CS 115
Functional Programming

Lecture 11:

The `IO` Monad

(for working Haskell programmers)

Today

- Practical interlude: the `IO` monad in practice
- Basic `IO` operations and functions
- The `>>` operator of the `Monad` class
- The `fail` method of the `MonadFail` class
- Monadic syntactic sugar: the `do` notation

Review

- The `Monad` type class contains two fundamental operations: `return` and `>>=`
- `return` is used to "lift" a regular value into a monadic value
- `>>=` is monadic application: it takes a monadic value, "unpacks" a regular value from it, and passes it to a monadic function to get a monadic value as a result

`IO` monadic functions

- `IO` is a type constructor which is an instance of the `Monad` constructor class
- `IO` has kind `* -> *`, as you'd expect
- `IO` monadic functions have the general form:

```
a -> IO b
```

- …representing a function that transforms a value of type `a` to a value of type `b`, possibly doing some I/O in the process

`IO` monadic values

- `IO` monadic values have the types `IO a` for some type `a`
- This represents a monadic "action" which (possibly) does some I/O and then returns a value of type `a`

The >>= operator

- The `>>=` operator for `IO` has this (specialized) type signature:

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

- This will be used in contexts like:

```
v >>= f
```

- Where:

`v :: IO a` (monadic value)

`f :: a -> IO b` (monadic function)

The >>= operator

`v >>= f`

`v :: IO a` (monadic value)

`f :: a -> IO b` (monadic function)

- Interpretation of `>>=` operator:
- `>>=` "unpacks" a value of type `a` from `v`, and passes it to `f`, which returns a (monadic) value of type `IO b`
- The trick is in the unpacking! (details omitted)

The `return` function

- `return` in the context of `IO` has the specialized type:

```
return :: a -> IO a
```

- `return` takes a regular Haskell value and turns it into a monadic value *i.e.* one that can (possibly) do some I/O and then return that value
- In fact, a value returned by `return` will not do I/O
  - What could it reasonably do?

Simple example

- Consider these two `IO` values/functions:
  - `getLine :: IO String`
  - `putStrLn :: String -> IO ()`
- `getLine` reads a newline-terminated line of text from the terminal, returning the text as a `String` (without the newline)
- `putStrLn` takes a `String`, prints it to the terminal, adds a newline, and returns the `()` value (of no significance)

Simple example

- How do we join `getLine` to `putStrLn` so that the line read by `getLine` is printed by `putStrLn`?

```
getAndPrintLine :: IO ()
getAndPrintLine = getLine >>= putStrLn
```

- `getAndPrintLine` is an `IO` monadic value ("action") which does some I/O (reading a line from the terminal, printing the line back to the terminal) and returns a value of type `()`
- The line read by `getLine` is "unpacked" from the `IO String` action by `>>=` and passed to `putStrLn`

Simple example

- More explicitly, we could write this as:

```
getAndPrintLine :: IO ()
getAndPrintLine = getLine >>= (\s -> putStrLn s)
```

  - The last line could more cleanly be written as:

    ```
    getAndPrintLine = getLine >>= \s -> putStrLn s
    ```

- Note that `\s -> putStrLn s` is the same function as just `putStrLn` (*eta expansion*)
- This makes it clear that the line read by `getLine` is "unpacked" into the name `s` and passed to `putStrLn`

Second example

- We want to read a line from the terminal, convert it to uppercase, and then print it back to the terminal
- We will use the **`toUpper`** function which is from the **`Data.Char`** module
- At the beginning of the file, we will need to write

```
import Data.Char
```

in order to use **`toUpper`**

- Type signature of **`toUpper`**:

```
toUpper :: Char -> Char
```

Second example

- Here's our "function" (monadic value AKA action):

```
readAndPrintUppercase :: IO ()
readAndPrintUppercase =
  getLine >>= \s -> putStrLn (map toUpper s)
```

- We read a line from the terminal, unpack it into `s`, convert it into upper case, then print it
- We could write this more succinctly as:

```
readAndPrintUppercase =
  getLine >>= putStrLn . map toUpper
```

Third example

- Read a line from the terminal and print it twice:

```
readAndPrintTwice :: IO ()
readAndPrintTwice =
  getLine >>=
    \s -> (putStrLn s >>= (\_ -> putStrLn s))
```

- We can drop some parentheses due to low precedence of `>>=` to get:

```
readAndPrintTwice =
  getLine >>=
    \s -> putStrLn s >>= \_ -> putStrLn s
```

Third example

- Notice that:

```
putStrLn s
```

- has the type `IO ()` (since it returns the place-holder `()` value)
- In the expression:

```
putStrLn s >>= (\_ -> putStrLn s)
```

- We unpack the value of type `()` and pass it to the function `(\_ -> putStrLn s)`, which ignores the `()` and prints the string `s` again

Third example

- This kind of situation (chaining together `IO` functions where some do not require the return values of others) is very common
- For instance, no monadic function is going to need the `()` return value of `putStrLn`
- Having to write a monadic function in the form

```
\_ -> ...
```

  - (*i.e.* ignoring the unpacked value)
- is gross!
- Fortunately, there is a nice alternative

The >> monadic operator

- I mentioned that the `return` function and the `>>=` operator were the "core" monadic operations, but that there was one more "non-core" operation
- This "non-core" operation is called `>>`
- It is used to combine two monadic values (actions) into a single monadic value (action), where neither action depends on the return value of the other
- It's sometimes pronounced "then"

The >> monadic operator

- Default definition of `>>` for a monad `m`:

```
(>>) :: m a -> m b -> m b
x >> y = x >>= \_ -> y
```

  - Rarely if ever need to override the default definition
- Intuition: perform "action" `x`, ignore the return value, then perform "action" `y`
- For the `IO` monad, this has the type signature

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

The >> monadic operator

- We can use **>>** to simplify previous example
- Old definition:

```
readAndPrintTwice =
  getLine >>=
    \s -> putStrLn s >>= \_ -> putStrLn s
```

- New definition:

```
readAndPrintTwice =
  getLine >>= \s -> putStrLn s >> putStrLn s
```

- Expresses programmer's intent much more directly

Digression: lambda expressions

- We've seen anonymous functions (lambda expressions):

```
\x -> 2 * x
```

- They have the general form:

```
\<pattern> -> <expression>
```

- Most of the time, the `<pattern>` is just one variable
- Can have more elaborate patterns *e.g.*:

```
Prelude> map (\(x, y) -> x + y) [(1, 2), (3, 4)]
[3,7]
```

Digression: lambda expressions

- Pattern match failures can occur:

```
Prelude> map (\(x:xs) -> x:x:xs) [[1,2], [3,4]]
  warning:
    Pattern match(es) are non-exhaustive
    In a lambda abstraction: Patterns of type `[a]' not
matched: []
[[1,1,2],[3,3,4]]
Prelude> map (\(x:xs) -> x:x:xs) [[1,2], []]
[[1,1,2],*** Exception:
Non-exhaustive patterns in lambda
```

Digression: lambda expressions

- This can happen with the **>>=** operator when used with anonymous functions:

```
Prelude> getLine >>= \(x:xs) -> putStrLn (x:x:xs)
[user enters "foobar"]
ffoobar
Prelude> getLine >>= \(x:xs) -> putStrLn (x:x:xs)
[user enters nothing (hits return)]
*** Exception: Non-exhaustive patterns in lambda
```

- We'll see why this is important soon

do notation

- Recall the third example:

```
readAndPrintTwice :: IO ()
readAndPrintTwice =
  getLine >>= \s -> putStrLn s >> putStrLn s
```

- Very common to see code which is a sequence of monadic operations chained together with `>>=` and/or `>>`
- This code is sometimes hard to read
- Haskell provides an interesting kind of syntactic sugar to make writing monadic code easier

do notation

- We can write this:

```
readAndPrintTwice :: IO ()
readAndPrintTwice =
  getLine >>= \s -> putStrLn s >> putStrLn s
```

- as:

```
readAndPrintTwice :: IO ()
readAndPrintTwice =
   do s <- getLine
      putStrLn s
      putStrLn s
```

do notation

- do-notation makes code look very imperative:

```
readAndPrintTwice :: IO ()
readAndPrintTwice =
   do s <- getLine
      putStrLn s
      putStrLn s
```

- Interpretation: get a line, bind it to `s`, print `s` once, then print it again
- This "desugars" to standard monadic code
- Can be used with *any* monad (not just `IO` monad)

Desugaring do notation

- There are two ways to describe how `do` notation is desugared to regular monadic code:
  - Way #1: simple, works most of the time
  - Way #2: more complex, works the rest of the time
- We will describe both, because way #1 is usually sufficient to understand what's going on
- Way #2 is necessary when pattern match failures are possible
  - otherwise ways 1 and 2 are equivalent

Desugaring do notation: way 1

- Desugar this:

```
do x <- m
   f x
```

- to:

```
m >>= \x -> f x
```

Desugaring do notation: way 1

- Desugar this:

```
do x <- m1
   y <- m2
   f x y
```

- to:

```
m1 >>= \x -> (m2 >>= \y -> f x y)
```

- or just:

```
m1 >>= \x -> m2 >>= \y -> f x y
```

Desugaring do notation: way 1

- Desugar this:

```
do x
   y
```

- to:

```
x >> y
```

Desugaring do notation: way 1

- Desugar this:

```
do x
   y
   z
```

- to:

```
(x >> y) >> z
```

- or just:

```
x >> y >> z  -- >> is left-associative
```

Desugaring do notation: way 1

- This way of desugaring is what's usually taught in textbooks
- It works fine as long as only simple variables are being bound in arrows e.g.

```
x <- m1
```

- Note that right-hand side of `<-` has type `m a` (`IO a`) but left-hand side has type `a`
- Intuitively: monadic value `m1` "unpacks" value of type `a` into `x`

Desugaring do notation: way 2

- We can also put patterns on left-hand side of `<-` *e.g.*

```
(x:xs) <- return [1,2,3]
```

- Idea: bind `x` to `1`, `xs` to `[2,3]`
- Works for any monad (`return` is generic)
- What about here?

```
(x:xs) <- return []
```

- With way 1 desugaring, this is a "non-exhaustive patterns in lambda" error
- This isn't what actually happens!

Desugaring do notation: way 2

- We would like to give our monads control over what happens when a pattern match fails in a `do` expression
- This is done by way of a method of type class `MonadFail`: `fail`
- When a pattern match failure occurs in a `do` expression, `fail` is called
- `fail` has the type signature

```
fail :: MonadFail m => String -> m a
```

MonadFail

- Originally, the `fail` method was part of the `Monad` type class
- It was moved out of `Monad` and into its own class, called `MonadFail`
  - Reasons are technical and not that interesting
- Any instance of `Monad` that needs to be able to handle pattern match failures in a `do` block has to also be an instance of `MonadFail`
- All monads we'll be discussing are also instances of `MonadFail`

Desugaring do notation: way 2

- Consider this code:

```
do (x:_) <- return []
   return x
```

- Desugaring according to way 1 gives:

```
return [] >>= \(x:_) -> return x
```

- This would give the error message:

```
Non-exhaustive patterns in lambda
```

- The actual error message (in the `IO` monad) is:

```
user error (Pattern match failure in 'do' block
at <location>)
```

Desugaring do notation: way 2

```
do (x:_) <- return []
   return x
```

- Desugaring according to way 2 gives:

```
return [] >>=
  \y -> case y of
    (x:_) -> return x
    _ -> fail "Pattern match failure in do block at ..."
```

- `fail` invoked on pattern match failures (hence the name)

Desugaring do notation: way 2

- Any monad that can be used with a pattern match failure must also be an instance of `MonadFail` and define the `fail` method
  - Otherwise, the compiler won't compile code with potential pattern match failures!
- `IO` monad redefines `fail` to something similar to `error` but which wraps the string `"user error(...)"` around the error message supplied

Desugaring do notation: way 2

- Haskell desugars using way 1 when pattern-match failures are impossible, and using way 2 when pattern-match failures can happen
- If pattern match failures can happen but the monad is not also an instance of `MonadFail`, it's a compile-time error

`let` inside `do` expressions

- `let` expressions (without an `in`) can be put inside `do` expressions
- Example:

```
do line <- getLine
   let line2 = "got line: " ++ line
   putStrLn line2
```

- How would this desugar?
- We'll use way 1 for convenience (no patterns on left-hand side of `<-` here)

`let` inside `do` expressions

```
do line <- getLine
   let line2 = "got line: " ++ line
   putStrLn line2
```

- desugars to:

```
getLine >>=
   \line ->
      let line2 = "got line: " ++ line in
        putStrLn line2
```

- `let` binding is in scope for the rest of the `do` expression

More about do expressions

- The last line of a `do` expression cannot be a binding
  - It must be an expression
- Consider

```
do line1 <- getLine
   line2 <- getLine
```

- How to desugar this?

```
getLine >>= \line1 -> (getLine >>= \line2 -> ???)
```

- This doesn't make sense, so not allowed
- Get this error message:

```
The last statement in a 'do' block must be an
expression
```

More about do expressions

- We can use a more explicit syntax instead of relying on whitespace in `do` expressions:

```
do { line1 <- getLine ;
     line2 <- getLine ;
     putStrLn (line1 ++ line2) }
```

- Some Haskell programmers write this as:

```
do { line1 <- getLine
   ; line2 <- getLine
   ; putStrLn (line1 ++ line2)
   }
```

Next time

- Practical interlude, part 2
- Writing and compiling stand-alone programs
- `ghci` and the `IO` monad
- References: `IORef`
- Coming up later: arrays: `Array` and `IOArray`