Typeclassopedia

Brent Yorgey

September 2013

By Brent Yorgey, byorgey@cis.upenn.edu

Originally published 12 March 2009 in issue 13 of the Monad.Reader. Ported to the Haskell wiki in November 2011 by Geheimdienst.

This is now the official version of the Typeclassopedia and supersedes the version published in the Monad.Reader. Please help update and extend it by editing it yourself or by leaving comments, suggestions, and questions on the talk page.

Abstract

The standard Haskell libraries feature a number of type classes with algebraic or category-theoretic underpinnings. Becoming a fluent Haskell hacker requires intimate familiarity with them all, yet acquiring this familiarity often involves combing through a mountain of tutorials, blog posts, mailing list archives, and IRC logs.

The goal of this document is to serve as a starting point for the student of Haskell wishing to gain a firm grasp of its standard type classes. The essentials of each type class are introduced, with examples, commentary, and extensive references for further reading.

Contents

Introduction

Have you ever had any of the following thoughts?

- What the heck is a monoid, and how is it different from a monad?
- I finally figured out how to use Parsec with do-notation, and someone told me I should use something called `Applicative` instead. Um, what?
- Someone in the #haskell IRC channel used `(***)`, and when I asked lambdabot to tell me its type, it printed out scary gobbledygook that didn't even fit on one line! Then someone used `fmap fmap fmap` and my brain exploded.
- When I asked how to do something I thought was really complicated, people started typing things like `zip.ap fmap.(id &&& wtf)` and the scary thing is that they worked! Anyway, I think those people must actually be robots because there's no way anyone could come up with that in two seconds off the top of their head.

If you have, look no further! You, too, can write and understand concise, elegant, idiomatic Haskell code with the best of them.

There are two keys to an expert Haskell hacker's wisdom:

1. Understand the types.
2. Gain a deep intuition for each type class and its relationship to other type classes, backed up by familiarity with many examples.

It's impossible to overstate the importance of the first; the patient student of type signatures will uncover many profound secrets. Conversely, anyone ignorant of the types in their code is doomed to eternal uncertainty. "Hmm, it doesn't compile ... maybe I'll stick in an `fmap` here ... nope, let's see ... maybe I need another `(.)` somewhere? ... um ..."

The second key—gaining deep intuition, backed by examples—is also important, but much more difficult to attain. A primary goal of this document is to set you on the road to gaining such intuition. However—

There is no royal road to Haskell.

This document can only be a starting point, since good intuition comes from hard work, not from learning the right metaphor. Anyone who reads and understands all of it will still have an arduous journey ahead—but sometimes a good starting point makes a big difference.

It should be noted that this is not a Haskell tutorial; it is assumed that the reader is already familiar with the basics of Haskell, including the standard `[http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html Prelude]`, the type system, data types, and type classes.

The type classes we will be discussing and their interrelationships:

Image:Typeclassopedia-diagram.png

- Solid arrows point from the general to the specific; that is, if there is an arrow from `Foo` to `Bar` it means that every `Bar` is (or should be, or can be made into) a `Foo`.
- Dotted arrows indicate some other sort of relationship.
- `Monad` and `ArrowApply` are equivalent.
- `Semigroup`, `Apply` and `Comonad` are greyed out since they are not actually (yet?) in the standard Haskell libraries .

One more note before we begin. The original spelling of "type class" is with two words, as evidenced by, for example, the Haskell 98 Revised Report, early papers on type classes like Type classes in Haskell and Type classes: exploring the design space, and Hudak et al.'s history of Haskell. However, as often happens with two-word phrases that see a lot of use, it has started to show up as one word ("typeclass") or, rarely, hyphenated ("type-class"). When wearing my prescriptivist hat, I prefer "type class", but realize (after changing into my descriptivist hat) that there's probably not much I can do about it.

We now begin with the simplest type class of all: `Functor`.

Functor

The `Functor` class (haddock) is the most basic and ubiquitous type class in the Haskell libraries. A simple intuition is that a `Functor` represents a "container" of some sort, along with the ability to apply a function uniformly to every element in the container. For example, a list is a container of elements, and we can apply a function to every element of a list, using `map`. As another example, a binary tree is also a container of elements, and it's not hard to come up with a way to recursively apply a function to every element in a tree.

Another intuition is that a `Functor` represents some sort of "computational context". This intuition is generally more useful, but is more difficult to explain, precisely because it is so general. Some examples later should help to clarify the `Functor`-as-context point of view.

In the end, however, a `Functor` is simply what it is defined to be; doubtless there are many examples of `Functor` instances that don't exactly fit either of the above intuitions. The wise student will focus their attention on definitions and examples, without leaning too heavily on any particular metaphor. Intuition will come, in time, on its own.

Definition

Here is the type class declaration for `Functor`:

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

`Functor` is exported by the `Prelude`, so no special imports are needed to use it.

First, the `f a` and `f b` in the type signature for `fmap` tell us that `f` isn't just a type; it is a *type constructor* which takes another type as a parameter. (A more precise way to say this is that the *kind* of `f` must be `* -> *`.) For example, `Maybe` is such a type constructor: `Maybe` is not a type in and of itself, but requires another type as a parameter, like `Maybe Integer`. So it would not make sense to say `instance Functor Integer`, but it could make sense to say `instance Functor Maybe`.

Now look at the type of `fmap`: it takes any function from `a` to `b`, and a value of type `f a`, and outputs a value of type `f b`. From the container point of view, the intention is that `fmap` applies a function to each element of a container, without altering the structure of the container. From the context point of view, the intention is that `fmap` applies a function to a value without altering its context. Let's look at a few specific examples.

Instances

As noted before, the list constructor `[]` is a functor ; we can use the standard list function `map` to apply a function to each element of a list . The `Maybe` type constructor is also a functor, representing a container which might hold a single element. The function `fmap g` has no effect on `Nothing` (there are no elements to which `g` can be applied), and simply applies `g` to the single element inside a `Just`. Alternatively, under the context interpretation, the list functor represents a context of nondeterministic choice; that is, a list can be thought of as representing a single value which is nondeterministically chosen from among several possibilities (the elements of the list). Likewise, the `Maybe` functor represents a context with possible failure. These instances are:

```haskell
instance Functor [] where
  fmap _ []     = []
  fmap g (x:xs) = g x : fmap g xs
  -- or we could just say fmap = map

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap g (Just a) = Just (g a)
```

As an aside, in idiomatic Haskell code you will often see the letter `f` used to stand for both an arbitrary `Functor` and an arbitrary function. In this document, `f` represents only `Functor`s, and `g` or `h` always represent functions, but you should be aware of the potential confusion. In practice, what `f` stands for should always be clear from the context, by noting whether it is part of a type or part of the code.

There are other `Functor` instances in the standard libraries; below are a few. Note that some of these instances are not exported by the `Prelude`; to access them, you can import `Control.Monad.Instances`.

- `Either e` is an instance of `Functor`; `Either e a` represents a container which can contain either a value of type `a`, or a value of type `e` (often representing some sort of error condition). It is similar to `Maybe` in that it represents possible failure, but it can carry some extra information about the failure as well.

- `((,) e)` represents a container which holds an "annotation" of type `e` along with the actual value it holds. It might be clearer to write it as `(e,)`, by analogy with an operator section like `(1+)`, but that syntax is not allowed in types (although it is allowed in expressions with the `TupleSections` extension enabled). However, you can certainly *think* of it as `(e,)`.

- `((->) e)` (which can be thought of as `(e ->)`; see above), the type of functions which take a value of type `e` as a parameter, is a `Functor`. As a container, `(e -> a)` represents a (possibly infinite) set of values of `a`, indexed by values of `e`. Alternatively, and more usefully, `((->) e)` can be thought of as a context in which a value of type `e` is available to be consulted in a read-only fashion. This is also why `((->) e)` is sometimes referred to as the *reader monad*; more on this later.

- `IO` is a `Functor`; a value of type `IO a` represents a computation producing a value of type `a` which may have I/O effects. If `m` computes the value `x` while producing some I/O effects, then `fmap g m` will compute the value `g x` while producing the same I/O effects.

- Many standard types from the containers library (such as `Tree`, `Map`, and `Sequence`) are instances of `Functor`. A notable exception is `Set`, which cannot be made a `Functor` in Haskell (although it is certainly a mathematical functor) since it requires an `Ord` constraint on its elements; `fmap` must be applicable to *any* types `a` and `b`. However, `Set` (and other similarly restricted data types) can be made an instance of a suitable generalization of `Functor`, either by making `a` and `b` arguments to the `Functor` type class themselves, or by adding an associated constraint.

Laws

As far as the Haskell language itself is concerned, the only requirement to be a `Functor` is an implementation of `fmap` with the proper type. Any sensible `Functor` instance, however, will also satisfy the *functor laws*, which are part of the definition of a mathematical functor. There are two:

```haskell
fmap id = id
fmap (g . h) = (fmap g) . (fmap h)
```

Together, these laws ensure that `fmap g` does not change the *structure* of a container, only the elements. Equivalently, and more simply, they ensure that `fmap g` changes a value without altering its context .

The first law says that mapping the identity function over every item in a container has no effect. The second says that mapping a composition of two functions over every item in a container is the same as first mapping one function, and then mapping the other.

As an example, the following code is a "valid" instance of `Functor` (it typechecks), but it violates the functor laws. Do you see why?

```haskell
-- Evil Functor instance
instance Functor [] where
  fmap _ [] = []
  fmap g (x:xs) = g x : g x : fmap g xs
```

Any Haskeller worth their salt would reject this code as a gruesome abomination.

Unlike some other type classes we will encounter, a given type has at most one valid instance of `Functor`. This can be proven via the *free theorem* for the type of `fmap`. In fact, GHC can automatically derive `Functor` instances for many data types.

A similar argument also shows that any `Functor` instance satisfying the first law (`fmap id = id`) will automatically satisfy the second law as well. Practically, this means that only the first law needs to be checked (usually by a very straightforward induction) to ensure that a `Functor` instance is valid.

Intuition

There are two fundamental ways to think about `fmap`. The first has already been mentioned: it takes two parameters, a function and a container, and applies the function "inside" the container, producing a new container. Alternately, we can think of `fmap` as applying a function to a value in a context (without altering the context).

Just like all other Haskell functions of "more than one parameter", however, `fmap` is actually *curried*: it does not really take two parameters, but takes a single parameter and returns a function. For emphasis, we can write `fmap`'s type with extra parentheses: `fmap :: (a -> b) -> (f a -> f b)`. Written in this form, it is apparent that `fmap` transforms a "normal" function (`g :: a -> b`) into one which operates over containers/contexts (`fmap g :: f a -> f b`). This transformation is often referred to as a *lift*; `fmap` "lifts" a function from the "normal world" into the "`f` world".

Further reading

A good starting point for reading about the category theory behind the concept of a functor is the excellent Haskell wikibook page on category theory.

Applicative

A somewhat newer addition to the pantheon of standard Haskell type classes, *applicative functors* represent an abstraction lying in between `Functor` and `Monad` in expressivity, first described by McBride and Paterson. The title of their classic paper, Applicative Programming with Effects, gives a hint at the intended intuition behind the `Applicative` type class. It encapsulates certain sorts of "effectful" computations in a functionally pure way, and encourages an "applicative" programming style. Exactly what these things mean will be seen later.

Definition

Recall that `Functor` allows us to lift a "normal" function to a function on computational contexts. But `fmap` doesn't allow us to apply a function which is itself in a context to a value in a context. `Applicative` gives us just such a tool, `(<*>)`. It also provides a method, `pure`, for embedding values in a default, "effect free" context. Here is the type class declaration for `Applicative`, as defined in `Control.Applicative`:

```haskell
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Note that every `Applicative` must also be a `Functor`. In fact, as we will see, `fmap` can be implemented using the `Applicative` methods, so every `Applicative` is a functor whether we like it or not; the `Functor` constraint forces us to be honest.

As always, it's crucial to understand the type signatures. First, consider `(<*>)`: the best way of thinking about it comes from noting that the type of `(<*>)` is similar to the type of `($)` , but with everything enclosed in an `f`. In other words, `(<*>)` is just function application within a computational context. The type of `(<*>)` is also very similar to the type of `fmap`; the only difference is that the first parameter is `f (a -> b)`, a function in a context, instead of a "normal" function `(a -> b)`.

`pure` takes a value of any type `a`, and returns a context/container of type `f a`. The intention is that `pure` creates some sort of "default" container or "effect free" context. In fact, the behavior of `pure` is quite constrained by the laws it should satisfy in conjunction with `(<*>)`. Usually, for a given implementation of `(<*>)` there is only one possible implementation of `pure`.

(Note that previous versions of the Typeclassopedia explained `pure` in terms of a type class `Pointed`, which can still be found in the `pointed` package. However, the current consensus is that `Pointed` is not very useful after all. For a more detailed explanation, see Why not Pointed?)

Laws

Traditionally, there are four laws that `Applicative` instances should satisfy . In some sense, they are all concerned with making sure that `pure` deserves its name:

- The identity law:

  ```haskell
  pure id <*> v = v
  ```

- Homomorphism:

  ```haskell
  pure f <*> pure x = pure (f x)
  ```

  Intuitively, applying a non-effectful function to a non-effectful argument in an effectful context is the same as just applying the function to the argument and then injecting the result into the context with `pure`.
- Interchange:

  ```haskell
  u <*> pure y = pure ($ y) <*> u
  ```

  Intuitively, this says that when evaluating the application of an effectful function to a pure argument, the order in which we evaluate the function and its argument doesn't matter.
- Composition:

  ```haskell
  u <*> (v <*> w) = pure (.) <*> u <*> v <*> w
  ```

  This one is the trickiest law to gain intuition for. In some sense it is expressing a sort of associativity property of `(<*>)`. The reader may wish to simply convince themselves that this law is type-correct.

Considered as left-to-right rewrite rules, the homomorphism, interchange, and composition laws actually constitute an algorithm for transforming any expression using `pure` and `(<*>)` into a canonical form with only a single use of `pure` at the very beginning and only left-nested occurrences of `(<*>)`. Composition allows reassociating `(<*>)`; interchange allows moving occurrences of `pure` leftwards; and homomorphism allows collapsing multiple adjacent occurrences of `pure` into one.

There is also a law specifying how `Applicative` should relate to `Functor`:

```
fmap g x = pure g <*> x
```

It says that mapping a pure function `g` over a context `x` is the same as first injecting `g` into a context with `pure`, and then applying it to `x` with `(<*>)`. In other words, we can decompose `fmap` into two more atomic operations: injection into a context, and application within a context. The `Control.Applicative` module also defines `(<$>)` as a synonym for `fmap`, so the above law can also be expressed as:

`g <$> x = pure g <*> x`.

Instances

Most of the standard types which are instances of `Functor` are also instances of `Applicative`.

`Maybe` can easily be made an instance of `Applicative`; writing such an instance is left as an exercise for the reader.

The list type constructor `[]` can actually be made an instance of `Applicative` in two ways; essentially, it comes down to whether we want to think of lists as ordered collections of elements, or as contexts representing multiple results of a nondeterministic computation (see Wadler's How to replace failure by a list of successes).

Let's first consider the collection point of view. Since there can only be one instance of a given type class for any particular type, one or both of the list instances of `Applicative` need to be defined for a `newtype` wrapper; as it happens, the nondeterministic computation instance is the default, and the collection instance is defined in terms of a `newtype` called `ZipList`. This instance is:

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
  pure = undefined   -- exercise
  (ZipList gs) <*> (ZipList xs) = ZipList (zipWith ($) gs xs)
```

To apply a list of functions to a list of inputs with `(<*>)`, we just match up the functions and inputs elementwise, and produce a list of the resulting outputs. In other words, we "zip" the lists together with function application, `($)`; hence the name `ZipList`.

The other `Applicative` instance for lists, based on the nondeterministic computation point of view, is:

```
instance Applicative [] where
  pure x    = [x]
  gs <*> xs = [ g x | g <- gs, x <- xs ]
```

Instead of applying functions to inputs pairwise, we apply each function to all the inputs in turn, and collect all the results in a list.

Now we can write nondeterministic computations in a natural style. To add the numbers `3` and `4` deterministically, we can of course write `(+) 3 4`. But suppose instead of `3` we have a nondeterministic computation that might result in `2`, `3`, or `4`; then we can write

```
  pure (+) <*> [2,3,4] <*> pure 4
```

or, more idiomatically,

```
  (+) <$> [2,3,4] <*> pure 4.
```

There are several other `Applicative` instances as well:

- `IO` is an instance of `Applicative`, and behaves exactly as you would think: to execute `m1 <*> m2`, first `m1` is executed, resulting in a function `f`, then `m2` is executed, resulting in a value `x`, and finally the value `f x` is returned as the result of executing `m1 <*> m2`.

- `((,) a)` is an `Applicative`, as long as `a` is an instance of `Monoid` (section Monoid). The `a` values are accumulated in parallel with the computation.

- The `Applicative` module defines the `Const` type constructor; a value of type `Const a b` simply contains an `a`. This is an instance of `Applicative` for any `Monoid a`; this instance becomes especially useful in conjunction with things like `Foldable` (section Foldable).

- The `WrappedMonad` and `WrappedArrow` newtypes make any instances of `Monad` (section Monad) or `Arrow` (section Arrow) respectively into instances of `Applicative`; as we will see when we study those type classes, both are strictly more expressive than `Applicative`, in the sense that the `Applicative` methods can be implemented in terms of their methods.

Intuition

McBride and Paterson's paper introduces the notation $[[g\ x_1\ x_2\ \cdots\ x_n]]$ to denote function application in a computational context. If each x_i has type $f\ t_i$ for some applicative functor f, and g has type $t_1 \to t_2 \to \cdots \to t_n \to t$, then the entire expression $[[g\ x_1\ \cdots\ x_n]]$ has type $f\ t$. You can think of this as applying a function to multiple "effectful" arguments. In this sense, the double bracket notation is a generalization of `fmap`, which allows us to apply a function to a single argument in a context.

Why do we need `Applicative` to implement this generalization of `fmap`? Suppose we use `fmap` to apply `g` to the first parameter `x1`. Then we get something of type `f (t2 -> ... t)`, but now we are stuck: we can't apply this function-in-a-context to the next argument with `fmap`. However, this is precisely what `(<*>)` allows us to do.

This suggests the proper translation of the idealized notation $[[g\ x_1\ x_2\ \cdots\ x_n]]$ into Haskell, namely

```
  g <$> x1 <*> x2 <*> ... <*> xn,
```

recalling that `Control.Applicative` defines `(<$>)` as convenient infix shorthand for `fmap`. This is what is meant by an "applicative style"—effectful computations can still be described in terms of function application; the only difference is that we have to use the special operator `(<*>)` for application instead of simple juxtaposition.

Note that `pure` allows embedding "non-effectful" arguments in the middle of an idiomatic application, like

```
  g <$> x1 <*> pure x2 <*> x3
```

which has type `f d`, given

```
g  :: a -> b -> c -> d
x1 :: f a
x2 :: b
x3 :: f c
```

The double brackets are commonly known as "idiom brackets", because they allow writing "idiomatic" function application, that is, function application that looks normal but has some special, non-standard meaning (determined by the particular instance of `Applicative` being used). Idiom brackets are not supported by GHC, but they are supported by the Strathclyde Haskell Enhancement, a preprocessor which (among many other things) translates idiom brackets into standard uses of `(<$>)` and `(<*>)`. This can result in much more readable code when making heavy use of `Applicative`.

Alternative formulation

An alternative, equivalent formulation of `Applicative` is given by

```haskell
class Functor f => Monoidal f where
  unit :: f ()
  (**) :: f a -> f b -> f (a,b)
```

Intuitively, this states that a monoidal functor is one which has some sort of "default shape" and which supports some sort of "combining" operation. `pure` and `(<*>)` are equivalent in power to `unit` and `(**)` (see the Exercises below).

Furthermore, to deserve the name "monoidal" (see the section on Monoids), instances of `Monoidal` ought to satisfy the following laws, which seem much more straightforward than the traditional `Applicative` laws:

- Naturality:

  ```haskell
  fmap (g *** h) (u ** v) = fmap g u ** fmap h v
  ```

- Left identity:

  `unit ** v ≅ v`

- Right identity:

  `u ** unit ≅ u`

- Associativity:

  `u ** (v ** w) ≅ (u ** v) ** w`

These turn out to be equivalent to the usual `Applicative` laws.

Much of this section was taken from a blog post by Edward Z. Yang; see his actual post for a bit more information.

Further reading

There are many other useful combinators in the standard libraries implemented in terms of `pure` and `(<*>)`: for example, `(*>)`, `(<*)`, `(<**>)`, `(<$)`, and so on (see haddock for Applicative). Judicious use of such secondary combinators can often make code using `Applicative`s much easier to read.

McBride and Paterson's original paper is a treasure-trove of information and examples, as well as some perspectives on the connection between `Applicative` and category theory. Beginners will find it difficult to make it through the entire paper, but it is extremely well-motivated—even beginners will be able to glean something from reading as far as they are able.

Conal Elliott has been one of the biggest proponents of `Applicative`. For example, the Pan library for functional images and the reactive library for functional reactive programming (FRP) make key use of it; his blog also contains many examples of `Applicative` in action. Building on the work of McBride and Paterson, Elliott also built the TypeCompose library, which embodies the observation (among others) that `Applicative` types are closed under composition; therefore, `Applicative` instances can often be automatically derived for complex types built out of simpler ones.

Although the Parsec parsing library (paper) was originally designed for use as a monad, in its most common use cases an `Applicative` instance can be used to great effect; Bryan O'Sullivan's blog post is a good starting point. If the extra power provided by `Monad` isn't needed, it's usually a good idea to use `Applicative` instead.

A couple other nice examples of `Applicative` in action include the ConfigFile and HSQL libraries and the formlets library.

Gershom Bazerman's post contains many insights into applicatives.

Monad

It's a safe bet that if you're reading this, you've heard of monads—although it's quite possible you've never heard of `Applicative` before, or `Arrow`, or even `Monoid`. Why are monads such a big deal in Haskell? There are several reasons.

- Haskell does, in fact, single out monads for special attention by making them the framework in which to construct I/O operations.
- Haskell also singles out monads for special attention by providing a special syntactic sugar for monadic expressions: the `do`-notation.
- `Monad` has been around longer than other abstract models of computation such as `Applicative` or `Arrow`.
- The more monad tutorials there are, the harder people think monads must be, and the more new monad tutorials are written by people who think they finally "get" monads (the monad tutorial fallacy).

I will let you judge for yourself whether these are good reasons.

In the end, despite all the hoopla, `Monad` is just another type class. Let's take a look at its definition.

Definition

The type class declaration for `Monad` is:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> n = m >>= \_ -> n

  fail   :: String -> m a
```

The `Monad` type class is exported by the `Prelude`, along with a few standard instances. However, many utility functions are found in `Control.Monad`, and there are also several instances (such as `((->) e)`) defined in `Control.Monad.Instances`.

Let's examine the methods in the `Monad` class one by one. The type of `return` should look familiar; it's the same as `pure`. Indeed, `return` *is* `pure`, but with an unfortunate name. (Unfortunate, since someone coming from an imperative programming background might think that `return` is like the C or Java keyword of the same name, when in fact the similarities are minimal.) From a mathematical point of view, every monad is an applicative functor, but for historical reasons, the `Monad` type class declaration unfortunately does not require this.

We can see that `(>>)` is a specialized version of `(>>=)`, with a default implementation given. It is only included in the type class declaration so that specific instances of `Monad` can override the default implementation of `(>>)` with a more efficient one, if desired. Also, note that although `_ >> n = n` would be a type-correct implementation of `(>>)`, it would not correspond to the intended semantics: the intention is that `m >> n` ignores the *result* of `m`, but not its *effects*.

The `fail` function is an awful hack that has no place in the `Monad` class; more on this later.

The only really interesting thing to look at—and what makes `Monad` strictly more powerful than `Applicative`—is `(>>=)`, which is often called *bind*. An alternative definition of `Monad` could look like:

```
class Applicative m => Monad' m where
  (>>=) :: m a -> (a -> m b) -> m b
```

We could spend a while talking about the intuition behind `(>>=)`—and we will. But first, let's look at some examples.

Instances

Even if you don't understand the intuition behind the `Monad` class, you can still create instances of it by just seeing where the types lead you. You may be surprised to find that this actually gets you a long way towards understanding the intuition; at the very least, it will give you some concrete examples to play with as you read more about the `Monad` class in general. The first few examples are from the standard `Prelude`; the remaining examples are from the `transformers` package.

1. The simplest possible instance of `Monad` is `Identity`, which is described in Dan Piponi's highly recommended blog post on The Trivial Monad. Despite being "trivial", it is a great introduction to the `Monad` type class, and contains some good exercises to get your brain working.
2. The next simplest instance of `Monad` is `Maybe`. We already know how to write `return`/`pure` for `Maybe`. So how do we write `(>>=)`? Well, let's think about its type. Specializing for `Maybe`, we have

   ```
   (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b.
   ```

   If the first argument to `(>>=)` is `Just x`, then we have something of type `a` (namely, `x`), to which we can apply the second argument—resulting in a `Maybe b`, which is exactly what we wanted. What if the first argument to `(>>=)` is `Nothing`? In that case, we don't have anything to which we can apply the `a -> Maybe b` function, so there's only one thing we can do: yield `Nothing`. This instance is:

   ```
   instance Monad Maybe where
     return = Just
     (Just x) >>= g = g x
     Nothing  >>= _ = Nothing
   ```

   We can already get a bit of intuition as to what is going on here: if we build up a computation by chaining together a bunch of functions with `(>>=)`, as soon as any one of them fails, the entire computation will fail (because `Nothing >>= f` is `Nothing`, no matter what `f` is). The entire computation succeeds only if all the constituent functions individually succeed. So the `Maybe` monad models computations which may fail.
3. The `Monad` instance for the list constructor `[]` is similar to its `Applicative` instance; see the exercise below.
4. Of course, the `IO` constructor is famously a `Monad`, but its implementation is somewhat magical, and may in fact differ from compiler to compiler. It is worth emphasizing that the `IO` monad is the *only* monad which is magical. It allows us to build up, in an entirely pure way, values representing possibly effectful computations. The special value `main`, of type `IO ()`, is taken by the runtime and actually executed, producing actual effects. Every other monad is functionally pure, and requires no special compiler support. We often speak of monadic values as "effectful computations", but this is because some monads allow us to write code *as if* it has side effects, when in fact the monad is hiding the plumbing which allows these apparent side effects to be implemented in a functionally pure way.
5. As mentioned earlier, `((->) e)` is known as the *reader monad*, since it describes computations in which a value of type `e` is available as a read-only environment.
   The `Control.Monad.Reader` module provides the `Reader e a` type, which is just a convenient `newtype` wrapper around `(e -> a)`, along with an appropriate `Monad` instance and some `Reader`-specific utility functions such as `ask` (retrieve the environment), `asks` (retrieve a function of the environment), and `local` (run a subcomputation under a different environment).
6. The `Control.Monad.Writer` module provides the `Writer` monad, which allows information to be collected as a computation progresses. `Writer w a` is isomorphic to `(a,w)`, where the output value `a` is carried along with an annotation or "log" of type `w`, which must be an instance of `Monoid` (see section Monoid); the special function `tell` performs logging.
7. The `Control.Monad.State` module provides the `State s a` type, a `newtype` wrapper around `s -> (a,s)`. Something of type `State s a` represents a stateful computation which produces an `a` but can access and modify the state of type `s` along the way. The module also provides `State`-specific utility functions such as `get` (read the current state), `gets` (read a function of the current state), `put` (overwrite the state), and `modify` (apply a function to the state).
8. The `Control.Monad.Cont` module provides the `Cont` monad, which represents computations in continuation-passing style. It can be used to suspend and resume computations, and to implement non-local transfers of control, co-routines, other complex control structures—all in a functionally pure way. `Cont` has been called the "mother of all monads" because of its universal properties.

Intuition

Let's look more closely at the type of `(>>=)`. The basic intuition is that it combines two computations into one larger computation. The first argument, `m a`, is the first computation. However, it would be boring if the second argument were just an `m b`; then there would be no way for the computations to interact with one another (actually, this is exactly the situation with `Applicative`). So, the second argument to `(>>=)` has type `a -> m b`: a function of this type, given a *result* of the first computation, can produce a second computation to be run. In other words, `x >>= k` is a computation which runs `x`, and then uses the result(s) of `x` to *decide* what computation to run second, using the output of the second computation as the result of the entire computation.

Intuitively, it is this ability to use the output from previous computations to decide what computations to run next that makes `Monad` more powerful than `Applicative`. The structure of an `Applicative` computation is fixed, whereas the structure of a `Monad` computation can change based on intermediate results. This also means that parsers built using an `Applicative` interface can only parse context-free languages; in order to parse context-sensitive languages a `Monad` interface is needed.

To see the increased power of `Monad` from a different point of view, let's see what happens if we try to implement `(>>=)` in terms of `fmap`, `pure`, and `(<*>)`. We are given a value `x` of type `m a`, and a function `k` of type `a -> m b`, so the only thing we can do is apply `k` to `x`. We can't apply it directly, of course; we have to use `fmap` to lift it over the `m`. But what is the type of `fmap k`? Well, it's `m a -> m (m b)`. So after we apply it to `x`, we are left with something of type `m (m b)`—but now we are stuck; what we really want is an `m b`, but there's no way to get there from here. We can *add* `m`'s using `pure`, but we have no way to *collapse* multiple `m`'s into one.

This ability to collapse multiple `m`'s is exactly the ability provided by the function `join :: m (m a) -> m a`, and it should come as no surprise that an alternative definition of `Monad` can be given in terms of `join`:

```haskell
class Applicative m => Monad'' m where
  join :: m (m a) -> m a
```

In fact, the canonical definition of monads in category theory is in terms of `return`, `fmap`, and `join` (often called η, T, and μ in the mathematical literature). Haskell uses an alternative formulation with `(>>=)` instead of `join` since it is more convenient to use . However, sometimes it can be easier to think about `Monad` instances in terms of `join`, since it is a more "atomic" operation. (For example, `join` for the list monad is just `concat`.)

Utility functions

The `Control.Monad` module provides a large number of convenient utility functions, all of which can be implemented in terms of the basic `Monad` operations (`return` and `(>>=)` in particular). We have already seen one of them, namely, `join`. We also mention some other noteworthy ones here; implementing these utility functions oneself is a good exercise. For a more detailed guide to these functions, with commentary and example code, see Henk-Jan van Tuyl's tour.

- `liftM :: Monad m => (a -> b) -> m a -> m b`. This should be familiar; of course, it is just `fmap`. The fact that we have both `fmap` and `liftM` is an unfortunate consequence of the fact that the `Monad` type class does not require a `Functor` instance, even though mathematically speaking, every monad is a functor. However, `fmap` and `liftM` are essentially interchangeable, since it is a bug (in a social rather than technical sense) for any type to be an instance of `Monad` without also being an instance of `Functor` .

- `ap :: Monad m => m (a -> b) -> m a -> m b` should also be familiar: it is equivalent to `(<*>)`, justifying the claim that the `Monad` interface is strictly more powerful than `Applicative`. We can make any `Monad` into an instance of `Applicative` by setting `pure = return` and `(<*>) = ap`.

- `sequence :: Monad m => [m a] -> m [a]` takes a list of computations and combines them into one computation which collects a list of their results. It is again something of a historical accident that `sequence` has a `Monad` constraint, since it can actually be implemented only in terms of `Applicative`. There is an additional generalization of `sequence` to structures other than lists, which will be discussed in the section on `Traversable`.

- `replicateM :: Monad m => Int -> m a -> m [a]` is simply a combination of `replicate` and `sequence`.

- `when :: Monad m => Bool -> m () -> m ()` conditionally executes a computation, evaluating to its second argument if the test is `True`, and to `return ()` if the test is `False`. A collection of other sorts of monadic conditionals can be found in the `IfElse` package.

- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]` maps its first argument over the second, and `sequence`s the results. The `forM` function is just `mapM` with its arguments reversed; it is called `forM` since it models generalized `for` loops: the list `[a]` provides the loop indices, and the function `a -> m b` specifies the "body" of the loop for each index.

- `(=<<) :: Monad m => (a -> m b) -> m a -> m b` is just `(>>=)` with its arguments reversed; sometimes this direction is more convenient since it corresponds more closely to function application.

- `(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c` is sort of like function composition, but with an extra `m` on the result type of each function, and the arguments swapped. We'll have more to say about this operation later. There is also a flipped variant, `(<=<)`.

- The `guard` function is for use with instances of `MonadPlus`, which is discussed at the end of the `Monoid` section.

Many of these functions also have "underscored" variants, such as `sequence_` and `mapM_`; these variants throw away the results of the computations passed to them as arguments, using them only for their side effects.

Other monadic functions which are occasionally useful include `filterM`, `zipWithM`, `foldM`, and `forever`.

Laws

There are several laws that instances of `Monad` should satisfy (see also the Monad laws wiki page). The standard presentation is:

```
return a >>= k  =  k a
m >>= return    =  m
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h

fmap f xs  =  xs >>= return . f  =  liftM f xs
```

The first and second laws express the fact that `return` behaves nicely: if we inject a value `a` into a monadic context with `return`, and then bind to `k`, it is the same as just applying `k` to `a` in the first place; if we bind a computation `m` to `return`, nothing changes. The third law essentially says that `(>>=)` is associative, sort of. The last law ensures that `fmap` and `liftM` are the same for types which are instances of both `Functor` and `Monad`—which, as already noted, should be every instance of `Monad`.

However, the presentation of the above laws, especially the third, is marred by the asymmetry of `(>>=)`. It's hard to look at the laws and see what they're really saying. I prefer a much more elegant version of the laws, which is formulated in terms of `(>=>)` . Recall that `(>=>)` "composes" two functions of type `a -> m b` and `b -> m c`. You can think of something of type `a -> m b` (roughly) as a function from `a` to `b` which may also have some sort of effect in the context corresponding to `m`. `(>=>)` lets us compose these "effectful functions", and we would like to know what properties `(>=>)` has. The monad laws reformulated in terms of `(>=>)` are:

```
return >=> g  =  g
g >=> return  =  g
(g >=> h) >=> k  =  g >=> (h >=> k)
```

Ah, much better! The laws simply state that `return` is the identity of `(>=>)`, and that `(>=>)` is associative .

There is also a formulation of the monad laws in terms of `fmap`, `return`, and `join`; for a discussion of this formulation, see the Haskell wikibook page on category theory.

do notation

Haskell's special `do` notation supports an "imperative style" of programming by providing syntactic sugar for chains of monadic expressions. The genesis of the notation lies in realizing that something like `a >>= \x -> b >> c >>= \y -> d` can be more readably written by putting successive computations on separate lines:

```
a >>= \x ->
b >>
c >>= \y ->
d
```

This emphasizes that the overall computation consists of four computations `a`, `b`, `c`, and `d`, and that `x` is bound to the result of `a`, and `y` is bound to the result of `c` (`b`, `c`, and `d` are allowed to refer to `x`, and `d` is allowed to refer to `y` as well). From here it is not hard to imagine a nicer notation:

```
do { x <- a
   ;      b
   ; y <- c
   ;      d
   }
```

(The curly braces and semicolons may optionally be omitted; the Haskell parser uses layout to determine where they should be inserted.) This discussion should make clear that `do` notation is just syntactic sugar. In fact, `do` blocks are recursively translated into monad operations (almost) like this:

```
                  do e → e
       do { e; stmts } → e >> do { stmts }
  do { v <- e; stmts } → e >>= \v -> do { stmts }
do { let decls; stmts} → let decls in do { stmts }
```

This is not quite the whole story, since `v` might be a pattern instead of a variable. For example, one can write

```
do (x:xs) <- foo
   bar x
```

but what happens if `foo` produces an empty list? Well, remember that ugly `fail` function in the `Monad` type class declaration? That's what happens. See section 3.14 of the Haskell Report for the full details. See also the discussion of `MonadPlus` and `MonadZero` in the section on other monoidal classes.

A final note on intuition: `do` notation plays very strongly to the "computational context" point of view rather than the "container" point of view, since the binding notation `x <- m` is suggestive of "extracting" a single `x` from `m` and doing something with it. But `m` may represent some sort of a container, such as a list or a tree; the meaning of `x <- m` is entirely dependent on the implementation of `(>>=)`. For example, if `m` is a list, `x <- m` actually means that `x` will take on each value from the list in turn.

Further reading

Philip Wadler was the first to propose using monads to structure functional programs. His paper is still a readable introduction to the subject.

There are, of course, numerous monad tutorials of varying quality .

A few of the best include Cale Gibbard's Monads as containers and Monads as computation; Jeff Newbern's All About Monads, a comprehensive guide with lots of examples; and Dan Piponi's You Could Have Invented Monads!, which features great exercises. If you just want to know how to use `IO`, you could consult the Introduction to IO. Even this is just a sampling; the monad tutorials timeline is a more complete list. (All these monad tutorials have prompted parodies like think of a monad ... as well as other kinds of backlash like Monads! (and Why Monad Tutorials Are All Awful) or Abstraction, intuition, and the “monad tutorial fallacy”.)

Other good monad references which are not necessarily tutorials include Henk-Jan van Tuyl’s tour of the functions in `Control.Monad`, Dan Piponi’s field guide, Tim Newsham’s What’s a Monad?, and Chris Smith’s excellent article Why Do Monads Matter?. There are also many blog posts which have been written on various aspects of monads; a collection of links can be found under Blog articles/Monads.

For help constructing monads from scratch, and for obtaining a “deep embedding” of monad operations suitable for use in, say, compiling a domain-specific language, see apfelmus’s operational package.

One of the quirks of the `Monad` class and the Haskell type system is that it is not possible to straightforwardly declare `Monad` instances for types which require a class constraint on their data, even if they are monads from a mathematical point of view. For example, `Data.Set` requires an `Ord` constraint on its data, so it cannot be easily made an instance of `Monad`. A solution to this problem was first described by Eric Kidd, and later made into a library named rmonad by Ganesh Sittampalam and Peter Gavin.

There are many good reasons for eschewing `do` notation; some have gone so far as to consider it harmful.

Monads can be generalized in various ways; for an exposition of one possibility, see Robert Atkey’s paper on parameterized monads, or Dan Piponi’s Beyond Monads.

For the categorically inclined, monads can be viewed as monoids (From Monoids to Monads) and also as closure operators Triples and Closure. Derek Elkins’s article in issue 13 of the Monad.Reader contains an exposition of the category-theoretic underpinnings of some of the standard `Monad` instances, such as `State` and `Cont`. Jonathan Hill and Keith Clarke have an early paper explaining the connection between monads as they arise in category theory and as used in functional programming. There is also a web page by Oleg Kiselyov explaining the history of the IO monad.

Links to many more research papers related to monads can be found under Research papers/Monads and arrows.

Monad transformers

One would often like to be able to combine two monads into one: for example, to have stateful, nondeterministic computations (`State` + `[]`), or computations which may fail and can consult a read-only environment (`Maybe` + `Reader`), and so on. Unfortunately, monads do not compose as nicely as applicative functors (yet another reason to use `Applicative` if you don’t need the full power that `Monad` provides), but some monads can be combined in certain ways.

Standard monad transformers

The transformers library provides a number of standard *monad transformers*. Each monad transformer adds a particular capability/feature/effect to any existing monad.

- `IdentityT` is the identity transformer, which maps a monad to (something isomorphic to) itself. This may seem useless at first glance, but it is useful for the same reason that the `id` function is useful -- it can be passed as an argument to things which are parameterized over an arbitrary monad transformer, when you do not actually want any extra capabilities.
- `StateT` adds a read-write state.
- `ReaderT` adds a read-only environment.
- `WriterT` adds a write-only log.
- `RWST` conveniently combines `ReaderT`, `WriterT`, and `StateT` into one.
- `MaybeT` adds the possibility of failure.
- `ErrorT` adds the possibility of failure with an arbitrary type to represent errors.
- `ListT` adds non-determinism (however, see the discussion of `ListT` below).
- `ContT` adds continuation handling.

For example, `StateT s Maybe` is an instance of `Monad`; computations of type `StateT s Maybe a` may fail, and have access to a mutable state of type `s`. Monad transformers can be multiply stacked. One thing to keep in mind while using monad transformers is that the order of composition matters. For example, when a `StateT s Maybe a` computation fails, the state ceases being updated (indeed, it simply disappears); on the other hand, the state of a `MaybeT (State s) a` computation may continue to be modified even after the computation has "failed". This may seem backwards, but it is correct. Monad transformers build composite monads "inside out"; `MaybeT (State s) a` is isomorphic to `s -> (Maybe a, s)`. (Lambdabot has an indispensable `@unmtl` command which you can use to "unpack" a monad transformer stack in this way.) Intuitively, the monads become "more fundamental" the further down in the stack you get, and the effects of a given monad "have precedence" over the effects of monads further up the stack. Of course, this is just handwaving, and if you are unsure of the proper order for some monads you wish to combine, there is no substitute for using `@unmtl` or simply trying out the various options.

Definition and laws

All monad transformers should implement the `MonadTrans` type class, defined in `Control.Monad.Trans.Class`:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

It allows arbitrary computations in the base monad `m` to be "lifted" into computations in the transformed monad `t m`. (Note that type application associates to the left, just like function application, so `t m a = (t m) a`.)

`lift` must satisfy the laws

```
lift . return   =  return
lift (m >>= f)  =  lift m >>= (lift . f)
```

which intuitively state that `lift` transforms `m a` computations into `t m a` computations in a "sensible" way, which sends the `return` and `(>>=)` of `m` to the `return` and `(>>=)` of `t m`.

Transformer type classes and "capability" style

There are also type classes (provided by the `mtl` package) for the operations of each transformer. For example, the `MonadState` type class provides the state-specific methods `get` and `put`, allowing you to conveniently use these methods not only with `State`, but with any monad which is an instance of `MonadState`—including `MaybeT (State s)`, `StateT s (ReaderT r IO)`, and so on. Similar type classes exist for `Reader`, `Writer`, `Cont`, `IO`, and others .

These type classes serve two purposes. First, they get rid of (most of) the need for explicitly using `lift`, giving a type-directed way to automatically determine the right number of calls to `lift`. Simply writing `put` will be automatically translated into `lift . put`, `lift . lift . put`, or something similar depending on what concrete monad stack you are using.

Second, they give you more flexibility to switch between different concrete monad stacks. For example, if you are writing a state-based algorithm, don't write

```
foo :: State Int Char
foo = modify (*2) >> return 'x'
```

but rather

```
foo :: MonadState Int m => m Char
foo = modify (*2) >> return 'x'
```

Now, if somewhere down the line you realize you need to introduce the possibility of failure, you might switch from `State Int` to `MaybeT (State Int)`. The type of the first version of `foo` would need to be modified to reflect this change, but the second version of `foo` can still be used as-is.

However, this sort of "capability-based" style (e.g. specifying that `foo` works for any monad with the "state capability") quickly runs into problems when you try to naively scale it up: for example, what if you need to maintain two independent states? A framework for solving this and related problems is described by Schrijvers and Olivera (Monads, zippers and views: virtualizing the monad stack, ICFP 2011) and is implemented in the `Monatron` package.

Composing monads

Is the composition of two monads always a monad? As hinted previously, the answer is no. For example, *XXX insert example here.*

Since `Applicative` functors are closed under composition, the problem must lie with `join`. Indeed, suppose `m` and `n` are arbitrary monads; to make a monad out of their composition we would need to be able to implement

```haskell
join :: m (n (m (n a))) -> m (n a)
```

but it is not clear how this could be done in general. The `join` method for `m` is no help, because the two occurrences of `m` are not next to each other (and likewise for `n`).

However, one situation in which it can be done is if `n` *distributes* over `m`, that is, if there is a function

```haskell
distrib :: n (m a) -> m (n a)
```

satisfying certain laws. See Jones and Duponcheel (Composing Monads); see also the section on Traversable.

Further reading

Much of the monad transformer library (originally `mtl`, now split between `mtl` and `transformers`), including the `Reader`, `Writer`, `State`, and other monads, as well as the monad transformer framework itself, was inspired by Mark Jones's classic paper Functional Programming with Overloading and Higher-Order Polymorphism. It's still very much worth a read—and highly readable—after almost fifteen years.

See Edward Kmett's mailing list message for a description of the history and relationships among monad transformer packages (`mtl`, `transformers`, `monads-fd`, `monads-tf`).

There are two excellent references on monad transformers. Martin Grabmüller's Monad Transformers Step by Step is a thorough description, with running examples, of how to use monad transformers to elegantly build up computations with various effects. Cale Gibbard's article on how to use monad transformers is more practical, describing how to structure code using monad transformers to make writing it as painless as possible. Another good starting place for learning about monad transformers is a blog post by Dan Piponi.

The `ListT` transformer from the `transformers` package comes with the caveat that `ListT m` is only a monad when `m` is *commutative*, that is, when `ma >>= \a -> mb >>= \b -> foo` is equivalent to `mb >>= \b -> ma >>= \a -> foo` (i.e. the order of `m`'s effects does not matter). For one explanation why, see Dan Piponi's blog post "Why isn't `<nowiki>ListT []</nowiki>` a monad". For more examples, as well as a design for a version of `ListT` which does not have this problem, see `ListT` done right.

There is an alternative way to compose monads, using coproducts, as described by Lüth and Ghani. This method is interesting but has not (yet?) seen widespread use.

MonadFix

Note: `MonadFix` is included here for completeness (and because it is interesting) but seems not to be used much. Skipping this section on a first read-through is perfectly OK (and perhaps even recommended).

mdo/do rec notation

The MonadFix class describes monads which support the special fixpoint operation `mfix :: (a -> m a) -> m a`, which allows the output of monadic computations to be defined via (effectful) recursion. This is supported in GHC by a special "recursive do" notation, enabled by the `-XDoRec` flag. Within a `do` block, one may have a nested `rec` block, like so:

```
do { x <- foo
   ; rec { y <- baz
         ; z <- bar
         ;      bob
         }
   ; w <- frob
   }
```

Normally (if we had `do` in place of `rec` in the above example), `y` would be in scope in `bar` and `bob` but not in `baz`, and `z` would be in scope only in `bob`. With the `rec`, however, `y` and `z` are both in scope in all three of `baz`, `bar`, and `bob`. A `rec` block is analogous to a `let` block such as

```
let { y = baz
    ; z = bar
    }
in bob
```

because, in Haskell, every variable bound in a `let`-block is in scope throughout the entire block. (From this point of view, Haskell's normal `do` blocks are analogous to Scheme's `let*` construct.)

What could such a feature be used for? One of the motivating examples given in the original paper describing `MonadFix` (see below) is encoding circuit descriptions. A line in a `do`-block such as

```
  x <- gate y z
```

describes a gate whose input wires are labeled `y` and `z` and whose output wire is labeled `x`. Many (most?) useful circuits, however, involve some sort of feedback loop, making them impossible to write in a normal `do`-block (since some wire would have to be mentioned as an input *before* being listed as an output). Using a `rec` block solves this problem.

Examples and intuition

Of course, not every monad supports such recursive binding. However, as mentioned above, it suffices to have an implementation of `mfix :: (a -> m a) -> m a`, satisfying a few laws. Let's try implementing `mfix` for the `Maybe` monad. That is, we want to implement a function

```
maybeFix :: (a -> Maybe a) -> Maybe a
```

Let's think for a moment about the implementation of the non-monadic `fix :: (a -> a) -> a`:

```
fix f = f (fix f)
```

Inspired by `fix`, our first attempt at implementing `maybeFix` might be something like

```
maybeFix :: (a -> Maybe a) -> Maybe a
maybeFix f = maybeFix f >>= f
```

This has the right type. However, something seems wrong: there is nothing in particular here about `Maybe`; `maybeFix` actually has the more general type `Monad m => (a -> m a) -> m a`. But didn't we just say that not all monads support `mfix`?

The answer is that although this implementation of `maybeFix` has the right type, it does *not* have the intended semantics. If we think about how `(>>=)` works for the `Maybe` monad (by pattern-matching on its first argument to see whether it is `Nothing` or `Just`) we can see that this definition of `maybeFix` is completely useless: it will just recurse infinitely, trying to decide whether it is going to return `Nothing` or `Just`, without ever even so much as a glance in the direction of `f`.

The trick is to simply *assume* that `maybeFix` will return `Just`, and get on with life!

```
maybeFix :: (a -> Maybe a) -> Maybe a
maybeFix f = ma
  where ma = f (fromJust ma)
```

This says that the result of `maybeFix` is `ma`, and assuming that `ma = Just x`, it is defined (recursively) to be equal to `f x`.

Why is this OK? Isn't `fromJust` almost as bad as `unsafePerformIO`? Well, usually, yes. This is just about the only situation in which it is justified! The interesting thing to note is that `maybeFix` *will never crash* -- although it may, of course, fail to terminate. The only way we could get a crash is if we try to evaluate `fromJust ma` when we know that `ma = Nothing`. But how could we know `ma = Nothing`? Since `ma` is defined as `f (fromJust ma)`, it must be that this expression has already been evaluated to `Nothing` -- in which case there is no reason for us to be evaluating `fromJust ma` in the first place!

To see this from another point of view, we can consider three possibilities. First, if `f` outputs `Nothing` without looking at its argument, then `maybeFix f` clearly returns `Nothing`. Second, if `f` always outputs `Just x`, where `x` depends on its argument, then the recursion can proceed usefully: `fromJust ma` will be able to evaluate to `x`, thus feeding `f`'s output back to it as input. Third, if `f` tries to use its argument to decide whether to output `Just` or `Nothing`, then `maybeFix f` will not terminate: evaluating `f`'s argument requires evaluating `ma` to see whether it is `Just`, which requires evaluating `f (fromJust ma)`, which requires evaluating `ma`, ... and so on.

There are also instances of `MonadFix` for lists (which works analogously to the instance for `Maybe`), for `ST`, and for `IO`. The instance for `IO` is particularly amusing: it creates a new `IORef` (with a dummy value), immediately reads its contents using `unsafeInterleaveIO` (which delays the actual reading lazily until the value is needed), uses the contents of the `IORef` to compute a new value, which it then writes back into the `IORef`. It almost seems, spookily, that `mfix` is sending a value back in time to itself through the `IORef` -- though of course what is really going on is that the reading is delayed just long enough (via `unsafeInterleaveIO`) to get the process bootstrapped.

GHC 7.6 changes

GHC 7.6 reinstated the old `mdo` syntax, so the example at the start of this section can be written

```
mdo { x <- foo
    ; y <- baz
    ; z <- bar
    ;      bob
    ; w <- frob
    }
```

which will be translated into the original example (assuming that, say, `bar` and `bob` refer to `y`. The difference is that `mdo` will analyze the code in order to find minimal recursive blocks, which will be placed in `rec` blocks, whereas `rec` blocks desugar directly into calls to `mfix` without any further analysis.

Further reading

For more information (such as the precise desugaring rules for `rec` blocks), see Levent Erkök and John Launchbury's 2002 Haskell workshop paper, A Recursive do for Haskell, or for full details, Levent Erkök's thesis, Value Recursion in Monadic Computations. (Note, while reading, that `MonadFix` used to be called `MonadRec`.) You can also read the GHC user manual section on recursive do-notation.

Semigroup

A semigroup is a set S together with a binary operation $\oplus$ which combines elements from S. The $\oplus$ operator is required to be associative (that is, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, for any a, b, c which are elements of S).

For example, the natural numbers under addition form a semigroup: the sum of any two natural numbers is a natural number, and $(a + b) + c = a + (b + c)$ for any natural numbers a, b, and c. The integers under multiplication also form a semigroup, as do the integers (or rationals, or reals) under $\max$ or $\min$, Boolean values under conjunction and disjunction, lists under concatenation, functions from a set to itself under composition ... Semigroups show up all over the place, once you know to look for them.

Definition

Semigroups are not (yet?) defined in the base package, but the package provides a standard definition.

The definition of the `Semigroup` type class (haddock) is as follows:

```
class Semigroup a where
  (<>) :: a -> a -> a

  sconcat :: NonEmpty a -> a
  sconcat = sconcat (a :| as) = go a as where
    go b (c:cs) = b <> go c cs
    go b []     = b

  times1p :: Whole n => n -> a -> a
  times1p = ...
```

The really important method is `(<>)`, representing the associative binary operation. The other two methods have default implementations in terms of `(<>)`, and are included in the type class in case some instances can give more efficient implementations than the default. `sconcat` reduces a nonempty list using `(<>)`; `times1p n` is equivalent to (but more efficient than) `sconcat . replicate n`. See the haddock documentation for more information on `sconcat` and `times1p`.

Laws

The only law is that `(<>)` must be associative:

```
(x <> y) <> z = x <> (y <> z)
```

Monoid

Many semigroups have a special element e for which the binary operation $\oplus$ is the identity, that is, $e \oplus x = x \oplus e = x$ for every element x. Such a semigroup-with-identity-element is called a *monoid*.

Definition

The definition of the `Monoid` type class (defined in `Data.Monoid`; haddock) is:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

The `mempty` value specifies the identity element of the monoid, and `mappend` is the binary operation. The default definition for `mconcat` "reduces" a list of elements by combining them all with `mappend`, using a right fold. It is only in the `Monoid` class so that specific instances have the option of providing an alternative, more efficient implementation; usually, you can safely ignore `mconcat` when creating a `Monoid` instance, since its default definition will work just fine.

The `Monoid` methods are rather unfortunately named; they are inspired by the list instance of `Monoid`, where indeed `mempty = []` and `mappend = (++)`, but this is misleading since many monoids have little to do with appending (see these Comments from OCaml Hacker Brian Hurt on the haskell-cafe mailing list). This was improved in GHC 7.4, where `(<>)` was added as an alias to `mappend`.

Laws

Of course, every `Monoid` instance should actually be a monoid in the mathematical sense, which implies these laws:

```
mempty `mappend` x = x
x `mappend` mempty = x
(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)
```

Instances

There are quite a few interesting `Monoid` instances defined in `Data.Monoid`.

1. `[a]` is a `Monoid`, with `mempty = []` and `mappend = (++)`. It is not hard to check that `(x ++ y) ++ z = x ++ (y ++ z)` for any lists `x`, `y`, and `z`, and that the empty list is the identity: `[] ++ x = x ++ [] = x`.
2. As noted previously, we can make a monoid out of any numeric type under either addition or multiplication. However, since we can't have two instances for the same type, `Data.Monoid` provides two `newtype` wrappers, `Sum` and `Product`, with appropriate `Monoid` instances.

   ```
   > getSum (mconcat . map Sum $ [1..5])
   15
   > getProduct (mconcat . map Product $ [1..5])
   120
   ```

   This example code is silly, of course; we could just write `sum [1..5]` and `product [1..5]`. Nevertheless, these instances are useful in more generalized settings, as we will see in the section on `Foldable`.
3. `Any` and `All` are `newtype` wrappers providing `Monoid` instances for `Bool` (under disjunction and conjunction, respectively).
4. There are three instances for `Maybe`: a basic instance which lifts a `Monoid` instance for `a` to an instance for `Maybe a`, and two `newtype` wrappers `First` and `Last` for which `mappend` selects the first (respectively last) non-`Nothing` item.
5. `Endo a` is a newtype wrapper for functions `a -> a`, which form a monoid under composition.
6. There are several ways to "lift" `Monoid` instances to instances with additional structure. We have already seen that an instance for `a` can be lifted to an instance for `Maybe a`. There are also tuple instances: if `a` and `b` are instances of `Monoid`, then so is `(a,b)`, using the monoid operations for `a` and `b` in the obvious pairwise manner.

Finally, if `a` is a `Monoid`, then so is the function type `e -> a` for any `e`; in particular, `g ‘mappend‘ h` is the function which applies both `g` and `h` to its argument and then combines the results using the underlying `Monoid` instance for `a`. This can be quite useful and elegant (see example).

7. The type `Ordering = LT | EQ | GT` is a `Monoid`, defined in such a way that `mconcat (zipWith compare xs ys)` computes the lexicographic ordering of `xs` and `ys` (if `xs` and `ys` have the same length). In particular, `mempty = EQ`, and `mappend` evaluates to its leftmost non-`EQ` argument (or `EQ` if both arguments are `EQ`). This can be used together with the function instance of `Monoid` to do some clever things (example).
8. There are also `Monoid` instances for several standard data structures in the containers library (haddock), including `Map`, `Set`, and `Sequence`.

`Monoid` is also used to enable several other type class instances. As noted previously, we can use `Monoid` to make `((,) e)` an instance of `Applicative`:

```
instance Monoid e => Applicative ((,) e) where
  pure x = (mempty, x)
  (u, f) <*> (v, x) = (u `mappend` v, f x)
```

`Monoid` can be similarly used to make `((,) e)` an instance of `Monad` as well; this is known as the *writer monad*. As we’ve already seen, `Writer` and `WriterT` are a newtype wrapper and transformer for this monad, respectively.

`Monoid` also plays a key role in the `Foldable` type class (see section Foldable).

Other monoidal classes: Alternative, MonadPlus, ArrowPlus

The `Alternative` type class (haddock) is for `Applicative` functors which also have a monoid structure:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Of course, instances of `Alternative` should satisfy the monoid laws

```
empty <|> x = x
x <|> empty = x
(x <|> y) <|> z = x <|> (y <|> z)
```

Likewise, `MonadPlus` (haddock) is for `Monad`s with a monoid structure:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

The `MonadPlus` documentation states that it is intended to model monads which also support “choice and failure”; in addition to the monoid laws, instances of `MonadPlus` are expected to satisfy

```
mzero >>= f   =  mzero
v >> mzero    =  mzero
```

which explains the sense in which `mzero` denotes failure. Since `mzero` should be the identity for `mplus`, the computation `m1 ‘mplus‘ m2` succeeds (evaluates to something other than `mzero`) if either `m1` or `m2` does; so `mplus` represents choice. The `guard` function can also be used with instances of `MonadPlus`; it requires a condition to be satisfied and fails (using `mzero`) if it is not. A simple example of a `MonadPlus` instance is `[]`, which is exactly the same as the `Monoid` instance for `[]`: the empty list represents failure, and list concatenation represents choice. In general, however, a `MonadPlus` instance for a type need not be the same as its `Monoid` instance; `Maybe` is an example of such a type. A great introduction to the `MonadPlus` type class, with interesting examples of its use, is Doug Auclair's *MonadPlus: What a Super Monad!* in the Monad.Reader issue 11.

There used to be a type class called `MonadZero` containing only `mzero`, representing monads with failure. The `do`-notation requires some notion of failure to deal with failing pattern matches. Unfortunately, `MonadZero` was scrapped in favor of adding the `fail` method to the `Monad` class. If we are lucky, someday `MonadZero` will be restored, and `fail` will be banished to the bit bucket where it belongs (see MonadPlus reform proposal). The idea is that any `do`-block which uses pattern matching (and hence may fail) would require a `MonadZero` constraint; otherwise, only a `Monad` constraint would be required.

Finally, `ArrowZero` and `ArrowPlus` (haddock) represent `Arrows` (see below) with a monoid structure:

```
class Arrow arr => ArrowZero arr where
  zeroArrow :: b `arr` c

class ArrowZero arr => ArrowPlus arr where
  (<+>) :: (b `arr` c) -> (b `arr` c) -> (b `arr` c)
```

Further reading

Monoids have gotten a fair bit of attention recently, ultimately due to a blog post by Brian Hurt, in which he complained about the fact that the names of many Haskell type classes (`Monoid` in particular) are taken from abstract mathematics. This resulted in a long haskell-cafe thread arguing the point and discussing monoids in general.

However, this was quickly followed by several blog posts about `Monoid` . First, Dan Piponi wrote a great introductory post, Haskell Monoids and their Uses. This was quickly followed by Heinrich Apfelmus's Monoids and Finger Trees, an accessible exposition of Hinze and Paterson's classic paper on 2-3 finger trees, which makes very clever use of `Monoid` to implement an elegant and generic data structure. Dan Piponi then wrote two fascinating articles about using `Monoids` (and finger trees): Fast Incremental Regular Expressions and Beyond Regular Expressions

In a similar vein, David Place's article on improving `Data.Map` in order to compute incremental folds (see the Monad Reader issue 11) is also a good example of using `Monoid` to generalize a data structure.

Some other interesting examples of `Monoid` use include building elegant list sorting combinators, collecting unstructured information, combining probability distributions, and a brilliant series of posts by Chung-Chieh Shan and Dylan Thurston using `Monoids` to elegantly solve a difficult combinatorial puzzle (followed by part 2, part 3, part 4).

As unlikely as it sounds, monads can actually be viewed as a sort of monoid, with `join` playing the role of the binary operation and `return` the role of the identity; see Dan Piponi's blog post.

Foldable

The `Foldable` class, defined in the `Data.Foldable` module (haddock), abstracts over containers which can be "folded" into a summary value. This allows such folding operations to be written in a container-agnostic way.

Definition

The definition of the `Foldable` type class is:

```
class Foldable t where
  fold    :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m

  foldr   :: (a -> b -> b) -> b -> t a -> b
  foldl   :: (a -> b -> a) -> a -> t b -> a
  foldr1  :: (a -> a -> a) -> t a -> a
  foldl1  :: (a -> a -> a) -> t a -> a
```

This may look complicated, but in fact, to make a Foldable instance you only need to implement one method: your choice of foldMap or foldr. All the other methods have default implementations in terms of these, and are presumably included in the class in case more efficient implementations can be provided.

Instances and examples

The type of foldMap should make it clear what it is supposed to do: given a way to convert the data in a container into a Monoid (a function a -> m) and a container of a's (t a), foldMap provides a way to iterate over the entire contents of the container, converting all the a's to m's and combining all the m's with mappend. The following code shows two examples: a simple implementation of foldMap for lists, and a binary tree example provided by the Foldable documentation.

```haskell
instance Foldable [] where
  foldMap g = mconcat . map g

data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)

instance Foldable Tree where
  foldMap f Empty        = mempty
  foldMap f (Leaf x)     = f x
  foldMap f (Node l k r) = foldMap f l `mappend` f k `mappend` foldMap f r
```

The foldr function has a type similar to the foldr found in the Prelude, but more general, since the foldr in the Prelude works only on lists.

The Foldable module also provides instances for Maybe and Array; additionally, many of the data structures found in the standard containers library (for example, Map, Set, Tree, and Sequence) provide their own Foldable instances.

Derived folds

Given an instance of Foldable, we can write generic, container-agnostic functions such as:

```haskell
-- Compute the size of any container.
containerSize :: Foldable f => f a -> Int
containerSize = getSum . foldMap (const (Sum 1))

-- Compute a list of elements of a container satisfying a predicate.
filterF :: Foldable f => (a -> Bool) -> f a -> [a]
filterF p = foldMap (\a -> if p a then [a] else [])

-- Get a list of all the Strings in a container which include the
-- letter a.
aStrings :: Foldable f => f String -> [String]
aStrings = filterF (elem 'a')
```

The Foldable module also provides a large number of predefined folds, many of which are generalized versions of Prelude functions of the same name that only work on lists: concat, concatMap, and, or, any, all, sum, product, maximum(By), minimum(By), elem, notElem, and find.

The important function toList is also provided, which turns any Foldable structure into a list of its elements in left-right order; it works by folding with the list monoid.

There are also generic functions that work with Applicative or Monad instances to generate some sort of computation from each element in a container, and then perform all the side effects from those computations, discarding the results: traverse_, sequenceA_, and others. The results must be discarded because the Foldable class is too weak to specify what to do with them: we cannot, in general, make an arbitrary Applicative or Monad instance into a

Monoid, but we can make m () into a Monoid for any such m. If we do have an Applicative or Monad with a monoid structure—that is, an Alternative or a MonadPlus—then we can use the asum or msum functions, which can combine the results as well. Consult the Foldable documentation for more details on any of these functions.

Note that the Foldable operations always forget the structure of the container being folded. If we start with a container of type t a for some Foldable t, then t will never appear in the output type of any operations defined in the Foldable module. Many times this is exactly what we want, but sometimes we would like to be able to generically traverse a container while preserving its structure—and this is exactly what the Traversable class provides, which will be discussed in the next section.

Foldable actually isn't

The generic term "fold" is often used to refer to the more technical concept of catamorphism. Intuitively, given a way to summarize "one level of structure" (where recursive subterms have already been replaced with their summaries), a catamorphism can summarize an entire recursive structure. It is important to realize that Foldable does not correspond to catamorphisms, but to something weaker. In particular, Foldable allows observing only the left-right order of elements within a structure, not the actual structure itself. Put another way, every use of Foldable can be expressed in terms of toList. For example, fold itself is equivalent to mconcat . toList.

This is sufficient for many tasks, but not all. For example, consider trying to compute the depth of a Tree: try as we might, there is no way to implement it using Foldable. However, it can be implemented as a catamorphism.

Further reading

The Foldable class had its genesis in McBride and Paterson's paper introducing Applicative, although it has been fleshed out quite a bit from the form in the paper.

An interesting use of Foldable (as well as Traversable) can be found in Janis Voigtländer's paper Bidirectionalization for free!.

Traversable

Definition

The Traversable type class, defined in the Data.Traversable module (haddock), is:

```
class (Functor t, Foldable t) => Traversable t where
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM      ::       Monad m => (a -> m b) -> t a -> m (t b)
  sequence  ::       Monad m => t (m a) -> m (t a)
```

As you can see, every Traversable is also a foldable functor. Like Foldable, there is a lot in this type class, but making instances is actually rather easy: one need only implement traverse or sequenceA; the other methods all have default implementations in terms of these functions. A good exercise is to figure out what the default implementations should be: given either traverse or sequenceA, how would you define the other three methods? (Hint for mapM: Control.Applicative exports the WrapMonad newtype, which makes any Monad into an Applicative. The sequence function can be implemented in terms of mapM.)

Intuition

The key method of the Traversable class, and the source of its unique power, is sequenceA. Consider its type:

```
  sequenceA :: Applicative f => t (f a) -> f (t a)
```

This answers the fundamental question: when can we commute two functors? For example, can we turn a tree of lists into a list of trees?

The ability to compose two monads depends crucially on this ability to commute functors. Intuitively, if we want to build a composed monad `M a = m (n a)` out of monads `m` and `n`, then to be able to implement `join :: M (M a) -> M a`, that is, `join :: m (n (m (n a))) -> m (n a)`, we have to be able to commute the `n` past the `m` to get `m (m (n (n a)))`, and then we can use the `join`s for `m` and `n` to produce something of type `m (n a)`. See Mark Jones's paper for more details.

Alternatively, looking at the type of `traverse`,

```
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

leads us to view `Traversable` as a generalization of `Functor`. `traverse` is an "effectful `fmap`": it allows us to map over a structure of type `t a`, applying a function to every element of type `a` and in order to produce a new structure of type `t b`; but along the way the function may have some effects (captured by the applicative functor `f`).

Instances and examples

What's an example of a `Traversable` instance? The following code shows an example instance for the same `Tree` type used as an example in the previous `Foldable` section. It is instructive to compare this instance with a `Functor` instance for `Tree`, which is also shown.

```
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)

instance Traversable Tree where
  traverse g Empty        = pure Empty
  traverse g (Leaf x)     = Leaf <$> g x
  traverse g (Node l x r) = Node <$> traverse g l
                                 <*> g x
                                 <*> traverse g r

instance Functor Tree where
  fmap     g Empty        = Empty
  fmap     g (Leaf x)     = Leaf $ g x
  fmap     g (Node l x r) = Node (fmap g l)
                                 (g x)
                                 (fmap g r)
```

It should be clear that the `Traversable` and `Functor` instances for `Tree` are almost identical; the only difference is that the `Functor` instance involves normal function application, whereas the applications in the `Traversable` instance take place within an `Applicative` context, using `(<$>)` and `(<*>)`. In fact, this will be true for any type.

Any `Traversable` functor is also `Foldable`, and a `Functor`. We can see this not only from the class declaration, but by the fact that we can implement the methods of both classes given only the `Traversable` methods.

The standard libraries provide a number of `Traversable` instances, including instances for `[]`, `Maybe`, `Map`, `Tree`, and `Sequence`. Notably, `Set` is not `Traversable`, although it is `Foldable`.

Laws

Any instance of `Traversable` must satisfy the following two laws, where `Identity` is the identity functor (as defined in the `Data.Functor.Identity` module from the `transformers` package), and `Compose` wraps the composition of two functors (as defined in `Data.Functor.Compose`):

1. `traverse Identity = Identity`

2. `traverse (Compose . fmap g . f) = Compose . fmap (traverse g) . traverse f`

The first law essentially says that traversals cannot make up arbitrary effects. The second law explains how doing two traversals in sequence can be collapsed to a single traversal.

Additionally, suppose `eta` is an "`Applicative` morphism", that is,

```
eta :: forall a f g. (Applicative f, Applicative g) => f a -> g a
```

and `eta` preserves the `Applicative` operations: `eta (pure x) = pure x` and `eta (x <*> y) = eta x <*> eta y`. Then, by parametricity, any instance of `Traversable` satisfying the above two laws will also satisfy `eta . traverse f = traverse (eta . f)`.

Further reading

The `Traversable` class also had its genesis in McBride and Paterson's `Applicative` paper, and is described in more detail in Gibbons and Oliveira, The Essence of the Iterator Pattern, which also contains a wealth of references to related work.

`Traversable` forms a core component of Edward Kmett's lens library. Watching Edward's talk on the subject is a highly recommended way to gain better insight into `Traversable`, `Foldable`, `Applicative`, and many other things besides.

For references on the `Traversable` laws, see Russell O'Connor's mailing list post (and subsequent thread).

Category

`Category` is a relatively recent addition to the Haskell standard libraries. It generalizes the notion of function composition to general "morphisms".

The definition of the `Category` type class (from `Control.Category`; haddock) is shown below. For ease of reading, note that I have used an infix type variable `` `arr` ``, in parallel with the infix function type constructor `(->)`. This syntax is not part of Haskell 2010. The second definition shown is the one used in the standard libraries. For the remainder of this document, I will use the infix type constructor `` `arr` `` for `Category` as well as `Arrow`.

```
class Category arr where
  id  :: a `arr` a
  (.) :: (b `arr` c) -> (a `arr` b) -> (a `arr` c)

-- The same thing, with a normal (prefix) type constructor
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c
```

Note that an instance of `Category` should be a type constructor which takes two type arguments, that is, something of kind `* -> * -> *`. It is instructive to imagine the type constructor variable `cat` replaced by the function constructor `(->)`: indeed, in this case we recover precisely the familiar identity function `id` and function composition operator `(.)` defined in the standard `Prelude`.

Of course, the `Category` module provides exactly such an instance of `Category` for `(->)`. But it also provides one other instance, shown below, which should be familiar from the previous discussion of the `Monad` laws. `Kleisli m a b`, as defined in the `Control.Arrow` module, is just a `newtype` wrapper around `a -> m b`.

```
newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Category (Kleisli m) where
  id = Kleisli return
  Kleisli g . Kleisli h = Kleisli (h >=> g)
```

The only law that `Category` instances should satisfy is that `id` and `(.)` should form a monoid—that is, `id` should be the identity of `(.)`, and `(.)` should be associative.

Finally, the `Category` module exports two additional operators: `(<<<)`, which is just a synonym for `(.)`, and `(>>>)`, which is `(.)` with its arguments reversed. (In previous versions of the libraries, these operators were defined as part of the `Arrow` class.)

Further reading

The name `Category` is a bit misleading, since the `Category` class cannot represent arbitrary categories, but only categories whose objects are objects of `Hask`, the category of Haskell types. For a more general treatment of categories within Haskell, see the category-extras package. For more about category theory in general, see the excellent Haskell wikibook page, [http://books.google.com/books/about/Category_theory.html?id=-MCJ6x2lC7oC Steve Awodey's new book], Benjamin Pierce's Basic category theory for computer scientists, or Barr and Wells's category theory lecture notes. Benjamin Russell's blog post is another good source of motivation and category theory links. You certainly don't need to know any category theory to be a successful and productive Haskell programmer, but it does lend itself to much deeper appreciation of Haskell's underlying theory.

Arrow

The `Arrow` class represents another abstraction of computation, in a similar vein to `Monad` and `Applicative`. However, unlike `Monad` and `Applicative`, whose types only reflect their output, the type of an `Arrow` computation reflects both its input and output. Arrows generalize functions: if `arr` is an instance of `Arrow`, a value of type `b ‘arr‘ c` can be thought of as a computation which takes values of type `b` as input, and produces values of type `c` as output. In the `(->)` instance of `Arrow` this is just a pure function; in general, however, an arrow may represent some sort of "effectful" computation.

Definition

The definition of the `Arrow` type class, from `Control.Arrow` (haddock), is:

```haskell
class Category arr => Arrow arr where
  arr :: (b -> c) -> (b `arr` c)
  first :: (b `arr` c) -> ((b, d) `arr` (c, d))
  second :: (b `arr` c) -> ((d, b) `arr` (d, c))
  (***) :: (b `arr` c) -> (b' `arr` c') -> ((b, b') `arr` (c, c'))
  (&&&) :: (b `arr` c) -> (b `arr` c') -> (b `arr` (c, c'))
```

The first thing to note is the `Category` class constraint, which means that we get identity arrows and arrow composition for free: given two arrows `g :: b ‘arr‘ c` and `h :: c ‘arr‘ d`, we can form their composition `g >>> h :: b ‘arr‘ d` .

As should be a familiar pattern by now, the only methods which must be defined when writing a new instance of `Arrow` are `arr` and `first`; the other methods have default definitions in terms of these, but are included in the `Arrow` class so that they can be overridden with more efficient implementations if desired.

Intuition

Let's look at each of the arrow methods in turn. Ross Paterson's web page on arrows has nice diagrams which can help build intuition.

- The `arr` function takes any function `b -> c` and turns it into a generalized arrow `b ‘arr‘ c`. The `arr` method justifies the claim that arrows generalize functions, since it says that we can treat any function as an arrow. It is intended that the arrow `arr g` is "pure" in the sense that it only computes `g` and has no "effects" (whatever that might mean for any particular arrow type).

- The `first` method turns any arrow from `b` to `c` into an arrow from `(b,d)` to `(c,d)`. The idea is that `first g` uses `g` to process the first element of a tuple, and lets the second element pass through unchanged. For the function instance of `Arrow`, of course, `first g (x,y) = (g x, y)`.

- The `second` function is similar to `first`, but with the elements of the tuples swapped. Indeed, it can be defined in terms of `first` using an auxiliary function `swap`, defined by `swap (x,y) = (y,x)`.

- The `(***)` operator is "parallel composition" of arrows: it takes two arrows and makes them into one arrow on tuples, which has the behavior of the first arrow on the first element of a tuple, and the behavior of the second arrow on the second element. The mnemonic is that `g *** h` is the *product* (hence `*`) of `g` and `h`. For the function instance of `Arrow`, we define `(g *** h) (x,y) = (g x, h y)`. The default implementation of `(***)` is in terms of `first`, `second`, and sequential arrow composition `(>>>)`. The reader may also wish to think about how to implement `first` and `second` in terms of `(***)`.

- The `(&&&)` operator is "fanout composition" of arrows: it takes two arrows `g` and `h` and makes them into a new arrow `g &&& h` which supplies its input as the input to both `g` and `h`, returning their results as a tuple. The mnemonic is that `g &&& h` performs both `g` *and* `h` (hence `&`) on its input. For functions, we define `(g &&& h) x = (g x, h x)`.

Instances

The `Arrow` library itself only provides two `Arrow` instances, both of which we have already seen: `(->)`, the normal function constructor, and `Kleisli m`, which makes functions of type `a -> m b` into `Arrow`s for any `Monad m`. These instances are:

```
instance Arrow (->) where
  arr g = g
  first g (x,y) = (g x, y)

newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Arrow (Kleisli m) where
  arr f = Kleisli (return . f)
  first (Kleisli f) = Kleisli (\ ~(b,d) -> do c <- f b
                                              return (c,d) )
```

Laws

There are quite a few laws that instances of `Arrow` should satisfy :

```
                        arr id  =  id
                   arr (h . g)  =  arr g >>> arr h
                 first (arr g)  =  arr (g *** id)
               first (g >>> h)  =  first g >>> first h
   first g >>> arr (id *** h)   =  arr (id *** h) >>> first g
          first g >>> arr fst   =  arr fst >>> g
first (first g) >>> arr assoc   =  arr assoc >>> first g

assoc ((x,y),z) = (x,(y,z))
```

Note that this version of the laws is slightly different than the laws given in the first two above references, since several of the laws have now been subsumed by the `Category` laws (in particular, the requirements that `id` is the identity arrow and that `(>>>)` is associative). The laws shown here follow those in Paterson's Programming with Arrows, which uses the `Category` class.

The reader is advised not to lose too much sleep over the `Arrow` laws , since it is not essential to understand them in order to program with arrows. There are also laws that `ArrowChoice`, `ArrowApply`, and `ArrowLoop` instances should satisfy; the interested reader should consult Paterson: Programming with Arrows.

ArrowChoice

Computations built using the `Arrow` class, like those built using the `Applicative` class, are rather inflexible: the structure of the computation is fixed at the outset, and there is no ability to choose between alternate execution paths based on intermediate results. The `ArrowChoice` class provides exactly such an ability:

```
class Arrow arr => ArrowChoice arr where
  left  :: (b `arr` c) -> (Either b d `arr` Either c d)
  right :: (b `arr` c) -> (Either d b `arr` Either d c)
  (+++) :: (b `arr` c) -> (b' `arr` c') -> (Either b b' `arr` Either c c')
  (|||) :: (b `arr` d) -> (c `arr` d) -> (Either b c `arr` d)
```

A comparison of `ArrowChoice` to `Arrow` will reveal a striking parallel between `left`, `right`, `(+++)`, `(|||)` and `first`, `second`, `(***)`, `(&&&)`, respectively. Indeed, they are dual: `first`, `second`, `(***)`, and `(&&&)` all operate on product types (tuples), and `left`, `right`, `(+++)`, and `(|||)` are the corresponding operations on sum types. In general, these operations create arrows whose inputs are tagged with `Left` or `Right`, and can choose how to act based on these tags.

- If `g` is an arrow from `b` to `c`, then `left g` is an arrow from `Either b d` to `Either c d`. On inputs tagged with `Left`, the `left g` arrow has the behavior of `g`; on inputs tagged with `Right`, it behaves as the identity.
- The `right` function, of course, is the mirror image of `left`. The arrow `right g` has the behavior of `g` on inputs tagged with `Right`.
- The `(+++)` operator performs "multiplexing": `g +++ h` behaves as `g` on inputs tagged with `Left`, and as `h` on inputs tagged with `Right`. The tags are preserved. The `(+++)` operator is the *sum* (hence `+`) of two arrows, just as `(***)` is the product.
- The `(|||)` operator is "merge" or "fanin": the arrow `g ||| h` behaves as `g` on inputs tagged with `Left`, and `h` on inputs tagged with `Right`, but the tags are discarded (hence, `g` and `h` must have the same output type). The mnemonic is that `g ||| h` performs either `g` *or* `h` on its input.

The `ArrowChoice` class allows computations to choose among a finite number of execution paths, based on intermediate results. The possible execution paths must be known in advance, and explicitly assembled with `(+++)` or `(|||)`. However, sometimes more flexibility is needed: we would like to be able to *compute* an arrow from intermediate results, and use this computed arrow to continue the computation. This is the power given to us by `ArrowApply`.

ArrowApply

The `ArrowApply` type class is:

```
class Arrow arr => ArrowApply arr where
  app :: (b `arr` c, b) `arr` c
```

If we have computed an arrow as the output of some previous computation, then `app` allows us to apply that arrow to an input, producing its output as the output of `app`. As an exercise, the reader may wish to use `app` to implement an alternative "curried" version, `app2 :: b ‘arr‘ ((b ‘arr‘ c) ‘arr‘ c)`.

This notion of being able to *compute* a new computation may sound familiar: this is exactly what the monadic bind operator `(>>=)` does. It should not particularly come as a surprise that `ArrowApply` and `Monad` are exactly equivalent in expressive power. In particular, `Kleisli m` can be made an instance of `ArrowApply`, and any instance of `ArrowApply` can be made a `Monad` (via the `newtype` wrapper `ArrowMonad`). As an exercise, the reader may wish to try implementing these instances:

```haskell
instance Monad m => ArrowApply (Kleisli m) where
  app =    -- exercise

newtype ArrowApply a => ArrowMonad a b = ArrowMonad (a () b)

instance ArrowApply a => Monad (ArrowMonad a) where
  return               =    -- exercise
  (ArrowMonad a) >>= k =    -- exercise
```

ArrowLoop

The `ArrowLoop` type class is:

```haskell
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c

trace :: ((b,d) -> (c,d)) -> b -> c
trace f b = let (c,d) = f (b,d) in c
```

It describes arrows that can use recursion to compute results, and is used to desugar the `rec` construct in arrow notation (described below).

Taken by itself, the type of the `loop` method does not seem to tell us much. Its intention, however, is a generalization of the `trace` function which is also shown. The `d` component of the first arrow's output is fed back in as its own input. In other words, the arrow `loop g` is obtained by recursively "fixing" the second component of the input to `g`.

It can be a bit difficult to grok what the `trace` function is doing. How can `d` appear on the left and right sides of the `let`? Well, this is Haskell's laziness at work. There is not space here for a full explanation; the interested reader is encouraged to study the standard `fix` function, and to read Paterson's arrow tutorial.

Arrow notation

Programming directly with the arrow combinators can be painful, especially when writing complex computations which need to retain simultaneous reference to a number of intermediate results. With nothing but the arrow combinators, such intermediate results must be kept in nested tuples, and it is up to the programmer to remember which intermediate results are in which components, and to swap, reassociate, and generally mangle tuples as necessary. This problem is solved by the special arrow notation supported by GHC, similar to `do` notation for monads, that allows names to be assigned to intermediate results while building up arrow computations. An example arrow implemented using arrow notation, taken from Paterson, is:

```haskell
class ArrowLoop arr => ArrowCircuit arr where
  delay :: b -> (b `arr` b)

counter :: ArrowCircuit arr => Bool `arr` Int
counter = proc reset -> do
            rec output <- idA     -< if reset then 0 else next
                next   <- delay 0 -< output + 1
            idA -< output
```

This arrow is intended to represent a recursively defined counter circuit with a reset line.

There is not space here for a full explanation of arrow notation; the interested reader should consult Paterson's paper introducing the notation, or his later tutorial which presents a simplified version.

Further reading

An excellent starting place for the student of arrows is the arrows web page, which contains an introduction and many references. Some key papers on arrows include Hughes's original paper introducing arrows, Generalising monads to arrows, and Paterson's paper on arrow notation.

Both Hughes and Paterson later wrote accessible tutorials intended for a broader audience: Paterson: Programming with Arrows and Hughes: Programming with Arrows.

Although Hughes's goal in defining the `Arrow` class was to generalize `Monads`, and it has been said that `Arrow` lies "between `Applicative` and `Monad`" in power, they are not directly comparable. The precise relationship remained in some confusion until analyzed by Lindley, Wadler, and Yallop, who also invented a new calculus of arrows, based on the lambda calculus, which considerably simplifies the presentation of the arrow laws (see The arrow calculus). There is also a precise technical sense in which `Arrow` can be seen as the intersection of `Applicative` and `Category`.

Some examples of `Arrows` include Yampa, the Haskell XML Toolkit, and the functional GUI library Grapefruit.

Some extensions to arrows have been explored; for example, the `BiArrows` of Alimarine et al., for two-way instead of one-way computation.

The Haskell wiki has links to many additional research papers relating to `Arrows`.

Comonad

The final type class we will examine is `Comonad`. The `Comonad` class is the categorical dual of `Monad`; that is, `Comonad` is like `Monad` but with all the function arrows flipped. It is not actually in the standard Haskell libraries, but it has seen some interesting uses recently, so we include it here for completeness.

Definition

The `Comonad` type class, defined in the `Control.Comonad` module of the comonad library, is:

```haskell
class Functor w => Comonad w where
  extract :: w a -> a

  duplicate :: w a -> w (w a)
  duplicate = extend id

  extend :: (w a -> b) -> w a -> w b
  extend f = fmap f . duplicate
```

As you can see, `extract` is the dual of `return`, `duplicate` is the dual of `join`, and `extend` is the dual of `(=<<)`. The definition of `Comonad` is a bit redundant, giving the programmer the choice on whether extend or duplicate are implemented; the other operation then has a default implementation.

A prototypical example of a `Comonad` instance is:

```haskell
-- Infinite lazy streams
data Stream a = Cons a (Stream a)

-- 'duplicate' is like the list function 'tails'
-- 'extend' computes a new Stream from an old, where the element
--   at position n is computed as a function of everything from
--   position n onwards in the old Stream
instance Comonad Stream where
  extract (Cons x _) = x
  duplicate s@(Cons x xs) = Cons s (duplicate xs)
  extend g s@(Cons x xs)  = Cons (g s) (extend g xs)
                       -- = fmap g (duplicate s)
```

Further reading

Dan Piponi explains in a blog post what cellular automata have to do with comonads. In another blog post, Conal Elliott has examined a comonadic formulation of functional reactive programming. Sterling Clover's blog post Comonads in everyday life explains the relationship between comonads and zippers, and how comonads can be used to design a menu system for a web site.

Uustalu and Vene have a number of papers exploring ideas related to comonads and functional programming:

- Comonadic Notions of Computation
- The dual of substitution is redecoration (Also available as ps.gz.)
- Recursive coalgebras from comonads
- Recursion schemes from comonads
- The Essence of Dataflow Programming.

Gabriel Gonzalez's Comonads are objects points out similarities between comonads and object-oriented programming.

The comonad-transformers package contains comonad transfomers.

Acknowledgements

A special thanks to all of those who taught me about standard Haskell type classes and helped me develop good intuition for them, particularly Jules Bean (quicksilver), Derek Elkins (ddarius), Conal Elliott (conal), Cale Gibbard (Cale), David House, Dan Piponi (sigfpe), and Kevin Reid (kpreid).

I also thank the many people who provided a mountain of helpful feedback and suggestions on a first draft of the Typeclassopedia: David Amos, Kevin Ballard, Reid Barton, Doug Beardsley, Joachim Breitner, Andrew Cave, David Christiansen, Gregory Collins, Mark Jason Dominus, Conal Elliott, Yitz Gale, George Giorgidze, Steven Grady, Travis Hartwell, Steve Hicks, Philip Hölzenspies, Edward Kmett, Eric Kow, Serge Le Huitouze, Felipe Lessa, Stefan Ljungstrand, Eric Macaulay, Rob MacAulay, Simon Meier, Eric Mertens, Tim Newsham, Russell O'Connor, Conrad Parker, Walt Rorie-Baety, Colin Ross, Tom Schrijvers, Aditya Siram, C. Smith, Martijn van Steenbergen, Joe Thornber, Jared Updike, Rob Vollmert, Andrew Wagner, Louis Wasserman, and Ashley Yakeley, as well as a few only known to me by their IRC nicks: b__jonas, maltem, tehgeekmeister, and ziman. I have undoubtedly omitted a few inadvertently, which in no way diminishes my gratitude.

Finally, I would like to thank Wouter Swierstra for his fantastic work editing the Monad.Reader, and my wife Joyia for her patience during the process of writing the Typeclassopedia.

About the author

Brent Yorgey (blog, homepage) is (as of November 2011) a fourth-year Ph.D. student in the programming languages group at the University of Pennsylvania. He enjoys teaching, creating EDSLs, playing Bach fugues, musing upon category theory, and cooking tasty lambda-treats for the denizens of #haskell.

Colophon

The Typeclassopedia was written by Brent Yorgey and initially published in March 2009. Painstakingly converted to wiki syntax by User:Geheimdienst in November 2011, after asking Brent's permission.

If something like this tex to wiki syntax conversion ever needs to be done again, here are some vim commands that helped:

- `%s/\\section{\([^}]*\)}/=\1=/gc`
- `%s/\\subsection{\([^}]*\)}/==\1==/gc`

- %s/^ *\\item /\r* /gc
- %s/---/—/gc
- %s/\$\([^$]*\)\$/<math>\1\\ <\/math>/gc ”Appending “\ ” forces images to be rendered. Otherwise, Mediawiki would go back and forth between one font for short <math> tags, and another more Tex-like font for longer tags (containing more than a few characters)""
- %s/|\([^|]*\)|/<code>\1<\/code>/gc
- %s/\\dots/.../gc
- %s/^\\label{.*$//gc
- %s/\\emph{\([^}]*\)}/”\1”/gc
- %s/\\term{\([^}]*\)}/”\1”/gc

The biggest issue was taking the academic-paper-style citations and turning them into hyperlinks with an appropriate title and an appropriate target. In most cases there was an obvious thing to do (e.g. online PDFs of the cited papers or Citeseer entries). Sometimes, however, it’s less clear and you might want to check the original Typeclassopedia PDF with the original bibliography file.

To get all the citations into the main text, I first tried processing the source with Tex or Lyx. This didn’t work due to missing unfindable packages, syntax errors, and my general ineptitude with Tex.

I then went for the next best solution, which seemed to be extracting all instances of “\cite{something}” from the source and *in that order* pulling the referenced entries from the .bib file. This way you can go through the source file and sorted-references file in parallel, copying over what you need, without searching back and forth in the .bib file. I used:

- egrep -o "\cite\{[^\}]*\}" ~/typeclassopedia.lhs | cut -c 6- | tr "," "\n" | tr -d "}" > /tmp/citations
- for i in $(cat /tmp/citations); do grep -A99 "$i" ~/typeclassopedia.bib|egrep -B99 ’^\}$’ -m1 ; done > ~/typeclasso-refs-sorted

Converted to PDF by Norman Ramsey using Pandoc 1.11 and a little hand editing.