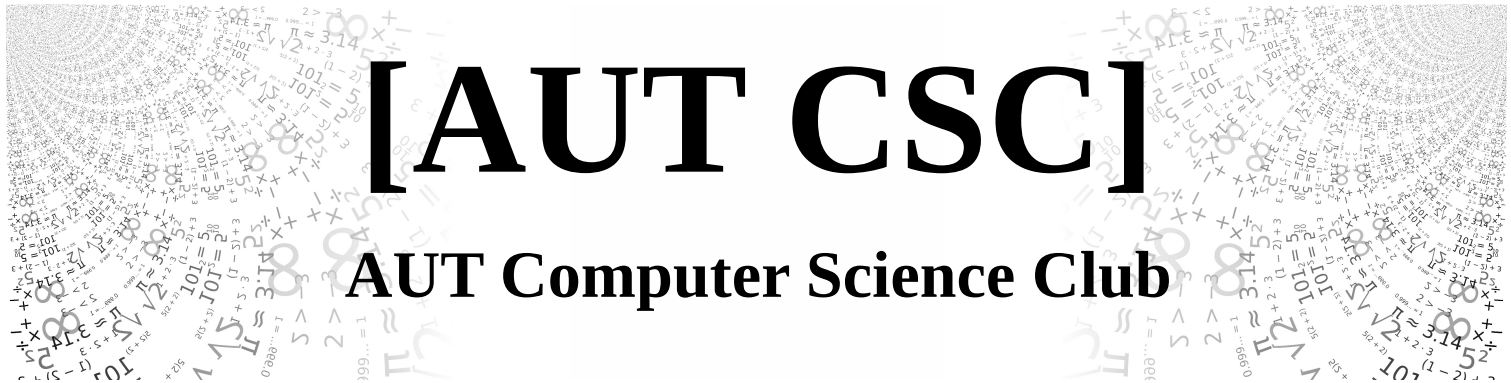
Functors

Or: why your abstractions are weak

Koz Ross

18th May, 2017



Outline

Things we want to have when programming

When we write code, all the following are pretty vital:

Things we want to have when programming

When we write code, all the following are pretty vital:

- Error handling (unlabelled and labelled)

Things we want to have when programming

When we write code, all the following are pretty vital:

- Error handling (unlabelled and labelled)
- Nondeterminism (multiple answers)

Things we want to have when programming

When we write code, all the following are pretty vital:

- Error handling (unlabelled and labelled)
- Nondeterminism (multiple answers)
- Side effects (file handling, GUIs, database operations,...)

Things we want to have when programming

When we write code, all the following are pretty vital:

- Error handling (unlabelled and labelled)
- Nondeterminism (multiple answers)
- Side effects (file handling, GUIs, database operations,...)
- Metadata handling (information about data)

Things we want to have when programming

When we write code, all the following are pretty vital:

- Error handling (unlabelled and labelled)
- Nondeterminism (multiple answers)
- Side effects (file handling, GUIs, database operations,...)
- Metadata handling (information about data)
- Containers (lists, dictionaries, trees,...)

Things we want to have when programming

When we write code, all the following are pretty vital:

- Error handling (unlabelled and labelled)
- Nondeterminism (multiple answers)
- Side effects (file handling, GUIs, database operations,...)
- Metadata handling (information about data)
- Containers (lists, dictionaries, trees,...)

These things are very different, aren't they?

Things we want to have when programming

When we write code, all the following are pretty vital:

- Error handling (unlabelled and labelled)
- Nondeterminism (multiple answers)
- Side effects (file handling, GUIs, database operations,...)
- Metadata handling (information about data)
- Containers (lists, dictionaries, trees,...)

These things are very different, aren't they? Your programming languages sure seem to think so...

The truth

The truth

Yes, your languages have been *lying* to you.

The fantastic functor

The *functor* is an abstraction which unifies *all* of the above: we can code to functors, and that code will *provably* work for every single one of those cases.

The fantastic functor

The *functor* is an abstraction which unifies *all* of the above: we can code to functors, and that code will *provably* work for every single one of those cases. Well, and plenty more besides...

The fantastic functor

The *functor* is an abstraction which unifies *all* of the above: we can code to functors, and that code will *provably* work for every single one of those cases. Well, and plenty more besides...

Functors have been known about for *over forty years*, and have been implemented in programming languages *since 1990*.

The fantastic functor

The *functor* is an abstraction which unifies *all* of the above: we can code to functors, and that code will *provably* work for every single one of those cases. Well, and plenty more besides...

Functors have been known about for *over forty years*, and have been implemented in programming languages *since 1990*. They are easy to understand, easy to use, and generally awesome.

The fantastic functor

The *functor* is an abstraction which unifies *all* of the above: we can code to functors, and that code will *provably* work for every single one of those cases. Well, and plenty more besides...

Functors have been known about for *over forty years*, and have been implemented in programming languages *since 1990*. They are easy to understand, easy to use, and generally awesome.

If your 'high-level' language doesn't have them, your language is *weak*.

The fantastic functor

The *functor* is an abstraction which unifies *all* of the above: we can code to functors, and that code will *provably* work for every single one of those cases. Well, and plenty more besides...

Functors have been known about for *over forty years*, and have been implemented in programming languages *since 1990*. They are easy to understand, easy to use, and generally awesome.

If your 'high-level' language doesn't have them, your language is *weak*. If you haven't been *taught* them, your *instructors* are weak (pointing no fingers, of course).

The fantastic functor

The *functor* is an abstraction which unifies *all* of the above: we can code to functors, and that code will *provably* work for every single one of those cases. Well, and plenty more besides...

Functors have been known about for *over forty years*, and have been implemented in programming languages *since 1990*. They are easy to understand, easy to use, and generally awesome.

If your 'high-level' language doesn't have them, your language is *weak*. If you haven't been *taught* them, your *instructors* are weak (pointing no fingers, of course).

Let's fix this, shall we?

Types

A *type* is a set of rules.

Types

A *type* is a set of rules. *Values* can have types (which indicate what we can do with them), but *variables* can have them too (and that tells us what values they can hold).

Types

A *type* is a set of rules. *Values* can have types (which indicate what we can do with them), but *variables* can have them too (and that tells us what values they can hold).

```
-- I'm a comment!
1 :: Int -- we're asserting that 1 is an Int
1 :: Real -- same look, different meaning
(3, 2.5) :: (Int, Real) -- a pair
["foo", "bar", "baz"] :: [String] --a list
```

Types

A *type* is a set of rules. *Values* can have types (which indicate what we can do with them), but *variables* can have them too (and that tells us what values they can hold).

```
-- I'm a comment!
1 :: Int -- we're asserting that 1 is an Int
1 :: Real -- same look, different meaning
(3, 2.5) :: (Int, Real) -- a pair
["foo", "bar", "baz"] :: [String] --a list

x :: Int -- the variable x holds ints (declaration)
x = 10 -- specifically this one (definition)
```

Types

A *type* is a set of rules. *Values* can have types (which indicate what we can do with them), but *variables* can have them too (and that tells us what values they can hold).

```
-- I'm a comment!
1 :: Int -- we're asserting that 1 is an Int
1 :: Real -- same look, different meaning
(3, 2.5) :: (Int, Real) -- a pair
["foo", "bar", "baz"] :: [String] --a list

x :: Int -- the variable x holds ints (declaration)
x = 10 -- specifically this one (definition)

plus :: Int -> Int -> Int -- functions have types too!
plus x y = x + y -- and definitions
```

Type parameterization

Instead of giving a type, we can use a *type parameter*:

Type parameterization

Instead of giving a type, we can use a *type parameter*:

```
-- take a value of any type, return one of the same
id :: a -> a
id x = x -- we're gonna see this one too!
```

Type parameterization

Instead of giving a type, we can use a *type parameter*:

```
-- take a value of any type, return one of the same
id :: a -> a
id x = x -- we're gonna see this one too!

-- we can have as many type parameters as we want!
swap :: (a, b) -> (b, a)
swap (x,y) = (y,x) -- just like you expected
```

Type parameterization

Instead of giving a type, we can use a *type parameter*:

```
-- take a value of any type, return one of the same
id :: a -> a
id x = x -- we're gonna see this one too!

-- we can have as many type parameters as we want!
swap :: (a, b) -> (b, a)
swap (x,y) = (y,x) -- just like you expected
```

We will typically use `a`,`b` for type parameters.

Our own types

We can define our own types by putting together descriptions of what they contain, based on existing types:

Our own types

We can define our own types by putting together descriptions of what they contain, based on existing types:

```
-- similar to a struct
-- type name on left, constructor name on right
data Name = Name { first :: String, last :: String }
```

Our own types

We can define our own types by putting together descriptions of what they contain, based on existing types:

```
-- similar to a struct
-- type name on left, constructor name on right
data Name = Name { first :: String, last :: String }

-- similar to an enum
data Colour = Red | Green | Blue
```

Our own types

We can define our own types by putting together descriptions of what they contain, based on existing types:

```
-- similar to a struct
-- type name on left, constructor name on right
data Name = Name { first :: String, last :: String }

-- similar to an enum
data Colour = Red | Green | Blue

-- we can mix these two forms

-- can be parametrized by type(s) if we wish
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Our own types

We can define our own types by putting together descriptions of what they contain, based on existing types:

```
-- similar to a struct
-- type name on left, constructor name on right
data Name = Name { first :: String, last :: String }

-- similar to an enum
data Colour = Red | Green | Blue

-- we can mix these two forms

-- can be parametrized by type(s) if we wish
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Type classes

Similar to interfaces or protocols — define some operations which might exist on different types, but should behave similarly.

Type classes

Similar to interfaces or protocols — define some operations which might exist on different types, but should behave similarly. Like interfaces, also tend to have additional rules about what they should do which the compiler cannot check:

Type classes

Similar to interfaces or protocols — define some operations which might exist on different types, but should behave similarly. Like interfaces, also tend to have additional rules about what they should do which the compiler cannot check:

```
-- a type class for things which can be compared
class Eq a where
-- instances must have these defined for them
  == :: a -> a -> Bool
  /= :: a -> a -> Bool
```

Type classes

Similar to interfaces or protocols — define some operations which might exist on different types, but should behave similarly. Like interfaces, also tend to have additional rules about what they should do which the compiler cannot check:

```
-- a type class for things which can be compared
class Eq a where
-- instances must have these defined for them
  == :: a -> a -> Bool
  /= :: a -> a -> Bool

instance Eq Int where
  == x y = -- fill in whatever
  /= x y = -- same here
```

Type classes

Similar to interfaces or protocols — define some operations which might exist on different types, but should behave similarly. Like interfaces, also tend to have additional rules about what they should do which the compiler cannot check:

```
-- a type class for things which can be compared
class Eq a where
-- instances must have these defined for them
  == :: a -> a -> Bool
  /= :: a -> a -> Bool

instance Eq Int where
  == x y = -- fill in whatever
  /= x y = -- same here
```

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects.

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't.

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't. Code which is not pure is *impure* (or *side-effecting*).

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't. Code which is not pure is *impure* (or *side-effecting*).

Most languages default to *everything* being as impure as possible — any line of code can do anything at all, and it's up to *us* to make sure they all behave.

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't. Code which is not pure is *impure* (or *side-effecting*).

Most languages default to *everything* being as impure as possible — any line of code can do anything at all, and it's up to *us* to make sure they all behave. This is a huge pain:

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't. Code which is not pure is *impure* (or *side-effecting*).

Most languages default to *everything* being as impure as possible — any line of code can do anything at all, and it's up to *us* to make sure they all behave. This is a huge pain:

- We have to keep it straight in our heads

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't. Code which is not pure is *impure* (or *side-effecting*).

Most languages default to *everything* being as impure as possible — any line of code can do anything at all, and it's up to *us* to make sure they all behave. This is a huge pain:

- We have to keep it straight in our heads
- No telling what an arbitrary function will do

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't. Code which is not pure is *impure* (or *side-effecting*).

Most languages default to *everything* being as impure as possible — any line of code can do anything at all, and it's up to *us* to make sure they all behave. This is a huge pain:

- We have to keep it straight in our heads
- No telling what an arbitrary function will do
- No guarantees that our impurity is limited in any way!

Pure and impure code

Code is *pure* if it *always* returns the same result when given the same arguments, *and* has no side effects. For example, the + function is pure, but something like `print` or `random` aren't. Code which is not pure is *impure* (or *side-effecting*).

Most languages default to *everything* being as impure as possible — any line of code can do anything at all, and it's up to *us* to make sure they all behave. This is a huge pain:

- We have to keep it straight in our heads
- No telling what an arbitrary function will do
- No guarantees that our impurity is limited in any way!

This seems rather strange — why not have *pure* code be the default, and then use the type system to *state* what other behaviour we want to have?

Marking impurity using the type system

Marking impurity using the type system

```
-- unlabelled error (divide by zero)
divide :: Real -> Real -> Maybe Real
```

Marking impurity using the type system

```
-- unlabelled error (divide by zero)
divide :: Real -> Real -> Maybe Real

-- labelled error (lots can go wrong)
parseInt :: String -> Either ParseError Int
```

Marking impurity using the type system

```
-- unlabelled error (divide by zero)
divide :: Real -> Real -> Maybe Real

-- labelled error (lots can go wrong)
parseInt :: String -> Either ParseError Int

-- nondeterminism (from zero to two roots)
sqrt :: Real -> [Real]
```

Marking impurity using the type system

```
-- unlabelled error (divide by zero)
divide :: Real -> Real -> Maybe Real

-- labelled error (lots can go wrong)
parseInt :: String -> Either ParseError Int

-- nondeterminism (from zero to two roots)
sqrt :: Real -> [Real]

-- or whatever the hell (probably web-based here...)
getMeme :: URL -> MemeType -> IO Meme
```

Marking impurity using the type system

```
-- unlabelled error (divide by zero)
divide :: Real -> Real -> Maybe Real

-- labelled error (lots can go wrong)
parseInt :: String -> Either ParseError Int

-- nondeterminism (from zero to two roots)
sqrt :: Real -> [Real]

-- or whatever the hell (probably web-based here...)
getMeme :: URL -> MemeType -> IO Meme

-- we know these functions' side effects
```

Marking impurity using the type system

```
-- unlabelled error (divide by zero)
divide :: Real -> Real -> Maybe Real

-- labelled error (lots can go wrong)
parseInt :: String -> Either ParseError Int

-- nondeterminism (from zero to two roots)
sqrt :: Real -> [Real]

-- or whatever the hell (probably web-based here...)
getMeme :: URL -> MemeType -> IO Meme

-- we know these functions' side effects
-- and we *didn't even have to read them*!
```

A problem

A problem

```
-- suppose we have this eminently-sensible function
square :: Real -> Real
square x = x * x
```

A problem

```
-- suppose we have this eminently-sensible function
square :: Real -> Real
square x = x * x

-- we want to do this, but it won't work
lifeTheUniverseAndEverything = square(sqrt(42))
```

A problem

```
-- suppose we have this eminently-sensible function
square :: Real -> Real
square x = x * x

-- we want to do this, but it won't work
lifeTheUniverseAndEverything = square(sqrt(42))

-- this is because sqrt gives us a [Real]
-- but square wants a plain Real
```

A problem

```
-- suppose we have this eminently-sensible function
square :: Real -> Real
square x = x * x

-- we want to do this, but it won't work
lifeTheUniverseAndEverything = square(sqrt(42))

-- this is because sqrt gives us a [Real]
-- but square wants a plain Real
```

It seems all we can do is re-write every single thing to handle every single condition possible, which is ridiculous and impossible.

A problem

```
-- suppose we have this eminently-sensible function
square :: Real -> Real
square x = x * x

-- we want to do this, but it won't work
lifeTheUniverseAndEverything = square(sqrt(42))

-- this is because sqrt gives us a [Real]
-- but square wants a plain Real
```

It seems all we can do is re-write every single thing to handle every single condition possible, which is ridiculous and impossible. We deserve better from our languages!

What you're probably thinking now

What a functor really is

What a functor really is

```
class Functor f where
  map :: a -> b -> f a -> f b
```

What a functor really is

```
class Functor f where
  map :: a -> b -> f a -> f b
```

Furthermore, we require that the following hold for any functor:

What a functor really is

```
class Functor f where
  map :: a -> b -> f a -> f b
```

Furthermore, we require that the following hold for any functor:

```
--compiler can't check this, sadly, so we must
map id == id
```

What a functor really is

```
class Functor f where
  map :: a -> b -> f a -> f b
```

Furthermore, we require that the following hold for any functor:

```
--compiler can't check this, sadly, so we must
map id == id
```

We call this the *functor law*.

Wait, what?

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor.

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor. So as long as our type has a functor definition, we can use *any* pure function with it.

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor. So as long as our type has a functor definition, we can use *any* pure function with it.

We can define (correct) functor instances for `Maybe`,

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor. So as long as our type has a functor definition, we can use *any* pure function with it.

We can define (correct) functor instances for `Maybe`, `Either`,

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor. So as long as our type has a functor definition, we can use *any* pure function with it.

We can define (correct) functor instances for `Maybe`, `Either`, `[]`,

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor. So as long as our type has a functor definition, we can use *any* pure function with it.

We can define (correct) functor instances for `Maybe`, `Either`, `[]`, `(,)`,

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor. So as long as our type has a functor definition, we can use *any* pure function with it.

We can define (correct) functor instances for `Maybe`, `Either`, `[]`, `(,)`, `IO`,

Wait, what?

Maybe this will help (it's the same thing really):

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

`map` transforms a pure function into a function with a side effect described by the type of the functor. So as long as our type has a functor definition, we can use *any* pure function with it.

We can define (correct) functor instances for `Maybe`, `Either`, `[]`, `(,)`, `IO`, most container types,...

Solving both our problems

Because of `map`, our original problems of 'type incompatibility' no longer exist:

Solving both our problems

Because of `map`, our original problems of 'type incompatibility' no longer exist:

```
-- this now works!
lifeTheUniverseAndEverything = map square (sqrt 42)
-- what do you think its type would be?
```

Solving both our problems

Because of `map`, our original problems of 'type incompatibility' no longer exist:

```
-- this now works!
lifeTheUniverseAndEverything = map square (sqrt 42)
-- what do you think its type would be?
lifeTheUniverseAndEverything :: [Real]
```

Solving both our problems

Because of `map`, our original problems of 'type incompatibility' no longer exist:

```
-- this now works!
lifeTheUniverseAndEverything = map square (sqrt 42)
-- what do you think its type would be?
lifeTheUniverseAndEverything :: [Real]
```

Furthermore, we can define (correct) functors for *all* the types which describe all the effects we mentioned earlier. Now, we can treat them all the same, just by writing functions against `Functor` instead.

Solving both our problems

Because of `map`, our original problems of 'type incompatibility' no longer exist:

```
-- this now works!
lifeTheUniverseAndEverything = map square (sqrt 42)
-- what do you think its type would be?
lifeTheUniverseAndEverything :: [Real]
```

Furthermore, we can define (correct) functors for *all* the types which describe all the effects we mentioned earlier. Now, we can treat them all the same, just by writing functions against `Functor` instead. What's more, this requires *no special support from the language* — if we can define the functor, we can do this!

An example functor: Maybe

An example functor: Maybe

```
instance Functor Maybe where
```

An example functor: Maybe

```
instance Functor Maybe where
  map :: (a -> b) -> (Maybe a -> Maybe b)
```

An example functor: Maybe

```
instance Functor Maybe where
  map :: (a -> b) -> (Maybe a -> Maybe b)
  map g Nothing =
```

An example functor: Maybe

```
instance Functor Maybe where
  map :: (a -> b) -> (Maybe a -> Maybe b)
  map g Nothing = Nothing -- no other choice
```

An example functor: Maybe

```
instance Functor Maybe where
  map :: (a -> b) -> (Maybe a -> Maybe b)
  map g Nothing = Nothing -- no other choice
  map g (Just x) =
```

An example functor: Maybe

```
instance Functor Maybe where
  map :: (a -> b) -> (Maybe a -> Maybe b)
  map g Nothing = Nothing -- no other choice
  map g (Just x) = Just (g x) -- apply and rewrap
```

An example functor: Maybe

```
instance Functor Maybe where
  map :: (a -> b) -> (Maybe a -> Maybe b)
  map g Nothing = Nothing -- no other choice
  map g (Just x) = Just (g x) -- apply and rewrap
```

Of course, we now need to check that the functor law is obeyed.

An example functor: Maybe

```
instance Functor Maybe where
  map :: (a -> b) -> (Maybe a -> Maybe b)
  map g Nothing = Nothing -- no other choice
  map g (Just x) = Just (g x) -- apply and rewrap
```

Of course, we now need to check that the functor law is obeyed. We can *prove* that this is the case (and pretty easily too).

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

```
map id Nothing == id Nothing
```

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

```
map id Nothing == id Nothing
=> Nothing == Nothing
```

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

```
map id Nothing == id Nothing
=> Nothing == Nothing
```

Otherwise, `Maybe a == Just x`, and we have:

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

```
map id Nothing == id Nothing
=> Nothing == Nothing
```

Otherwise, `Maybe a == Just x`, and we have:

```
map id (Just x) == id (Just x)
```

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

```
map id Nothing == id Nothing
=> Nothing == Nothing
```

Otherwise, `Maybe a == Just x`, and we have:

```
map id (Just x) == id (Just x)
=> Just (id x) == Just x
```

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

```
map id Nothing == id Nothing
=> Nothing == Nothing
```

Otherwise, `Maybe a == Just x`, and we have:

```
map id (Just x) == id (Just x)
=> Just (id x) == Just x
=> Just x == Just x
```

Proving the functor law for Maybe

Lemma

Our definition of `Maybe` *follows the functor law.*

Proof.

We must show that, for our definition of `Maybe`, we have:

```
map id == id
```

If `Maybe a == Nothing`, we have:

```
map id Nothing == id Nothing
=> Nothing == Nothing
```

Otherwise, `Maybe a == Just x`, and we have:

```
map id (Just x) == id (Just x)
=> Just (id x) == Just x
=> Just x == Just x
```

Thus, our definition of `Maybe` follows the functor law. □

What functors *can't* do

What functors *can't* do

- Can't apply functions with side-effect markers. If we want to apply a `Maybe (Int -> Int)` to a `Maybe Int`, functors are of no help.

What functors *can't* do

- Can't apply functions with side-effect markers. If we want to apply a `Maybe (Int -> Int)` to a `Maybe Int`, functors are of no help.
- 'Chaining together' functors stacks markers instead of collapsing them. If we tried do `map sqrt (sqrt 42)`, we probably want a `[Real]`, but what we actually get is `[[Real]]`.

What functors *can't* do

- Can't apply functions with side-effect markers. If we want to apply a `Maybe (Int -> Int)` to a `Maybe Int`, functors are of no help.
- 'Chaining together' functors stacks markers instead of collapsing them. If we tried do `map sqrt (sqrt 42)`, we probably want a `[Real]`, but what we actually get is `[[Real]]`.

Luckily for us, there are other, more powerful abstractions built on the functor which solve *both* of these problems.

Questions?