Morris Inorder (Non-Recursive) Re-Visited
IFMSIG  Winter Meeting 1999-2000
Hugh Gibbons
Computer Science Dept.,
Trinity College Dublin

Knuth Challenge:
Non-recursive inorder of a binary tree,
without using an explicit stack or 'boolean flags'

Inorder Traversal (Recursive Version)

```plaintext
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,

\[
\text{Inorder}(t) = \text{Inorder}(t.\text{left}) + [t.\text{value}] + \text{Inorder}(t.\text{right}) \\
= \text{Inorder}(t.\text{left}) + \text{Inorder}(\text{build}(t.\text{value}, \text{void}, t.\text{right})) \\
= \text{Inorder}(b_1) + \text{Inorder}(b_2) \\
\text{where } b_1 = t.\text{left} \\
\text{and } b_2 = \text{build}(t.\text{value}, \text{void}, t.\text{right})
\]

Diagram:
**Join operator on Trees**

Consider a function `join` s.t.

\[ \text{join}(b_1, b_2) \text{ ‘joins’ } b_2 \text{ to the right most of } b_1 \]

\[ \text{Join } \begin{array}{c} b_1 \end{array} \begin{array}{c} b_2 \end{array} = \begin{array}{c} b_1 \end{array} \begin{array}{c} b_2 \end{array} \]

**Note:**

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

**Inorder with join**

\[ \text{Inorder } \begin{array}{c} \text{L} \\ \text{R} \end{array} = \text{Inorder } \begin{array}{c} \text{L} \\ \text{R} \end{array} \]

i.e.

\[
\begin{align*}
\text{inorder}(t) &= \text{inorder}(b_1) + \text{inorder}(b_2) \\
&= \text{inorder}(\text{join}(b_1,b_2)) \\
&\text{where } b_1 = t.\text{left} \\
&\quad b_2 = \text{build}(t.\text{value}, \text{void}, t.\text{right})
\end{align*}
\]
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 â 2n

   right : N f  N    -- total
      n â 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**

- \( x \rightarrow^R y \equiv (E k \mid k \geq 0 \land \text{right}^k x = y) \)
  -- "right reaches"

**Note:** \( x \rightarrow^R x \)

Similarly,

- \( x \rightarrow^L y \equiv (E k \mid k \geq 0 \land \text{left}^k x = y) \)
  -- "left reaches"

**Inorder first and Inorder Successor**

- \( y = \text{right}_\text{most} x \)
  \[ \equiv x \rightarrow^R y \land \text{right } y \notin \text{dom val} \]

Similarly,

- \( y = \text{left}_\text{most} x \)
  \[ \equiv x \rightarrow^L y \land \text{left } y \notin \text{dom val} \]
• first = left_most root

• y = succ x  
  ≡ (left y ∈ dom val ∧ x = right_most (left y)) ∨ (right x ∈ dom val ∧ y = left_most (right x))

• x = pred y  ≡  y = succ x

Morris Inorder

marked y  
≡ 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  ≡  right x = y

i.e. when (left y) ∈ dom val

\[
x = \text{pred}' y
\equiv (\text{left } y \in \text{dom val} \land \text{left } y \rightarrow_{R} x \land (\text{right } x \notin \text{dom val} \lor \text{right } x = y)
\]

We define a function, \(\text{mor } (q, \text{lt}, \text{rt}, S, n)\) such that
mor (t.root, t.left, t.right, [ ], t.size) = (t.right, inorder t)

\[
\begin{align*}
mor (q, lt, rt, S, n) \\
&| S.size = n \quad f \quad (rt, S) \\
&| left q \notin \text{dom val} \quad f \quad mor (rt \ q, \ rt, S ++ [\text{val } q]) \\
&| \text{marked } q \quad f \quad mor (rt \ q, rt \uparrow \{p \atop 2p+1\}, S ++ [\text{val } q]) \\
&\neg \text{marked } q \quad f \quad mor (lt \ q, rt \uparrow \{p \atop q\}, S) \\
&\text{where } p = \text{pred}' q
\end{align*}
\]

Notation:
\[\uparrow\] 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(left 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 -- mor
Termination
Let \( n \) = number of nodes in tree,
\( m \) = number of marked nodes
\( s \) = number of S, output list.

First attempt:
? variant?: \( 2n - (m + s) \)

but for call
\[
\text{marked q \& mor (rt p, rt \{p \# 2p+1\}, S + [val p])}
\]

\( p \) 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
= \( (n - s) + \#\text{left_edges} \)
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: