IO Without Breaking a Sweat

Explaining Haskell's IO without Monads

by

Chris Wilson

March 3, 2014

Contents

1. IO as a Value

    See how IO is just another value like anything else

2. Working with IO

    Use the fact that *IO is a value* to get stuff done

3. Next Steps

    Some thoughts and links

Introduction

Before I Start

This presentation is based on Neil Mitchell's excellent blog post, *Haskell IO Without Monads*: `http://neilmitchell.blogspot.co.uk/2010/01/haskell-io-without-monads.html`

IO as a Value

Four IO Functions

- **`readFile :: FilePath -> IO String`**
  read in a file
- **`writeFile :: FilePath -> String -> IO ()`**
  write out a file
- **`getArgs :: IO [String]`**
  get command line arguments (as a list of strings)
- **`putStrLn :: String -> IO ()`**
  write a string to the console followed by a newline

Simple IO Pattern

```
main :: IO ()
main = do
   src <- readFile ”file.in”
   writeFile ”file.out” (operate src)

operate :: String -> String
operate = -- your code here
```

- The “processor” function, `operate`, is just a normal function.

Action List Pattern

```haskell
main :: IO ()
main = do
    x1 <- expr1
    x2 <- expr2
    -- ...
    xN <- exprN
    return ()
```

Action List Pattern

```haskell
main :: IO ()
main = do
    [arg1,arg2] <- getArgs
    src <- readFile arg1
    res <- return (operate src)
    _ <- writeFile arg2 res
    return ()
```

Simplifying IO

You can simplify IO according to three rules:

1. `_ <- x` can be written as `x`.
2. If the second-to-last thing in a do block has no binding arrow (`<-`) and is of type `IO ()`, then you can leave off the `return ()`.
3. `x <- return y` can be re-written as `let x = y`

Simplifying IO

```
main :: IO ()
main = do
    [arg1,arg2] <- getArgs
    src <- readFile arg1
    res <- return (operate src)
    _ <- writeFile arg2 res
    return ()
```

- We can re-factor our code using these rules!

Simplifying IO

```
main :: IO ()
main = do
    [arg1,arg2] <- getArgs
    src <- readFile arg1
    res <- return (operate src)
    writeFile arg2 res
    return ()
```

- Rule 1

Simplifying IO

```
main :: IO ()
main = do
    [arg1,arg2] <- getArgs
    src <- readFile arg1
    res <- return (operate src)
    writeFile arg2 res
```

- Rule 2

Simplifying IO

```haskell
main :: IO ()
main = do
    [arg1,arg2] <- getArgs
    src <- readFile arg1
    let res = operate src
    writeFile arg2 res
```

- Rule 3

Nested IO

- We can also *nest* IO actions...

```
title :: String -> IO ()
title str = do
    putStrLn str
    putStrLn (replicate (length str) ’-’)
    putStrLn ””
```

- ...and then use it in `main`.

```
main :: IO ()
main = do
    title ”Hello”
    title ”Goodbye”
```

Returning IO Values

- We're not just limited to the **IO ()** type, we can return values from IO
- This function returns the first two command line args as a tuple:

```
readArgs :: IO (String,String)
readArgs = do
    xs <- getArgs
    let x1 = if length xs > 0
               then xs !! 0 else ”file.in”
    let x2 = if length xs > 1
               then xs !! 1 else ”file.out”
    return (x1,x2)
```

Returning IO Values

- Now we can use `readArgs` in `main`:

```haskell
main :: IO ()
main = do
    (arg1,arg2) <- readArgs
    src <- readFile arg1
    let res = operate src
    writeFile arg2 res
```

Optional IO

- In any *real* program, we need to *optionally* run code in response to input:

```
main :: IO ()
main = do
    xs <- getArgs
    if null xs then do
        putStrLn ”You␣entered␣no␣arguments”
     else do
        putStrLn (”You␣entered␣” ++ show xs)
```

Working with IO

Remember, IO is a Value

- Recall that the `title` function had type `IO ()`
- Which means it can be used as-is in a do block to run the action three times
- That is, we *don't* have to immediately execute **IO** actions

```
main :: IO ()
main = do
    let x = title ”Welcome”
    x
    x
    x
```

We Can Pass Arguments to IO

```
replicateM_ :: Int -> IO () -> IO ()
replicateM_ n act = do
    if n == 0 then do
        return ()
     else do
        act
        replicateM_ (n-1) act
```

- We recursively run the `IO ()` as many times as we need,
- so, rewriting our last example:

```
main :: IO ()
main = do
    let x = title ”Welcome”
    replicateM_ 3 x
```

Store IO in Structures

- **sequence_** runs a list of actions in turn:

```
sequence_ :: [IO ()] -> IO ()
sequence_ xs = do
    if null xs then do
        return ()
     else do
        head xs
        sequence_ (tail xs)
```

- We can refactor `replicateM_` using **sequence_**:

```
replicateM_ :: Int -> IO () -> IO ()
replicateM_ n act = sequence_ (replicate n act)
```

Pattern Match

- Keeping in mind that **IO** is just a value, we can pattern match on it
- let’s refactor that definition of **sequence_**:

```
sequence_ :: [IO ()] -> IO ()
sequence_ []       = return ()
sequence_ (x:xs) = do
    x
    sequence_ xs
```

A Short Example

```
main :: IO ()
main = do
    xs <- getArgs
    sequence_ (map operateFile xs)

operateFile :: FilePath -> IO ()
operateFile x = do
    src <- readFile x
    writeFile (x ++ ”.out”) (operate src)

operate :: String -> String
operate = -- ...your code here
```

- This performs `operate` on each file given on the command line.

Next Steps

Other Things to Check Out

- *Programming in Haskell* by Graham Hutton (chapters 8 & 9)
- Monads as Containers
- Many more useful functions can be found in the Control.Monad package