



M-Structures: Programming with State and Nondeterminism

Arvind
Laboratory for Computer Science
M.I.T.

October 23, 2002

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

Limitations of Functional Programming

- Forces an *obscure coding style* - threading the “state” - for some problems
- Requires too much *storage*
- Cannot express the *parallelism* in some algorithms
- Cannot express *non-deterministic algorithms*
 - histograms
 - graph traversals
- Cannot express *non-determinism inherent in*
 - access to shared resources
 - storage allocator



Language extensions

- I-structures: “write once” variables
 - Multiple writes cause an “inconsistency” and blowup the program. A flavor of logic variables
 - Benign side-effects but equational reasoning is weakened
- M-structures: “synchronized reads and writes”.
 - each read “empties” the variable and a write to a “full” variable causes a program blowup
 - simultaneously requires the notion of a “barrier” to control the order of evaluation of some expressions
 - equational reasoning is weakened dramatically
- Monads: a new way of manipulating programs (has become very popular in the last decade)
 - preserves equational reasoning
 - not obvious how to use it for expressing parallelism

Nov 4, 6

October 23, 2002

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

I-Cell: The Simplest I-Structure

```
data ICell a = ICell {contents :: . a}
```

Constructor

```
ICell :: a -> ICell a
```

I-Structure field

```
ICell e          or          ICell {contents = e}
```

or create an empty cell and fill it (a “side-effect”)

```
ic = ICell {}
contents ic := e
```

Selector (iFetch)

```
contents ic      or
case ic of
  ICell x -> ... x ...
```

October 23, 2002

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

I-Cell: Dynamic Behavior

- Let allocated I-cells be represented by objects $o_1, o_2, \dots$
- Let the states of an I-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 I-cell
 $(x = \text{iFetch}(o) ; \text{full}(o,v)) \Rightarrow (x = v ; \text{full}(o,v))$
- Storing into an I-cell
 $(\text{iStore}(o,v) ; \text{empty}(o)) \Rightarrow \text{full}(o,v)$
 $(\text{iStore}(o,v) ; \text{full}(o,v')) \Rightarrow \text{error}(o) ; \text{full}(o,v')$

October 23, 2002

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

Multiple-Store Error

Multiple assignments to an I-cell cause a multiple store error

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

Program $\rightarrow T$

The Top represents a contradiction

October 23, 2002

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

M-Cell: The Simplest M-Structure

```
data MCell a = MCell {contents :: & a}
```

Constructor

```
MCell :: a -> MCell a
```

M-Structure field

```
MCell e          or          MCell {contents = e}
```

or create an empty cell and fill it

```
mc = MCell {}
```

```
contents mc := e
```

overloaded notation

Selector (mFetch)

```
contents & mc
```

pattern matching ?

October 23, 2002

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



M-Cell: Dynamic Behavior

- Let allocated M-cells be represented by objects $o_1, o_2, \dots$
- 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

$(\text{mStore}(o,v) ; \text{empty}(o)) \Rightarrow \text{full}(o,v)$
 $(\text{mStore}(o,v) ; \text{full}(o,v')) \Rightarrow \text{error}(o) ; \text{full}(o,v')$

October 23, 2002

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



The Need of Barriers

Suppose we want to replace the contents of M-Cell **mc** by zero.

We need to empty it first to avoid a double store error.

First attempt:

```
let old = content & mc
  content mc := 0
in ...
```

Correct ?

Second attempt:

October 23, 2002

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



M-Cell: Imperative Reads and Writes

Examine: like a read operation

```
contents mc ≡ let v = contents & mc
               contents mc := v
               in
               v
```

Replace: like an update operation

```
contents & mc := e ≡
  v = e
  ( _ = v >>>
    _ = contents & mc >>>
    contents mc := v )
```

M-structures with barriers have the full expressive power of imperative languages *but the language is not sequential!*

October 23, 2002

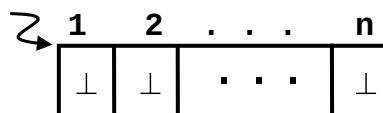
<http://www.csg.lcs.mit.edu/6.827>



M-Arrays

- *Allocate*

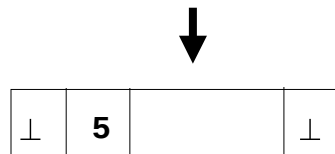
$x = \text{mArray } (1, n) []$



- *Put*

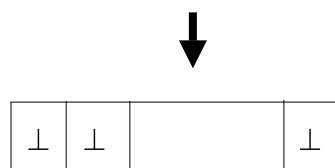
$x!2 := 5$

A put operation on a full slot is an error



- *Take*

$x!&2$



October 23, 2002

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


Three Examples

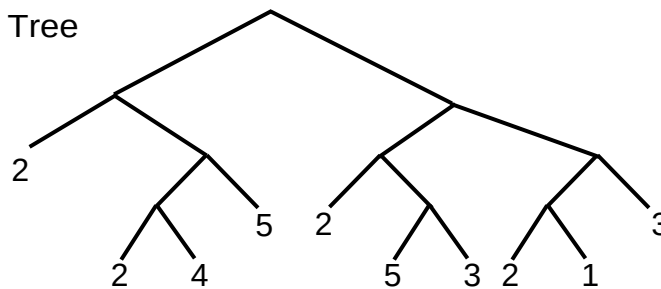
- Histograms
- Inserting an element in a list
- Graph traversal

October 23, 2002

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


Histogram of Elements in a Tree

Tree



Histogram

1	2	3	4	5
1				2

October 23, 2002

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


Histogram: A Functional Solution

Thread the histogram array

```
data Tree = Leaf Int | Node Tree Tree
traverse :: Tree -> (ArrayI Int) -> (ArrayI Int)
traverse (Leaf i) hist = incr hist i
traverse (Node ltree rtree) hist =
```

```
incr hist j =
  let inc i = if i == j then (hist!i)+1
              else hist!i
  in mkArray (bounds hist) inc
```

```
mkHistogram tree =
  let hist = array (1,5) [ 0 | i <- [1..5] ]
  in traverse tree hist
```

October 23, 2002

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


Histogram : Using M-structures

```
mkHistogram tree =
  let hist = mArray (1,5) [ 0 | i <- [1..5]]
    ( traverse tree hist
      >>>
      hist' = hist )
  in hist'

traverse :: Tree -> (MArrayI Int) -> ()
traverse (Leaf i) hist =

traverse (Node ltree rtree) hist =
```

?

?

No threading, No copying
+ Natural coding style and more parallelism

October 23, 2002

<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}
```

M-structure slot

Take

```
tl & x
```

Put

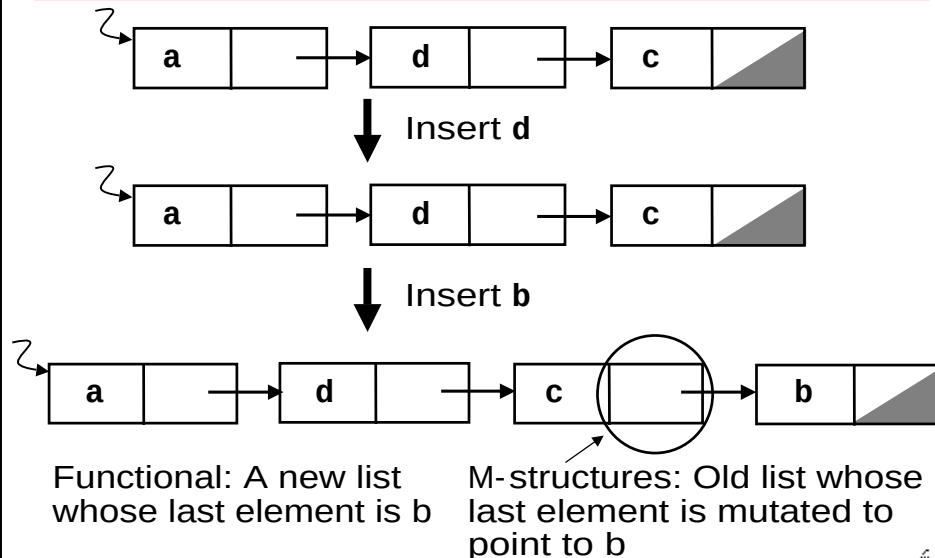
```
tl x := v
```

No side-effects while pattern matching

October 23, 2002

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

Inserting an element in a list



October 23, 2002

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

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
```

?

October 23, 2002

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

Subtle Issues

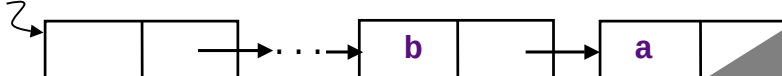
Compare

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

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

assuming **a** and **b** are not in **ys**.

ys2 Can the following list be produced?



October 23, 2002

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


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
```

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

ys1 can be returned before the insertion of **a** is complete but **(tl&ys)** can't be read again before **(tl&ys)** is set

Can you replace **tl&ys** by **ys'**?

October 23, 2002

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


Membership and Insertion

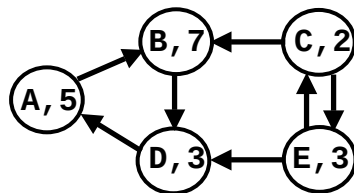
`insertm'` is the same as `insertm` except that it also returns a flag that indicates if a match was found

```
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)
```

October 23, 2002

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

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]
```

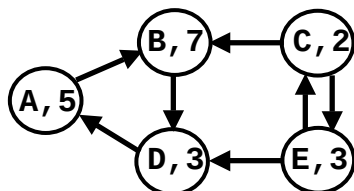
Write function `rsum` to sum the nodes reachable from a given node.

`rsum a ==> ?`

October 23, 2002

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

Graph Traversal: First Attempt



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

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

October 23, 2002

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

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}
```

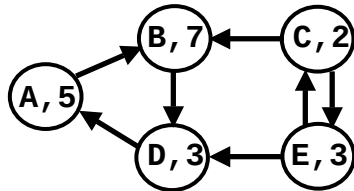
A procedure to return the current flag value of a node and to simultaneously set it to true

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

October 23, 2002

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

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))

```

October 23, 2002

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

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 -> ()
```

October 23, 2002

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

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 s
```

?

October 23, 2002

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

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

October 23, 2002

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