IO monad
Imperative programming in Haskell

Deian Stefan

(adopted from my & Edward Yang’s CSE242 slides)

Can we do IO as usual?

```
ls :: [(), ()]
ls = [putChar ‘x’, 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 ls` vs. `head ls`
- Laziness forces us to take a more principled approach!

Monad IO

- Extend category of values with actions

- A value of type (IO a) is an action

- When performed, the action of type IO a may perform some I/O before it delivers a result of type a

- How to think about actions:

  ➤ `type IO a = World -> (a, 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

```haskell
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 =  if x
            then return “what”
            else return “no way!”
  ```

How do we create actions?

- The compose function (>>)
  - Given an IO action act_1 and action act_2 produce a bigger action, which when executed:
    - executes act_1
    - execute act_2 and deliver the value produced by act_2
  - `(>>) :: IO a -> IO b -> 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: >>=

- `return 42 >>= (\i -> putChar (chr i))`

-
  ```
  echo :: IO ()
  echo = getChar >>= (\c -> putChar c)
  ```

-
  ```
  echoTwice :: IO ()
  echoTwice =  getChar >>= \c ->
               putChar c >>= \_ ->
               putChar c
  ```

Do notation

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

-
  ```
  getTwoChars :: IO (Char, Char)
  getTwoChars = do
     c1 <- getChar
     c2 <- getChar
     return (c1, c2)
  ```

Do notation: de-sugaring

• do x <- e s	➡ e >>= \x -> do s
• do e s	➡ e >> do s
• do e	➡ e

How do we execute actions?

- Haskell program has to define main function
  - `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