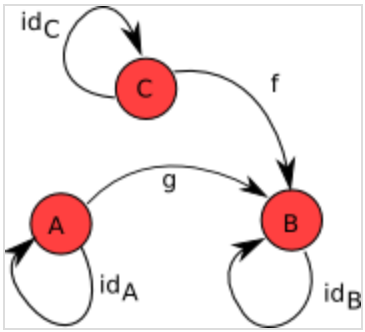
WIKIBOOKS

Haskell/Category theory

This article attempts to give an overview of category theory, in so far as it applies to Haskell. To this end, Haskell code will be given alongside the mathematical definitions. Absolute rigour is not followed; in its place, we seek to give the reader an intuitive feel for what the concepts of category theory are and how they relate to Haskell.

Introduction to categories



A simple category, with three objects *A*, *B* and *C*, three identity morphisms $\mathbf{id}_A$, $\mathbf{id}_B$ and $\mathbf{id}_C$, and two other morphisms $f : C \to B$ and $g : A \to B$. The third element (the specification of how to compose the morphisms) is not shown.

A category is, in essence, a simple collection. It has three components:

- A collection of **objects**.
- A collection of **morphisms**, each of which ties two objects (a *source object* and a *target object*) together. (These are sometimes called **arrows**, but we avoid that term here as it has other connotations in Haskell.) If *f* is a morphism with source object *C* and target object *B*, we write $f : C \to B$.
- A notion of **composition** of these morphisms. If $g : A \to B$ and $f : B \to C$ are two morphisms, they can be composed, resulting in a morphism $f \circ g : A \to C$.

Lots of things form categories. For example, **Set** is the category of all sets with morphisms as standard functions and composition being standard function composition. (Category names are often typeset in bold face.) **Grp** is the category of all groups with morphisms as functions that preserve group operations (the group homomorphisms), i.e. for any two groups, *G* with operation * and *H* with operation ·, a function $k : G \to H$ is a morphism in **Grp** if:

$$k(u * v) = k(u) \cdot k(v)$$

It may seem that morphisms are always functions, but this needn't be the case. For example, any partial order (P, $\leq$) defines a category where the objects are the elements of P, and there is a morphism between any two objects A and B iff $A \leq B$. Moreover, there are allowed to be multiple morphisms with the same source and target objects; using the **Set** example, **sin** and **cos** are both functions with source object $\mathbb{R}$ (the set of real numbers) and target object $[-1, 1]$, but they're most certainly not the same morphism!

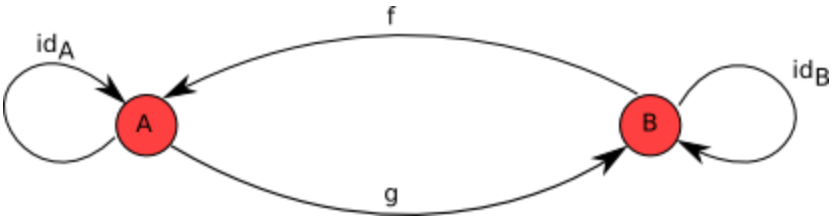
Category laws

There are three laws that categories need to follow. Firstly, and most simply, the composition of morphisms needs to be ***associative***. Symbolically,

$$f \circ (g \circ h) = (f \circ g) \circ h$$

Morphisms are applied right to left in Haskell and most commonly in mathematics, so with $f \circ g$ first g is applied, then f.

Secondly, the category needs to be ***closed*** under the composition operation. So if $f : B \to C$ and $g : A \to B$, then there must be some morphism $h : A \to C$ in the category such that $h = f \circ g$. We can see how this works using the following category:



f and g are both morphisms so we must be able to compose them and get another morphism in the category. So which is the morphism $f \circ g$? The only option is id_A. Similarly, we see that $g \circ f = \mathrm{id}_B$.

Lastly, given a category C there needs to be for every object A an ***identity*** *morphism*, $\mathrm{id}_A : A \to A$ that is an identity of composition with other morphisms. Put precisely, for every morphism $g : A \to B$:

$$g \circ \mathrm{id}_A = \mathrm{id}_B \circ g = g$$

Hask, the Haskell category

The main category we'll be concerning ourselves with in this article is **Hask**, which treats Haskell types as objects and Haskell functions as morphisms and uses `(.)` for composition: a function `f :: A -> B` for types `A` and `B` is a morphism in **Hask**. We can check the first and second law easily: we know `(.)` is an associative function, and clearly, for any `f` and `g`, `f . g` is another function. In **Hask**, the identity morphism is `id`, and we have trivially:

```
id . f = f . id = f
```

[1] This isn't an exact translation of the law above, though; we're missing subscripts. The function `id` in Haskell is *polymorphic* — it can take many different types for its domain and range, or, in category-speak, can have many different source and target objects. But morphisms in category theory are by definition *monomorphic* — each morphism has one specific source object and one specific target object (note: *monomorphic* here is not being used in the category theoretic sense). A polymorphic Haskell function can be made monomorphic by specifying its type (*instantiating* with a monomorphic type), so it would be more precise if we said that the identity morphism from **Hask** on a type `A` is `(id :: A -> A)`. With this in mind, the above law would be rewritten as:

```
(id :: B -> B) . f = f . (id :: A -> A) = f
```

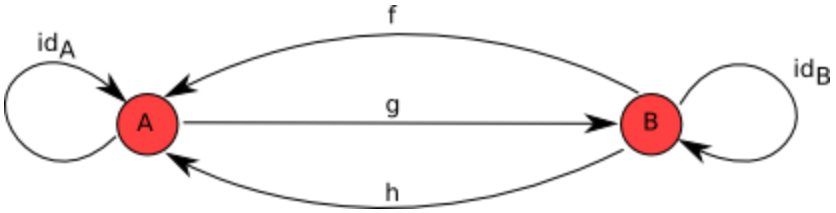
However, for simplicity, we will ignore this distinction when the meaning is clear.

Exercises

- As was mentioned, any partial order $(P, \leq)$ is a category with objects as the elements of P and a morphism between elements a and b iff a $\leq$ b. Which

of the above laws guarantees the transitivity of $\leq$?

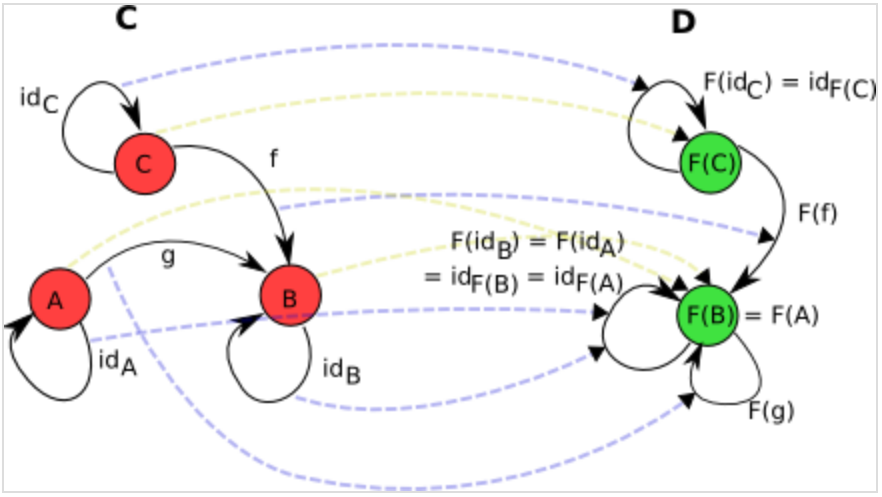
- (Harder.) If we add another morphism to the above example, as illustrated below, it fails to be a category. Why? Hint: think about associativity of the composition operation.



Functors

So we have some categories which have objects and morphisms that relate our objects together. The next Big Topic in category theory is the **functor**, which relates categories together. A functor is essentially a transformation between categories, so given categories *C* and *D*, a functor $F : C \to D$:

- Maps any object *A* in *C* to $F(A)$, in *D*.
- Maps morphisms $f : A \to B$ in *C* to $F(f) : F(A) \to F(B)$ in *D*.



A functor between two categories, **C** and **D**. Of note is that the objects *A* and *B* both get mapped to the same object in **D**, and that therefore *g* becomes a morphism with the same source and target object (but isn't necessarily an identity), and $\mathbf{id}_A$ and $\mathbf{id}_B$ become the same morphism. The arrows showing the mapping of objects are shown in a dotted, pale olive. The arrows showing the mapping of morphisms are shown in a dotted, pale blue.

One of the canonical examples of a functor is the forgetful functor $\mathbf{Grp} \to \mathbf{Set}$ which maps groups to their underlying sets and group morphisms to the functions which behave the same but are defined on sets instead of groups. Another example is the power set functor $\mathbf{Set} \to \mathbf{Set}$ which maps sets to their power sets and maps functions $f : X \to Y$ to functions $\mathcal{P}(X) \to \mathcal{P}(Y)$ which take inputs $U \subseteq X$ and return $f(U)$, the image of U under f, defined by $f(U) = \{ f(u) : u \in U \}$. For any category *C*, we can define a functor known as the identity functor on *C*, or $1_C : C \to C$, that just maps objects to themselves and morphisms to themselves. This will turn out to be useful in the monad laws section later on.

Once again there are a few axioms that functors have to obey. Firstly, given an identity morphism $\mathbf{id}_A$ on an object *A*, $F(\mathbf{id}_A)$ must be the identity morphism on $F(A)$, i.e.:

$$F(\mathbf{id}_A) = \mathbf{id}_{F(A)}$$

Secondly functors must distribute over morphism composition, i.e.

$$F(f \circ g) = F(f) \circ F(g)$$

Exercises

For the diagram given on the right, check these functor laws.

Functors on Hask

The Functor typeclass you have probably seen in Haskell does in fact tie in with the categorical notion of a functor. Remember that a functor has two parts: it maps objects in one category to objects in another and morphisms in the first category to morphisms in the second. Functors in Haskell are from **Hask** to *func*, where *func* is the subcategory of **Hask** defined on just that functor's types. E.g. the list functor goes from **Hask** to **Lst**, where **Lst** is the category containing only *list types*, that is, `[T]` for any type `T`. The morphisms in **Lst** are functions defined on list types, that is, functions `[T] -> [U]` for types `T`, `U`. How does this tie into the Haskell typeclass Functor? Recall its definition:

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

Let's have a sample instance, too:

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  = Nothing
```

Here's the key part: the *type constructor* Maybe takes any type `T` to a new type, `Maybe T`. Also, `fmap` restricted to Maybe types takes a function `a -> b` to a function `Maybe a -> Maybe b`. But that's it! We've defined two parts, something that takes objects in **Hask** to objects in another category (that of Maybe types and functions defined on Maybe types), and something that takes morphisms in **Hask** to morphisms in this category. So Maybe is a functor.

A useful intuition regarding Haskell functors is that they represent types that can be mapped over. This could be a list or a Maybe, but also more complicated structures like trees. A function that does some mapping could be written using `fmap`, then any functor structure could be passed into this function. E.g. you could write a generic function that covers all of Data.List.map, Data.Map.map, Data.Array.IArray.amap, and so on.

What about the functor axioms? The polymorphic function `id` takes the place of id_A for any A, so the first law states:

```
fmap id = id
```

With our above intuition in mind, this states that mapping over a structure doing nothing to each element is equivalent to doing nothing overall. Secondly, morphism composition is just `(.)`, so

```
fmap (f . g) = fmap f . fmap g
```

This second law is very useful in practice. Picturing the functor as a list or similar container, the right-hand side is a two-pass algorithm: we map over the structure, performing `g`, then map over it again, performing `f`. The functor axioms guarantee we can transform this into a single-pass algorithm that performs `f . g`. This is a process known as *fusion*.

Exercises

Check the laws for the Maybe and list functors.

Translating categorical concepts into Haskell

Functors provide a good example of how category theory gets translated into Haskell. The key points to remember are that:

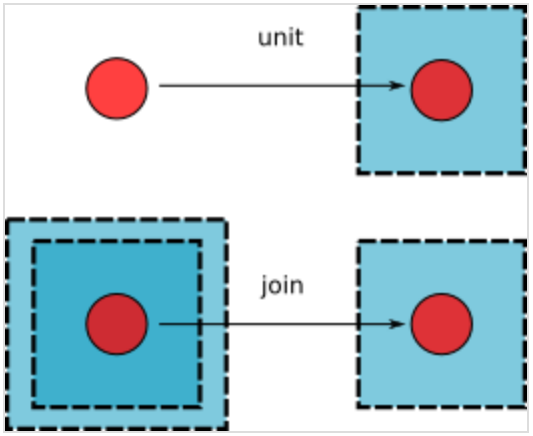
- We work in the category **Hask** and its subcategories.
- Objects are types.
- Morphisms are functions.
- Things that take a type and return another type are type constructors.
- Things that take a function and return another function are higher-order functions.
- Typeclasses, along with the polymorphism they provide, make a nice way of capturing the fact that in category theory things are often defined over a number of objects at once.

Monads

Monads are obviously an extremely important concept in Haskell, and in fact they originally came from category theory. A *monad* is a special type of functor, from a category to that same category, that supports some additional structure. So, down to definitions. A monad is a functor $M : C \to C$, along with two morphisms[2] for every object X in C:

- $\mathbf{unit}_X^M : X \to M(X)$
- $\mathbf{join}_X^M : M(M(X)) \to M(X)$

When the monad under discussion is obvious, we'll leave out the M superscript for these functions and just talk about $\mathbf{unit}_X$ and $\mathbf{join}_X$ for some X.



unit and *join*, the two morphisms that must exist for every object for a given monad.

Let's see how this translates to the Haskell typeclass Monad, then.

```
class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The class constraint of `Functor m` ensures that we already have the functor structure: a mapping of objects and of morphisms. `return` is the (polymorphic) analogue to $\mathbf{unit}_X$ for any X. But we have a problem. Although `return`'s type looks quite similar to that of *unit*; the other function, `(>>=)`, often

called *bind*, bears no resemblance to *join*. There is however another monad function, `join :: Monad m => m (m a) -> m a`, that looks quite similar. Indeed, we can recover `join` and `(>>=)` from each other:

```
join :: Monad m => m (m a) -> m a
join x = x >>= id

(>>=) :: Monad m => m a -> (a -> m b) -> m b
x >>= f = join (fmap f x)
```

So specifying a monad's `return`, `fmap`, and `join` is equivalent to specifying its `return` and `(>>=)`. It just turns out that the normal way of defining a monad in category theory is to give *unit* and *join*, whereas Haskell programmers like to give `return` and `(>>=)`.[3] Often, the categorical way makes more sense. Any time you have some kind of structure M and a natural way of taking any object X into $M(X)$, as well as a way of taking $M(M(X))$ into $M(X)$, you probably have a monad. We can see this in the following example section.

Example: the powerset functor is also a monad

The power set functor $P : \mathbf{Set} \to \mathbf{Set}$ described above forms a monad. For any set *S* you have a $\mathbf{unit}_S(x) = \{x\}$, mapping elements to their singleton set. Note that each of these singleton sets are trivially a subset of *S*, so $\mathbf{unit}_S$ returns elements of the powerset of *S*, as is required. Also, you can define a function $\mathbf{join}_S$ as follows: we receive an input $L \in \mathcal{P}(\mathcal{P}(S))$. This is:

- A member of the powerset of the powerset of *S*.
- So a member of the set of all subsets of the set of all subsets of *S*.
- So a set of subsets of *S*

We then return the union of these subsets, giving another subset of *S*. Symbolically,

$$\mathbf{join}_S(L) = \bigcup L.$$

Hence *P* is a monad [4].

In fact, *P* is almost equivalent to the list monad; with the exception that we're talking lists instead of sets, they're almost the same. Compare:

Power set functor on Set	
Function type	**Definition**
Given a set *S* and a morphism $f : A \to B$:	
$P(f) : \mathcal{P}(A) \to \mathcal{P}(B)$	$(P(f))(S) = \{f(a) : a \in S\}$
$\mathbf{unit}_S : S \to \mathcal{P}(S)$	$\mathbf{unit}_S(x) = \{x\}$
$\mathbf{join}_S : \mathcal{P}(\mathcal{P}(S)) \to \mathcal{P}(S)$	$\mathbf{join}_S(L) = \bigcup L$
List monad from Haskell	
Function type	**Definition**
Given a type T and a function `f :: A -> B`	
`fmap f :: [A] -> [B]`	`fmap f xs = [ f a | a <- xs ]`
`return :: T -> [T]`	`return x = [x]`
`join :: [[T]] -> [T]`	`join xs = concat xs`

The monad laws and their importance

Just as functors had to obey certain axioms in order to be called functors, monads have a few of their own. We'll first list them, then translate to Haskell, then see why they're important.

Given a monad $M : C \to C$ and a morphism $f : A \to B$ for $A, B \in C$,

1. $\mathbf{join} \circ M(\mathbf{join}) = \mathbf{join} \circ \mathbf{join}$
2. $\mathbf{join} \circ M(\mathbf{unit}) = \mathbf{join} \circ \mathbf{unit} = \mathbf{id}$
3. $\mathbf{unit} \circ f = M(f) \circ \mathbf{unit}$
4. $\mathbf{join} \circ M(M(f)) = M(f) \circ \mathbf{join}$

By now, the Haskell translations should be hopefully self-explanatory:

1. `join . fmap join = join . join`
2. `join . fmap return = join . return = id`
3. `return . f = fmap f . return`
4. `join . fmap (fmap f) = fmap f . join`

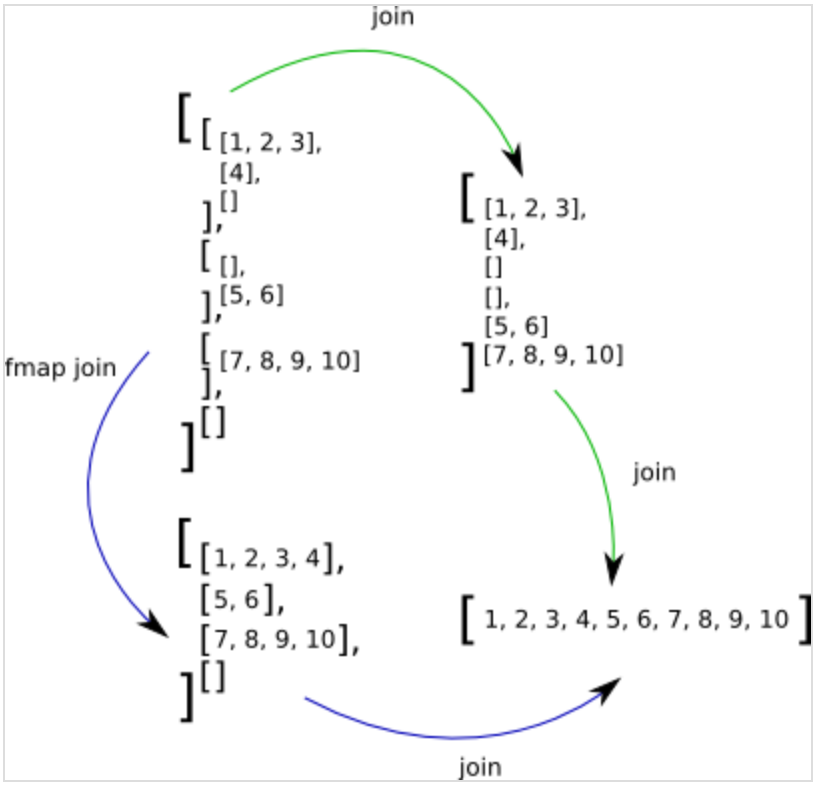
(Remember that `fmap` is the part of a functor that acts on morphisms.) These laws seem a bit impenetrable at first, though. What on earth do these laws mean, and why should they be true for monads? Let's explore the laws.

The first law

`join . fmap join = join . join`

In order to understand this law, we'll first use the example of lists. The first law mentions two functions, `join . fmap join` (the left-hand side) and `join . join` (the right-hand side). What will the types of these functions be? Remembering that `join`'s type is `[[a]] -> [a]` (we're talking just about lists for now), the types are both `[[[a]]] -> [a]` (the fact that they're the same is

handy; after all, we're trying to show they're completely the same function!). So we have a list of lists of lists. The left-hand side, then, performs `fmap join` on this 3-layered list, then uses `join` on the result. `fmap` is just the familiar `map` for lists, so we first map across each of the list of lists inside the top-level list, concatenating them down into a list each. So afterward, we have a list of lists, which we then run through `join`. In summary, we 'enter' the top level, collapse the second and third levels down, then collapse this new level with the top level.



A demonstration of the first law for lists. Remember that `join` is `concat` and `fmap` is `map` in the list monad.

What about the right-hand side? We first run `join` on our list of list of lists. Although this is three layers, and you normally apply a two-layered list to `join`, this will still work, because a `[[[a]]]` is just `[[b]]`, where `b = [a]`, so in a sense, a three-layered list is just a two layered list, but rather than the last layer being 'flat', it is composed of another list. So if we apply our list of lists (of lists) to `join`, it will flatten those outer two layers into one. As the second layer wasn't flat but instead contained a third layer, we will still end up with a list of lists, which the other `join` flattens. Summing up, the left-hand side will flatten the inner two layers into a new layer, then flatten this with the outermost layer. The right-hand side will flatten the outer two layers, then flatten this with the innermost layer. These two operations should be equivalent. It's sort of like a law of associativity for `join`.

`Maybe` is also a monad, with

```
return :: a -> Maybe a
return x = Just x

join :: Maybe (Maybe a) -> Maybe a
join Nothing         = Nothing
join (Just Nothing)  = Nothing
join (Just (Just x)) = Just x
```

So if we had a *three*-layered Maybe (i.e., it could be `Nothing`, `Just Nothing`, `Just (Just Nothing)` or `Just (Just (Just x))`), the first law says that collapsing the inner two layers first, then that with the outer layer is exactly the same as collapsing the outer layers first, then that with the innermost layer.

Exercises

Verify that the list and Maybe monads do in fact obey this law with some examples to see precisely how the layer flattening works.

The second law

```
join . fmap return = join . return = id
```

What about the second law, then? Again, we'll start with the example of lists. Both functions mentioned in the second law are functions `[a] -> [a]`. The left-hand side expresses a function that maps over the list, turning each element `x` into its singleton list `[x]`, so that at the end we're left with a list of singleton lists. This two-layered list is flattened down into a single-layer list again using the `join`. The right hand side, however, takes the entire list `[x, y, z, ...]`, turns it into the singleton list of lists `[[x, y, z, ...]]`, then flattens the two layers down into one again. This law is less obvious to state quickly, but it essentially says that applying `return` to a monadic value, then `join`ing the result should have the same effect whether you perform the `return` from inside the top layer or from outside it.

Exercises

Prove this second law for the Maybe monad.

The third and fourth laws

```
return . f = fmap f . return
```

```
join . fmap (fmap f) = fmap f . join
```

The last two laws express more self-evident facts about how we expect monads to behave. The easiest way to see how they are true is to expand them to use the expanded form:

1. `\x -> return (f x) = \x -> fmap f (return x)`
2. `\x -> join (fmap (fmap f) x) = \x -> fmap f (join x)`

Exercises

Convince yourself that these laws should hold true for any monad by exploring what they mean, in a similar style to how we explained the first and second laws.

Application to do-blocks

Well, we have intuitive statements about the laws that a monad must support, but why is that important? The answer becomes obvious when we consider do-blocks. Recall that a do-block is just syntactic sugar for a combination of statements involving `(>>=)` as witnessed by the usual translation:

```
do { x }                --->  x
do { let { y = v }; x }  --->  let y = v in do { x }
```

```
do { v <- y; x }         -->  y >>= \v -> do { x }
do { y; x }              -->  y >>= \_ -> do { x }
```

Also notice that we can prove what are normally quoted as the monad laws using `return` and `(>>=)` from our above laws (the proofs are a little heavy in some cases, feel free to skip them if you want to):

1. `return x >>= f = f x`. Proof:

```
   return x >>= f
 = join (fmap f (return x)) -- By the definition of (>>=)
 = join (return (f x))      -- By law 3
 = (join . return) (f x)
 = id (f x)                 -- By law 2
 = f x
```

2. `m >>= return = m`. Proof:

```
   m >>= return
 = join (fmap return m)    -- By the definition of (>>=)
 = (join . fmap return) m
 = id m                    -- By law 2
 = m
```

3. `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`. Proof (recall that `fmap f . fmap g = fmap (f . g)`):

```
   (m >>= f) >>= g
 = (join (fmap f m)) >>= g                          -- By the definition of (>>=)
 = join (fmap g (join (fmap f m)))                  -- By the definition of (>>=)
 = (join . fmap g) (join (fmap f m))
 = (join . fmap g . join) (fmap f m)
 = (join . join . fmap (fmap g)) (fmap f m)         -- By law 4
 = (join . join . fmap (fmap g) . fmap f) m
 = (join . join . fmap (fmap g . f)) m              -- By the distributive law of functors
 = (join . join . fmap (\x -> fmap g (f x))) m
 = (join . fmap join . fmap (\x -> fmap g (f x))) m -- By law 1
 = (join . fmap (join . (\x -> fmap g (f x)))) m    -- By the distributive law of functors
 = (join . fmap (\x -> join (fmap g (f x)))) m
 = (join . fmap (\x -> f x >>= g)) m                -- By the definition of (>>=)
 = join (fmap (\x -> f x >>= g) m)
 = m >>= (\x -> f x >>= g)                          -- By the definition of (>>=)
```

These new monad laws, using `return` and `(>>=)`, can be translated into do-block notation.

Points-free style	Do-block style
`return x >>= f = f x`	`do { v <- return x; f v } = do { f x }`
`m >>= return = m`	`do { v <- m; return v } = do { m }`
`(m >>= f) >>= g = m >>= (\x -> f x >>= g)`	`do { y <- do { x <- m; f x }; g y } = do { x <- m; y <- f x; g y }`

The monad laws are now common-sense statements about how do-blocks should function. If one of these laws were invalidated, users would become confused, as you couldn't be able to manipulate things within the do-blocks as would be expected. The monad laws are, in essence, usability guidelines.

Exercises

In fact, the two versions of the laws we gave:

```
-- Categorical:
join . fmap join = join . join
join . fmap return = join . return = id
return . f = fmap f . return
join . fmap (fmap f) = fmap f . join

-- Functional:
m >>= return = m
return m >>= f = f m
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

are entirely equivalent. We showed that we can recover the functional laws from the categorical ones. Go the other way; show that starting from the functional laws, the categorical laws hold. It may be useful to remember the following definitions:

```
join m = m >>= id
fmap f m = m >>= return . f
```

Thanks to Yitzchak Gale for suggesting this exercise.

Summary

We've come a long way in this chapter. We've looked at what categories are and how they apply to Haskell. We've introduced the basic concepts of category theory including functors, as well as some more advanced topics like monads, and seen how they're crucial to idiomatic Haskell. We haven't covered some of the basic category theory that wasn't needed for our aims, like natural transformations, but have instead provided an intuitive feel for the categorical grounding behind Haskell's structures.

Notes

1. Actually, there is a subtlety here: because `(.)` is a lazy function, if `f` is `undefined`, we have that `id . f = \_ -> ⊥`. Now, while this may seem equivalent to ⊥ for all intents and purposes, you can actually tell them apart using the strictifying function `seq`, meaning that the last category law is broken. We can define a new strict composition function, `f .! g = ((.) $! f) $! g`, that makes **Hask** a category. We proceed by using the normal `(.)`, though, and attribute any discrepancies to the fact that `seq` breaks an awful lot of the nice language properties anyway.
2. Experienced category theorists will notice that we're simplifying things a bit here; instead of presenting *unit* and *join* as natural transformations, we treat them explicitly as morphisms, and require naturality as extra axioms alongside the standard monad laws (laws 3 and 4). The reasoning is simplicity; we are not trying to teach category theory as a whole, simply give a

categorical background to some of the structures in Haskell. You may also notice that we are giving these morphisms names suggestive of their Haskell analogues, because the names η and μ don't provide much intuition.

3. This is perhaps due to the fact that Haskell programmers like to think of monads as a way of sequencing computations with a common feature, whereas in category theory the container aspect of the various structures is emphasised. `join` pertains naturally to containers (squashing two layers of a container down into one), but `(>>=)` is the natural sequencing operation (do something, feeding its results into something else).
4. If you can prove that certain laws hold, which we'll explore in the next section.
