

Monoids explained to an imperative programmer

Mateusz Kowalczyk

University of Bath

April 23, 2013

Originally this talk was going to be about dependent types.

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.

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.

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”.

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.

Quick overview

Presentation structure

Structure of this presentation

- Functions

Quick overview

Presentation structure

Structure of this presentation

- Functions
 - more friendly notation and function composition

Quick overview

Presentation structure

Structure of this presentation

- Functions
 - more friendly notation and function composition
- Generic monoid definition

Structure of this presentation

- Functions
 - more friendly notation and function composition
- Generic monoid definition
- Simple monoid example (set of integers under addition)

Structure of this presentation

- Functions
 - more friendly notation and function composition
- Generic monoid definition
- Simple monoid example (set of integers under addition)
- List monoid example

Structure of this presentation

- Functions
 - more friendly notation and function composition
- Generic monoid definition
- Simple monoid example (set of integers under addition)
- List monoid example
- Why would we ever want to abstract to a monoid?

Functions

Notation

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 we have a simple method from *int* to *int*

Functions

Notation

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 we have a simple method from *int* to *int*

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

Functions

Notation

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 we have a simple method from *int* to *int*

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

We see that x is given type *int*.

From here on, we shall write this down as $x : \textit{int}$.

Functions

Notation

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 we have a simple method from *int* to *int*

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

We see that x is given type *int*.

From here on, we shall write this down as $x : \textit{int}$.

In general, for all types τ , $v : \tau$ is 'v is a member of type τ '.

Functions

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.

Functions

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;  
    }  
}
```

Functions

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*.

Functions

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 $f : \text{int} \rightarrow \text{int}$.

Functions

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 $f : \text{int} \rightarrow \text{int}$. Note how $x : \text{int}$ and $f : \text{int} \rightarrow \text{int}$ use the same notation.

Functions

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 $f : \text{int} \rightarrow \text{int}$. Note how $x : \text{int}$ and $f : \text{int} \rightarrow \text{int}$ use the same notation. We also define $a \rightarrow b \rightarrow c \equiv (a \rightarrow b) \rightarrow c$, i.e. the type signature is left associative.

Functions

Notation

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

Functions

Notation

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

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


Functions

Notation

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

Functions

Notation

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

Functions

Composition

Consider the following.

```
public class TwoBasicFunc {  
    public static int f(int x) {  
        return x + 1;  
    }  
  
    public static int g(int x) {  
        return x * 2;  
    }  
}
```

Functions

Composition

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 $f : \text{int} \rightarrow \text{int}$ and $g : \text{int} \rightarrow \text{int}$.

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

Functions

Composition

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

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

Functions

Composition

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 $id : A \rightarrow A$; it's a trivial function that just returns its argument. What's important is that it accepts any type: $\langle A \rangle$ means a generic type A in Java.

Functions

Composition

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 $id : A \rightarrow A$; it's a trivial function that just returns its argument. What's important is that it accepts any type: $\langle A \rangle$ means a generic type A in Java.

In our notation, this is simply $id : A \rightarrow A$: nothing special about it!

Functions

Composition

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

Functions

Composition

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

Functions

Composition

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

Functions

Composition

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.

Functions

Composition

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* : $(a \rightarrow b) \rightarrow a \rightarrow b$. It might look different than all the previous ones but let's try to read it.

Functions

Composition

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: $applier : (a \rightarrow b) \rightarrow a \rightarrow b$. It might look different than all the previous ones but let's try to read it. $(a \rightarrow b)$ is just a function that takes a value of type a and returns a value of type b . So what's $(a \rightarrow b) \rightarrow a \rightarrow b$? Well, it's just a function that takes a value of type $a \rightarrow b$, a value of type a and returns a b . The first argument just happens to be a function!

Functions

Composition

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: $applier : (a \rightarrow b) \rightarrow a \rightarrow b$. It might look different than all the previous ones but let's try to read it. $(a \rightarrow b)$ is just a function that takes a value of type a and returns a value of type b . So what's $(a \rightarrow b) \rightarrow a \rightarrow b$? Well, it's just a function that takes a value of type $a \rightarrow b$, a value of type a and returns a b . The first argument just happens to be a function!

Functions

Composition

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.

Functions

Composition

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
```


Functions

Composition

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
```

Functions

Composition

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

$$h :: (a \rightarrow b) \rightarrow a \rightarrow b$$
$$h\ f\ x = f\ x$$

Functions

Composition

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

$$h :: (a \rightarrow b) \rightarrow a \rightarrow b$$

$$h\ f\ x = f\ x$$

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 $f x$?

Functions

Composition

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 $a \rightarrow b$ and a value of type a . As we saw with id , a and b can be anything: the types are polymorphic.

Functions

Composition

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 $f : b \rightarrow c$, $g : a \rightarrow b$ and $x : a$.

Functions

Composition

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 $f : b \rightarrow c$, $g : a \rightarrow b$ and $x : a$. So we want some $h : a \rightarrow c$. How can we produce it? This is where function composition steps in.

Functions

Composition

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 $f : b \rightarrow c$, $g : a \rightarrow b$ and $x : a$. So we want some $h : a \rightarrow c$. How can we produce it? This is where function composition steps in.

What we need is a function h with a signature $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$.

So: $h f g x = f (g x)$.

Functions

Composition

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 $f : b \rightarrow c$, $g : a \rightarrow b$ and $x : a$. So we want some $h : a \rightarrow c$. How can we produce it? This is where function composition steps in.

What we need is a function h with a signature $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$.

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 $a \rightarrow c$ and $(f \circ g)\ x$ is of type c .

Functions

Composition

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 $f : b \rightarrow c$, $g : a \rightarrow b$ and $x : a$. So we want some $h : a \rightarrow c$. How can we produce it? This is where function composition steps in.

What we need is a function h with a signature $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$.

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 $a \rightarrow c$ and $(f \circ g)\ x$ is of type c .

We can give this new function $(f \circ g)$ a name: $f' = f \circ g$.

Monoids

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. A monoid is a set S along with a binary operation \cdot satisfying three simple laws:

Monoids

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. A monoid is a set S along with a binary operation \cdot satisfying three simple laws:

- Closure

Monoids

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. A monoid is a set S along with a binary operation \cdot satisfying three simple laws:

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

Monoids

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. 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

Monoids

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. 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
 - $\forall x, y, z \in S : (x \cdot y) \cdot z = x \cdot (y \cdot z)$

Monoids

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. 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

Monoids

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. 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

Monoid definition

Now that we have a sane notation and function composition, we can finally define a monoid. 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

Monoids

\mathbb{Z} under 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 .

Monoids

\mathbb{Z} under 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

Monoids

\mathbb{Z} under 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

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

Monoids

\mathbb{Z} under 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

Monoids

\mathbb{Z} under 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

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

Monoids

\mathbb{Z} under 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

Monoids

\mathbb{Z} under 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
 - $\forall x \in \mathbb{Z} : 0 + x = x + 0 = x$

Monoids

\mathbb{Z} under 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

Monoids

\mathbb{Z} under 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

Monoids

\mathbb{Z} under 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

Monoids

\mathbb{Z} under 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.

Monoids

Functions under composition

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

Monoids

Functions under composition

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

- Closure

Monoids

Functions under composition

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

- Closure

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

Monoids

Functions under composition

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

- Closure

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

Monoids

Functions under composition

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

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

Monoids

Functions under composition

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

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

Monoids

Functions under composition

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

- Closure

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

- Associativity

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

Monoids

Functions under composition

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

- Closure

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

- Associativity

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

Monoids

Functions under composition

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

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

Monoids

Functions under composition

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

- Closure
 - $\forall f, g \in F : f \circ g \in F$
 - $((f : \tau \rightarrow \tau) \circ (g : \tau \rightarrow \tau)) : \tau \rightarrow \tau$
- 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
 - $\forall f \in F : id \circ f = f \circ id = f$

Monoids

Functions under composition

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

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

Monoids

Functions under composition

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

- Closure

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

- 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 $(\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$
- $\forall (x : \tau), \forall f \in F : (f \circ id) \ x = f \ (id \ x) = f \ x$

Monoids

Functions under composition

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

- Closure
 - $\forall f, g \in F : f \circ g \in F$
 - $((f : \tau \rightarrow \tau) \circ (g : \tau \rightarrow \tau)) : \tau \rightarrow \tau$
- 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 $(\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$
 - $\forall (x : \tau), \forall f \in F : (f \circ id) \ x = f \ (id \ x) = f \ x$
 - $\forall (x : \tau), \forall f \in F : (id \circ f) \ x = id \ (f \ x) = f \ x$

Monoids

Functions under composition

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

- Closure
 - $\forall f, g \in F : f \circ g \in F$
 - $((f : \tau \rightarrow \tau) \circ (g : \tau \rightarrow \tau)) : \tau \rightarrow \tau$
- 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 $(\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$
 - $\forall (x : \tau), \forall f \in F : (f \circ id) \ x = f (id \ x) = f \ x$
 - $\forall (x : \tau), \forall f \in F : (id \circ f) \ x = id(f \ x) = f \ x$
 - 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.

Monoids

Small caveat

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 $f^0 : a$, $f^1 : a \rightarrow a$, f^k takes k arguments of type a .

Monoids

Small caveat

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 $f^0 : a$, $f^1 : a \rightarrow a$, f^k takes k arguments of type a .

This means that a set of all functions $g : a \rightarrow b$ does not form a monoid under composition where $a \neq b$: it is only in a monoidal category; we can't compose two $a \rightarrow b$ functions together.

Monoids

Small caveat

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 $f^0 : a$, $f^1 : a \rightarrow a$, f^k takes k arguments of type a .

This means that a set of all functions $g : a \rightarrow b$ does not form a monoid under composition where $a \neq b$: it is only in a monoidal category; we can't compose two $a \rightarrow b$ 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!

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 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 ;)