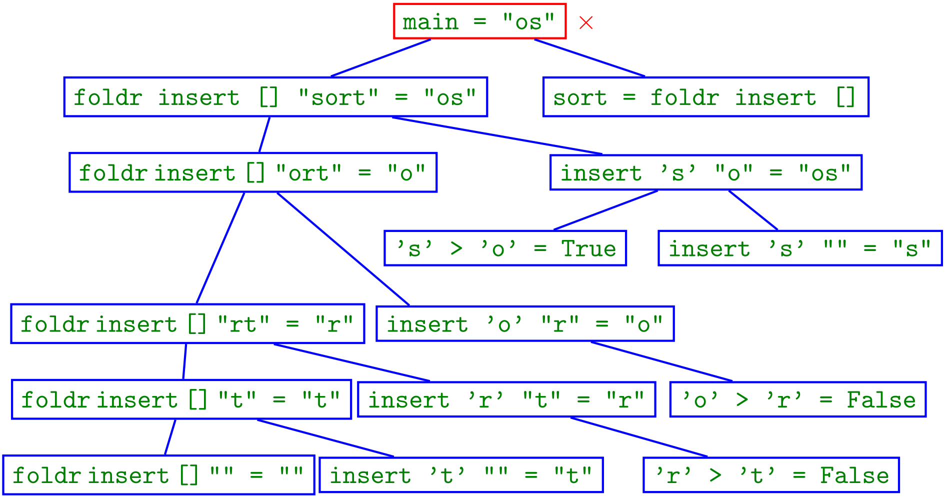
Comprehending Finite Maps for Algorithmic Debugging of Higher-Order Functional Programs

Olaf Chitil and Thomas Davie

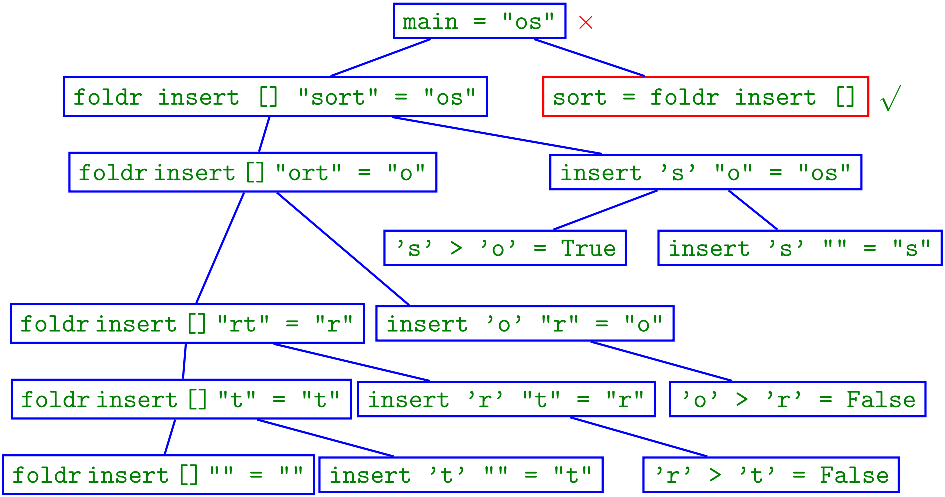
University of Kent, UK

16th July 2008

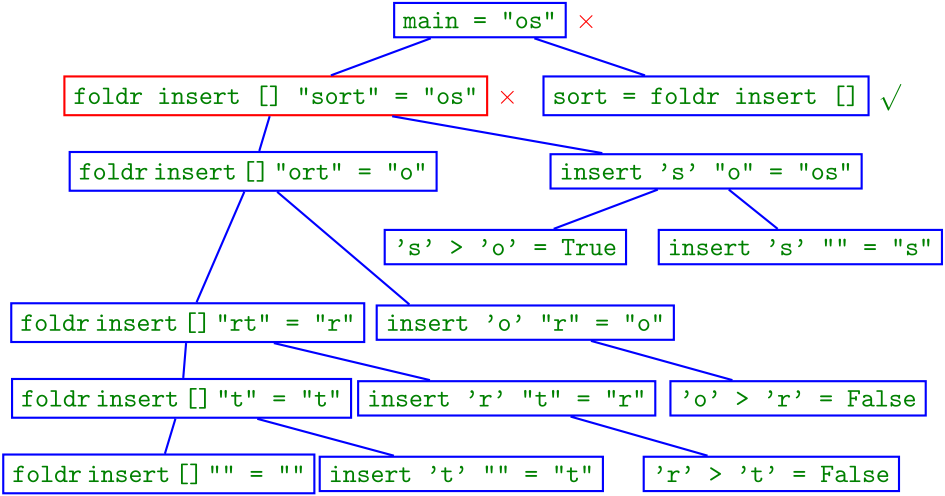
Algorithmic Debugging: Faulty Equation in Tree



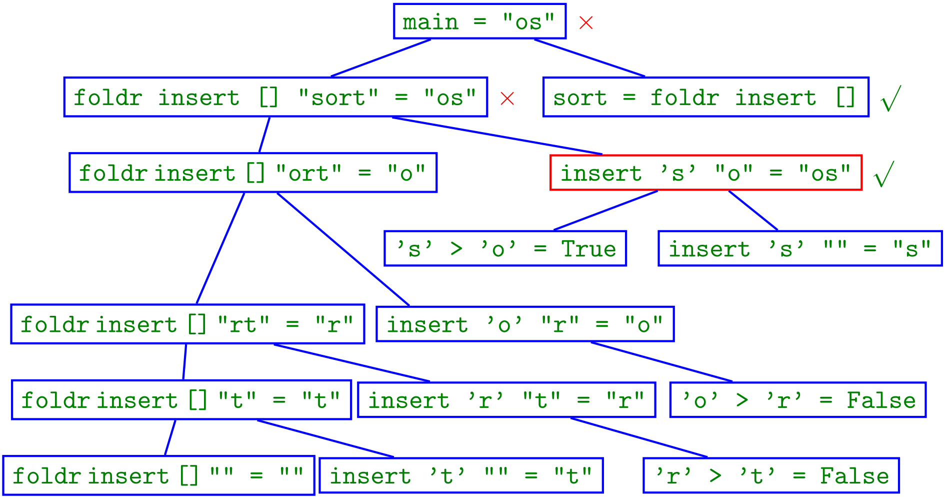
Algorithmic Debugging: Faulty Equation in Tree



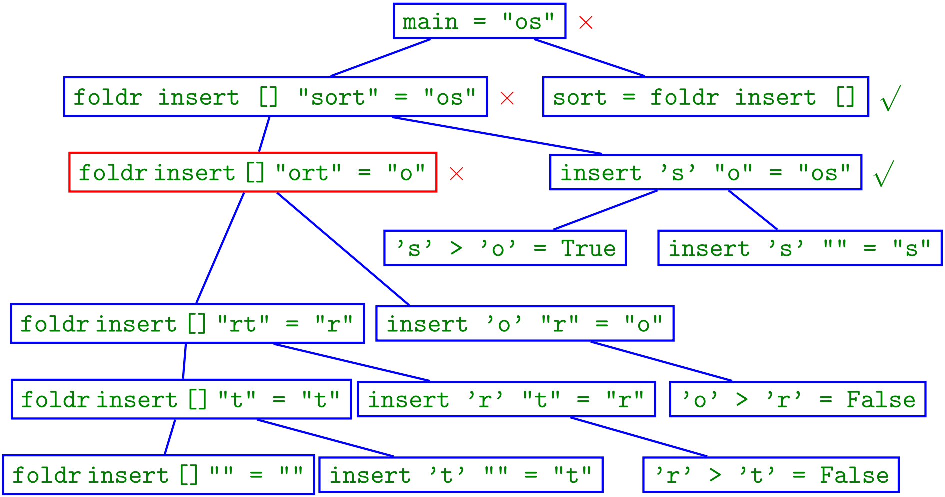
Algorithmic Debugging: Faulty Equation in Tree



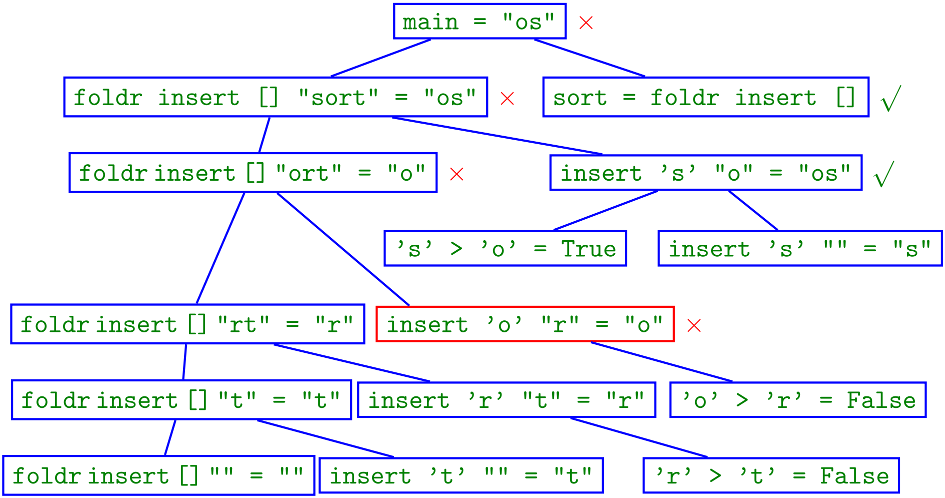
Algorithmic Debugging: Faulty Equation in Tree



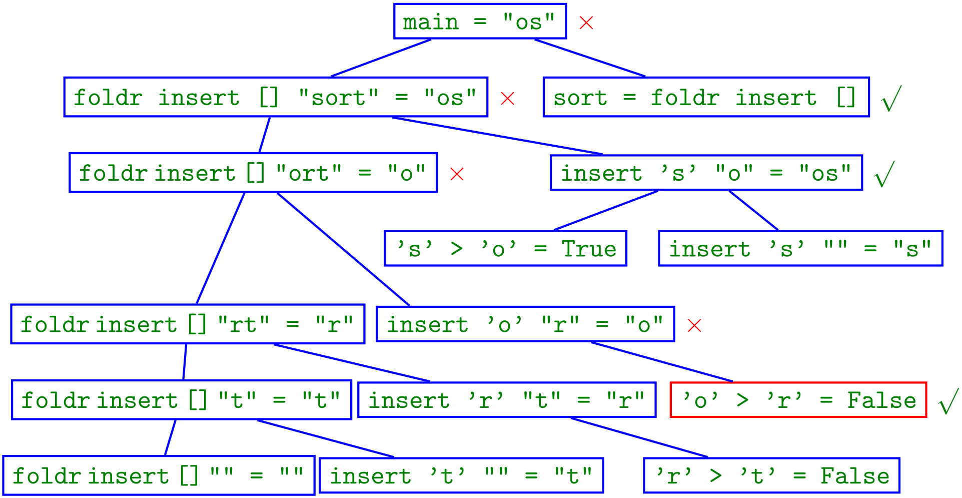
Algorithmic Debugging: Faulty Equation in Tree



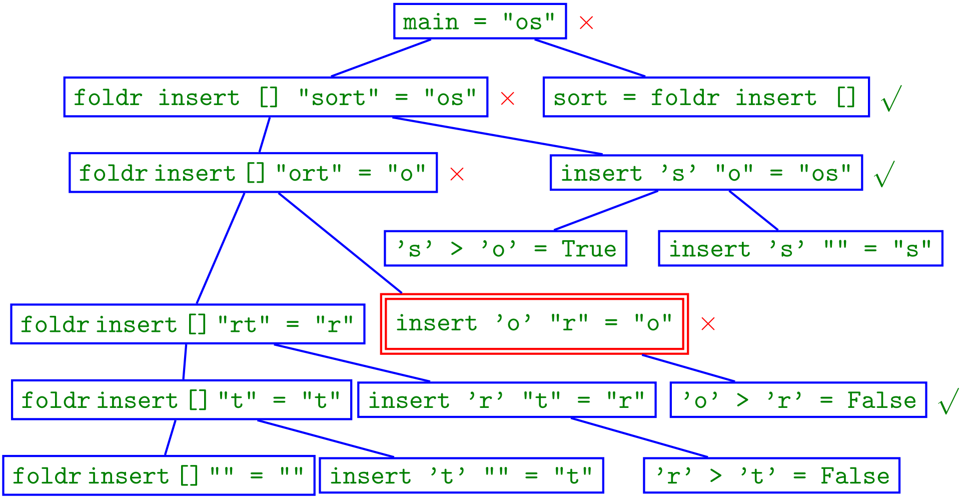
Algorithmic Debugging: Faulty Equation in Tree



Algorithmic Debugging: Faulty Equation in Tree



Algorithmic Debugging: Faulty Equation in Tree



Fault located!

Faulty computation: `insert 'o' "r" = "o"`

Program

```
main :: String
main = sort "sort"

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

Representation of a Functional Value

As Applicative term

```
parse
 (pSucc (flip ($)) <*> (pSucc (const 1) <*> pSym '1') <*>
  (pSucc id <|> pSucc combine <*>
   (pSucc (const 0) <*> pSym '0' <|> pSucc (const 0) <*> pSym '1') <*>
    (pSucc id <|> pSucc combine <*>
     (pSucc (const 0) <*> pSym '0' <|> pSucc (const 0) <*> pSym '1') <*>
      (pSucc id <|> pSucc combine <*>
       (pSucc (const 0) <*> pSym '0' <|> pSucc (const 0) <*> pSym '1') <*>
        _))))
"101"
= [4] ?
```

Program fragment

```
type Parser a = String -> [(a,String)]
parse :: Parser a -> String -> [a]
```

Representation of a Functional Value

As Applicative term

```
parse
 (pSucc (flip ($)) <*> (pSucc (const 1) <*> pSym '1') <*>
  (pSucc id <|> pSucc combine <*>
   (pSucc (const 0) <*> pSym '0' <|> pSucc (const 0) <*> pSym '1') <*>
    (pSucc id <|> pSucc combine <*>
     (pSucc (const 0) <*> pSym '0' <|> pSucc (const 0) <*> pSym '1') <*>
      (pSucc id <|> pSucc combine <*>
       (pSucc (const 0) <*> pSym '0' <|> pSucc (const 0) <*> pSym '1') <*>
        _))))
"101"
= [4] ?
```

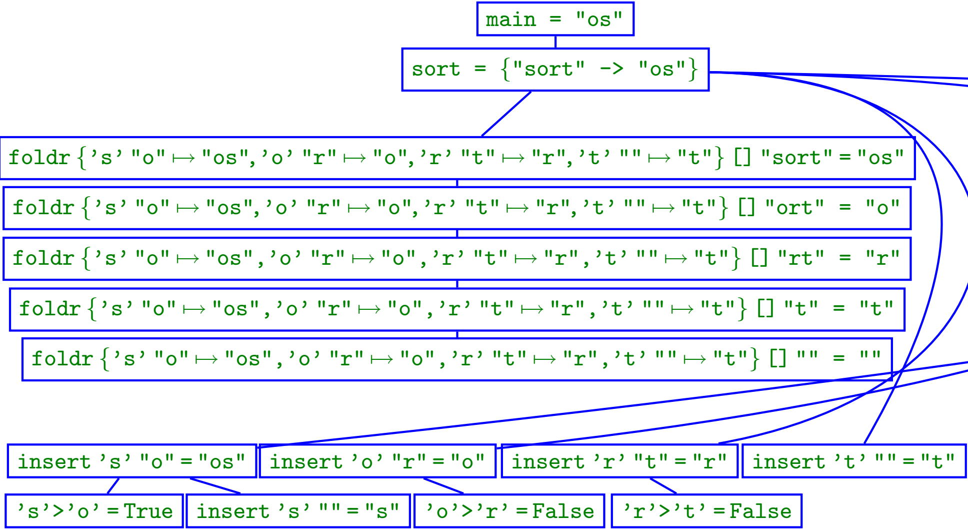
Program fragment

```
type Parser a = String -> [(a,String)]
parse :: Parser a -> String -> [a]
```

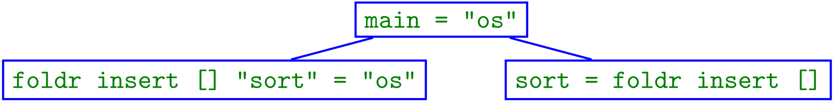
As Finite map

```
parse {"101" ↦ [(_,"01"),(_,"1"),(4,[])]} = [4]
```

Function Dependency Tree: Functions as Finite Maps



Compositional Tree enables Algorithmic Debugging




```
main
```
= {program equation of `main`}
```
sort "sort"
```
= {child}
```
foldr insert [] "sort"
```
= {child}
```
"os"
```

Program fragment

```
main = sort "sort"
```

Soundness of Algorithmic Debugging

If parent equation is incorrect and all child equations are correct, then program equation of parent is faulty.

Soundness for Functions as Finite Maps

`main = "os"`

`sort = {"sort" ↦ "os"}`

```
   main
```

$=$ {program equation of `main`}

```
   sort "sort"
```

$=$ {child}

```
   {"sort" ↦ "os"} "sort"
```

$=$ {assumed property of intended semantics}

```
   "os"
```

Program fragment

```
main = sort "sort"
```

Soundness of Algorithmic Debugging

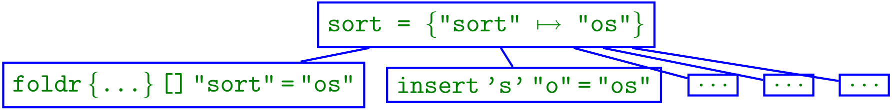
If parent equation is incorrect and all child equations are correct, then program equation of parent is faulty.

Intended Semantics with Consistency Properties

An intended semantics is a binary relation $\sqsupseteq$ on terms.

1. Reflexivity:
   $M \sqsupseteq M$
2. Transitivity:
   $M \sqsupseteq N \wedge N \sqsupseteq O \implies M \sqsupseteq O$
3. Closure:
   $M \sqsupseteq N \implies M\ O \sqsupseteq N\ O \ \wedge \ O\ M \sqsupseteq O\ N$
4. Least element:
   $M \sqsupseteq \{\}$
5. Application:
   $\{N_1 \mapsto M_1, \ldots, N_k \mapsto M_k\}\ N_i \sqsupseteq M_i$
6. Abstraction:
   $ON_1 \sqsupseteq M_1 \wedge \ldots \wedge ON_k \sqsupseteq M_k \implies O \sqsupseteq \{N_1 \mapsto M_1, \ldots, N_k \mapsto M_k\}$

Another Fragment of the Tree for Finite Maps



Program fragment

```
sort = foldr insert []
```

sub-proof 1

insert 's' "o" $\sqsupseteq$ "os" $\wedge \ldots$ {children}

$\implies$ {abstraction property}

insert $\sqsupseteq$ {'s' "o" $\mapsto$ "os", ...}

sub-proof 2

foldr {...} [] "sort" $\sqsupseteq$ "os" {child}

$\implies$ {abstraction property}

foldr {...} [] $\sqsupseteq$ {"sort" $\mapsto$ "os"}

sort

$\sqsupseteq$ {program equation of sort}

foldr insert []

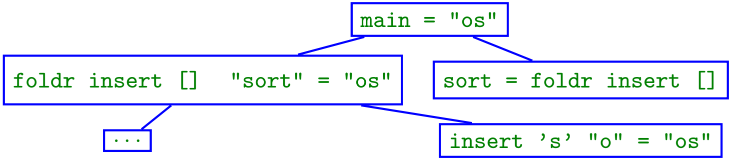
$\sqsupseteq$ {sub-proof 1}

foldr {'s' "o" $\mapsto$ "os", ...} []

$\sqsupseteq$ {sub-proof 2}

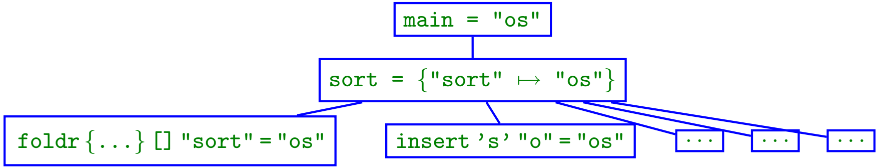
{"sort" $\mapsto$ "os"}

Structure of Trees for Different Function Representations



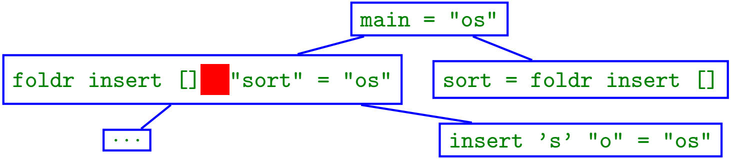
Program

```
main = sort "sort"
sort = foldr insert []
```



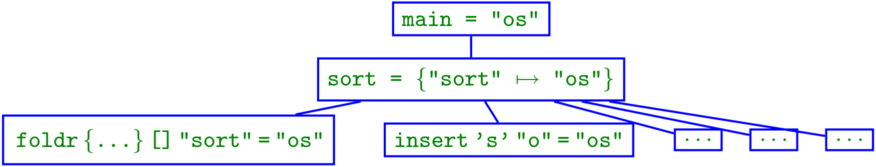
Structure of Trees for Different Function Representations

Applicative Terms: application appears in definition of parent



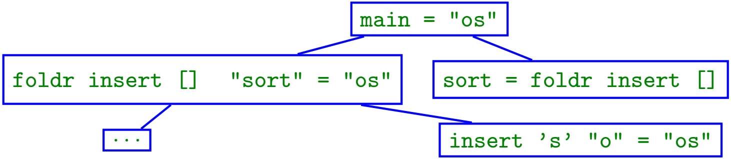
Program

```
main = sort "sort"
sort = foldr insert []
```



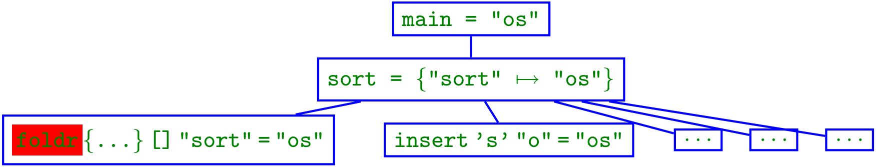
Structure of Trees for Different Function Representations

Applicative Terms: application appears in definition of parent



Program

```
main = sort  "sort"
sort = foldr insert []
```



Finite Map: function symbol appears in definition of parent

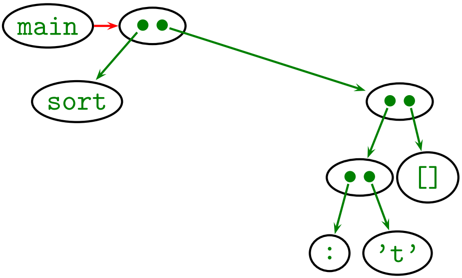
Basis of Trees: The Trace

main

Program fragment

```
main = sort "t"
sort = foldr insert []
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
insert x [] = [x]
```

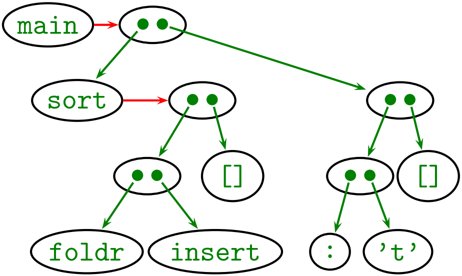
Basis of Trees: The Trace



Program fragment

```
main = sort "t"
sort = foldr insert []
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
insert x [] = [x]
```

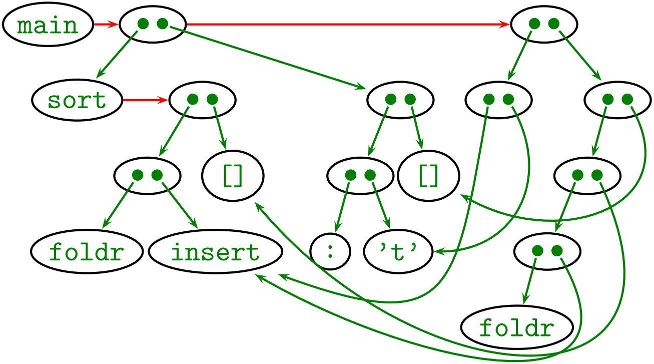
Basis of Trees: The Trace



Program fragment

```
main = sort "t"
sort = foldr insert []
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
insert x [] = [x]
```

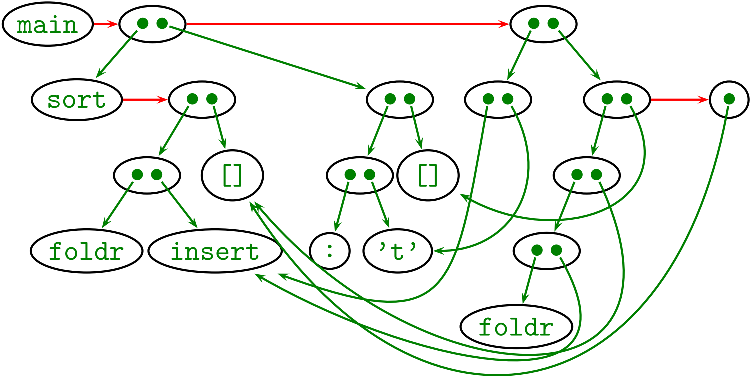
Basis of Trees: The Trace



Program fragment

```
main = sort "t"
sort = foldr insert []
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
insert x [] = [x]
```

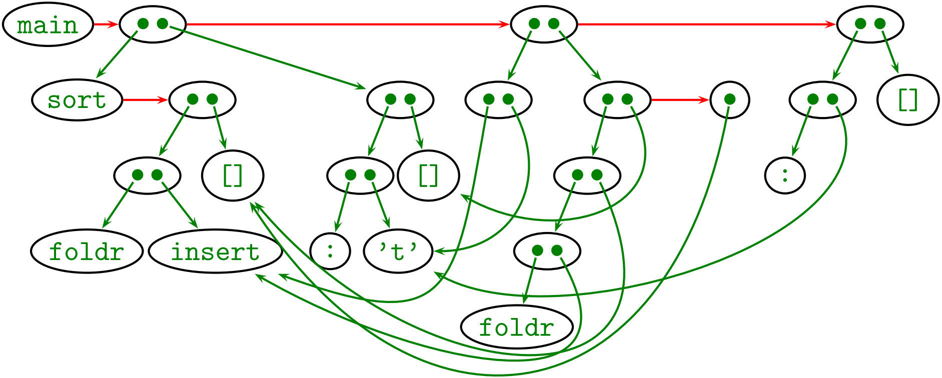
Basis of Trees: The Trace



Program fragment

```
main = sort "t"
sort = foldr insert []
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
insert x [] = [x]
```

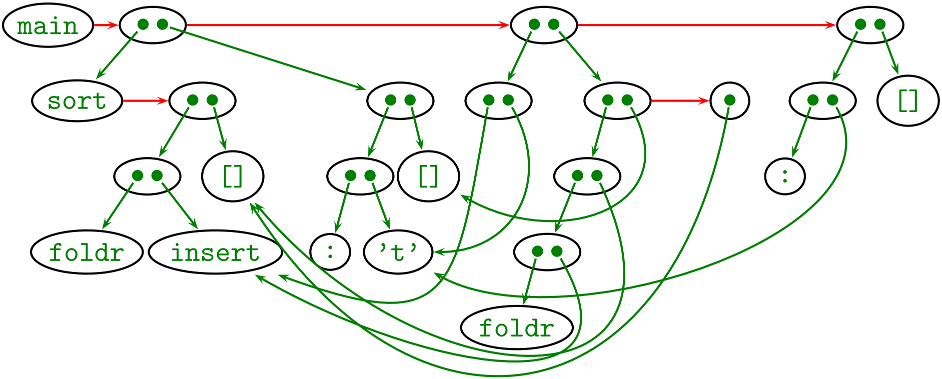
Basis of Trees: The Trace



Program fragment

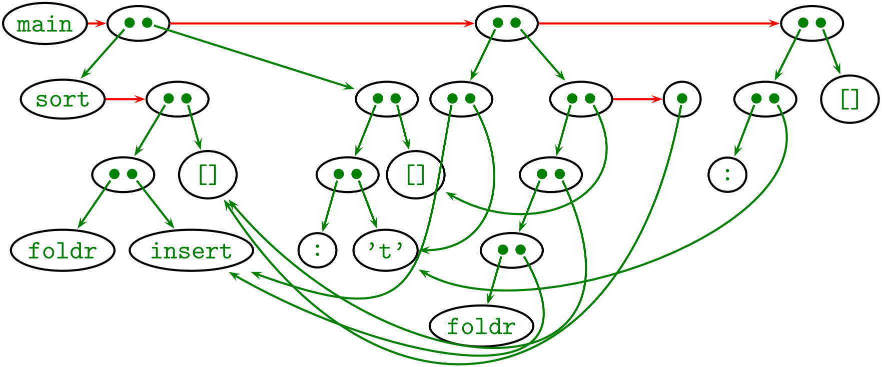
```
main = sort "t"
sort = foldr insert []
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
insert x [] = [x]
```

Basis of Trees: The Trace



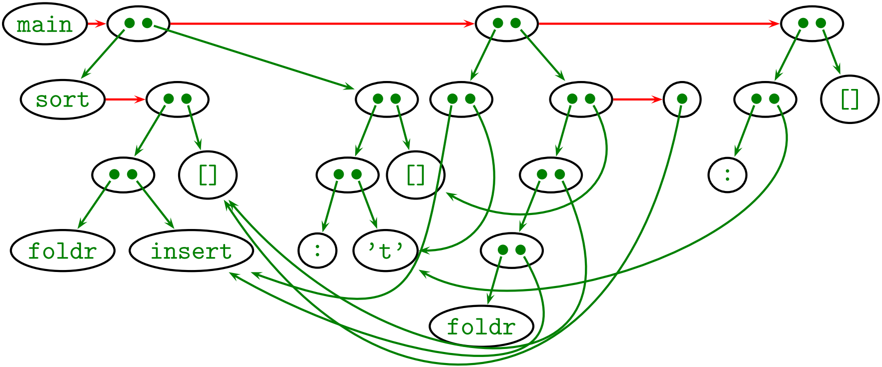
- New nodes for right-hand-side, connected via result pointer.
- Only add to graph, never remove.
- Sharing ensures compact representation.

From Trace to Computation Tree



- Each reduction edge gives rise to a tree node.
- Tree structure based on node parent:
  - applicative: parent of reduction node (application)
  - finite map: parent of function symbol (left-most)
- Most evaluated form of node: always follow reduction edges
  - finite map: show nodes representing functions differently

Finite Maps from the Trace

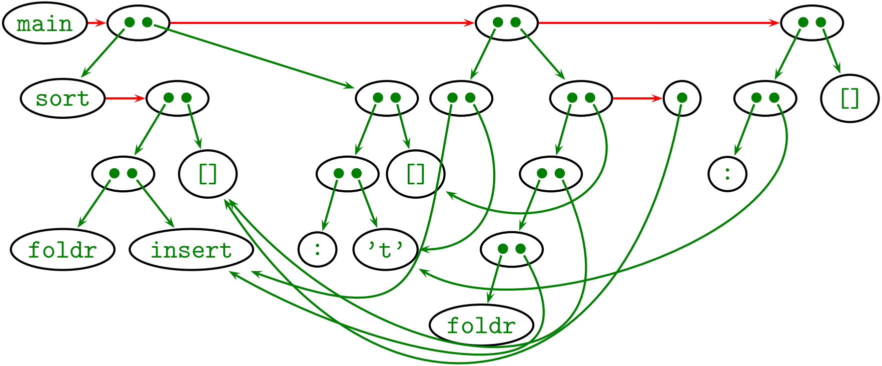


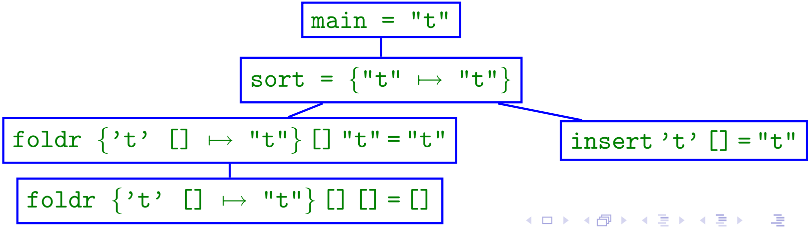
Finite map for a node: find all applications of node.

$$\text{fMap}_{\mathcal{G}}(\text{node of } \texttt{sort}) = \{\texttt{'t':[]} \mapsto \texttt{'t':[]}\}$$

$$\text{fMap}_{\mathcal{G}}(\text{node of } \texttt{insert}) = \{\texttt{'t'} \mapsto \{\texttt{[]} \mapsto \texttt{'t':[]}\}\} = \{\texttt{'t'}\ \ \texttt{[]} \mapsto \texttt{'t':[]}\}$$

From Trace to Tree for Finite Maps

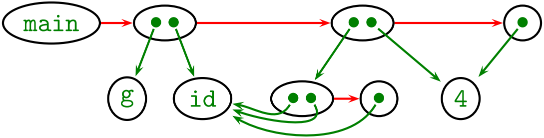




Well-Definedness of Finite Maps

Self-application

```
main = g id
id x = x
g h = (h h) 4
```

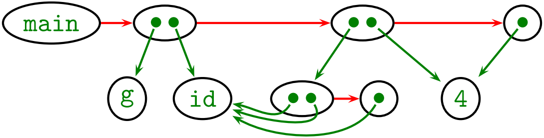


$\text{fMap}_{\mathcal{I}}(\text{node of } \texttt{id}) = \{\text{fMap}_{\mathcal{I}}(\text{node of } \texttt{id}) \mapsto \{4 \mapsto 4\}, 4 \mapsto 4\}$

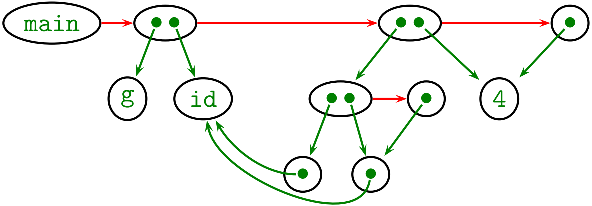
Well-Definedness of Finite Maps

Self-application

```
main = g id
id x = x
g h = (h h) 4
```

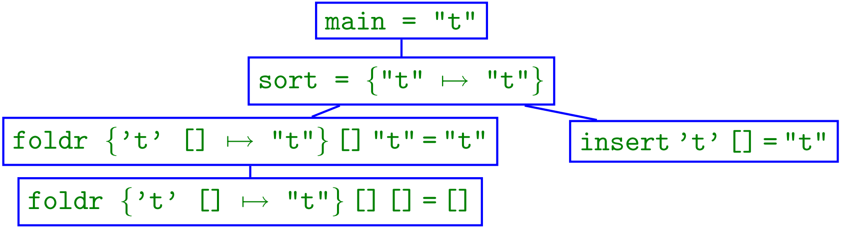


$\text{fMap}_{\mathcal{I}}(\text{node of } \texttt{id}) = \{\text{fMap}_{\mathcal{I}}(\text{node of } \texttt{id}) \mapsto \{4 \mapsto 4\}, 4 \mapsto 4\}$



$\text{fMap}_{\mathcal{J}}(\text{node of } \texttt{id}) = \{\{4 \mapsto 4\} \mapsto \{4 \mapsto 4\}, \ 4 \mapsto 4\ \}$

Conclusions



- A finite map is a useful alternative representation of a functional value.
- Trace provides framework for both applicative terms and finite maps.
  - formal definition
  - soundness proof
  - implementation