M-Structures *(Continued)*

plus

Introduction to the I/O Monad

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 24, 2006

Insert: Functional and Non Functional

Functional solution:

```
insertf  []     x = [x]
insertf (y:ys) x = if (x==y) then y:ys
                   else y:(insertf ys x)
```

M-structure solution:

```
insertm ys x =
 case ys of
    MNil        -> MCons x MNil
    MCons y ys’ ->
    if x == y then ys
       else let tl ys := insertm (tl&ys) x
            in  ys
```

In pattern matching m-fields have the “examine semantics”

Can we replace `tl&ys` by `ys’` ?

No

Out-of-order Insertion

Compare **ys2's** assuming **a** and **b** are not in **ys**.

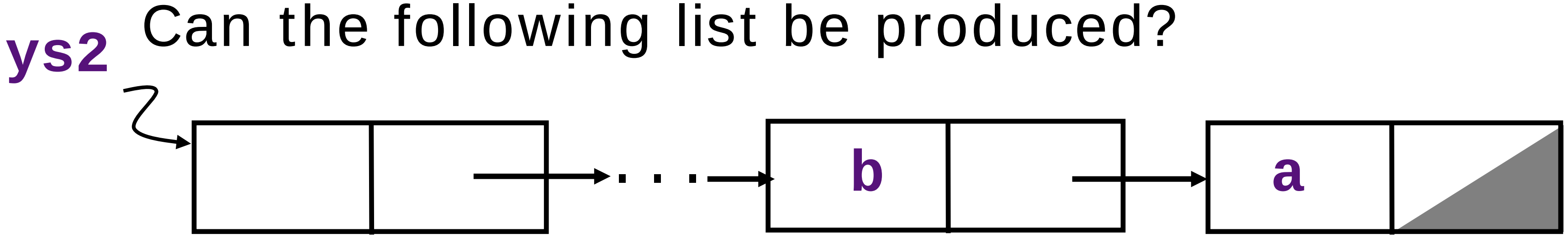
no

```
ys1 = insertf ys  a
ys2 = insertf ys1 b
```

```
ys1 = insertm ys  a
ys2 = insertm ys1 b
```

yes!

Can the following list be produced?



ys1 can be returned before the insertion of a is complete.

How can we stop the out of order insertion ?

insertm Reexamined

```
insertm ys x =
    case ys of
       MNil        -> MCons x MNil
       MCons y ys' ->
       if x == y then ys
          else let tl ys := insertm (tl&ys) x
                   in  ys
```

- In all cases to return the answer, ys has to be destructured and y has to be read
- In the `MNil` and `x==y` cases the answer is returned only after the insertion is complete
- However, in the `!(x==y)` case ys can be returned even before `insertm` begins

Avoiding out-of-order insertion

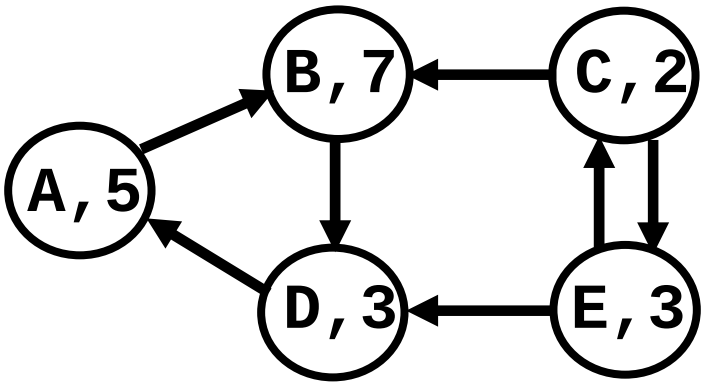
Do the take first!

```
insertm ys x =
     case ys of
        MNil          -> MCons x MNil
        MCons y ys' ->
        if x == y then ys
           else let   ( tlPtr = tl&ys     >>>
                        listToBeReturned = ys )
                        tl ys := insertm ~~(tl&ys)~~ x
                                         tlPtr

                in  ~~ys~~
                    listToBeReturned
```

Notice `(tl&ys)` can't be read again before `(tl ys)` is set

Graph Traversal

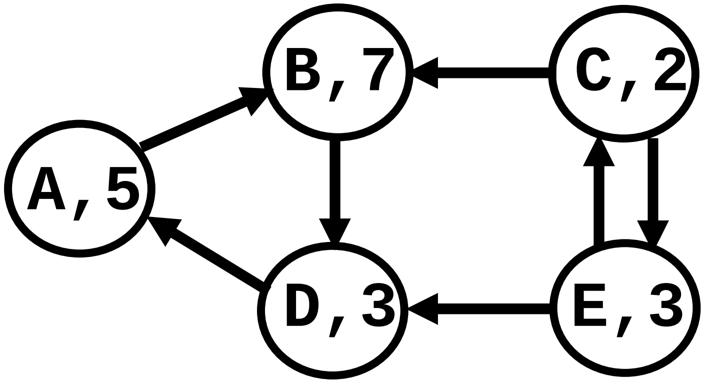



```
data GNode =
      GNode {id   :: Nodeid,
             val  :: Int,
             nbrs:: [GNode] }
a = GNode “A” 5 [b]
b = GNode “B” 7 [d]
c = GNode “C” 2 [b]
d = GNode “D” 3 [a]
e = GNode “E” 3 [c,d]
```

Write function **rsum** to sum the nodes reachable from a given node.

rsum a ==> $\boxed{15}$ **?**

Graph Traversal: First Attempt




```
data GNode =
      GNode {id  :: Nodeid,
             val :: Int,
             nbrs:: [GNode] }
```

```
rsum (GNode x i nbs) =
                  i + sum (map rsum nbs)
```

Wrong!

A node can get counted more than once and in case of a cycle, infinite number of times

Mutable Markings

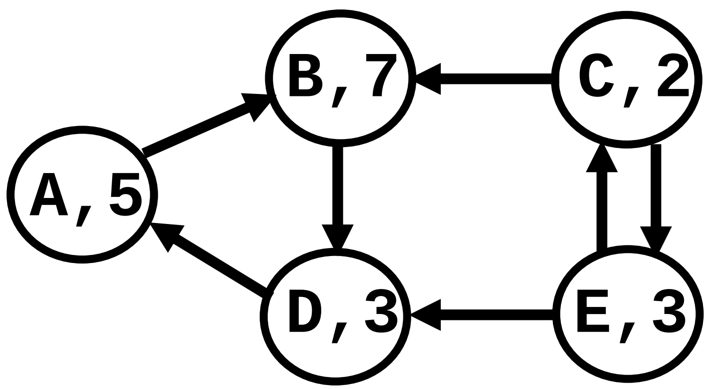
Keep an updateable boolean flag to record if a node has been visited. Initially the flag is set to false in all nodes.

```
data GNode = GNode {id::Nodeid, val::Int,
                    nbrs::[GNode], flag::&Bool}
```

A procedure to return the current flag value of a node and to simultaneously set it to true

```
marked node = let m = flag & node >>>
                      flag node := True
              in
                  m
```

Graph Traversal: Mutable Markings




```
data GNode =
      GNode {id  :: Nodeid,
             val :: Int,
             nbrs:: [GNode]
             flag::&Bool   }
```

```
rsum node =
      if marked node then 0
      else
            (val node)
          + sum (map rsum (nbrs node))
```

Problem: Parallel execution

(rsum a)+(rsum b) ?

Book-Keeping Information

```
data GNode = GNode {id::Nodeid, val::Int,
                    nbrs::[GNode], flag::&Bool}
```

The graph should not be mutated!

Keep the visited flags in a separate data structure - *a notebook* with the following functions

```
mkNotebook :: () -> Notebook
member     :: Notebook -> Nodeid -> Bool
```

Insertion: Immutable (functional) notebook

```
insert :: Notebook -> Nodeid -> Notebook
```

Insertion in a Mutable notebook causes a side-effect

```
insert :: Notebook -> Nodeid -> ()
```

Graph Traversal: *Immutable Notebook*

Thread the notebook and the current sum through the reachable nodes of the graph in any order

```
data GNode =
 GNode {id::Nodeid, val::Int, nbrs::[GNode]}

rsum node =
 let nb = mkNotebook ()         -- a new notebook
     (s,_) = thread (0, nb) node
     thread (s,nb) (GNode x i nbs) =
        if member nb x then (s,nb)
        else let nb' = insert nb x
                 s' = s + i
             in fold thread (s',nb') nbs  ?
 in  s
```

Graph Traversal: *Mutable Notebook*

```
rsum node =
  let nb = mkNotebook ()          -- a new notebook

      rsum' (GNode x i nbs) =
         if (member nb x) then 0
          else let
                   insert nb x >>>
                   s = i + sum (map rsum' nbs)
               in s
   in rsum' node
```

- *No threading*
- *No copying*

but wrong !!!

After we check for membership but before we do the insertion, some other insertion can get in.

Mutable Notebooks: *revisited*

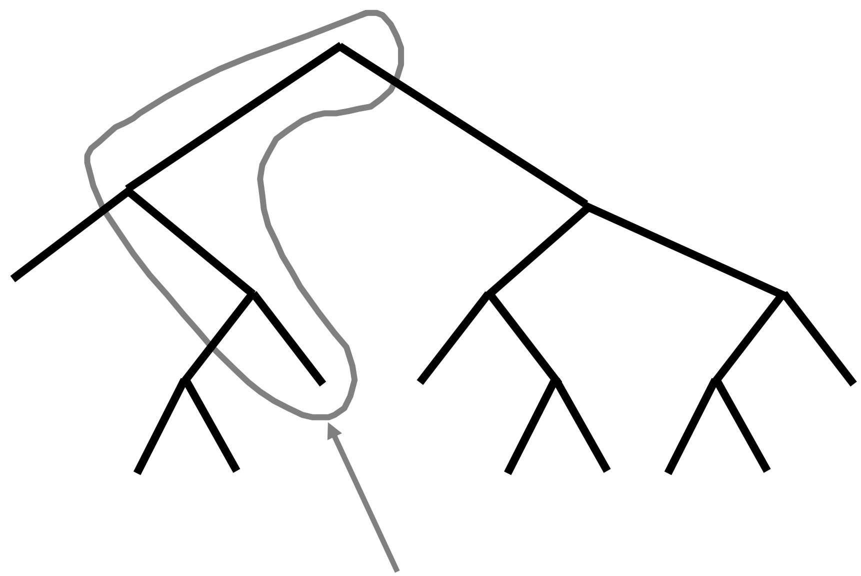
The test for membership and subsequent insertion have to be done atomically to avoid races.

```
isMemberInsertm :: Notebook -> Nodeid -> Bool
rsum node =
  let nb = mkNotebook ()                -- a new notebook
      rsum' (GNode x i nbs) =
          if (isMemberInsert nb x)
          then 0
          else i + sum (map rsum' nbs)
  in
      rsum' node
```

Notebook Representation: Tree

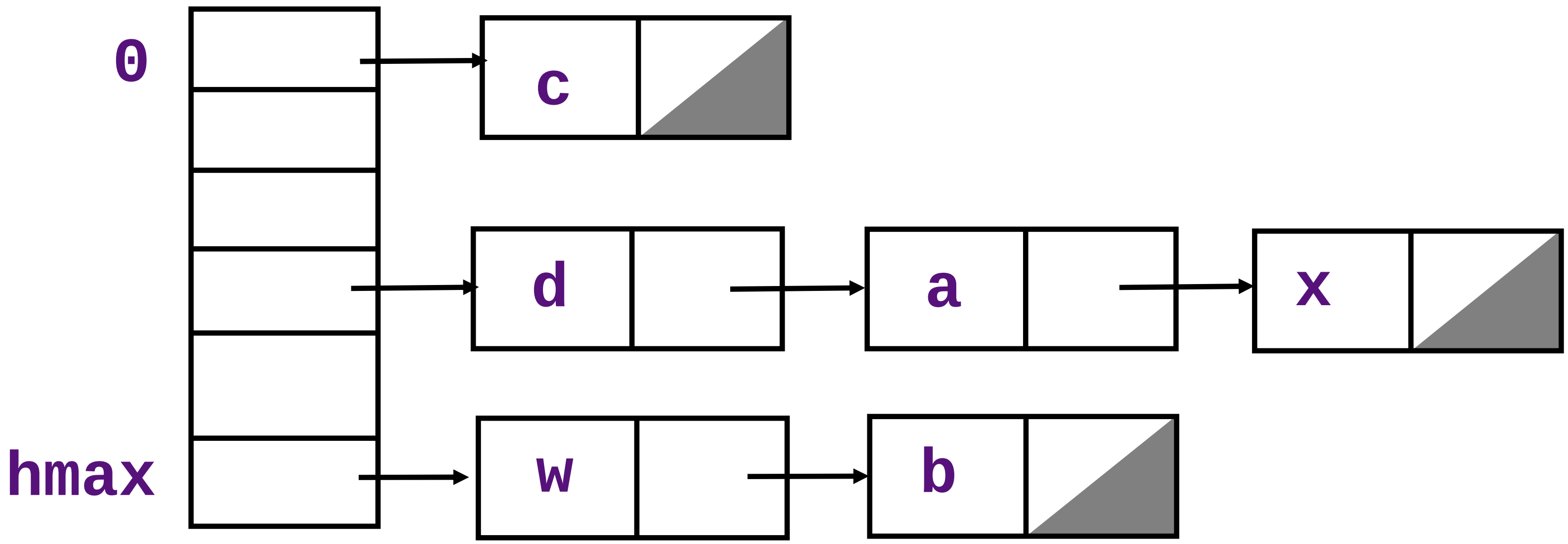
We can maintain the notebook as a (balanced) binary tree

```
data Tree = TEmpty | TNode Int Tree Tree
```



Nodes above the point of insertion have to be copied in a functional solution

Notebook Representation: Hash Table




```
data  MList t = MNil
              | MCons {hd::t, tl::&(MList t)}

mkNotebook () =
   mArray (0,hmax) [(j,MNil) | j <- [0..hmax]]
```

isMemberInsert

```
isMemberInsert nb x =
  let i = hash x
      ys = nb!&i
      (flag, ys’) = insertm' ys x
       nb!i := ys'
  in  flag
```

`insertm'` is the same as **`insertm`** except that it also returns a flag to indicate if a match was found

Membership and Insertion

insertm' is the same as **insertm** except that it also returns a flag that indicates if a match was found

```
insertm' ys x =
  case ys of
     MNil        -> (False,(MCons x MNil))
     MCons y ys’ ->
        if x == y then (True,ys)
        else let( tlPtr = tl&ys >>> ysTBR = ys)
                (flag,ys’’) = insertm’ tlPtr x
                 tl ys := ys’’
             in
                (flag, ysTBR)
```

Summary

- M-structures were used heavily to program
  - Monsoon dataflow machine run-time system, including I/O
  - Id compiler in Id
  - Non-deterministic numerical algorithms

- Programming with M-structures proved to be full of perils!
  - Encapsulate M-structures in functional data structures, if possible

Using Monads for Input and Output

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 24, 2006

(based on a lecture by Jan-Willem Maessen)

Functional Languages and I/O

```
z := f(x) + g(y);
```

In a functional language f and g can be evaluated in any order

This is not so if f or g had side-effects, e.g. print statements

Is I/O incompatible with FL?

What other languages do

- Execute programs in a fixed order (top-to-bottom, left-to-right):

```
(define (hello)
    (princ “Hello “)
    (princ “World “))
```

> Weakens equational reasoning:
>
> ```
> (let ((a (f x)))                (let ((b (g y)))
>  (let ((b (g y)))     vs         (let ((a (f x)))
>  (+ a b)))                       (+ a b)))
> ```

- Provide explicit constructs for sequencing in FL

```
(princ “Hello “) >>> (princ “World “)
```

Using Barriers

```
echo :: () -> ()
echo () =
     let c = getChar()
     in if c=='\n' then ()
        else let putChar c
                 >>>
                 echo ()
             in ()
```

Barriers can destroy modularity?

```
myProgram () =
   let input = acceptAllTheInput()
       >>>
       consumeAndOutput input
   in ()
```

Barriers don't work well when there is complex interleaving of producer and consumer

Another solution: Magic return value

getChar returns a magic value in addition to the character indicating that further I/O is safe.

```
echo :: World -> World
echo world0 =
    let (c, world1) = getChar world0
    in if c=='\n' then ()
         else let world2 = putChar c world1
                  world3 = echo world2
              in  world3
```

Used in Id and Clean

(Single)-threading is users responsibility

I/O and Computation

```
main :: World -> World
```

OS provides the initial state of the world and supports I/O actions on the world

Computation affects the world through these I/O actions

Is these another possible way of dealing with the world?

Example: Role of a Program Driver

Suppose by convention

```
main :: [string]
main = ["Hello", "world!"]
```

or

```
main = let a = "Hello"
           b = "World!"
       in [a,b]
```

Program is a *specification* of intended effect to be performed by the program driver

The driver, a primitive one indeed, takes a string and treats it as a sequence of commands to print.

Monadic I/O in Haskell

Treats a sequence of I/O commands as a specification to interact with the outside world.

The program produces an *actionspec*, which the program driver turns into real I/O actions.

A program that produces an *actionspec* remains purely functional!

Programs to produce actionspecs

```
main :: IO ()
putChar :: Char -> IO ()
getChar :: IO Char

main = putChar 'a'
```

is an actionspec that says that character “a” is to be output to some standard output device

How can we sequence actionspecs?

Sequencing

We need a way to compose actionspecs:

```
(>>)   :: IO () -> IO () -> IO ()
```

Example:

```
putChar ‘H’ >> putChar ‘i’ >>
putChar ‘!’                       :: IO ()

putString :: String -> IO ()
putString “”       = done
putString (c:cs) =
         putChar c >> putString cs
```

Monads: Composing Actionspecs

We need some way to get at the results of `getChar`

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

We read the "bind" operator as follows:

x_1 `>>= \a ->` x_2

- Perform the action represented by x_1, producing a value of type "a"
- Apply function `\a ->` x_2 to that value, producing a new actionspec x_2 `:: IO b`
- Perform the action represented by x_2, producing a value of type b

Example: `getChar >>= putChar`

the same as `getChar >>= \c -> putChar c`

An Example

```
main =
   let
      islc c = putChar (if ('a'<=c)&&(c<='z')
                        then 'y'
                        else 'n')
   in
      getChar >>= islc
```

Turning expressions into actions

```
return :: a -> IO a

getLine :: IO String

getLine = getChar >>= \c ->
          if (c == '\n') then
               return ""
          else getLine >>= \s ->
               return (c:s)
```

where **'\n'** represents the newline character

Monadic I/O

`IO a`: computation which does some I/O,
then produces a value of type **`a`**.

```
(>>)   :: IO a -> IO b -> IO b
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

Primitive actionspecs:

```
getChar   :: IO Char
putChar   :: Char -> IO ()
openFile, hClose, ...
```

Monadic I/O is a clever, type-safe idea which has become a rage in the FL community.