Reviewing IO Actions

• Recall properties of special type of IO actions.
• Basic operations have “side-effects”, e.g.

\[
\begin{align*}
\text{getChar} & : : \text{IO} \ \text{Char} \\
\text{putChar} & : : \text{Char} \rightarrow \text{IO} () \\
\text{isEOF} & : : \text{IO} \ \text{Bool}
\end{align*}
\]

• Operations are combined into **sequences** using “do”:

\[
\begin{align*}
\text{echo} & : : \text{IO} () \\
\text{echo} & = \text{do} \ b < - \ \text{isEOF} \\
& \quad \text{if not} \ b \ \text{then} \\
& \quad \quad \text{do} \ \{x < - \ \text{getChar} ; \ \text{putChar} \ x ; \ \text{echo}\} \\
& \quad \text{else} \ \text{return} ()
\end{align*}
\]

• Operations don't actually happen except at “top level” where we implicitly perform an operation with type

\[
\text{runIO} : : \text{IO} \ a \rightarrow a \quad \text{-- actually perform the IO}
\]
“do” and “bind”

• The special notation

\[
\text{do } v1 \leftarrow e1 \\
\quad e2
\]

is just “syntactic sugar” for the (ordinary) expression

\[
e1 \gg= \lambda v1 \rightarrow e2
\]

where \(\gg=\) (pronounced “bind”) is a built-in function

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

which turns a sequence of two IO actions into a single IO action.

• The value returned by the first action needs to be fed to the second action; that's why the second argument to \(\gg=\) is a function (normally, but not necessarily, an explicit lambda-definition).
More about “do”

- Actions of type IO() don't carry a useful value; they can be sequenced using the simpler function
  \[(\gg\gg) \:: IO() \to IO b \to IO b\]
  \[e_1 \gg e_2 = e_1 \gg\gg (\_ \to e_2)\]

- The full translation of “do” notation is
  \[do \{ e \} \equiv e\]
  \[do \{ e; es \} \equiv e \gg do \{es\}\]
  \[do \{ x <- e; es \} \equiv e \gg\gg (\_x \to do \{es\})\]
  \[do \{let ds; es\} \equiv let ds in do \{es\}\]

- Can always do without do if we want
  \[echo = getChar \gg\gg (\_x \to\]
  \[\quad putChar x \gg\]
  \[\quad echo)\]

  (Note: could drop parentheses)
Now for a different problem

- Recall code for interpreting simple arithmetic expressions

```haskell
data Exp = Plus Exp Exp |
          Minus Exp Exp |
          Times Exp Exp |
          Div Exp Exp |
          Const Int

eval :: Exp -> Int

eval (Plus e1 e2) = (eval e1) + (eval e2)
 eval (Minus e1 e2) = (eval e1) - (eval e2)
 eval (Times e1 e2) = (eval e1) * (eval e2)
 eval (Div e1 e2) = (eval e1) `div` (eval e2)
 eval (Const i) = i

answer = eval (Div (const 3)
              (Plus (Const 4) (Const 2)))
```
Adding Exceptions

- Suppose we want to improve this by trapping attempts to divide by zero.

  data Exception a = Ok a | Error String

  eval :: Exp -> Exception Int

  eval (Div e1 e2) =
  case eval e1 of
      Ok v1 ->
        case eval e2 of
            Ok v2 -> if v2 == 0 then Error "divby0"
                          else Ok (v1 `div` v2)
            Error s -> Error s
            Error s -> Error s

  -- Plus, Minus, Times must be changed similarly

  eval (Int i) = Ok i
Abstracting Exceptional Flow

- This solution exposes a lot of ugly plumbing.
- Notice that whenever an expression evaluates to Error, that Error propagates up to the final result.
- We can abstract this to a higher-order function

```haskell
andthen :: Exception a -> (a -> Exception b) -> Exception b

e `andthen` k =
  case e of
    Ok x -> k x
    Error s -> Error s

eval (Plus e1 e2) =
  eval e1 `andthen` (\v1 ->
    eval e2 `andthen` (\v2 ->
      Ok (v1 + v2)))
```
Exception and IO are Monads

- Compare the types of these functions:

```
andthen :: Exception a -> (a -> Exception b) -> Exception b
Ok :: a -> Exception a

(>>=) :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

- The similarities aren't accidental!
- IO, Exception, and many other type constructors are instances of a more general structure called a **monad**.
- Monads are suitable for describing many kinds of **computational effects** where there is a concept of **sequencing** (captured by >>=).
Monads, Formally

- Formally, a monad is a type constructor \(M\ a\) and two operations
  
  \[
  (\gg\gg) \::\:: \text{M } a \rightarrow (a \rightarrow \text{M } b) \rightarrow \text{M } b
  \]

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

- The operations must satisfy these three laws:

  \[
  \text{m1} \gg\gg (\lambda x \rightarrow (\text{m2} \gg\gg (\lambda y \rightarrow \text{m3})))
  = (\text{m1} \gg\gg (\lambda x \rightarrow \text{m2})) \gg\gg (\lambda y \rightarrow \text{m3})
  \]

  \[
  (\text{return } x) \gg\gg k = k x
  \]

  \[
  \text{m} \gg\gg \text{return} = \text{m}
  \]

- Note that we use the same names for the general case as for IO actions.
The Monad Type Class

- The Prelude defines a class for monadic behavior:
  ```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
  ```

- Unlike other classes we have seen, this one describes a type constructor class (m is a variable representing a type constructor, not a type).

- The IO type constructor is declared as an instance of this class, using built-in primitive defns. roughly like this
  ```haskell
  instance Monad IO where
      return = builtinReturnIO
      (>>=) = builtinBindIO
  ```

- The “do” notation can be used for any instance of the Monad class, including user-defined instances.
Exceptions revisited

- Can make Exception an instance

  ```haskell
  instance Monad Exception where
  return = Ok
  (>>=) = andthen
  ```

- Now can rewrite interpreter code using “do” notation, e.g.

  ```haskell
  eval (Plus e1 e2) =
  do v1 <- eval e1
     v2 <- eval e2
     return (v1+v2)
  ```

- In fact, the (very similar) Maybe type is already defined as an instance in the Prelude:

  ```haskell
  instance Monad Maybe where
  return = Just
  (Just x) >>= k = k x
  Nothing >>= k = Nothing
  ```
Threading Auxiliary Information

Suppose that we want to extend our (original) interpreter to produce a trace of operations in the order that they occur, in addition to a final answer.

\[
\text{eval :: Exp } \rightarrow \text{ String } \rightarrow (\text{String, Int})
\]
\[
\text{eval (Plus e1 e2) s } =
\]
\[
\quad \text{let } (s1,v1) = \text{eval e1}
\quad (s2,v2) = \text{eval e2}
\quad \text{in } (s ++ s1 ++ s2 ++ " +", e1 + e2)
\]

\[
\ldots
\]
\[
\text{eval (Const i) s } = (s ++ " " ++ \text{show i, i})
\]
\[
(\text{trace, answer}) =
\quad \text{eval (Div (Const 10) (Plus (Const 2) (Const 3))) ""}
\quad \text{-- returns (" 10 2 3 + /", 2)}
\]
Maintaining State

- In imperative language, would be more convenient to maintain trace info in a global variable (part of the program **state**) which is **updated** by each eval step.
- Avoids need to thread trace to/from each function call.
- Can capture this idiom using a (particular) **state monad**.

```haskell
newtype SM a = SM (String -> (String, a))
instance Monad SM where
  return a = SM (\s -> (s, a))
  (SM m1) >>= k = SM (\s -> let (s1,a) = m1 s
                           SM m2 = k a
                           in m2 s1)
runSM :: SM a -> (String, a)
runSM (SM m) = m ""
trace :: String -> SM ()
trace s0 = SM (\s -> (s ++ s0, ()))
```
Stateful computation using “do”

- Now can rewrite tracing eval in “do” notation:

  eval :: Exp -> SM Int
  eval (Plus e1 e2) =
    do v1 <- eval e1
       v2 <- eval e2
       trace " +"
       return (v1 + v2)

  ...
  eval (Const i) =
    do trace (" " ++ show i)
       return i
  (trace, answer) =
    runSM (eval (Div (Const 10) (Plus (Const 2) (Const 3))))
    -- returns (" 10 2 3 + /", 2)
Simulating the IO Monad

- The IO monad is “built-in” to Haskell, i.e., it cannot be implemented within the language itself.
  - Special primitives are needed to actually perform the IO actions and to sequence them.
  - The IO type is abstract (it has no constructors).
- But we can **simulate** the behavior of IO actions involving a single input and output stream, using the following type:

  ```haskell
  newtype IOX t = IOX (Input -> (t, Input, Output))
  type Input = String
  type Output = String
  ```

- Each IOX function takes the available input as argument, performs an IO action that consumes some of that input, and returns:
  - the result of the action (of type t)
  - the remaining input
  - any output produced by the action
The Simulated IO Monad

instance Monad IOX where
  (IOX m) >>= k =
    IOX (\input ->
      let (t, input', output) = m input
        IOX m' = k t
        (t', input'', output') = m' input'
        in (t', input'', output ++ output')
      return x = IOX (\input -> (x,input,""))

getChar :: IOX Char
getChar = IOX (\(i:is) -> (i,is,""))

putChar :: Char -> IOX ()
putChar c = IOX (\is -> ((),is,[c]))

isEOF :: IOX Bool
isEOF = IOX (\input -> (null input,input,""))