CS 115
Functional Programming

Lecture 9:

Monads, part 1

[Conceptual Introduction]

Today

- Introduction to monads
- Motivation
- Pure vs. impure functions
- Function composition and application
- "Notions of computation"

Introduction

- So far, we've been covering "core Haskell"
- Still don't know how to do some important things
- Example: input/output (I/O)
- Haskell uses a concept called *monads* to deal with input/output and other imperative features
- However, monads have *much* wider applicability than just emulating imperative coding idioms
- We will spend several lectures dissecting monads in great detail

Don't Panic!

- Monads cause a great deal of grief to new Haskell programmers
- I believe this is because they are rarely if ever explained properly
- Monads are not "hard" but they are very abstract
- We will take our time, and gain a comprehensive understanding

Background

- The concept of a monad comes to Haskell from Category Theory (CT)
  - like functors (previous lecture)
  - However, we won't require that you learn CT in order to learn Haskell monads
- Monads brought into Haskell by Eugenio Moggi (indirectly) and Philip Wadler (directly)
- Used to resolve a basic tension between pure and impure code

Background

- Haskell is a *pure* functional language
  - Functions are "referentially transparent"
  - A particular input to a function *always* gives the *same* output
  - *i.e.* functions behave like mathematical functions
- Some kinds of programming are difficult/impossible to do in a pure language
  - notably, I/O
- Pure languages like Haskell have struggled to come up with a way to do I/O while retaining purity

The problem

- Consider a function that reads a string input by the user on the terminal, and returns it
- What would be the type signature of such a function?
- Inputs: none (could use `()`)
- Output: `String`
- So, naively, type could be `() -> String`
- However, this is not a pure function!
  - each time called, returns a different `String`
- Do we have to abandon purity to do I/O?

Copping out

- Most functional languages cop out at this point
- They support imperative features for those kinds of programming where it's most "natural" to have impure functions
- Examples: Scheme, OCaml, Erlang, Scala
  - basically all functional languages but Haskell
  - (some minor languages like Clean are pure too)

Copping out: the problem

- Allowing impure functions certainly makes life simpler for someone coming to a language from an imperative background
- Unfortunately, imperative code tends to pollute the functional goodness of purely functional code
- Calling an imperative function from otherwise-functional code means that the code isn't functional any more
- Worse, no way to *know* that some code is purely functional in the presence of imperative features

Not copping out: Haskell

- Haskell refuses to compromise on purity and the benefits that come with it
  - easy composability, equational reasoning, referential transparency
- So a method had to be found to get the effect of imperative code without allowing arbitrary impure functions
- Haskell's solution involves monads
  - Other solutions exist (*e.g.* Clean has "uniqueness types")

Not copping out: Haskell

- Monads will allow us to write arbitrary imperative code in Haskell
- Even better: the type system will *label* such code so that there will be no way to inadvertently mix imperative and functional code (won't type check!)
- So monads solve the I/O problem in Haskell
- But also: they have *many* other uses, unrelated to I/O or simulating imperative code in a pure functional language

What are monads?

- "Executive summary" of what monads are in Haskell:

Monads are a generalization of functions, function application, and function composition to allow them to deal with richer notions of computation than standard functions.

Notions of computation

- What is a "notion of computation"?
- The most familiar one is *pure functions*
- A pure function takes in one or more inputs
  - Currying allows us to only consider functions of a single argument
- Always generates the *same* output given the same inputs
- In Haskell notation, we write such functions as:

```
f :: a -> b   -- for types a and b
```

Notions of computation

- Pure functions are *not* the only notion of computation
- All notions of computation involve function-like entities which take an input value of type `a` and produce an output value of type `b`
- However, other things may occur in the process of mapping the inputs to the outputs
- What are some of the alternative notions of computation?

Notions of computation

- In addition to mapping an input of type `a` to an output of type `b`, some function-like entities may
  - do file or terminal input/output
  - raise exceptions
  - read and/or write to/from local or global state variables
  - fail (not produce any results in some circumstances)
  - produce more than one result

In conventional languages

- Conventional programming languages (C, Java, Python) make no attempts to be "pure" so any "function" can support any notion of computation
- Advantage: simplifies the languages
- Disadvantage: no way to know if a function is pure, or if it isn't, what kind of extra notions of computation are embodied by the function
- In C/Java/Python, *all* functions can do terminal/file I/O, can raise exceptions, can fail, can access/modify state, *etc.*

In conventional languages

- Put differently: conventional languages do not support the mathematical notion of *pure functions*
- All functions can do arbitrary "extra stuff" in addition to mapping the input values to output values
- Benefits of supporting pure functions…
  - easy composability
  - easy debuggability
  - predictable behavior (referential transparency)
- …are thus lost

Monads

- Monads will provide a *controlled* way to model arbitrary notions of computation
- Monadic functions can embody single or multiple alternative notions of computation
- You only need to "dress up" a function with the specific extra behaviors it needs
  - *e.g.* allowing file/terminal I/O only, or allowing exceptions to be thrown only, or both

Monads

- Furthermore, monads will provide us with ways to compose monadic functions that are as simple as composing pure functions!
- This is the real payoff of monads: *allowing alternative notions of computation to be usable as conveniently as pure functions*
- Before we get into that, we need to review some basics

Function application

- In Haskell, function application is done by juxtaposing a function with its arguments

```
f x  -- apply function f to value x
```

- We can also use the `$` operator:

```
($) :: (a -> b) -> a -> b
f $ x     -- apply function f to value x
($) f x   -- same
```

Function application

- If we wanted to, we could define a "reverse apply" operator:

```
(>$>) :: a -> (a -> b) -> b
x >$> f = f x  -- or: (>$>) = flip ($)
```

- This is analogous to pipes in Unix shells:
  - produce some data
  - apply a function (program) to the produced data
- This is just a convenience

Function composition

- Recall how function composition is defined in Haskell:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

- You could also define a "reverse composition" operator:

```
(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
f >.> g = g . f  -- or: (>.>) = flip (.)
```

- More natural to use this instead of `.` in many situations

Function composition

- Benefit of function composition is that we can take existing functions and "snap them together" to form new ones very easily
- Not having to name the arguments to the function is particularly handy
- Consider what would happen if arguments needed to be explicitly named at all times
- Example: chaining together 10 functions of type (`Int -> Int`) to form an 11th

Function composition

- Without function composition:

```
f11 x =
  let x2 = f1 x
      x3 = f2 x2
      x4 = f3 x3
      x5 = f4 x4
      x6 = f5 x5
      x7 = f6 x6
      x8 = f7 x7
      x9 = f8 x8
      x10 = f9 x9
      x11 = f10 x10
in x11
```

Function composition

- With function composition:

```
f11 = f10 . f9 . f8 . f7 . f6 . f5 . f4 . f3 . f2 . f1
```

- Or:

```
f11 = f1 >.> f2 >.> f3 >.> f4 >.> f5
      >.> f6 >.> f7 >.> f8 >.> f9 >.> f10
```

- In either case, function composition makes it much simpler to create new functions from old ones

Monadic functions

- Before we talked about different "notions of computation"
- A function embodying a notion of computation (other than a pure function) will be called a "monadic function"
  - my terminology, not standard
- We need to understand
  - what these functions are like
  - how to compose them

Monadic functions

- A *monadic function* is a function which, in addition to the usual role of taking a specific value of the input type **`a`** and returning a specific value of the output type **`b`**, also has some other computational effect
- We will write such effects as **`E`**, **`E1`**, **`E2`**, etc.
- We can write a monadic function's "type" schematically as

```
a ----[E]---> b
```

- for some effect **`E`**

Monadic functions

- Recall: "effects" may include
  - doing file/terminal I/O
  - raising exceptions
  - failing
  - returning multiple values
  - interacting with local or global state
- Each of these has a particular kind of effect `E`
- [This is not Haskell yet!]
- Example effect: file/terminal I/O; effect is called `IO`

Monadic functions

- Conceptually, monadic functions with the `IO` effect have type signatures like this:

```
a ----[IO]----> b
```

- However, this is not legal Haskell syntax!
- Haskell requires that the "IO-ness" of this function (more generally, the "monad-ness" of a monadic function) be put into either the input type or the return type of the function

Monadic functions

- So instead of this:

```
a ----[IO]----> b
```

- A monadic type signature could only be one of:

```
a -> IO b
```

- or:

```
IO a -> b
```

- Note: Either way, `IO` must be what?
  - a (unary) type constructor!
  - kind: `* -> *`

Monadic functions

- Haskell actually uses types of this form to represent I/O effects:

```
a -> IO b
```

- or, more generally for a monad (type constructor) `m`:

```
a -> m b
```

- The other form:

```
m a -> b
```

- also exists in Haskell and is called a "comonad" (much less used)

Monadic values

- Notice that this type signature for a monadic function:

```
a -> m b
```

- is essentially describing such a function as a regular function from a normal Haskell type `a` to a "monadic value" of type `m b`
- Problem: the notion of a "monadic function" is fairly intuitive (a function with some extra computational effect)
- But the notion of a "monadic value" is *not* intuitive
- And yet, monadic values are what we actually work with

Monadic values

- The non-intuitiveness of monadic values causes endless problems for new Haskell programmers
- Makes it very hard to answer the question "What is a monad?" because the asker usually really wants to know "What does a monadic value signify?"
- Many Haskell tutorials make this worse by using poor analogies: a monad is like a box, a burrito, a nuclear reactor, a fire truck, a pogo stick, you name it...

Monadic values

- Remember: the intuitive concept is a *monadic function* (a function with effects)
- *Monadic values* are a kind of artifact of the way that Haskell represents monadic functions
- Therefore, monadic values don't have to represent anything particularly intuitive
- Nevertheless, there is one not totally bad way of intuiting what a monadic value in Haskell represents: a kind of "action"

Monadic values as "actions"

- Let's try to describe what Haskell programmers mean when they speak of "actions"
- Consider a monadic function `f` in a monad (type constructor) `m` with this type signature:

```
f :: a -> m b
```

- If the monad `m` was the `IO` monad, this becomes:

```
f :: a -> IO b
```

- This is the type signature of a function which transforms a value of type `a` to a value of type `b`, possibly also doing some file or terminal I/O

Monadic values as "actions"

```
f :: a -> IO b
```

- This should be fairly intuitive
- If `f` is applied to a value of type `a`, we get a value of type `IO b`:

```
x :: a
```

```
f x :: IO b
```

- What kind of intuition can we attach to the monadic value `f x`?

Monadic values as "actions"

- Consider the very simple function `g`:

```
g :: a -> () -> a
g x () = x
```

- What `g` does is take a value and wrap it into a function which takes the unit value `()` and returns the original value
- Consider

```
g (f x)
```

- where `f x` is the monadic value on the previous slide

Monadic values as "actions"

- Type signature of `g (f x)`:

```
g (f x) :: () -> IO b
```

- This is the type signature of a monadic function which transforms a value of type `()` to a value of type `b`, possibly doing some I/O in the process

Monadic values as "actions"

- The point: if you need to have some kind of intuition for what a monadic value "is", about as close as you can get is this: it's like a monadic function that maps a meaningless value to a value of the return type, performing some effect along the way
- This is what we refer to as a monadic "action"
- Kind of like a function, but without a function type or input argument
- For `IO`, we say an `IO` value is an "action" which may do some I/O and then "returns" a value of some type `b`

Example

- Consider the "functions" to read and write a line from the terminal and their types:

```
getLine :: IO String
putStrLn :: String -> IO ()
```

- `getLine` is a monadic value (monadic "action") which reads a line from the terminal and "returns" it
- `putStrLn` is a monadic function which takes a string, prints it along with a newline, and returns the `()` value

Example

- In a more conventional language, these would be functions with these types:

```
getLine :: () -> String  -- not Haskell
putStrLn :: String -> () -- not Haskell
```

- Here, **`getLine`** takes the **`()`** (unit) value (of no significance), reads a line from the terminal, and returns it as a **`String`**
- **`putStrLn`** takes a **`String`**, prints it to the terminal with a newline, and returns the **`()`** (unit) value (of no significance)

Example

- The only purpose of the `()`s in:

```
getLine :: () -> String  -- not Haskell
putStrLn :: String -> () -- not Haskell
```

- is to make sure that `getLine` and `putStrLn` are actually functions
- Without the `()`s, they would become:

```
getLine :: String
putStrLn :: String
```

- which is clearly wrong
- `()` types make sure the functions have well-defined inputs and outputs

Example

- In Haskell, though, we have:

```
getLine :: IO String
```

```
putStrLn :: String -> IO ()
```

- `putStrLn`'s type signature makes sense
  - a monadic function mapping a `String` to `()`, doing some I/O along the way
- `getLine`'s type signature is a bit mysterious
- If we think of it as `() -> IO String`, not mysterious anymore!
- Maps an irrelevant value `()` to a `String`, doing some I/O along the way

Example

- However, **`getLine`**'s type signature is **`IO String`**, *not* **`() -> IO String`**
- Therefore: a monadic value AKA monadic "action" is like a monadic function with an *implicit input argument* of type **`()`**
- This is about as much meaning as you can give to monadic values/actions
- We often informally say "**`getLine`** is an action which does some I/O and returns a **`String`**"

Next time

- Monads, part 2
- The `Monad` type class in Haskell
- The two fundamental monadic operations:
  - the `return` function
  - the `>>=` (bind) operator