M-Structures: Programming with State and Nondeterminism

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

October 19, 2006

Corrected October23

Limitations of Functional Programming

- For some problems
  - Forces an *obscure coding style* - thread the “state”
  - 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
  - also 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

October 26

I-Cell: The Simplest I-Structure

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

I-Structure field

Constructor

```
ICell :: a -> ICell a

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 ...
```

I-Cell: Dynamic Behavior

- Let allocated I-cells be represented by objects o_1, o_2, ...
- Let the states of an I-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 I-cell
  (x=iFetch(o) ; full(o,v)) $\Rightarrow$ (x=v ; full(o,v))

- Storing into an I-cell
  (iStore(o,v) ; empty(o)) $\Rightarrow$ full(o,v)
  (iStore(o,v) ; full(o,v')) $\Rightarrow$ (error(o); full(o,v'))

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 --> T

The Top represents a contradiction

All functional data structures in pH are implemented as I-structures.

I-structures are *non functional*

```
f x y = let x!1 := 10
            y!1 := 20
        in ()
```

```
let x = iArray (1,2) []
in  f x x
```

≡

```
f  (iArray (1,2) []) (iArray (1,2) []) ?
```

The example

```
f x y = let x!1 := 10
            y!1 := 20

        in ()
```

```
let
 x = iArray (1,2) []
in
 f x x
 ⇓
let
 x = iArray (1,2) []
 x!1 := 10
 x!1 := 20
 ⇓
 "blow up"
```

```
f  (iArray (1,2) [])
   (iArray (1,2) [])
⇓
let
 t1 = iArray (1,2) []
 t2 = iArray (1,2) []
 t1!1 := 10
 t2!1 := 20
in ()
```

M-Cell: The Simplest M-Structure

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

M-Structure field

Constructor

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

```
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 ?

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)) $\Rightarrow$ (x=v ; empty(o))
- Storing into an M-cell
  (mStore(o,v) ; empty(o)) $\Rightarrow$ full(o,v)
  (mStore(o,v) ; full(o,v')) $\Rightarrow$ (error(o); full(o,v'))

The Need of Barriers

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

First attempt:

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

Correct ?

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

Second attempt:

barrier

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

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!*

Barriers: Dynamic Behavior

- A barrier discharges when all the bindings in its pre-region *terminate*, i.e., all expressions become *values.*

```
let
  ( y = 1+7
   >>>
    z = 3 )
in
  z
```

⇒

```
let
  ( y = 8
   >>>
    z = 3 )
in
  z
```

⇒

```
let
  ( y = 8

    z = 3 )
in
  z
```

⇒

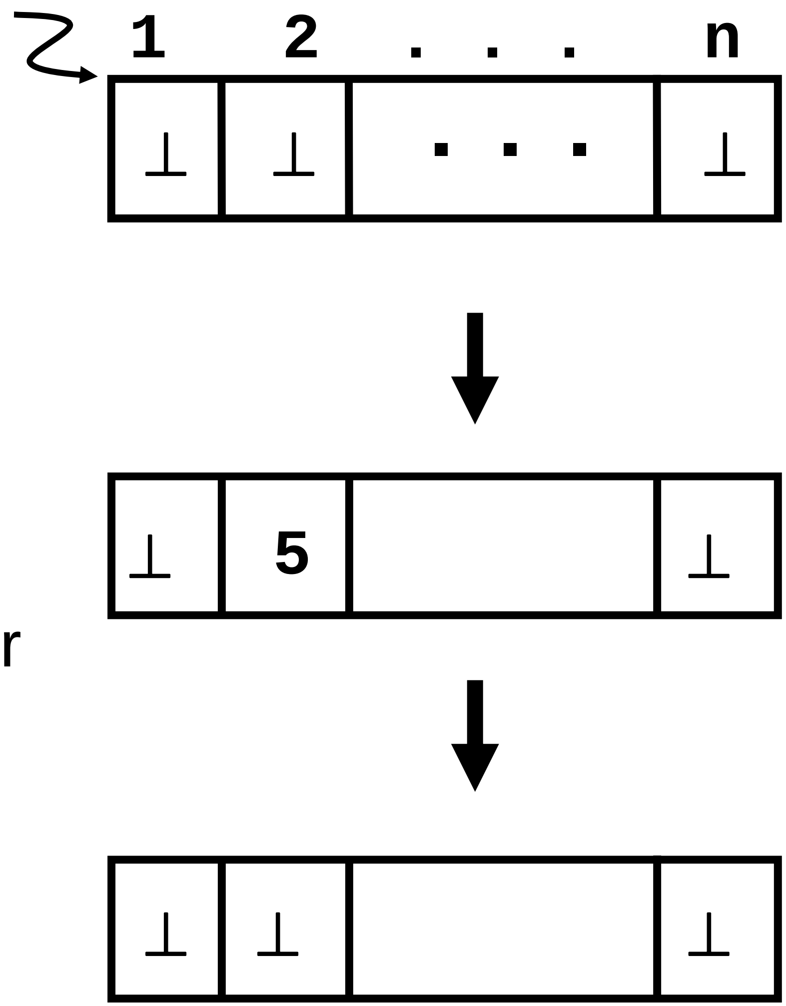
```
let
    y = 8

    z = 3
in
    3
```

M-Arrays

- *Allocate*

  ```
  x = mArray (1,n) []
  ```



- *Put*

  ```
  x!2 := 5
  ```

  A put operation on a full slot is an error

- *Take*

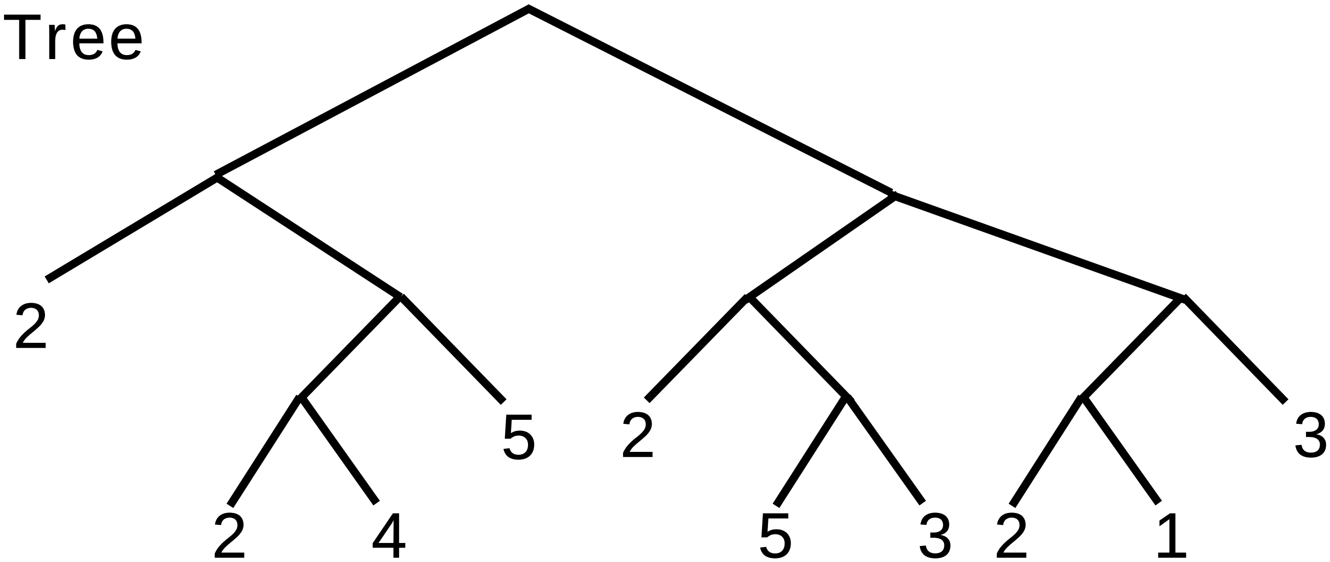
  ```
  x!&2
  ```

Three Examples

- Histograms
- Inserting an element in a list
- Graph traversal (next lecture)

Histogram of Elements in a Tree

Tree



Histogram

1	2	3	4	5
1	4	2	1	2

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 =        ?
```

```
let  hL = traverse ltree hist
in   traverse rtree hL
```

```
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
```

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 = Let hist!i := hist!&i + 1      ?
                         in ()
traverse (Node ltree rtree) hist =                      ?

                         Let traverse ltree hist
                             traverse rtree hist
                         in ()
```

No threading, No copying

+ Natural coding style and more parallelism

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
```

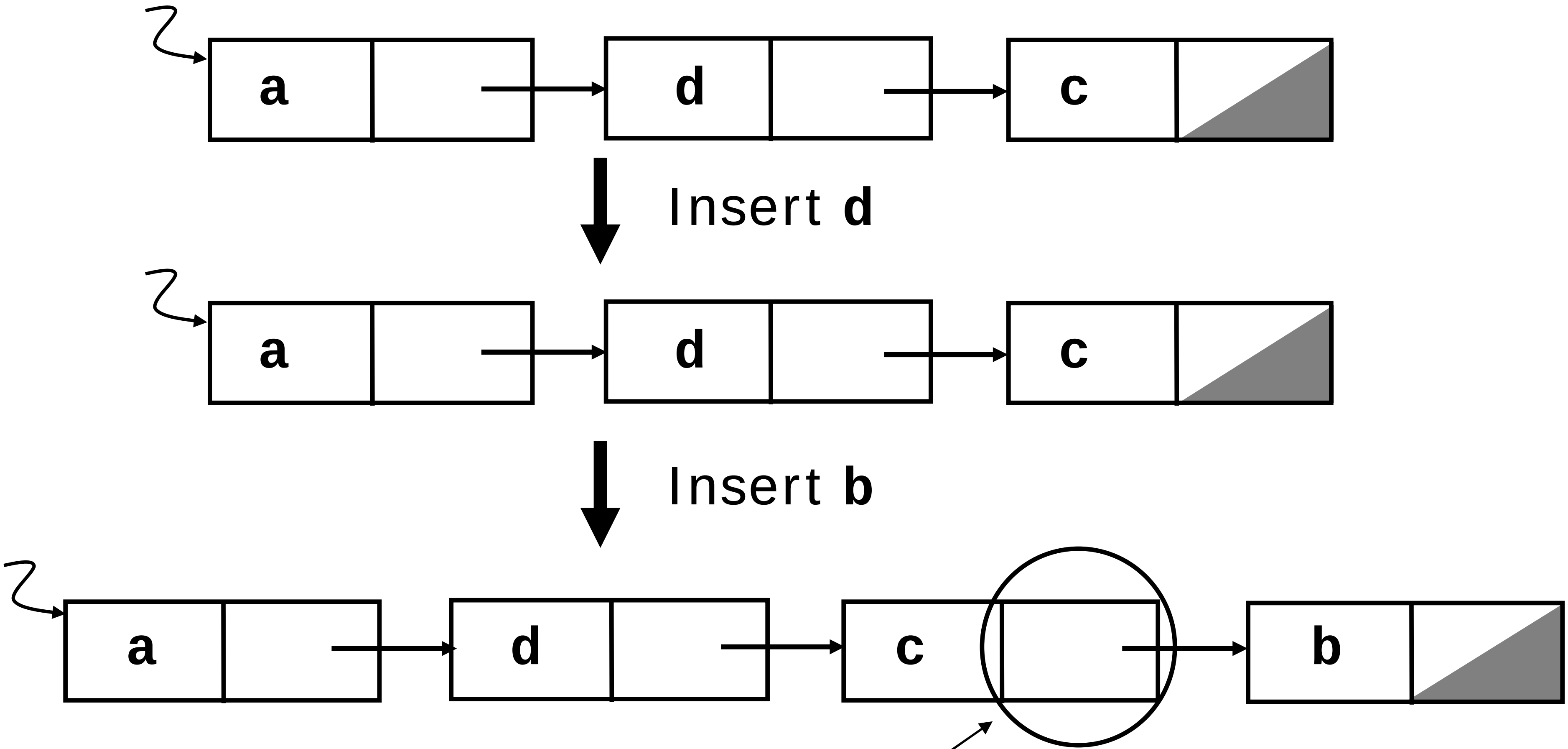
Put

```
tl x := v
```

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

No side-effects while pattern matching

Inserting an element in a list



Functional: A new list whose last element is b

M-structures: Old list whose last element is mutated to point to b

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
```

Can you replace **tl&ys** by **ys’** ? No

Subtle Issues

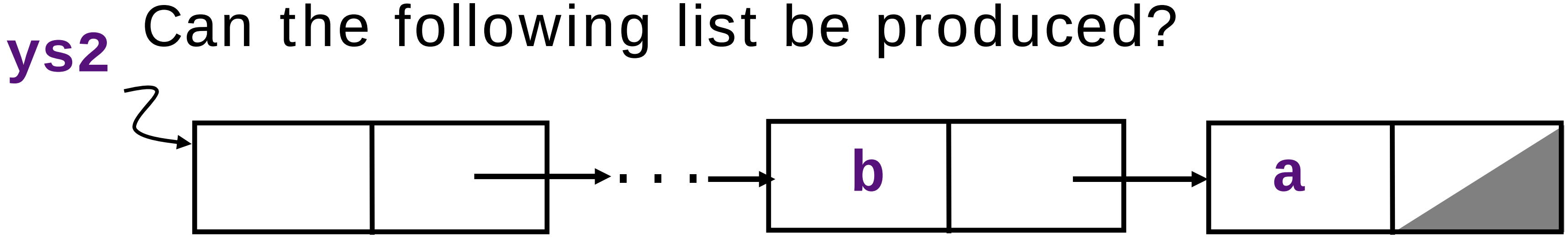
Compare

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

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

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

Can the following list be produced?



Out-of-order Insertion

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

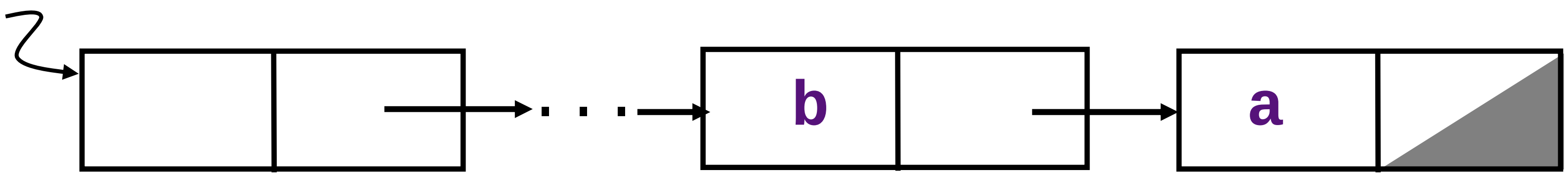
no

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

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

yes!

ys2 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