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 $ys_2$’s assuming $a$ and $b$ are not in $ys$.

**no**

$ys_1 = \text{insertf } ys \ a$

$ys_2 = \text{insertf } ys_1 \ b$

**yes!**

$ys_1 = \text{insertm } ys \ a$

$ys_2 = \text{insertm } ys_1 \ b$

Can the following list be produced?

$ys_2$

Can the following list be produced?

$ys_1$ can be returned before the insertion of $a$ is complete.

How can we stop the out of order insertion?
\textbf{insertm Reexamined}

\begin{align*}
\text{insertm } ys \ x = \\
\text{\hspace{1cm} case } ys \ \text{of} \\
\text{\hspace{2cm} MNil } & \rightarrow \ MCons \ x \ MNil \\
\text{\hspace{2cm} MCons } y \ ys' & \rightarrow \\
\text{\hspace{3cm} if } x \ \text{==} \ y \ \text{then } ys \\
\text{\hspace{3cm} else } \text{let } tl \ ys := \text{insertm} \ (tl\&ys) \ x \\
\text{\hspace{3.4cm} in } \ ys
\end{align*}

- 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

\[
\text{insertm } \text{ys } x = \\
\text{case } \text{ys } \text{ of} \\
\text{MNil } \rightarrow \text{MCons } x \text{ MNil} \\
\text{MCons } y \text{ ys'} \rightarrow \\
\text{if } x == y \text{ then } \text{ys} \\
\text{else let} \\
\text{tl } \text{ys} := \text{insertm } \text{(tl&ys)} \text{ x} \\
\text{tlPtr} \\
\text{in } \text{ys} \\
\text{listToBeReturned}
\]

Do the take first!

Notice \((\text{tl&ys})\) can’t be read again before \((\text{tl } \text{ys})\) is set
Graph Traversal

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

```
rsum a ==> 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.

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

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

\[
\text{marked node} = \text{let } m = \text{flag} \& \text{node} >>> \\
\text{flag node} := \text{True} \\
\text{in } \\
\text{in } \\
m
\]
**Graph Traversal: Mutable Markings**

The graph is shown with nodes labeled A, B, C, D, E and their respective values 5, 7, 2, 3. The edges are as follows:
- A → B
- A → D
- B → C
- C → B
- D → A
- D → E
- E → D

The data structure `GNode` is defined as:

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

The function `rsum` is defined as:

```haskell
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

```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

- `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

```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 fold thread (s',nb') nbs
          in s
```

October 24, 2006
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*

*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.

```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
```

The test for membership and subsequent insertion have to be done atomically to avoid races.
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 \( nb \ x = \)
\[
\begin{align*}
    & \quad \text{let } i = \text{hash } x \\
    & \quad \text{ys} = nb!&i \\
    & \quad (\text{flag}, \text{ys'}) = \text{insertm'} \text{ ys } x \\
    & \quad nb!i := \text{ys'} \\
    & \quad \text{in \ flag}
\end{align*}
\]

\text{insertm'} is the same as \text{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.

```haskell
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):

```lisp
(define (hello)
  (princ "Hello ")
  (princ "World "))
```

Weakens equational reasoning:

```lisp
(let ((a (f x)))
  (let ((b (g y)))
    (+ a b)))
```

vs

```lisp
(let ((a (f x)))
  (+ a b)))
```

- Provide explicit constructs for sequencing in FL

```lisp
(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.

\begin{verbatim}
  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
\end{verbatim}

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

\[
\begin{align*}
\text{main} &:: \text{[string]} \\
\text{main} = &\text{[“Hello”, “world!”]} \\
\text{or} \\
\text{main} = &\text{let } a = \text{“Hello”} \\
&\text{b = “World!”} \\
&\text{in } [a,b]
\end{align*}
\]

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:

\[
(\gg\gg) : IO () \rightarrow IO () \rightarrow IO ()
\]

Example:

\[
\begin{align*}
\text{putChar 'H'} & \gg \text{putChar 'i'} \gg \\
\text{putChar '!'} & : IO ()
\end{align*}
\]

\[
\begin{align*}
\text{putString} & : \text{String} \rightarrow IO () \\
\text{putString} & \text{""} = \text{done} \\
\text{putString} & (c:cs) = \\
& \text{putChar c} \gg \text{putString} \text{ cs}
\end{align*}
\]
Monads: Composing Actionspecs

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

\[(\ggg) : \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b\]

We read the “bind” operator as follows:

\[x_1 \ggg \lambda a \rightarrow x_2\]

- Perform the action represented by \(x_1\), producing a value of type “a”
- Apply function \(\lambda a \rightarrow x_2\) to that value, producing a new actionspec \(x_2 : \text{IO } b\)
- Perform the action represented by \(x_2\), producing a value of type b

Example: \(\text{getChar} \ggg \text{putChar}\) the same as \(\text{getChar} \ggg \lambda c \rightarrow \text{putChar } c\)
An Example

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

\[ \text{return} :: \text{a} \rightarrow \text{IO}\ \text{a} \]

\[ \text{getLine} :: \text{IO}\ \text{String} \]

\[
\text{getLine} = \text{getChar} \gg= \ \backslash c \rightarrow \\
\quad \text{if}\ (c == \ \backslash n)\ \text{then} \\
\quad \quad \text{return} \ \text{""} \\
\quad \text{else}\ \text{getLine} \gg= \ \backslash s \rightarrow \\
\quad \quad \text{return} \ (c:s)
\]

where `\n` represents the newline character
Monadic I/O

\textbf{IO a}: computation which does some I/O, then produces a value of type \textit{a}.

\begin{align*}
\text{(>>)} & : \text{IO a} \rightarrow \text{IO b} \rightarrow \text{IO b} \\
\text{(>>=)} & : \text{IO a} \rightarrow (\text{a} \rightarrow \text{IO b}) \rightarrow \text{IO b} \\
\text{return} & : \text{a} \rightarrow \text{IO a}
\end{align*}

Primitive actionspecs:

- \text{getChar} :: \text{IO Char}
- \text{putChar} :: \text{Char} \rightarrow \text{IO ()}
- \text{openFile, hClose, ...}

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