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)}
```

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 $o_1, o_2, \ldots$
- Let the states of an M-cell be represented as:
  
  \[
  \text{empty}(o) \mid \text{full}(o,v) \mid \text{error}(o)
  \]

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

- Reading an M-cell
  
  \[
  (x=\text{mFetch}(o) \ ; \ \text{full}(o,v)) \ \Rightarrow \ (x=v \ ; \ \text{empty}(o))
  \]

- Storing into an M-cell
  
  \[
  (\text{mStore}(o,v) \ ; \ \text{empty}(o)) \ \Rightarrow \ \text{full}(o,v)
  \]
  \[
  (\text{mStore}(o,v) \ ; \ \text{full}(o,v')) \ \Rightarrow \ ?(\text{error}(o); \ \text{full}(o,v'))
  \]

Barriers

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

\[
\{ \ (y = 1+7 \\
\quad \gggg \ z = 3 \ ) \ \Rightarrow \ \\
\quad \quad \ \text{in} \\
\quad \quad \quad z \ 
\} \]
Insert: Functional and Non Functional

Functional solution:
\[
\text{insertf} \ [ \ x = [x] \\
\text{insertf} \ (y:ys) \ x = \text{if} \ (x==y) \ \text{then} \ y:ys \\
\text{else} \ y:(\text{insertf} \ ys \ x)
\]

M-structure solution:
\[
\text{insertm} \ ys \ x = \\
\text{case} \ ys \ of \\
\text{MNil} \quad \rightarrow \ MCons \ x \ \text{MNil} \\
\text{MCons} \ y \ ys' \ \rightarrow \\
\text{if} \ x == y \ \text{then} \ ys \\
\text{else let} \ tl \ ys := \text{insertm} \ (\text{tl &} \ ys) \ x \\
\text{in} \ ys
\]

Can we replace \( \text{tl &} \ ys \) by \( ys' \)?

Out-of-order Insertion

Compare \( ys2's \) assuming \( a \) and \( b \) are not in \( ys \).
\[
\begin{align*}
ys1 &= \text{insertf} \ ys \ a \\
ys2 &= \text{insertf} \ ys1 \ b
\end{align*}
\]

\( ys2 \) 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**

```haskell
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.

```haskell
rsum a ==> ?```

---

**Graph Traversal: First Attempt**

```haskell
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.

\[
\text{data GNode} = \text{GNode} \{ \text{id :: Nodeid, val :: Int, nprs :: [GNode], flag :: &Bool} \}
\]

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

\[
\text{marked node} = \text{let } m = \text{flag} \& \text{node} >>> \text{flag node} := \text{True in m}
\]

Graph Traversal: Mutable Markings

\[
\text{rsum node} = \begin{cases} 
0 & \text{if marked node} \\
(\text{val node}) + \text{sum (map rsum (nprs node))} & \text{else}
\end{cases}
\]

\[
\text{data GNode} = \text{GNode} \{ \text{id :: Nodeid, val :: Int, nprs :: [GNode], flag :: &Bool} \}
\]

\[
\text{rsum node} = \begin{cases} 
0 & \text{if marked node} \\
(\text{val node}) + \text{sum (map rsum (nprs node))} & \text{else}
\end{cases}
\]
Book-Keeping Information

```haskell
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

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

Immutable (functional) notebook

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

Mutable notebook: insertion causes a side-effect

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

Graph Traversal: *Immutable Notebook*

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

```haskell
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 s’
in s
```

http://www.csg.lcs.mit.edu/6.827
Graph Traversal: *Mutable Notebook*

```haskell
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.

```haskell
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

```haskell
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**

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

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

[Diagram showing a hash table with keys 0 to hmax and values c, d, w, a, b, x.]
**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 $\lambda_S$ Calculus

- An extension of $\lambda_{let}$ with side-effects and barriers

$\lambda_S$ Syntax

$$E ::= x \mid \lambda x.E \mid E \ E \mid \{ S \ in \ E \} \mid \text{Cond} \ (E, E, E) \mid \text{PF}_k(E_1, \ldots, E_k) \mid \text{CN}_0 \mid \text{CN}_k(E_1, \ldots, E_k) \mid \text{CN}_k(x_1, \ldots, x_k) \mid \text{allocate}() \mid o_i \text{ object descriptors}$$

$$\text{PF}_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots \mid \text{ifetch} \mid \text{mfetch}$$

$$\text{CN}_0 ::= \text{Number} \mid \text{Boolean} \mid ()$$

$$S ::= \varepsilon \mid x = E \mid S; S \mid S \bowtie \bowtie S \mid \text{sstore}(E, E) \mid \text{allocator} \mid \text{empty}(o_i) \mid \text{full}(o_i, E) \mid \text{error}(o_i)$$

Not in initial expressions
Values and Heap Terms

Values
\[ V ::= \lambda x.E \mid CN_0 \mid CN_k(x_1,\ldots,x_k) \mid o_i \]

Simple expressions
\[ SE ::= x \mid V \]

Heap Terms
\[ H ::= x = V \mid H; H \mid allocator \]
\[ \mid empty(o_i) \mid full(o_i, V) \]

Terminal Expressions
\[ E^T ::= V \mid let H in SE \]

Side-effect Rules

• Allocation rule
  \[ (allocator; x=allocate()) \Rightarrow (allocator; x = o; empty(o)) \]
  where o is a new object descriptor

• Fetch and Take rules
  \[ (x=iFetch(o) ; full(o,v)) \Rightarrow (x=v ; full(o,v)) \]
  \[ (x=mFetch(o) ; full(o,v)) \Rightarrow (x=v ; empty(o)) \]

• Store rules
  \[ (mStore(o,v) ; empty(o)) \Rightarrow full(o,v) \]
  \[ (mStore(o,v) ; full(o,v')) \Rightarrow \epsilon(error(o); full(o,v')) \]

• Lifting rules
  \[ sstore(\{ S in e \}, e_2) \Rightarrow (S ; sstore(\epsilon, e_2)) \]
  \[ sstore(\epsilon, \{ S in e \}) \Rightarrow \epsilon(S ; sstore(\epsilon, e)) \]
Barrier Rules

• Barrier discharge

$(\epsilon \ggg S) \Rightarrow S$

• Barrier equivalence

$((H ; S_1) \ggg S_2) = (H ; (S_1 \ggg S_2))$

$(H \ggg S) \Rightarrow (H ; S) \text{ (derivable)}$

Multiple-Store Error

A program with “exposed” store error is supposed 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