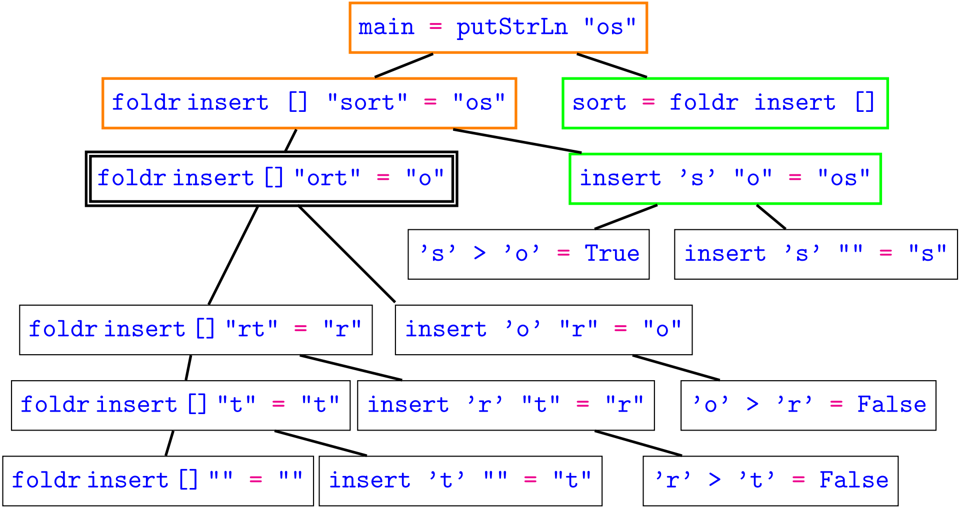
`'r' > 't' = False`

Higher-Order Insertion Sort: Evaluation Dependence Tree



```
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"
foldr insert [] "t" = "t"
insert 'r' "t" = "r"
'o' > 'r' = False
foldr insert [] "" = ""
insert 't' "" = "t"
'r' > 't' = False
```

Higher-Order Insertion Sort: Evaluation Dependence Tree



`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"`

`foldr insert [] "t" = "t"`

`insert 'r' "t" = "r"`

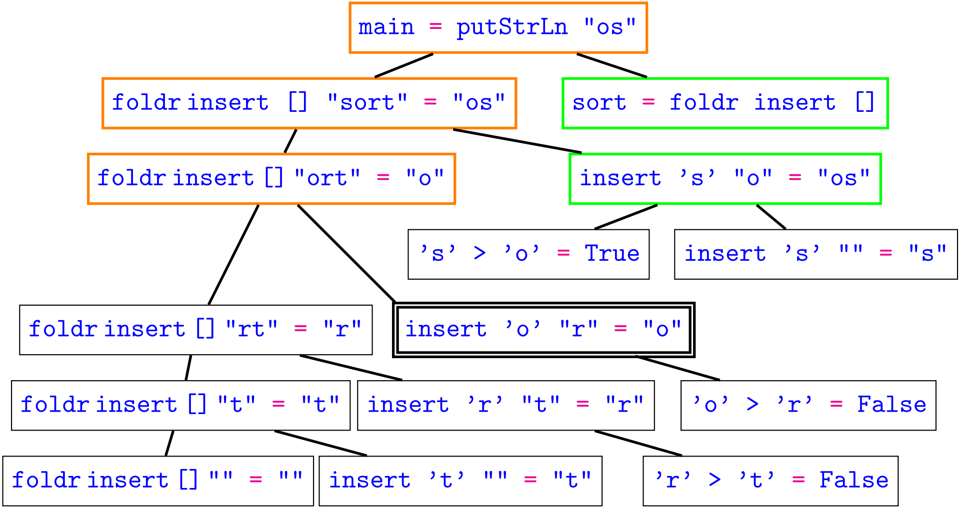
`'o' > 'r' = False`

`foldr insert [] "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

Higher-Order Insertion Sort: Evaluation Dependence Tree



`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"`

`foldr insert [] "t" = "t"`

`insert 'r' "t" = "r"`

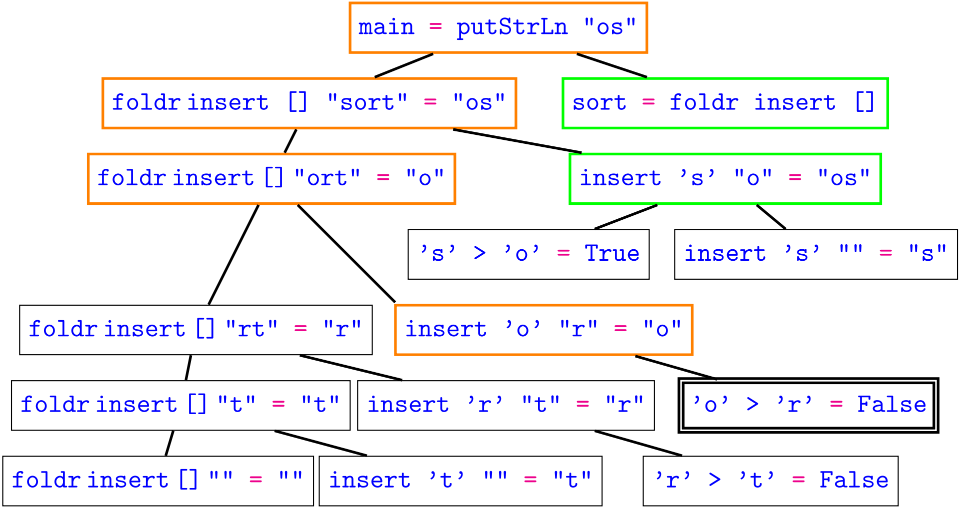
`'o' > 'r' = False`

`foldr insert [] "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

Higher-Order Insertion Sort: Evaluation Dependence Tree



`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"`

`foldr insert [] "t" = "t"`

`insert 'r' "t" = "r"`

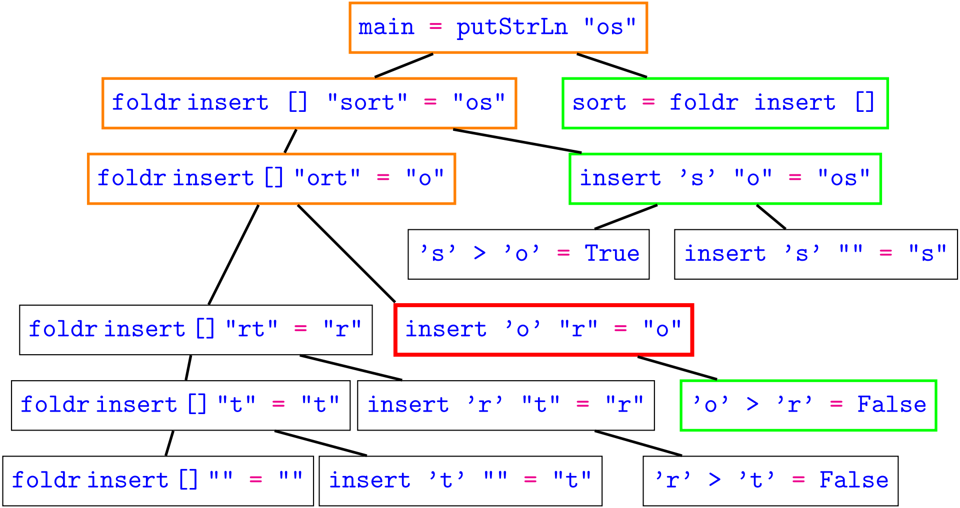
`'o' > 'r' = False`

`foldr insert [] "" = ""`

`insert 't' "" = "t"`

`'r' > 't' = False`

Higher-Order Insertion Sort: Evaluation Dependence Tree



`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"`

`foldr insert [] "t" = "t"`

`insert ’r’ "t" = "r"`

`’o’ > ’r’ = False`

`foldr insert [] "" = ""`

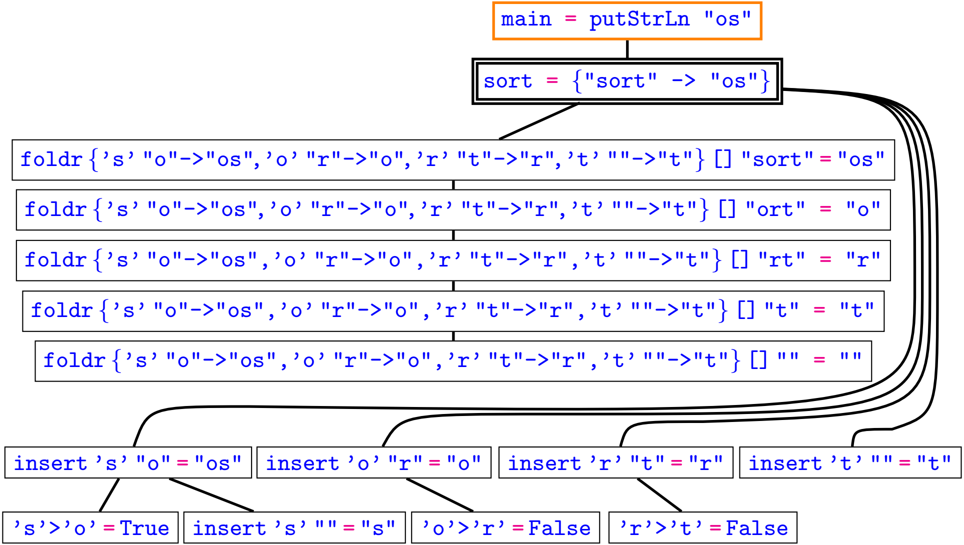
`insert ’t’ "" = "t"`

`’r’ > ’t’ = False`

Function Dependence Tree with Functions as Finite Maps

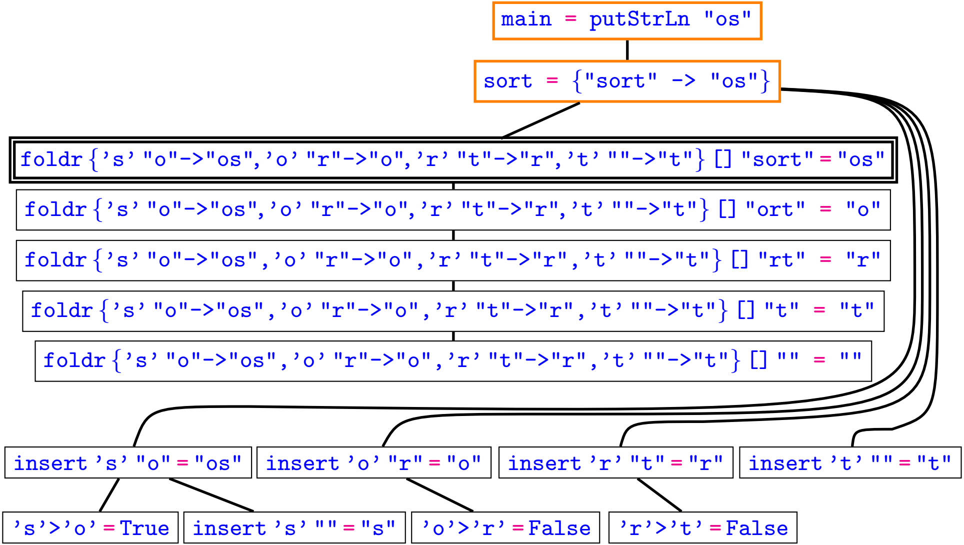
```
main = putStrLn "os"
sort = {"sort" -> "os"}
foldr {'s' "o"->"os",'o' "r"->"o",'r' "t"->"r",'t' ""->"t"} [] "sort" = "os"
foldr {'s' "o"->"os",'o' "r"->"o",'r' "t"->"r",'t' ""->"t"} [] "ort" = "o"
foldr {'s' "o"->"os",'o' "r"->"o",'r' "t"->"r",'t' ""->"t"} [] "rt" = "r"
foldr {'s' "o"->"os",'o' "r"->"o",'r'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



```
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"

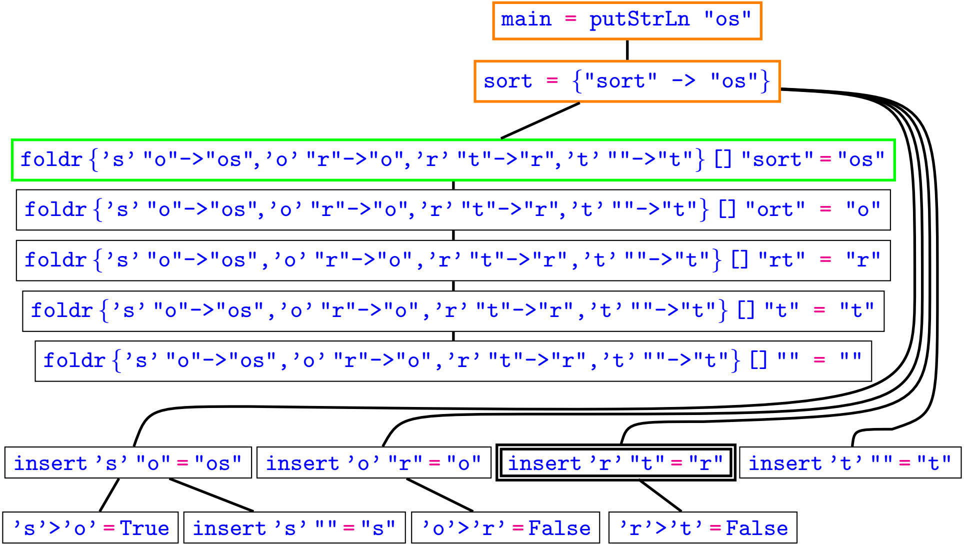
’s’>’o’ = True

insert ’s’ "" = "s"

’o’>’r’ = False

’r’>’t’ = False

Function Dependence Tree with Functions as Finite Maps



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"

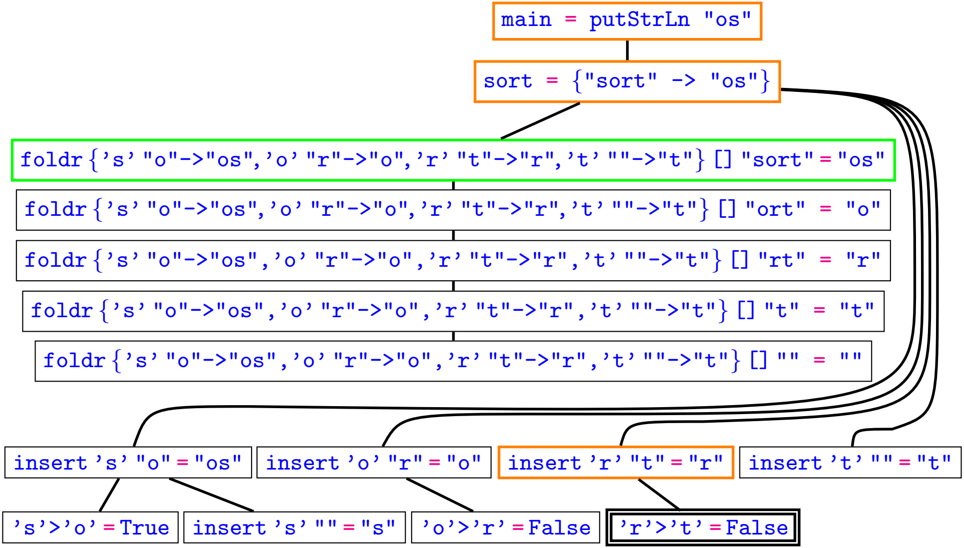
's'>'o' = True

insert 's' "" = "s"

'o'>'r' = False

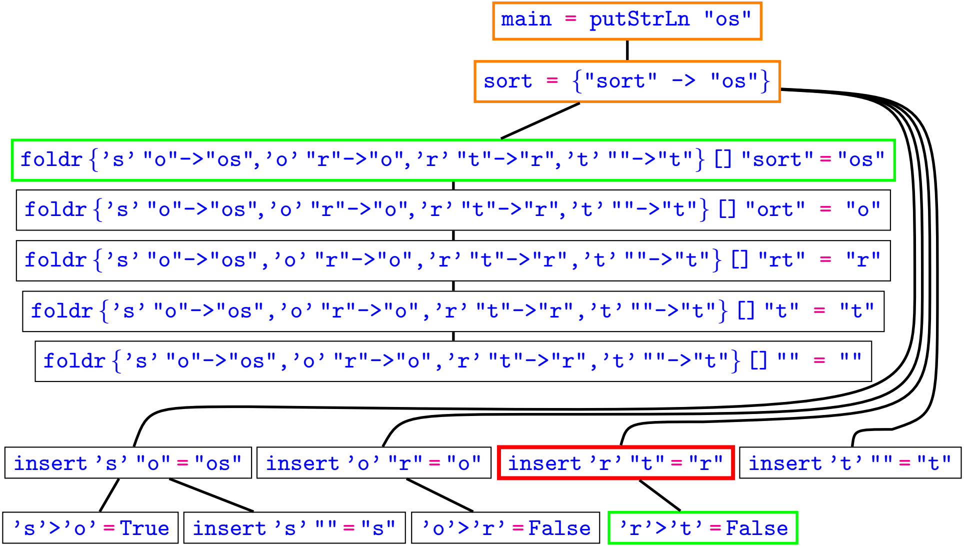
'r'>'t' = False

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



```
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
```

Combining Free Tree Navigation, Source, Program Slicing

```
==== Hat-Explore 2.00 ==== Call 2/2 ========================
 1.  main = putStrLn "os"
 2.  sort "sort" = "os"
 3.  sort "ort" = "o"

---- Insert.hs ---- lines 3 to 8 --------------------------
sort :: [Char] -> [Char]
sort []      = []
sort (x:xs) = insert x (sort xs)

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

Reminds of stepping debugger, but freely going forwards and backwards.

Following a Redex Trail

```
Output: ------------------------------------
os\n
Trail:  ------- Insert.hs -----------------------------
```

Following a Redex Trail

```
Output:  -----------------------------------------------
os\n
Trail:  ------- Insert.hs ------------------------------
<- putStrLn "os"
```

Following a Redex Trail

```
Output: ----------------------------------------
os\n
Trail: ------- Insert.hs --------------------------
<- putStrLn "os"
<- insert 's' "o" | if True
```

Following a Redex Trail

```
Output:  -----------------------------------------------
os\n
Trail:  ------- Insert.hs ------------------------------
<- putStrLn "os"
<- insert 's' "o" | if True
<- insert 'o' "r" | if False
```

Following a Redex Trail

```
Output: ---------------------------------------
os\n
Trail:  ------- Insert.hs ---------------------------
<- putStrLn "os"
<- insert 's' "o" | if True
<- insert 'o' "r" | if False
<- insert 'r' "t" | if False
```

Following a Redex Trail

```
Output:  ------------------------------------------------
os\n
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\n
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

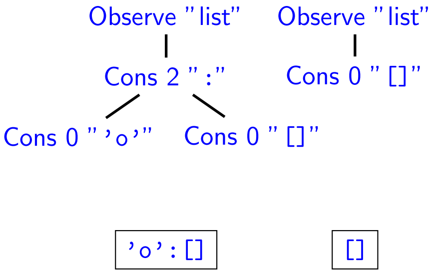
All program transformations and modified abstract machine are complex.

HOOD: To Observe Values, Generate an Event Sequence

```
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 ":"
7	...	
8	Parent 6 Left	Cons 0 "'o'"
9	...	
10	Parent 6 Right	Cons 0 "[]"



HOOD: Generate Event Sequence

```
observe :: Observable a => String -> a -> a
observe label orig = unsafePerformObs $ 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 =
  unsafePerformObs (observer parent orig)
```

Reconstructing Computation Tree Nodes

Observe all suspected top-level functions as follows:

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

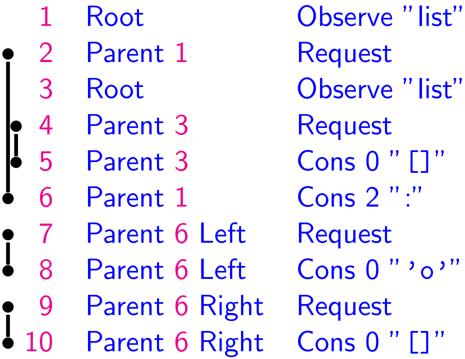
HOOD gives

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

So we get the nodes of the computation tree:

```
insert 'o' "r" = "o"
...
```

Reconstructing Computation Tree Edges: Event Brackets

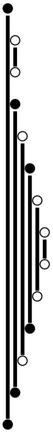


1	Root	Observe "list"
2	Parent 1	Request
3	Root	Observe "list"
4	Parent 3	Request
5	Parent 3	Cons 0 "[]"
6	Parent 1	Cons 2 ":"
7	Parent 6 Left	Request
8	Parent 6 Left	Cons 0 "'o'"
9	Parent 6 Right	Request
10	Parent 6 Right	Cons 0 "[]"

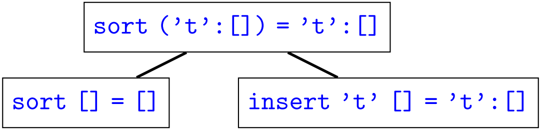
- Every non-Root event is preceded by a request event.
- Request + response event are like brackets in event sequence:
  - either in sequence (one pair after another)
  - or nested

Reconstructing Computation Tree Edges: Nesting

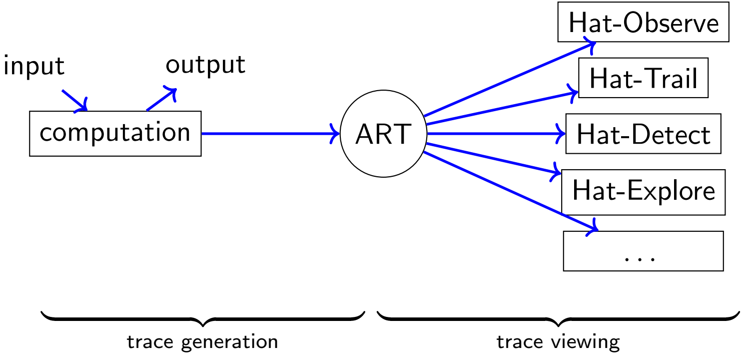
- Event brackets for results are directly nested.
- Brackets for any argument make a gap in surrounding nesting.



request	result of 1^{st} `sort`	
request	argument of 1^{st} `sort`	
response	argument of 1^{st} `sort`	is `:`
request	result of `insert`	
request	2^{nd} argument of `insert`	
request	result of 2^{nd}`sort`	
request	argument of 2^{nd} `sort`	
request	part of argument of 1^{st} `sort`	
response	part of argument of 1^{st} `sort`	is `[]`
response	argument of 2^{nd} `sort`	is `[]`
response	result of 2^{nd} `sort`	is `[]`
response	2^{nd} argument of `insert`	is `[]`
response	result of `insert`	is `:`
response	result of 1^{st} `sort`	is `:`



A Universal Trace: The Augmented Redex Trail (ART)


input
output
computation
ART
Hat-Observe
Hat-Trail
Hat-Detect
Hat-Explore
. . .
trace generation
trace viewing

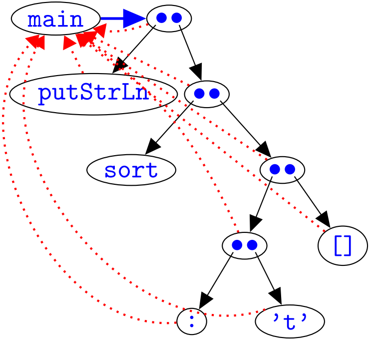
ART contains wealth of information for numerous views.

Structure of the Augmented Redex Trail (ART)

The ART is a graph of nodes and three types of edges.

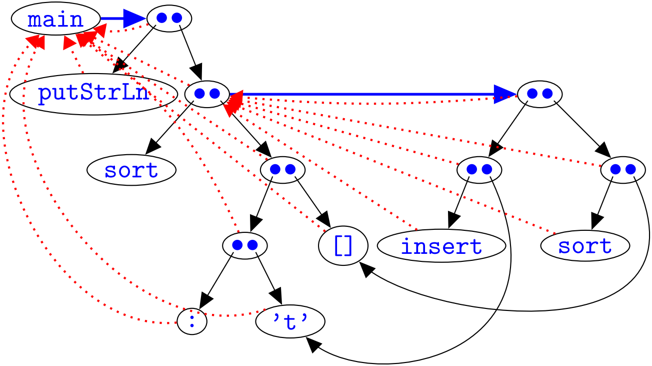
```
main = putStrLn (sort [’t’])

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
```

Structure of the Augmented Redex Trail (ART)



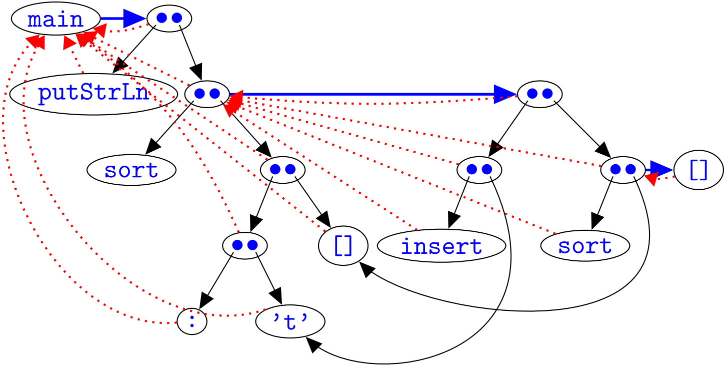
```
main = putStrLn (sort ['t'])

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
```

Structure of the Augmented Redex Trail (ART)



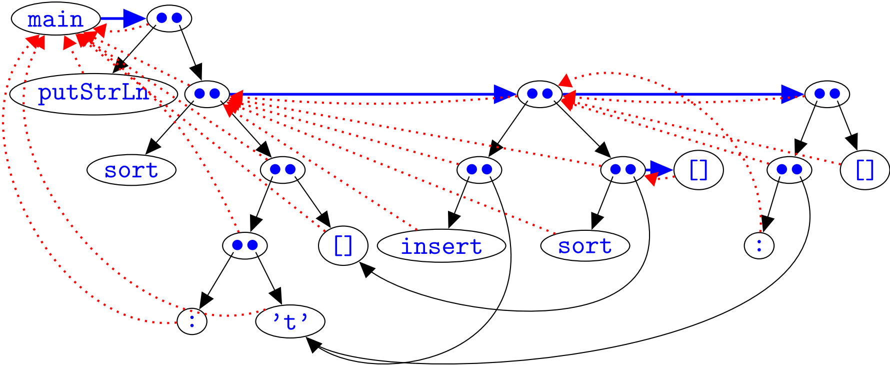
```
main = putStrLn (sort ['t'])

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
```

Structure of the Augmented Redex Trail (ART)



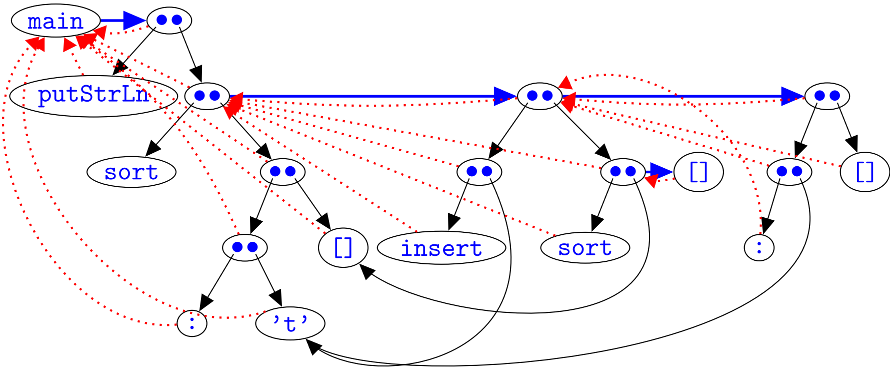
```
main = putStrLn (sort ['t'])

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
```

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
```

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

```
sort :: [Char] -> [Char]
sort [] = con "[]" 0 []
sort (x:xs) =
  app2 (var "insert" insert) (lamVar [R,L,R] x)
    (app (var "sort" sort) (lamVar [R,R] xs))

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

More invasive transformation, but all types are unchanged.

Part V

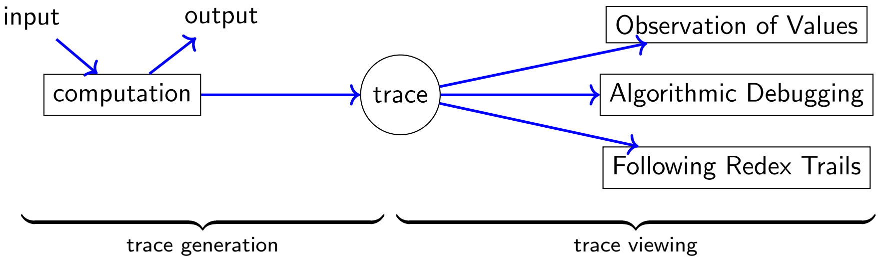
Open Challenges

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..]
```

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

```
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..])
```

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..]))
 ⇝ or (False : map (== 42) [2..])
 ⇝ or (map (== 42) [2..])
```

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

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

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..]))
 ⇝ or (False : map (== 42) [2..])
 ⇝ or (map (== 42) [2..])
 ⇝ or (map (== 42) (2:[3..]))
 ⇝ or (False : map (== 42) [3..])
 ⇝ or (map (== 42) [3..])
 ⋮   ⋮
```

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..]))
 ⤳ or (False : map (== 42) [2..])
 ⤳ or (map (== 42) [2..])
 ⤳ or (map (== 42) (2:[3..]))
 ⤳ or (False : map (== 42) [3..])
 ⤳ or (map (== 42) [3..])
 ⋮   ⋮
 ⤳ True
```

Here reduction steps for `map` and `or` are skipped.

Hat-Trans: Transforming Haskell for Tracing

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] -> [a] -> [a]
```

becomes

```
T.Fun (T.List a) (T.Fun (T.List a) (T.List a))
```

Hat-Trans: An Example

```
rev :: [a] -> [a] -> [a]
rev [] ys = ys
rev (x:xs) ys = rev xs (x:ys)
```

is transformed into

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

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

tMain = T.mkModule "Main" "Test.hs" Prelude.True
arev = T.mkVariable tMain 50001 60028 3 2 "rev" Prelude.False
p5v13v5v14 = T.mkSrcPos tMain 50013 50014
p6v17v6v28 = T.mkSrcPos tMain 60017 60028
p6v17v6v19 = T.mkSrcPos tMain 60017 60019
```

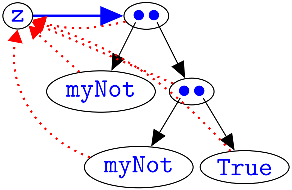
Idea: Use Hood's Instrumentation Method

Instrument code with side effects that write an event sequence.

```
sendEvent :: String -> IO a

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

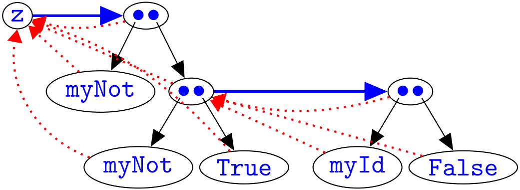
Instrumentation: `True` ⇝ `tr "True" True`

Essential to ART Structure: Chains of Reductions



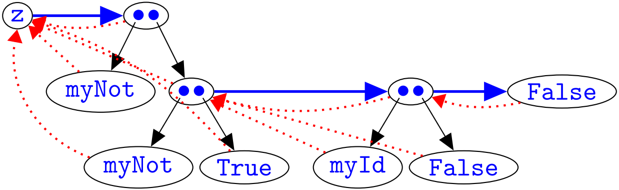
```
myId True = True
myId False = False
myNot True = myId False
myNot False = myId True
z = myNot (myNot True)
```

Essential to ART Structure: Chains of Reductions



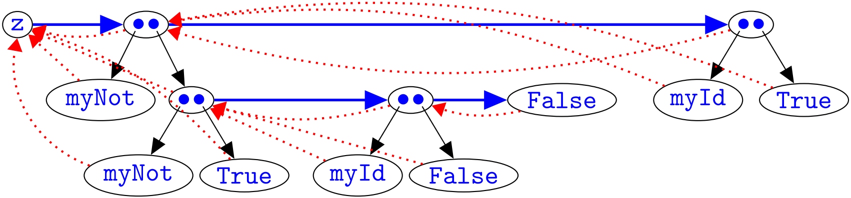
```
myId True = True
myId False = False

myNot True = myId False
myNot False = myId True

z = myNot (myNot True)
```

Essential to ART Structure: Chains of Reductions



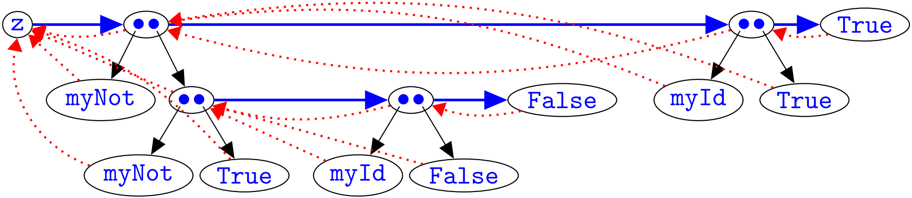
```
myId True = True
myId False = False
myNot True = myId False
myNot False = myId True
z = myNot (myNot True)
```

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)
```

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)
```

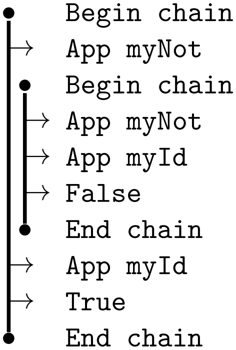
There are two chains of reductions, one nested within the other.

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.

```
ev :: a -> a
ev x = unsafePerformIO $ do
  sendEvent "Begin chain"
  x ’seq’ sendEvent "End chain"
  return x
```

An event sequence:



```
● Begin chain
→ App myNot
 ● Begin chain
 → App myNot
 → App myId
 → False
 ● End chain
→ App myId
→ True
● End chain
```

HatLight's Events and Tracing Combinators

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

type EventId = Int
data Branch = L | R
```

```haskell
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

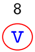
```
myId :: Bool -> Bool
myId True = con "True" 0 True
myId False = con "False" 0 False

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

Chain 8
⋮

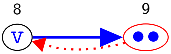


Translation from Event Sequence to ART

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 9
⋮



Translation from Event Sequence to ART

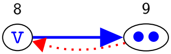
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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 9 L
Chain 9
⋮

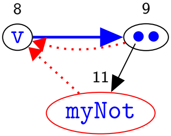


Translation from Event Sequence to ART

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 11
Chain 9
⋮

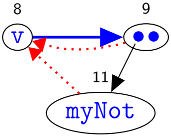


Translation from Event Sequence to ART

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 9
⋮



Translation from Event Sequence to ART

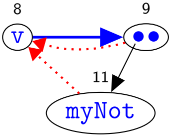
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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 9 R
Chain 9
⋮



Translation from Event Sequence to ART

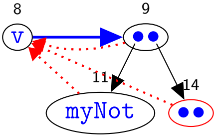
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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
⋮



Translation from Event Sequence to ART

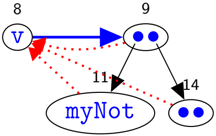
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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
⋮



Translation from Event Sequence to ART

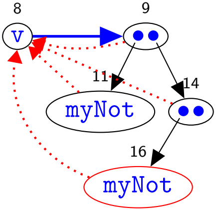
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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
⋮

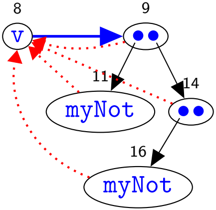


Translation from Event Sequence to ART

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

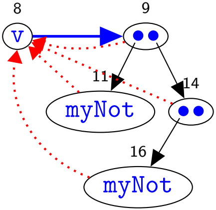
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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

Context 14 R
Chain 14
Chain 9
⋮

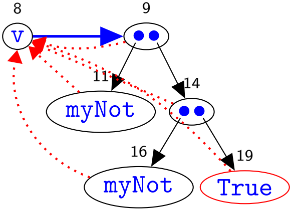


Translation from Event Sequence to ART

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 19
Chain 14
Chain 9
⋮

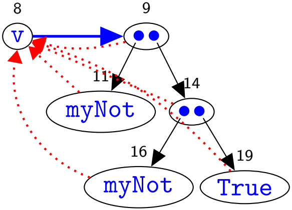


Translation from Event Sequence to ART

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
⋮

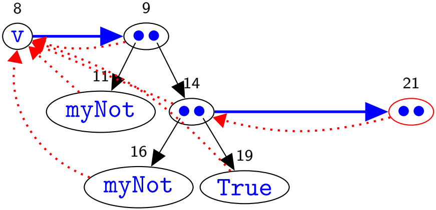


Translation from Event Sequence to ART

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
⋮

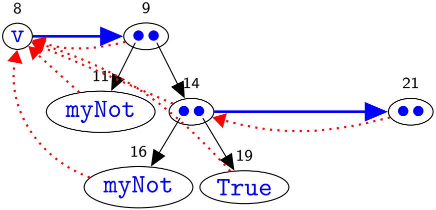


Translation from Event Sequence to ART

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
⋮

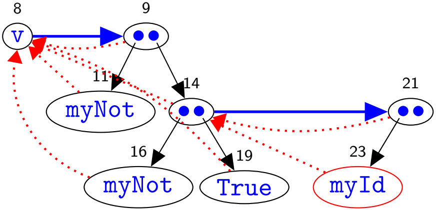


Translation from Event Sequence to ART

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
⋮

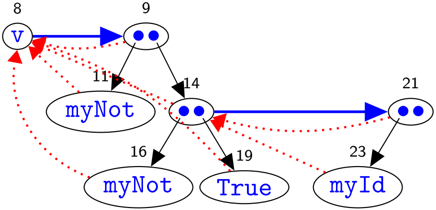


Translation from Event Sequence to ART

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
⋮

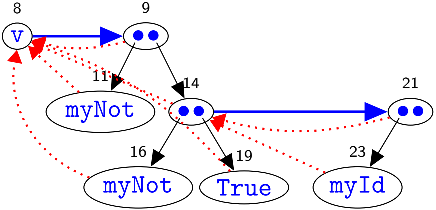


Translation from Event Sequence to ART

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
⋮

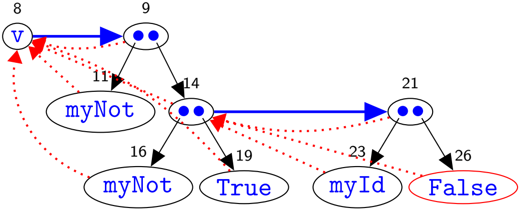


Translation from Event Sequence to ART

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
⋮

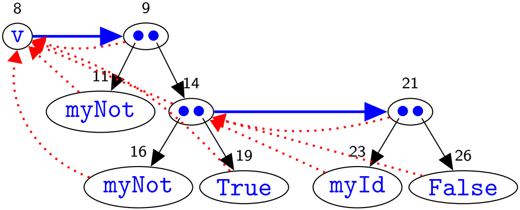


Translation from Event Sequence to ART

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
⋮

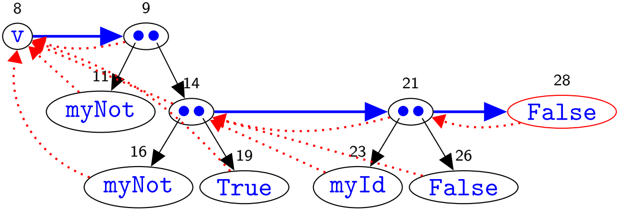


Translation from Event Sequence to ART

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
⋮



Translation from Event Sequence to ART

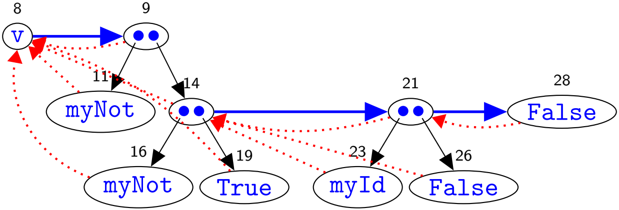
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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
⋮

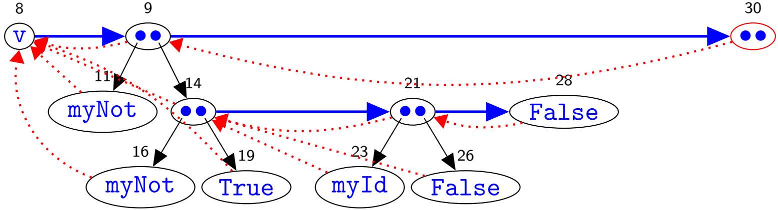


Translation from Event Sequence to ART

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

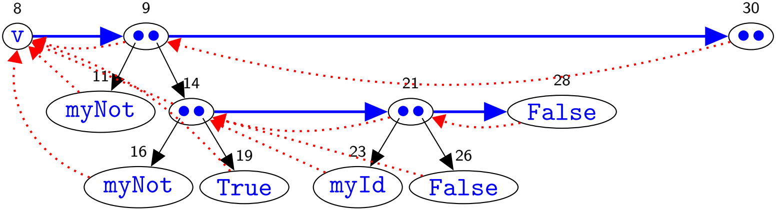
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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

Context 30 L
Chain 30
⋮

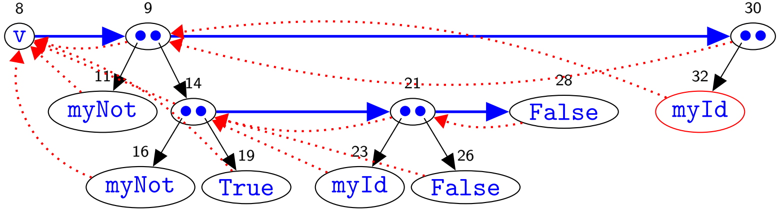


Translation from Event Sequence to ART

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 32
Chain 30
⋮

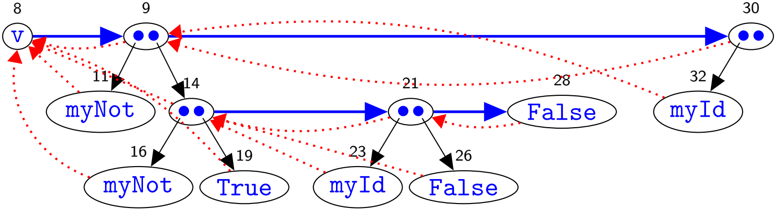


Translation from Event Sequence to ART

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 30
⋮



Translation from Event Sequence to ART

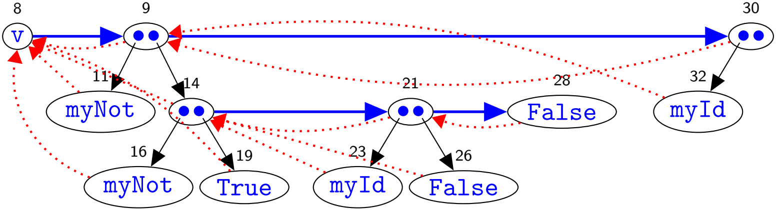
8: Var "z"
9: App
10: Enter 9 L
11: Var "myNot"
12: Value
13: Enter 9 R
14: App
15: Enter 14 L
16: Var "myNot"
17: Value

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

Context 30 R
Chain 30
⋮

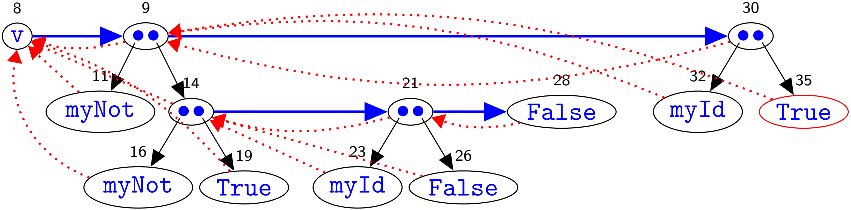


Translation from Event Sequence to ART

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
⋮

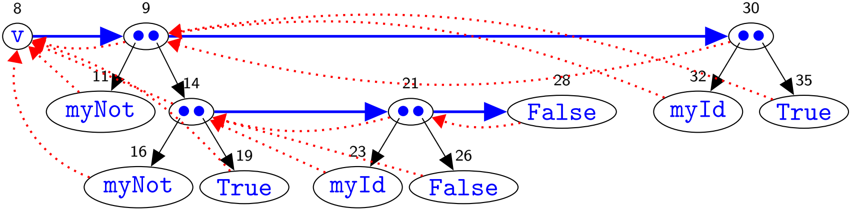


Translation from Event Sequence to ART

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
⋮

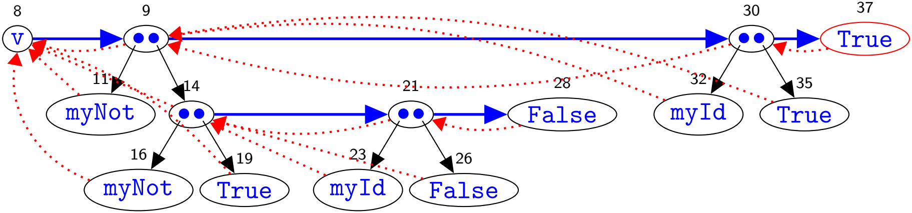


Translation from Event Sequence to ART

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
⋮

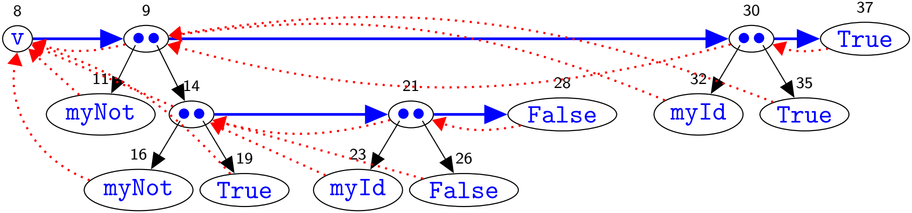


Translation from Event Sequence to ART

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

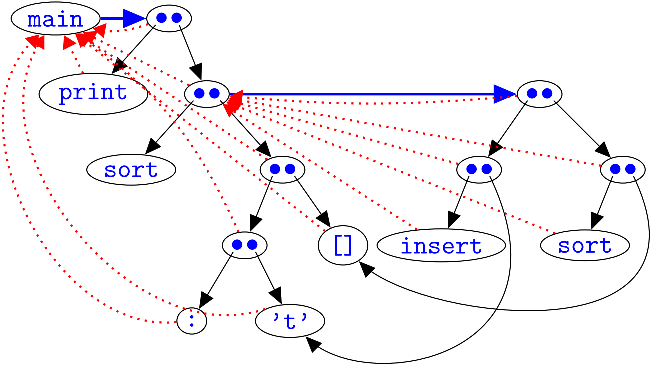
Stack

Chain 37
⋮



Still to Cover: Parameter Variables

`sort (x:xs) = insert x (sort xs)`

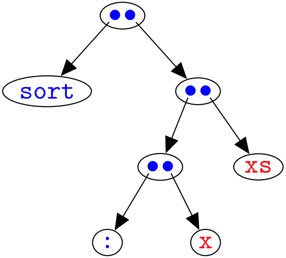


No ART nodes for parameter variables,
but component edges point backwards, to share.

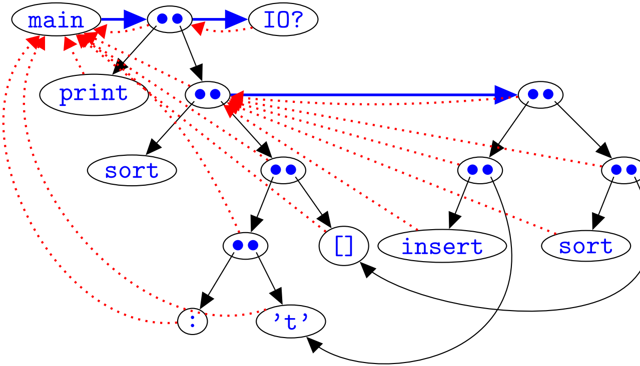
Solution for Parameter Variables

A list of branches, `x`: `[R,L,R]` and `xs`: `[R,R]`,
locates the variable / computation in the left-hand side / ART redex.

syntax tree of
left-hand side



ART



`sort (x:xs) = insert x (sort xs)`

Additions to Tracing for Parameter (λ-bound) Variables

Additional event:

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

Additional tracing combinator:

```
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`:

```
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.