# Munchy Monads

By Jonathan Louie

# Monads are Confusing

- Monad tutorial fallacy: http://wadler.blogspot.com/2013/11/the-monad-tutorial-fallacy.html
- Many people confuse the FP definition with the category theory definition they are based on
  - https://en.wikipedia.org/wiki/Monad_(functional_programming)
  - https://en.wikipedia.org/wiki/Monad_(category_theory)

# Why Monads?

```
def foo(): Option[Int] = Some(1)

def bar(): Option[Int] = Some(2)

def baz(a: Int, b: Int): Option[Int] = Some(a + b)

def getResult(): Option[Int] = {
  foo() match {
    case Some(a) => {
      bar() match {
        case Some(b) => baz(a, b)
        case None => None
      }
    }
    case None => None
  }
}
```

# With Monads

```
def getResult(): Option[Int] = {
  foo().flatMap(a =>
    bar().flatMap(b =>
      baz(a, b).map(c => c * 2)))
}
```

# ...and with Sugar

```
def getResult(): Option[Int] = {
  for {
    a <- foo()
    b <- bar()
    c <- baz(a, b)
  } yield c * 2
}
```

# What is a Monad?

- “Just” an interface supporting a “pure/return” function and a “bind/flatMap” function
  - Multiple ways to define a Monad typeclass, but this is the most common
  - Comes with 3 monad laws, but they are not enforced
- Allows us to generalize a very common pattern we see
- Naïve definition:

```
trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A], f: A => F[B]): F[B]
  def pure[A](a: A): F[A]
}
```

# Why don't we talk about them?

- Let’s try to define this interface in Rust:

```
trait Monad<A> {
    fn flat_map<B>(&self, f: fn(A) -> ???) -> ???;
}
```

- We need “Higher-Kinded Types” to talk about the type constructor (`F<_>`) needed

# Types vs. Type Constructors

- Is `Option` a type?
  - No, it's a type constructor
- A data constructor takes a value and constructs a new value
- A type constructor takes a type and constructs a new type
- Example: `Option<u32>` is the type produced by applying the `Option` type constructor to `u32`

# Hidden in Plain Sight

```
async fn foo() -> u32 { 1 }
async fn bar() -> u32 { 2 }
async fn baz(a: u32, b: u32) -> u32 { a + b }

async fn get_result() -> u32 {
    let a = foo().await;
    let b = bar().await;
    let c = baz(a, b).await;
    c * 2
}
```

# Comparison

Rust:

```rust
fn get_result() -> Option<u32> {
    let a = foo()?;
    let b = bar()?;
    let c = baz(a, b)?;
    Some(c * 2)
}
```

Scala:

```scala
def getResult(): Option[Int] = {
  for {
    a <- foo()
    b <- bar()
    c <- baz(a, b)
  } yield c * 2
}
```

# Questions?

# Thanks for listening!