Morris Inorder (Non-Recursive) Re-Visited

IFMSIG Winter Meeting 1999-2000

Hugh Gibbons
Computer Science Dept.,
Trinity College Dublin

Knuth Challenge:

Non-recursive <u>inorder</u> of a binary tree,
without using an explicit stack or 'boolean flags'

Inorder Traversal (Recursive Version)

```
inorder(t : TREE[G]) : LIST[G] is
    do
        if  is_empty (t)  then
            Result := [ ] -- empty list
        else
            Result := inorder(t.left)
                        ++ [t.value] ++
                        inorder(t.right)
        end
    end -- Inorder
```

Notation

++ is the join operator on lists

Approach to Non-Recursive version

For non-empty t, we get,

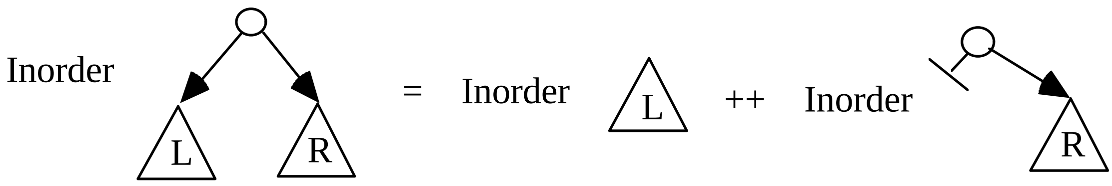
Inorder(t)

= Inorder(t.left) ++[t.value] ++Inorder(t.right)

= Inorder(t.left) ++
Inorder(build(t.value, void, t.right))

= Inorder(b1) ++ Inorder(b2)
where b1 = t.left
b2 = build(t.value, void, t.right)

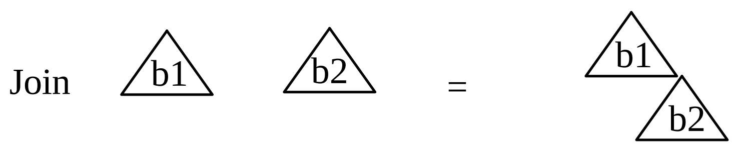
Diagram:



Join operator on Trees

Consider a function <u>join</u> s.t.

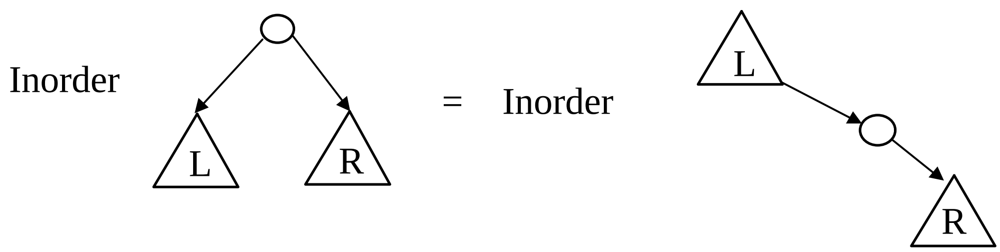
join(b1, b2) ‘joins’ b2 to the right most of b1



<u>Note</u>:

Trees with the operator, join, and the identity, empty, is a monoid.

Inorder with join



i.e.

```
inorder(t)
=  inorder(b1) ++inorder(b2)
=  inorder(join(b1,b2))
       where  b1 = t.left
              b2 = build(t.value, void, t.right)
```

```
join(b1,b2 : TREE[G]):TREE[G] is
    do
        if is_empty(b1) then
            result := b2
        else
            result := build(b1.value, b1.left, Join(b1.right, b2))
        end if
    end --join
```

Morris Inorder -- Abstract Code

```
Morris_Inorder(t0 : TREE[G]) : LIST[G] is
        t : TREE[G]
        s : LIST[G]
    do
        from
            t := t0
            s := [ ]
        until
            t = void
        loop
            if t.left = void then
                t := t.right
                s := s ++ [t.value]
            else
                t := Join(t.left, build(t.value, void, t.right))
            end
        end
        Result := s
    end -- Morris_Inorder
```

Binary Tree Structure/Class

class TREE [Values]

feature

root : N **is 1**

size : N

val : {1..size} f **Values** **-- partial on** N

left : N f N **-- total**

n Œ **2n**

right : N f N **-- total**

n Œ **2n+1**

first: N
-- inorder first

succ : N f N
-- inorder succ

left_sub : TREE

right_sub : TREE

etc.

end -- TREE

The Nodes in the tree are natural numbers, or viewed a binary numerals, an element of {0,1}*, the set of finite sequences from 0 and 1.

Bi-Graph instead of a Tree

In the more concrete implementation of inorder, the function, right, will be updated (and later reset) so that the Tree will become a bi-graph, hence loosing the properties of being a Tree.

Reachability

- $\mathbf{x \xrightarrow{R} y \;\; \equiv \;\; (\exists\, k \mid k \geq 0 \wedge right^k\, x = y)}$
  -- "right reaches"

<u>Note</u>: $\mathbf{x \xrightarrow{R} x}$

Similarly,

- $\mathbf{x \xrightarrow{L} y \;\; \equiv \;\; (\exists\, k \mid k \geq 0 \wedge left^k\, x = y)}$
  -- "left reaches"

Inorder first and Inorder Successor

- **y = right_most x**
  $\mathbf{\equiv \;\; x \xrightarrow{R} y \;\wedge\; right\ y \notin dom\ val}$

Similarly,

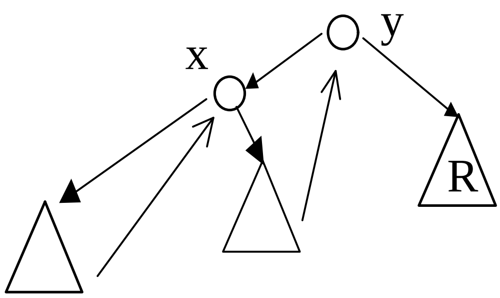
- **y = left_most x**
  $\mathbf{\equiv \;\; x \xrightarrow{L} y \;\wedge\; left\ y \notin dom\ val}$

- **first = left_most root**
- **y = succ x**

  $\equiv$ **(left y $\in$ dom val $\wedge$ x = right_most (left y))**

  $\vee$ **(right x $\in$ dom val $\wedge$ y = left_most (right x))**
- **x = pred y** $\equiv$ **y = succ x**

Morris Inorder



- **marked y**

$\equiv$ **right (pred y) = y**

A change is needed in the definition of pred as when y is marked a cycle is introduced.

If (marked y) then

x = pred' y $\equiv$ **right x = y**

i.e. when (left y) $\in$ dom val

x = pred' y

$\equiv$ (left y $\in$ dom val

$\wedge$ left y $\xrightarrow{R}$ x

$\wedge$ (right x $\notin$ dom val $\vee$ right x = y)

We define a function, mor (q, lt, rt, S, n)
such that

mor (t.root, t.left, t.right, [ ], t.size)
= (t.right, inorder t)

mor (q, lt, rt, S, n)
| S.size = n   f   (rt, S)
| left q ∉ dom val   f   mor (rt q, rt, S ⧺ [val q])
| marked q   f mor (rt q, rt † {p Œ 2p+1}, S ⧺ [val q])
|¬ marked q f   mor (lt q, rt † {p Œ q}, S)
where p = pred' q

Notation:
† is the override operator

Eiffel program

Using 'pointers', and using void for the 'undefined' links, we get the following Eiffel routine for inorder which is directly based on that of Joe Morris [Morris_79].

```
mor (t0:TREE[STRING]) is
    local
        rm,t : NODE[STRING]
    do
        from
            t := t0
        until
            t = void
        loop
            if t.left = void then
                print(t.value)
                t := t.right
            else
                from
                    rm := t.left
                until
                    rm.right=void or rm.right=t
                loop
                    rm := rm.right
                end
                { rm = right_most(leff t) }
                if  rm.right = void then
                    rm.right_set(t)
                    t := t.left
                    { marked t }
                else
                    print(t.value)
                    rm.Right_Set(void)
                    { ¬ marked t }
                    t := t.right
                end
            end
        end -- loop
    end -- mor
```

Termination

Let n = #nodes in tree,

m = # marked nodes

s = #S, output list.

First attempt:

? variant ?: 2n - (m + s)

but for call

| marked q f mor (rt p, rt † {p Œ 2p+1}, S ++ [val p])

p gets unmarked and so m decreases by 1 while s increases by 1 and so no overall decrease in variant.

Try

variant: 2(n-s) - m

In effect, 'processing a node' is counted double of 'marking a node'.

When program terminates, s = n and m = 0.

Note:

In the article [Morris_79], the following is suggested as a variant:

variant:

The number of nodes still to processed

+ the number of left edges

= (n - s) + #left_edges

Conclusion

The Functional Programming (FP) version of the non-recursive inorder program attempts to capture the essence of the imperative routine. Rather than verify the imperative routine directly, it is hoped to verify the FP one which then can be used in the verification of the imperative routine.

References:

- **[Morris_79] Morris, J.M.**
  "Traversing Binary Trees Simply and Cheaply"
  Informations Processing Letters
  Vol 9 Num 5 (December 1979)
- **Mateti, P. and Manghirmalani, R.**
  "Morris' Tree Traversal Algorithm Reconsidered"
  Science of Computer Programming
  Vol 11 (1988) p29-43 North Holland