CS 457/557 Functional Programming

Lecture 18
Monads
Reviewing IO Actions

• Recall properties of special type of IO actions.
• Basic operations have “side-effects”, e.g.
  
  \[
  \begin{align*}
  \text{getChar} & \,::\, \text{IO Char} \\
  \text{putChar} & \,::\, \text{Char} \rightarrow \text{IO ()} \\
  \text{isEOF} & \,::\, \text{IO Bool}
  \end{align*}
  \]

• Operations are combined into **sequences** using “do”:
  
  \[
  \begin{align*}
  \text{echo} & \,::\, \text{IO ()} \\
  \text{echo} = \text{do} \ b \leftarrow \text{isEOF} \\
  & \quad \text{if not } b \text{ then} \\
  & \quad \quad \text{do} \ \{x \leftarrow \text{getChar}; \ \text{putChar } x; \ \text{echo}\} \\
  & \quad \text{else return ()}
  \end{align*}
  \]

• 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
  
  
  do v1 <- e1
  
  e2
  
  is just “syntactic sugar” for the (ordinary) expression

  e1 >>= \v1 -> e2

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

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

  \[(\gg\gg) \; :: \; IO \; a \; \rightarrow \; IO \; b \; \rightarrow \; IO \; b\]

  \[e_1 \; \gg\gg \; e_2 \; = \; e_1 \; \gg\gg\; (\_ \; \rightarrow \; e_2)\]

• The full translation of “do” notation is

  \[do \; \{ \; e \; \} \; \equiv \; e\]

  \[do \; \{ \; e; \; es\} \; \equiv \; e \; \gg\gg \; do \; \{es\}\]

  \[do \; \{ \; x \; \leftarrow \; e; \; es\} \; \equiv \; e \; \gg\gg\; (\_x \; \rightarrow \; do \; \{es\})\]

  \[do \; \{let \; ds; \; es\} \; \equiv \; let \; ds \; in \; do \; \{es\}\]

• Can always do without \texttt{do} if we want

  \[echo \; = \; getChar \; \gg\gg\; (\_x \; \rightarrow \]

  \[\begin{align*}
  & putChar \; x \; \gg \\
  & \text{echo}
  \end{align*}\]

  (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.

\[
\begin{align*}
\text{data Exception } & \text{ a } = \text{ Ok } \text{ a } \mid \text{ Error } \text{ String} \\
\text{eval } & : : \text{ Exp } \rightarrow \text{ Exception } \text{ Int} \\
\text{eval } (\text{Div } e_1 e_2) & = \\
& \text{case eval } e_1 \text{ of} \\
& \text{ Ok } v_1 \rightarrow \\
& \text{ case eval } e_2 \text{ of} \\
& \text{ Ok } v_2 \rightarrow \text{ if } v_2 == 0 \text{ then Error } \text{ “divby0”} \\
& \text{ else } \text{ Ok } (v_1 \ `\text{div}` v_2) \\
& \text{ Error s } \rightarrow \text{ Error s} \\
& \text{ Error s } \rightarrow \text{ Error s} \\
& \text{ -- Plus, Minus, Times must be changed similarly} \\
\text{eval } (\text{Int } i) & = \text{ Ok } i
\end{align*}
\]
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

```plaintext
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:

  \text{andthen} :: \text{Exception} a \rightarrow (a \rightarrow \text{Exception} b) \rightarrow \text{Exception} b

  \text{Ok} :: a \rightarrow \text{Exception} a

  \text{(>>=)} :: \text{IO} a \rightarrow (a \rightarrow \text{IO} b) \rightarrow \text{IO} b

  \text{return} :: a \rightarrow \text{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

  \[
  (\op{>>=}) :: M \ a \to (a \to M \ b) \to M \ b
  \]

  \[
  \op{return} :: a \to M \ a
  \]

- The operations must satisfy these three laws:

  \[
  m_1 \op{>>=} (\lambda x \to (m_2 \op{>>=} (\lambda y \to m_3)))
  = (m_1 \op{>>=} (\lambda x \to m_2)) \op{>>=} (\lambda y \to m_3)
  \]

  provided that \( x \) does not appear in \( m_3 \)

  \[
  (\op{return} x) \op{>>=} k = k x
  \]

  \[
  m \op{>>=} \op{return} = m
  \]

- 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.

\[
\text{eval} :: \text{Exp} \rightarrow \text{String} \rightarrow (\text{String}, \text{Int})
\]
\[
\text{eval} (\text{Plus } e_1 e_2) s =
\]
\[
\text{let} \ (s_1,v_1) = \text{eval } e_1 \ s
\]
\[
\quad (s_2,v_2) = \text{eval } e_2 \ s_1
\]
\[
in \ (s_2 + + " +", e_1 + e_2)
\]
\[
\ldots
\]
\[
\text{eval} (\text{Const } i) s = (s + + " " + + \text{show } i, i)
\]
\[
(\text{trace}, \text{answer}) =
\]
\[
\text{eval} (\text{Div } (\text{Const } 10) (\text{Plus } (\text{Const } 2) (\text{Const } 3)) ""
\]
\[
-- \text{return} \ (" 10 2 3 + /", 2)
\]
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:

```haskell
 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 definitions:

  ```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
      in (t', input'', 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,"")