IO monad

Imperative programming in Haskell

Deian Stefan

(adopted from my & Edward Yang’s CSE242 slides)
Can we do IO as usual?

\[ l_\text{s} :: [(), ()] \]
\[ l_\text{s} = [\text{putChar} 'x', \text{putChar} 'y'] \]

Is this okay? A: yes, B: no
Laziness gets in the way?

- Depending on evaluation order order of effects may vary or may not even be observed

  - E.g., length 1s vs. head 1s

- Laziness forces us to take a more principled approach!
Monad IO

• Extend category of values with actions

• A value of type \( \text{IO } a \) is an action

• \textbf{When performed}, the action of type \( \text{IO } a \) may perform some I/O before it delivers a result of type \( a \)

• How to think about actions:
  
  ➤ \text{type IO } a = \text{World } \rightarrow (a, \text{World})
getChar :: IO Char
IO actions are first-class

• What does this mean? (Recall: first-class functions)
  ➤ Can return actions from function
  ➤ Can pass actions as arguments
  ➤ Can create actions in functions
putChar :: Char -> IO ()
How do we create actions?

• The return function:
  ➤ Worst name ever: has nothing to do with terminating early
  ➤ Given value produce IO action that doesn’t perform any IO and only delivers the value
  ➤ return :: a -> IO a
Example: return

- return 42

- \( f \ x = \ \text{if } x \)
  
  \begin{align*}
  & \text{then return "what" } \\
  & \text{else return "no way!"}
  \end{align*}
How do we create actions?

- The compose function (>>>)

  ➤ Given an IO action \( \text{act}_1 \) and action \( \text{act}_2 \) produce a bigger action, which when executed:

  ➤ executes \( \text{act}_1 \)

  ➤ execute \( \text{act}_2 \) and deliver the value produced by \( \text{act}_2 \)

  ➤ (>>>) :: IO a \rightarrow IO b \rightarrow IO b
Example: >>

- return 42 >> putChar ‘A’ >> putChar ‘B’

- f x = putStrLn “hello world” >>
  if x == “hello”
    then return x
  else return “bye bye!”
How do we create actions?

• The bind function ( >>= )
  ➤ Like (>>), but doesn’t drop the result of first action: it chains the result to the next action (which may use it)
  ➤ ( >>= ) :: IO a -> (a -> IO b) -> IO b

• Can we define (>>) in terms of ( >>= )?
(>>>) via (>>>=)

• Recall:
  ➤ (>>>=) :: IO a -> (a -> IO b) -> IO b
  ➤ (>>>)  :: IO a -> IO b    -> IO b

• From this:
  ➤ (>>>) act1 act2 = act1 >>= \\
  \_  -> act2
Example: >>=

- \( \text{return } 42 >>= (\ i \rightarrow \text{putChar} (\text{chr } i)) \)

- \( \text{echo} :: \text{IO } () \)
  \( \text{echo} = \text{getChar} >>= (\ c \rightarrow \text{putChar } c) \)

- \( \text{echoTwice} :: \text{IO } () \)
  \( \text{echoTwice} = \text{getChar} >>= \ (\ c \rightarrow \text{putChar } c) \)
  \( \rightarrow \ (\ c \rightarrow \text{putChar } c) \)
Do notation

- Syntactic sugar to make it easier create big actions from small actions

- `getTwoChars :: IO (Char, Char)`
  ```haskell
getTwoChars = do
c1 <- getChar
c2 <- getChar
return (c1, c2)
```
Do notation: de-sugaring

- \( \text{do } x \leftarrow e \)
  
  \( e >>= \lambda x \rightarrow \text{do } s \)

- \( \text{do } e \)
  
  \( e >> \text{do } s \)

- \( \text{do } e \)
  
  \( e \)
How do we execute actions?

- Haskell program has to define main function
  
  ```haskell
  main :: IO ()
  ```

- To execute an action it has to be bound!
Monads are cool!

- Principled way to expose imperative programming in FP languages
- Evaluation order is explicit
- Idea goes beyond IO: you can define your own monad
  - E.g., LIO monad does security checks before performing, say, a readFile to prevent data leaks