Monoids explained to an imperative programmer

Mateusz Kowalczyk

University of Bath

April 29, 2013

Quick overview

Originally this talk was going to be about dependent types.

Quick overview

Originally this talk was going to be about dependent types.
It turned out that I was running out of time before even getting into regular typing.

Quick overview

Originally this talk was going to be about dependent types.
It turned out that I was running out of time before even getting into regular typing.
Changed the topic to something that builds upon knowledge you most likely already have: imperative programming.

Quick overview

Originally this talk was going to be about dependent types.
It turned out that I was running out of time before even getting into regular typing.
Changed the topic to something that builds upon knowledge you most likely already have: imperative programming.
This talk is going to essentially be a cut-down, partial, 10-minute version of the excellent presentation given by Brian Beckman titled "Don't fear the Monad".

Quick overview

Originally this talk was going to be about dependent types.
It turned out that I was running out of time before even getting into regular typing.
Changed the topic to something that builds upon knowledge you most likely already have: imperative programming.
This talk is going to essentially be a cut-down, partial, 10-minute version of the excellent presentation given by Brian Beckman titled “Don't fear the Monad”.
Go watch it, it's awesome.

Structure of this presentation

- Monoid definition and simple example

Structure of this presentation

- Monoid definition and simple example
- Functions

Structure of this presentation

- Monoid definition and simple example
- Functions
  - more friendly notation and function composition

Structure of this presentation

- Monoid definition and simple example
- Functions
  - more friendly notation and function composition
- Monoid under composition example

Structure of this presentation

- Monoid definition and simple example
- Functions
  - more friendly notation and function composition
- Monoid under composition example
- Why would we ever want to abstract to a monoid?

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

- Closure

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

- Closure
  - $\forall x, y \in S : x \cdot y \in S$

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

- Closure
  - $\forall x, y \in S : x \cdot y \in S$
- Associativity

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

- Closure
  - $\forall x, y \in S : x \cdot y \in S$
- Associativity
  - $\forall x, y, z \in S : (x \cdot y) \cdot z = x \cdot (y \cdot z)$

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

- Closure
  - $\forall x, y \in S : x \cdot y \in S$
- Associativity
  - $\forall x, y, z \in S : (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Identity

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

- Closure
  - $\forall x, y \in S : x \cdot y \in S$
- Associativity
  - $\forall x, y, z \in S : (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Identity
  - $\exists e \in S : \forall x \in S : e \cdot x = x \cdot e = x$

A monoid is a set S along with a binary operation $\cdot$ satisfying three simple laws:

- Closure
  - $\forall x, y \in S : x \cdot y \in S$
- Associativity
  - $\forall x, y, z \in S : (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Identity
  - $\exists e \in S : \forall x \in S : e \cdot x = x \cdot e = x$
  - Monoids where $x \cdot y = y \cdot x$ are called commutative monoids

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity
  - $\forall x, y, z \in \mathbb{Z} : (x + y) + z = x + (y + z)$

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity
  - $\forall x, y, z \in \mathbb{Z} : (x + y) + z = x + (y + z)$
- Identity

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity
  - $\forall x, y, z \in \mathbb{Z} : (x + y) + z = x + (y + z)$
- Identity
  - $\forall x \in \mathbb{Z} : 0 + x = x + 0 = x$

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity
  - $\forall x, y, z \in \mathbb{Z} : (x + y) + z = x + (y + z)$
- Identity
  - $\forall x \in \mathbb{Z} : 0 + x = x + 0 = x$
  - Here 0 is our identity element

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity
  - $\forall x, y, z \in \mathbb{Z} : (x + y) + z = x + (y + z)$
- Identity
  - $\forall x \in \mathbb{Z} : 0 + x = x + 0 = x$
  - Here 0 is our identity element
  - Happens to be a commutative monoid thanks to addition

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity
  - $\forall x, y, z \in \mathbb{Z} : (x + y) + z = x + (y + z)$
- Identity
  - $\forall x \in \mathbb{Z} : 0 + x = x + 0 = x$
  - Here 0 is our identity element
  - Happens to be a commutative monoid thanks to addition

We can easily show that a set of all integers ($\mathbb{Z}$) is a monoid under addition: that is, we use $+$ for our $\cdot$.

- Closure
  - $\forall x, y \in \mathbb{Z} : x + y \in \mathbb{Z}$
- Associativity
  - $\forall x, y, z \in \mathbb{Z} : (x + y) + z = x + (y + z)$
- Identity
  - $\forall x \in \mathbb{Z} : 0 + x = x + 0 = x$
  - Here 0 is our identity element
  - Happens to be a commutative monoid thanks to addition

We can also very easily show that $\mathbb{Z}/\{0\}$ under multiplication is also a monoid: we use 1 as the identity element.

Here I'm going to define a notation that's more common in functional programming languages. The imperative language snippets are going to be in Java.

Here I'm going to define a notation that's more common in functional programming languages. The imperative language snippets are going to be in Java.

```java
public class SingleVar {
    int x;
}
```

Here I'm going to define a notation that's more common in functional programming languages. The imperative language snippets are going to be in Java.

```java
public class SingleVar {
    int x;
}
```

We see that x is given type *int*.
From here on, we shall write this down as $x : int$.

Here I'm going to define a notation that's more common in functional programming languages. The imperative language snippets are going to be in Java.

```
public class SingleVar {
    int x;
}
```

We see that x is given type int.
From here on, we shall write this down as $x : int$.
In general, for all types τ, $v : \tau$ is 'v is a member of type τ'.

Time for the same thing but on a function. I ask you to look past the fact that Java only has methods; making those methods static aims to lessen the difference.

Time for the same thing but on a function. I ask you to look past the fact that Java only has methods; making those methods static aims to lessen the difference.

```java
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

Time for the same thing but on a function. I ask you to look past the fact that Java only has methods; making those methods static aims to lessen the difference.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

Here we have a very simple function *f* that takes an *int* (and binds it to *x*) and returns an *int*.

Time for the same thing but on a function. I ask you to look past the fact that Java only has methods; making those methods static aims to lessen the difference.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

Here we have a very simple function f that takes an *int* (and binds it to x) and returns an *int*.
With the new notation, we can write the type of f as $f : int \to int$.

Time for the same thing but on a function. I ask you to look past the fact that Java only has methods; making those methods static aims to lessen the difference.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

Here we have a very simple function f that takes an *int* (and binds it to x) and returns an *int*.
With the new notation, we can write the type of f as $f : int \rightarrow int$.
Note how $x : int$ and $f : int \rightarrow int$ use the same notation.

Time for the same thing but on a function. I ask you to look past the fact that Java only has methods; making those methods static aims to lessen the difference.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

Here we have a very simple function f that takes an *int* (and binds it to x) and returns an *int*.
With the new notation, we can write the type of f as $f : int \to int$.
Note how $x : int$ and $f : int \to int$ use the same notation.
We also define $a \to b \to c \equiv (a \to b) \to c$, i.e. the type signature is left associative.

Time for simple function application. Consider the same code again.

Time for simple function application. Consider the same code again.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

Time for simple function application. Consider the same code again.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

We know that $f : int \to int$. Now consider any $v : int$. In Java, we might use the function like a = f(v);. What is the type of a? Well, we take a $f : int \to int$ and feed an int to it so we get an int back.

Time for simple function application. Consider the same code again.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

We know that $f : int \to int$. Now consider any $v : int$. In Java, we might use the function like `a = f(v);`. What is the type of a? Well, we take a $f : int \to int$ and feed an int to it so we get an int back. With our new notation, we shall write `f(v);` as $f\ v$.

Consider the following.

```
public class TwoBasicFunc {
    public static int f(int x) {
        return x + 1;
    }

    public static int g(int x) {
        return x * 2;
    }
}
```

Consider the following.

```
public class TwoBasicFunc {
    public static int f(int x) {
        return x + 1;
    }

    public static int g(int x) {
        return x * 2;
    }
}
```

We have $f : int \to int$ and $g : int \to int$.

Consider the following.

```
public class TwoBasicFunc {
    public static int f(int x) {
        return x + 1;
    }

    public static int g(int x) {
        return x * 2;
    }
}
```

We have $f : int \to int$ and $g : int \to int$.
Let's say we want to apply g first and then f to the result. In Java we'd right f(g(v));. With our new notation we write $f\ g\ v$. Note that application is left associative here.

Let's not stop on *int*s though and go generic.

```
public class Identity {

    public static <A> A id(A x) {
        return x;
    }

}
```

Let's not stop on *ints* though and go generic.

```
public class Identity {

    public static <A> A id(A x) {
        return x;
    }

}
```

Here we have a function $id : A \to A$; it's a trivial function that just returns its argument. What's important is that it accepts any type: $<$A$>$ means a generic type A in Java.

Let's not stop on *int*s though and go generic.

```
public class Identity {

    public static <A> A id(A x) {
        return x;
    }

}
```

Here we have a function $id : A \rightarrow A$; it's a trivial function that just returns its argument. What's important is that it accepts any type: $<$A$>$ means a generic type A in Java.
In our notation, this is simply $id : A \rightarrow A$: nothing special about it!

We can mix types without a problem! Consider $f : b \to c$, $g : a \to b$ and $x : a$.

We can mix types without a problem! Consider $f : b \to c$, $g : a \to b$ and $x : a$.
We can very easily see that $g\ x : b$. So what's the type of $f\ (g\ x)$?
Well, it's just c! $g\ x$ provides us a value of type b and we apply $f : b \to c$ to it.

We can mix types without a problem! Consider $f : b \to c$, $g : a \to b$ and $x : a$.
We can very easily see that $g\ x : b$. So what's the type of $f\ (g\ x)$?
Well, it's just c! $g\ x$ provides us a value of type b and we apply $f : b \to c$ to it.
Mind that we need the parenthesis in $f\ (g\ x)$ as we have defined function application to be left associative.
Without parenthesis, we'd get $f\ g\ x \equiv (f\ g)\ x$. But we can't have $f\ g$ because f takes a value of type b and g is of type $a \to b$.

Before we proceed, I'd like to draw attention to the fact that the type notation for 'variables' and for functions is basically the same.

Before we proceed, I'd like to draw attention to the fact that the type notation for 'variables' and for functions is basically the same. In fact, we can think of $x : a$ as a nullary function of type a: a function that takes no arguments! Consider a function with the following signature then: $applier : (a \to b) \to a \to b$. It might look different than all the previous ones but let's try to read it.

Before we proceed, I'd like to draw attention to the fact that the type notation for 'variables' and for functions is basically the same. In fact, we can think of $x : a$ as a nullary function of type a: a function that takes no arguments! Consider a function with the following signature then: $applier : (a \to b) \to a \to b$. It might look different than all the previous ones but let's try to read it. $(a \to b)$ is just a function that takes a value of type a and returns a value of type b. So what's $(a \to b) \to a \to b$?

Before we proceed, I'd like to draw attention to the fact that the type notation for 'variables' and for functions is basically the same. In fact, we can think of $x : a$ as a nullary function of type a: a function that takes no arguments! Consider a function with the following signature then: $\mathit{applier} : (a \to b) \to a \to b$. It might look different than all the previous ones but let's try to read it. $(a \to b)$ is just a function that takes a value of type a and returns a value of type b. So what's $(a \to b) \to a \to b$? Well, it's just a function that takes a value of type $a \to b$, a value of type a and returns a b. The first argument just happens to be a function!

Continuing from the previous slide, I can show that making such a function is fairly trivial. We need to define syntax for creating functions first though.

Continuing from the previous slide, I can show that making such a function is fairly trivial. We need to define syntax for creating functions first though.

```
public class BasicFunc {
    public static int f(int x) {
        return x + 1;
    }
}
```

We can write this function like this:

```
f :: int -> int
f x = x + 1
```

So going back to our $(a \to b) \to a \to b$ function, how can we make one? Let's call this function h:

```
h :: (a -> b) -> a -> b
h f x = f x
```

So going back to our $(a \to b) \to a \to b$ function, how can we make one? Let's call this function h:

```
h :: (a -> b) -> a -> b
h f x = f x
```

Simple stuff! It turns out that h is just function application that we've been doing all the time!

Isn't $h\ f\ x = f\ x$ pointless though? Why not just write $f\ x$?

Isn't $h\ f\ x = f\ x$ pointless though? Why not just write $f\ x$?
The idea is that we just wrote a function h that takes any function of type $a \to b$ and a value of type a. As we saw with id, a and b can be anything: the types are polymorphic.

When we previously did a $f\ (g\ x)$, what we really wanted is a function that does g and then f on x, in order.
Here $f : b \to c$, $g : a \to b$ and $x : a$.

When we previously did a $f\ (g\ x)$, what we really wanted is a function that does g and then f on x, in order.
Here $f : b \to c$, $g : a \to b$ and $x : a$.
So we want some $h : a \to c$. How can we produce it? This is where function composition steps in.

When we previously did a $f\ (g\ x)$, what we really wanted is a function that does g and then f on x, in order.
Here $f : b \to c$, $g : a \to b$ and $x : a$.
So we want some $h : a \to c$. How can we produce it? This is where function composition steps in.
What we need is a function h with a signature $(b \to c) \to (a \to b) \to (a \to c)$.
So: $h\ f\ g\ x = f\ (g\ x)$.

When we previously did a $f\ (g\ x)$, what we really wanted is a function that does g and then f on x, in order.
Here $f : b \to c$, $g : a \to b$ and $x : a$.
So we want some $h : a \to c$. How can we produce it? This is where function composition steps in.
What we need is a function h with a signature $(b \to c) \to (a \to b) \to (a \to c)$.
So: $h\ f\ g\ x = f\ (g\ x)$.
That's it! It's that simple. This operation is so common that we define $h\ f\ g\ x = f\ (g\ x)$ to be $(f \circ g)\ x$. So just $(f \circ g)$ is of type $a \to c$ and $(f \circ g)\ x$ is of type c.

When we previously did a $f\ (g\ x)$, what we really wanted is a function that does g and then f on x, in order.
Here $f : b \to c$, $g : a \to b$ and $x : a$.
So we want some $h : a \to c$. How can we produce it? This is where function composition steps in.
What we need is a function h with a signature $(b \to c) \to (a \to b) \to (a \to c)$.
So: $h\ f\ g\ x = f\ (g\ x)$.
That's it! It's that simple. This operation is so common that we define $h\ f\ g\ x = f\ (g\ x)$ to be $(f \circ g)\ x$. So just $(f \circ g)$ is of type $a \to c$ and $(f \circ g)\ x$ is of type c.
We can give this new function $(f \circ g)$ a name: $f' = f \circ g$.

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\, (g \circ h)\, x = f\, (g\, (h\, x))$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\ (g \circ h)\, x = f\ (g\ (h\, x))$
  - $\forall (x : \tau), \forall f, g, h \in F : f \circ (g \circ h)\, x = f\ ((g \circ h)\, x) = f\ (g\ (h\, x))$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\ (g \circ h)\, x = f\ (g\ (h\, x))$
  - $\forall (x : \tau), \forall f, g, h \in F : f \circ (g \circ h)\, x = f\ ((g \circ h)\, x) = f\ (g\ (h\, x))$
- Identity

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\, (g \circ h)\, x = f\, (g\, (h\, x))$
  - $\forall (x : \tau), \forall f, g, h \in F : f \circ (g \circ h)\, x = f\, ((g \circ h)\, x) = f\, (g\, (h\, x))$
- Identity
  - $\forall f \in F : id \circ f = f \circ id = f$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\, (g \circ h)\, x = f\, (g\, (h\, x))$
  - $\forall (x : \tau), \forall f, g, h \in F : f \circ (g \circ h)\, x = f\, ((g \circ h)\, x) = f\, (g\, (h\, x))$
- Identity
  - $\forall f \in F : id \circ f = f \circ id = f$
  - Here $id\ x = x$ is our identity element. id is polymorphic so it gets the type $(\tau \to \tau) \to (\tau \to \tau)$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\, (g \circ h)\, x = f\, (g\, (h\, x))$
  - $\forall (x : \tau), \forall f, g, h \in F : f \circ (g \circ h)\, x = f\, ((g \circ h)\, x) = f\, (g\, (h\, x))$
- Identity
  - $\forall f \in F : id \circ f = f \circ id = f$
  - Here $id\ x = x$ is our identity element. id is polymorphic so it gets the type $(\tau \to \tau) \to (\tau \to \tau)$
  - $\forall (x : \tau), \forall f \in F : (f \circ id)\, x = f\, (id\ x) = f\ x$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\ (g \circ h)\, x = f\ (g\ (h\, x))$
  - $\forall (x : \tau), \forall f, g, h \in F : f \circ (g \circ h)\, x = f\ ((g \circ h)\, x) = f\ (g\ (h\, x))$
- Identity
  - $\forall f \in F : id \circ f = f \circ id = f$
  - Here $id\ x = x$ is our identity element. id is polymorphic so it gets the type $(\tau \to \tau) \to (\tau \to \tau)$
  - $\forall (x : \tau), \forall f \in F : (f \circ id)\, x = f\ (id\ x) = f\ x$
  - $\forall (x : \tau), \forall f \in F : (id \circ f)\, x = id(f\ x) = f\ x$

Having talked about functions and composition, it's time to put them to use. Consider a set of all functions of type $\tau \to \tau$, F. We can form a monoid using $\circ$ as $\cdot$.

- Closure
  - $\forall f, g \in F : f \circ g \in F$
  - $((f : \tau \to \tau) \circ (g : \tau \to \tau)) : \tau \to \tau$
- Associativity
  - $\forall f, g, h \in F : (f \circ g) \circ h = f \circ (g \circ h)$
  - $\forall (x : \tau), \forall f, g, h \in F : (f \circ g) \circ h\, x = f\ (g \circ h)\, x = f\ (g\ (h\, x))$
  - $\forall (x : \tau), \forall f, g, h \in F : f \circ (g \circ h)\, x = f\ ((g \circ h)\, x) = f\ (g\ (h\, x))$
- Identity
  - $\forall f \in F : id \circ f = f \circ id = f$
  - Here $id\ x = x$ is our identity element. id is polymorphic so it gets the type $(\tau \to \tau) \to (\tau \to \tau)$
  - $\forall (x : \tau), \forall f \in F : (f \circ id)\ x = f\ (id\ x) = f\ x$
  - $\forall (x : \tau), \forall f \in F : (id \circ f)\ x = id(f\ x) = f\ x$
  - Mind that by definition of id, $id\ f\ x = f\ x$ and $f\ (id\ x) = f\ x$. Not applying to x, we just get f in both cases.

It's important to note that for functions to form a monoid under composition, the functions must be of a uniform type: that is, for any function f^n where $f^0 : a$, $f^1 : a \to a$, f^k takes k arguments of type a.

It's important to note that for functions to form a monoid under composition, the functions must be of a uniform type: that is, for any function f^n where $f^0 : a$, $f^1 : a \to a$, f^k takes k arguments of type a.
This means that a set of all functions $g : a \to b$ does not form a monoid under composition where $a \neq b$: it is only in a monoidal category; we can't compose two $a \to b$ functions together.

It's important to note that for functions to form a monoid under composition, the functions must be of a uniform type: that is, for any function f^n where $f^0 : a$, $f^1 : a \to a$, f^k takes k arguments of type a.
This means that a set of all functions $g : a \to b$ does not form a monoid under composition where $a \neq b$: it is only in a monoidal category; we can't compose two $a \to b$ functions together.

So, what's the point of monoids?

So, what's the point of monoids?

- We get a *fold* operation for free.

So, what's the point of monoids?

- We get a *fold* operation for free.
  - Python's *reduce* with the unit used as the initial value

So, what's the point of monoids?

- We get a *fold* operation for free.
  - Python's *reduce* with the unit used as the initial value
- For functions, it ensures that whatever we do, we stay in the monoid

So, what's the point of monoids?

- We get a *fold* operation for free.
  - Python's *reduce* with the unit used as the initial value
- For functions, it ensures that whatever we do, we stay in the monoid
  - Gives us assurance about what we can do

So, what's the point of monoids?

- We get a *fold* operation for free.
  - Python's *reduce* with the unit used as the initial value
- For functions, it ensures that whatever we do, we stay in the monoid
  - Gives us assurance about what we can do
  - Composing small parts is the way to control complexity

So, what's the point of monoids?

- We get a *fold* operation for free.
  - Python's *reduce* with the unit used as the initial value
- For functions, it ensures that whatever we do, we stay in the monoid
  - Gives us assurance about what we can do
  - Composing small parts is the way to control complexity
- Only a step away from the ever powerful monads!

Here's output from a proof of concept monoid code written in Java.

```
Folding left: [2, 3, 4]
9
Folding right: [2, 3, 4]
9
Folding left [Why, , hello , world!]
Why, hello world!
Folding right [Why, , hello , world!]
Why, hello world!
Folding left [f x = x * x, f x = x + 5, f x = x * 3]
Applying the result of the fold to 7: 676
Folding right [f x = x * x, f x = x + 5, f x = x * 3]
Applying the result of the fold to 7: 676
Folding left [[1, 3, 4], [25, 7, 16], [735, 17, 8]]
[1, 3, 4, 25, 7, 16, 735, 17, 8]
Folding right [[1, 3, 4], [25, 7, 16], [735, 17, 8]]
[1, 3, 4, 25, 7, 16, 735, 17, 8]
```

Questions

Hopefully I miraculously managed to fit in my assigned time.
Feel free to ask any questions or point out any mistakes.

Contact

mk440@bath.ac.uk; fuuzetsu@fuuzetsu.co.uk
Get these slides and all code at
https://github.com/ShanaTsunTsunLove/foundations-talk

A quote from the #haskell IRC channel

* roconnor: where are all the category theoriest? why don't they already have all the answers for us?
* edwardk: roconnor: this is the point in your career where you look around for the cavalry and realize that you're it ;)