M-Structures *Continued*

Arvind
Laboratory for Computer Science
M.I.T.

Lecture 13

http://www.csg.lcs.mit.edu/6.827

Mutable Lists

Any field in an algebraic type can be specified as an M-structure field by marking it with an “&”

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

M-structure slot

Allocate

```
x = MCons {hd = 5}
```

Take

```
tl & x
```

Put

```
tl x := v
```

No side-effects while pattern matching

M-Cell: Dynamic Behavior

- Let allocated M-cells be represented by objects o_1, o_2, …
- Let the states of an M-cell be represented as:

empty(o) | full(o,v) | error(o)

- When a cell is allocated it is assigned a new object descriptor o and is empty, i.e., empty(o)

- Reading an M-cell
  (x=mFetch(o) ; full(o,v)) ⇒ (x=v ; empty(o))

- Storing into an M-cell
  (mStore(o,v) ; empty(o)) ⇒ full(o,v)
  (mStore(o,v) ; full(o,v')) ⇒?(error(o); full(o,v'))

Barriers

- Barriers are needed to control to the execution of some operations
- A barrier discharges when all the bindings in its pre-region *terminate,* i.e., all expressions become *values.*

```
{ ( y = 1+7
    >>>
    z = 3 )    ⇒
in
    z }
```

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’`?

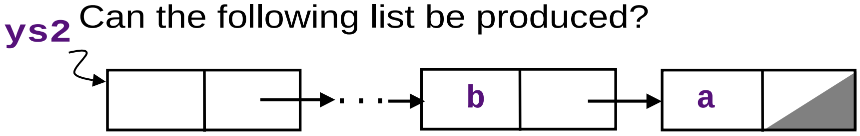
Out-of-order Insertion

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

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

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

Can the following list be produced?



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

Avoiding out-of-order insertion

```
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
```

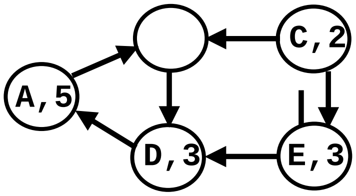
Notice **`(tl&ys)`** can’t be read again before **`(tl ys)`** is set

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
                (flag,ys’’) = (insertm’ (tl&ys) x)
                 tl ys := ys’’
             in
                (flag, ys)
```

Graph Traversal

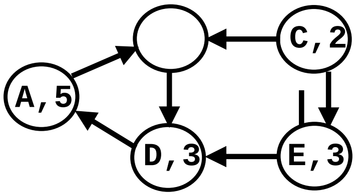


```
data GNode =
      GNode {id  :: Nodeid,
             val :: Int,
             nbrs:: [GNode] }
a = GNode “A” 5 [b]
b = GNode “B” 7 [d]
c = GNode “C” 2 [b]
d = GNode “D” 3 [a]
e = GNode “E” 3 [c,d]
```

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

`rsum a ==>` ?

Graph Traversal: First Attempt



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

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

Mutable Markings

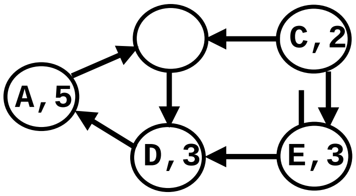
Keep an updateable boolean flag to record if a node has been visited. Initially the flag is set to false in all nodes.

```
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))
```

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
```

Immutable (functional) notebook

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

Mutable notebook: insertion 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                            ?
 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

Mutable Notebooks: *revisited*

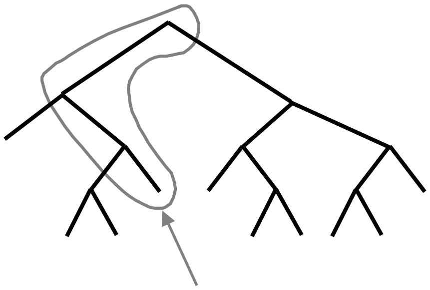
The test for membership and subsequent insertion has to be done atomically to avoid races.

```
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

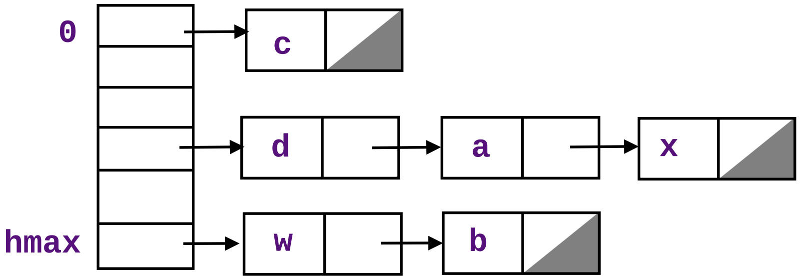
```
data Tree = TEmpty | TNode Int Tree Tree
```



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

---

Summary

- M-structures have been used heavily to program
  - Monsoon run-time system, including I/O
  - Id compiler in Id
  - Non-deterministic numerical algorithms
- Programming with M-structures is full of perils!
  - Encapsulate M-structures in functional data structures, if possible

The λ_S Calculus

- An extention of λ_{let} with side-effects and barriers

λ_S Syntax

E ::= x | λx.E | E E | { S *in* E }

| Cond (E, E, E)

| $PF_k(E_1,\ldots,E_k)$

| CN_0 | $CN_k(E_1,\ldots,E_k)$ | $\underline{CN_k}(x_1,\ldots,x_k)$

| allocate()

| o_i object descriptors

Not in initial expressions

PF_1 ::= negate | not | ... | Prj_1| Prj_2 | ... | ifetch | mfetch

...

CN_0 ::= Number | Boolean | ()

S ::= ε | x = E | S; S

| S >>> S

| sstore(E,E)

| allocator | empty(o_i) | full(o_i,E) | error(o_i)

Values and Heap Terms

Values
V ::= λx.E | CN_0 | $\underline{CN_k}(x_1,\ldots,x_k)$ | o_i

Simple expressions
SE ::= x | V

Heap Terms
H ::= x = V | H; H | allocator
| empty(o_i) | full(o_i,V)

Terminal Expressions
E^T ::= V | *let* H *in* SE

Side-effect Rules

- Allocation rule
  (allocator; x=allocate()) ⇒
  (allocator; x = o; empty(o))
  where o is a new object descriptor
- Fetch and Take rules
  (x=iFetch(o) ; full(o,v)) ⇒ (x=v ; full(o,v))
  (x=mFetch(o) ; full(o,v)) ⇒ (x=v ; empty(o))
- Store rules
  (mStore(o,v) ; empty(o)) ⇒ full(o,v)
  (mStore(o,v) ; full(o,v’)) ⇒?(error(o); full(o,v’))
- Lifting rules
  sstore({ S *in* e }, e_2) ⇒ (S ; sstore(e,e_2))
  sstore(e_1, { S *in* e }) ⇒ (??S ; sstore(e_1,e))

Barrier Rules

- Barrier discharge

$$(\varepsilon >>> S) \Rightarrow S$$

- Barrier equivalence

$$((H ; S_1) >>> S_2) \equiv (H ; (S_1 >>> S_2))$$

$$(H >>> S) \Rightarrow ?? (H ; S) \quad \textit{(derivable)}$$

Multiple-Store Error

A program with “exposed” store error is suppose to blow up!

Program --> T

The Top represents a contradiction

Exposed error: A error(o) cell that is not below a barrier, inside an arm of a conditional or inside a lambda abstraction