The Typeclassopedia

by Brent Yorgey (byorgey@cis.upenn.edu)

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 article 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.

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 article is to set you on the road to gaining such intuition. However—

*There is no royal road to Haskell.*
—Euclid

This article can only be a starting point, since good intuition comes from hard work, not from learning the right metaphor [1]. 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 Prelude, the type system, data types, and type classes.

Figure 1 on page 19 shows the type classes we will be discussing and their interrelationships. 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. The solid double arrow indicates that Monad and ArrowApply are equivalent. Pointed and Comonad are greyed out since they are not actually (yet) in the standard Haskell libraries (they are in the category-extras library [2]).

One more note before we begin. I’ve seen “type class” written as one word, “typeclass,” but let’s settle this once and for all: the correct spelling uses two words (the title of this article notwithstanding), as evidenced by, for example, the Haskell 98 Revised Report [3], early papers on type classes [4, 5], and Hudak et al.’s history of Haskell [6].

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

**Functor**

The **Functor** class [7] 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

---

1Well, he probably would have said it if he knew Haskell.
function to every element of a list using \texttt{map}. 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 \texttt{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 \texttt{Functor}-as-context point of view.

In the end, however, a \texttt{Functor} is simply what it is defined to be; doubtless there are many examples of \texttt{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**

The type class declaration for \texttt{Functor} is shown in Listing 5. \texttt{Functor} is exported by the \texttt{Prelude}, so no special imports are needed to use it.

First, the \texttt{f a} and \texttt{f b} in the type signature for \texttt{fmap} tell us that \texttt{f} isn’t just a type; it is a \texttt{type constructor} which takes another type as a parameter. (A more precise way to say this is that the \texttt{kind} of \texttt{f} must be \texttt{* -> *}. For ex-
class Functor f where
  fmap :: (a -> b) -> f a -> f b

Listing 5: The Functor type class

ample, 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;\(^2\) we can use the standard list function map to apply a function to each element of a list.\(^3\) 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 shown in Listing 6.

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 tutorial, I will use f only to represent Functors, and g or h to 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

\(^2\)Recall that [] has two meanings in Haskell: it can either stand for the empty list, or, as here, it can represent the list type constructor (pronounced “list-of”). In other words, the type [a] (list-of-a) can also be written ([] a).

\(^3\)You might ask why we need a separate map function. Why not just do away with the current list-only map function, and rename fmap to map instead? Well, that’s a good question. The usual argument is that someone just learning Haskell, when using map incorrectly, would much rather see an error about lists than about Functors.
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)

Listing 6: Two simple Functor instances

There are other Functor instances in the standard libraries; here are a few:

- **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.

- **((->) e)**, the type of functions which take a value of type e as a parameter, is a Functor. It would be clearer to write it as (e ->), by analogy with an operator section like (1+), but that syntax is not allowed. However, you can certainly think of it as (e ->). 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 [8] (such as Tree, Map, Sequence, and Stream) 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.

---

4Note that some of these instances are not exported by the Prelude; to access them, you can import Control.Monad.Instances.

21
A good exercise is to implement `Functor` instances for `Either e`, `((,) e)`, and `((->) e)`.

**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, shown in Listing 7; 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.⁵

---

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

**Listing 7: The Functor laws**

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 code shown in Listing 8 is a “valid” instance of `Functor` (it typechecks), but it violates the functor laws. Do you see why?

---

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

**Listing 8: A lawless Functor instance**

Any Haskeller worth their salt would reject the code in Listing 8 as a gruesome abomination.

**Intuition**

There are two fundamental ways to think about `fmap`. The first has already been touched on: it takes two parameters, a function and a container, and applies the

---

⁵Technically, these laws make `f` and `fmap` together an endofunctor on `Hask`, the category of Haskell types (ignoring ⊥, which is a party pooper). [9]
function “inside” the container, producing a new container. Alternately, we can think of \texttt{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, \texttt{fmap} is actually \textbf{curried}: it does not really take two parameters, but takes a single parameter and returns a function. For emphasis, we can write \texttt{fmap}’s type with extra parentheses: \texttt{fmap :: (a -> b) -> (f a -> f b)}. Written in this form, it is apparent that \texttt{fmap} transforms a “normal” function (\texttt{g :: a -> b}) into one which operates over containers/contexts (\texttt{fmap g :: f a -> f b}). This transformation is often referred to as a \textbf{lift}; \texttt{fmap} “lifts” a function from the “normal world” into the “\textit{f} world.”

\textbf{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 [9].

\textbf{Pointed*}

The \textbf{Pointed} type class represents \textbf{pointed functors}. It is not actually a type class in the standard libraries (hence the asterisk).\footnote{It is, however, a type class in the category-extras library [2].} But it \textbf{could} be, and it’s useful in understanding a few other type classes, notably \textbf{Applicative} and \textbf{Monad}, so let’s pretend for a minute.

Given a \textbf{Functor}, the \textbf{Pointed} class represents the additional ability to put a value into a “default context.” Often, this corresponds to creating a container with exactly one element, but it is more general than that. The type class declaration for \textbf{Pointed} is shown in Listing 9.

\begin{verbatim}
class Functor f => Pointed f where
  pure :: a -> f a -- aka singleton, return, unit, point
\end{verbatim}

\textbf{Listing 9:} The \textbf{Pointed} type class

Most of the standard \textbf{Functor} instances could also be instances of \textbf{Pointed}—for example, the \textbf{Maybe} instance of \textbf{Pointed} is \texttt{pure = Just}; there are many possible implementations for lists, the most natural of which is \texttt{pure x = [x]}; for \texttt{(->) e} it is...well, I’ll let you work it out. (Just follow the types!)
One example of a **Functor** which is not **Pointed** is \((\cdot, e)\). If you try implementing `pure :: a -> (e,a)` you will quickly see why: since the type `e` is completely arbitrary, there is no way to generate a value of type `e` out of thin air! However, as we will see, \((\cdot, e)\) can be made **Pointed** if we place an additional restriction on `e` which allows us to generate a default value of type `e` (the most common solution is to make `e` an instance of **Monoid**).

The **Pointed** class has only one law, shown in Listing 10.

\[
\text{fmap } g \cdot \text{pure} = \text{pure} \cdot g
\]

**Listing 10**: The **Pointed** law

However, you need not worry about it: this law is actually a so-called “free theorem” guaranteed by parametricity [10]; it’s impossible to write an instance of **Pointed** which does not satisfy it.\(^8\)

**Applicative**

A somewhat newer addition to the pantheon of standard Haskell type classes, applicative functors [11] represent an abstraction lying exactly in between **Functor** and **Monad**, first described by McBride and Paterson [12]. The title of McBride and Paterson’s 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**

The **Applicative** class adds a single capability to **Pointed** functors. 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 another context. **Applicative** gives us just such a tool. Listing 11 shows the type class declaration for **Applicative**, which is defined in `Control.Applicative`. Note that every **Applicative** must also be a **Functor**. In fact, as we will see, **fmap** can be implemented using the **Applicative** methods, so

\(^7\)For those interested in category theory, this law states precisely that `pure` is a natural transformation from the identity functor to \(f\).

\(^8\)... modulo \(\bot\), \text{seq}, and assuming a lawful **Functor** instance.
every Applicative is a functor whether we like it or not; the Functor constraint forces us to be honest.

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

Listing 11: The Applicative type class

As always, it’s crucial to understand the type signature of (<*>). The best way of thinking about it comes from noting that the type of (<*>) is similar to the type of ($),9 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).

Of course, pure looks rather familiar. If we actually had a Pointed type class, Applicative could instead be defined as shown in Listing 12.

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

Listing 12: Alternate definition of Applicative using Pointed

**Laws**

There are several laws that Applicative instances should satisfy [11, 12], but only one is crucial to developing intuition, because it specifies how Applicative should relate to Functor (the other four mostly specify the exact sense in which pure deserves its name). This law is shown in Listing 13.

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

Listing 13: Law relating Applicative to Functor

The law 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

---

9Recall that ($) is just function application: f $ x = f x.
words, we can decompose \texttt{fmap} into two more atomic operations: injection into a context, and application within a context. The \texttt{Control.Applicative} module also defines \((<<\$>>)\) as a synonym for \texttt{fmap}, so the above law can also be expressed as \(g \ <\$\> \ x = \text{pure} \ g \ <\!\,*\!\> \ x\).

\textbf{Instances}

Most of the standard types which are instances of \texttt{Functor} are also instances of \texttt{Applicative}.

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

The list type constructor \([\ ]\) can actually be made an instance of \texttt{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 [13].

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 \texttt{Applicative} need to be defined for a \texttt{newtype} wrapper; as it happens, the nondeterministic computation instance is the default, and the collection instance is defined in terms of a \texttt{newtype} called \texttt{ZipList}. This instance is shown in Listing 14.

\begin{verbatim}
newtype ZipList a = ZipList { getZipList :: [a] }

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

Listing 14: ZipList instance of Applicative
\end{verbatim}

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 \texttt{ZipList}. As an exercise, determine the correct definition of \texttt{pure}—there is only one implementation that satisfies the law in Listing 13.

The other \texttt{Applicative} instance for lists, based on the nondeterministic computation point of view, is shown in Listing 15. 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.
instance Applicative [] where
    pure x = [x]
    gs <*> xs = [ g x | g <- gs, x <- xs ]

Listing 15: [] instance of Applicative

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: when g <$> m1 <*> m2 <*> m3 is executed, the effects from the mi’s happen in order from left to right.
- ((,) a) is an Applicative, as long as a is an instance of Monoid (page 39). 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 (page 44).
- The WrappedMonad and WrappedArrow newtypes make any instances of Monad (page 29) or Arrow (page 51) 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 \(t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t\), 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 \texttt{fmap}, which allows us to apply a function to a single argument in a context.

Why do we need \texttt{Applicative} to implement this generalization of \texttt{fmap}? Suppose we use \texttt{fmap} to apply \(g\) to the first parameter \(x_1\). Then we get something of type \(f \ (t_2 \to \cdots \ t)\), but now we are stuck: we can’t apply this function-in-a-context to the next argument with \texttt{fmap}. However, this is precisely what \((<*>\)) allows us to do.

This suggests the proper translation of the idealized notation \(\llbracket g \ x_1 \ x_2 \ \cdots \ x_n \rrbracket\) into Haskell, namely

\[
g \ <$> \ x_1 \ <*> \ x_2 \ <*> \ \cdots \ <*> \ x_n,
\]

recalling that \texttt{Control.Applicative} defines \((<$>)\) as a convenient infix shorthand for \texttt{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.

\section*{Further reading}

There are many other useful combinators in the standard libraries implemented in terms of \texttt{pure} and \((<*>\)): for example, \((*>\)), \((<<*)\), \((<<*>\)), \((<$>)\), and so on [11]. Judicious use of such secondary combinators can often make code using \texttt{Applicatives} much easier to read.

McBride and Paterson’s original paper [12] is a treasure-trove of information and examples, as well as some perspectives on the connection between \texttt{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 \texttt{Applicative}. For example, the Pan library for functional images [14] and the reactive library for functional reactive programming (FRP) [15] make key use of it; his blog also contains many examples of \texttt{Applicative} in action [16]. Building on the work of McBride and Paterson, Elliott also built the TypeCompose library [17], which embodies the observation (among others) that \texttt{Applicative} types are closed under composition; therefore, \texttt{Applicative} instances can often be automatically derived for complex types built out of simpler ones.

Although the Parsec parsing library [18, 19] was originally designed for use as a monad, in its most common use cases an \texttt{Applicative} instance can be used to great effect; Bryan O’Sullivan’s blog post is a good starting point [20]. If the extra power provided by \texttt{Monad} isn’t needed, it’s usually a good idea to use \texttt{Applicative} instead.
A couple other nice examples of Applicative in action include the ConfigFile and HSQL libraries [21] and the formlets library [22].

Monad

It’s a safe bet that if you’re reading this article, 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 various 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 [1].

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 [23] is shown in Listing 16. 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.

```haskell
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

Listing 16: The Monad type class
```

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 \texttt{\textbf{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 a pointed functor (indeed, an applicative functor), but for historical reasons, the \texttt{Monad} type class declaration unfortunately does not require this.

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

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

The only really interesting thing to look at—and what makes \texttt{Monad} strictly more powerful than \texttt{Pointed} or \texttt{Applicative}—is \texttt{(>>=)}, which is often called \texttt{bind}. An alternative definition of \texttt{Monad} could look like Listing 17.

\begin{lstlisting}[language=Haskell]
class Applicative m => Monad' m where
  (>>=) :: m a -> (a -> m b) -> m b
\end{lstlisting}

\textbf{Listing 17:} An alternative definition of \texttt{Monad}

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

\textbf{Instances}

Even if you don’t understand the intuition behind the \texttt{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 \texttt{Monad} class in general. The first few examples are from the standard \texttt{Prelude}; the remaining examples are from the monad transformer library (\texttt{mtl}) [24].

\begin{itemize}
  \item The simplest possible instance of \texttt{Monad} is \texttt{Identity} [25], which is described in Dan Piponi’s highly recommended blog post on “The Trivial Monad” [26]. Despite being “trivial,” it is a great introduction to the \texttt{Monad} type class, and contains some good exercises to get your brain working.
\end{itemize}
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

\[ (\gg=) \colon \text{Maybe } a \to (a \to \text{Maybe } b) \to \text{Maybe } b. \]

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 \( a \to \text{Maybe } b \) function, so there’s only one thing we can do: yield Nothing. This instance is shown in Listing 18. 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.

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

Listing 18: The Maybe instance of Monad

- The Monad instance for the list constructor [] is similar to its Applicative instance; I leave its implementation as an exercise. Follow the types!
- 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.
- As mentioned earlier, \((\to)\ e\) is known as the reader monad, since it describes computations in which a value of type \( e \) is available as a read-only environment. It is worth trying to write a Monad instance for \((\to)\ e\)
The Monad.Reader Issue 13

yourself.

The Control.Monad.Reader module [27] 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).

▶ The Control.Monad.Writer module [28] 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 (page 39); the special function tell performs logging.

▶ The Control.Monad.State module [29] 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).

▶ The Control.Monad.Cont module [30] 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” [31] 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. 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.

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 \( a \to m \ b \), 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 \( m \ a \to m \ (m \ b) \). 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 :: \( m \ (m \ a) \to m \ a \), and it should come as no surprise that an alternative definition of Monad can be given in terms of join, as shown in Listing 19.

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

**Listing 19:** An alternative definition of Monad in terms of join

In fact, monads in category theory are defined in terms of return, fmap, and join (often called \( \eta \), \( T \), and \( \mu \) in the mathematical literature). Haskell uses the equivalent formulation in terms of (>>=) 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.) An excellent exercise is to implement (>>=) in terms of fmap and join, and to implement join in terms of (>>=).

**Utility functions**

The Control.Monad module [32] 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 [33].

- 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.
The Monad.Reader Issue 13

- **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 also an additional generalization of sequence to structures other than lists, which will be discussed in the section on Traversable (page 47).
- **replicateM :: Monad m => Int -> 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 [34].
- **mapM :: Monad m => (a -> m b) -> [a] -> m [b]** maps its first argument over the second, and sequences 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.
- 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.

**Laws**

There are several laws that instances of Monad should satisfy [35]. The standard presentation is shown in Listing 20.

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

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

Listing 20: The Monad laws

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 (>=>).

Ah, much better! The laws simply state that return is the identity of (>=>), and that (>=>) is associative. Working out the equivalence between these two formulations, given the definition g >>= h = \x -> g x >>= h, is left as an exercise.

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 [9].

10 I like to pronounce this operator “fish,” but that’s probably not the canonical pronunciation...

11 As fans of category theory will note, these laws say precisely that functions of type a -> m b are the arrows of a category with (>=>) as composition! Indeed, this is known as the Kleisli category of the monad m. It will come up again when we discuss Arrows.
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 \( a >>= \lambda x \to b >> c >>= \lambda y \to d \) can be more readably written by putting successive computations on separate lines:

\[
\begin{align*}
a & \mapsto \lambda x \to \\
b & >> \\
c & \mapsto \lambda y \to \\
d & 
\end{align*}
\]

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:

\[
\begin{align*}
do \{ x \leftarrow a ; b ; y \leftarrow c ; d \}
\end{align*}
\]

(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) as shown in Listing 22.

```
\[
\begin{align*}
do \ e & \mapsto e \\
\quad & \quad do \ \{ e ; stmts \} \mapsto e >> do \ \{ stmts \} \\
\quad & \quad do \ \{ v \leftarrow e ; stmts \} \mapsto e >>= \lambda v \to do \ \{ stmts \} \\
\quad & \quad do \ \{ let \ decls ; stmts \} \mapsto let \ decls \ in \ do \ \{ stmts \}
\end{align*}
\]
```

Listing 22: Desugaring of do blocks (almost)

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

\[
\begin{align*}
do \ (x:xs) \leftarrow \text{foo} \\
\quad & \quad \text{bar x}
\end{align*}
\]
but what happens if \texttt{foo} produces an empty list? Well, remember that ugly \texttt{fail} function in the \texttt{Monad} type class declaration? That’s what happens. See section 3.14 of the Haskell Report for the full details [3]. See also the discussion of \texttt{MonadPlus} and \texttt{MonadZero} (page 42).

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

\section*{Monad transformers}

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

The monad transformer library [24] provides a number of \texttt{monad transformers}, such as \texttt{StateT}, \texttt{ReaderT}, \texttt{ErrorT} [36], and (soon) \texttt{MaybeT}, which can be applied to other monads to produce a new monad with the effects of both. For example, \texttt{StateT s Maybe} is an instance of \texttt{Monad}; computations of type \texttt{StateT s Maybe a} may fail, and have access to a mutable state of type \texttt{s}. These 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 \texttt{StateT s Maybe a} computation fails, the state ceases being updated; on the other hand, the state of a \texttt{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”; for example, \texttt{MaybeT (State s) a} is isomorphic to \texttt{s -> Maybe (a, s)}. Lambdabot has an indispensable \texttt{@unmtl} command which you can use to “unpack” a monad transformer stack in this way.)

All monad transformers should implement the \texttt{MonadTrans} type class (Listing 23), defined in \texttt{Control.Monad.Trans}. It allows arbitrary computations in the base monad \texttt{m} to be “lifted” into computations in the transformed monad \texttt{t m}. (Note that type application associates to the left, just like function application, so \texttt{t m a = (t m) a}. As an exercise, you may wish to work out \texttt{t}'s kind, which is rather more interesting than most of the kinds we’ve seen up to this point.) However, you should only have to think about \texttt{MonadTrans} when defining your own monad transformers, not when using predefined ones.
class MonadTrans t where
    lift :: Monad m => m a -> t m a

Listing 23: The MonadTrans type class

There are also type classes such as MonadState, which provides state-specific methods like 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.12

There are two excellent references on monad transformers. Martin Grabmüller’s Monad Transformers Step by Step [37] 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 [38] 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 [39].

MonadFix

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 recursion. This is supported in GHC and Hugs by a special “recursive do” notation, mdo. For more information, see Levent Erkők’s thesis, Value Recursion in Monadic Computations [40].

Further reading

Philip Wadler was the first to propose using monads to structure functional programs [41]. His paper is still a readable introduction to the subject.

Much of the monad transformer library (mtl) [24], 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 [42]. It’s still very much worth a read—and highly readable—after almost fifteen years.

12The only problem with this scheme is the quadratic number of instances required as the number of standard monad transformers grows—but as the current set of standard monad transformers seems adequate for most common use cases, this may not be that big of a deal.
There are, of course, numerous monad tutorials of varying quality [43, 44, 45, 46, 47, 48, 49, 50, 51, 52]. A few of the best include Cale Gibbard’s Monads as containers [44] and Monads as computation [51]; Jeff Newbern’s All About Monads [43], a comprehensive guide with lots of examples; and Dan Piponi’s You could have invented monads!, which features great exercises [47]. If you just want to know how to use IO, you could consult the Introduction to IO [53]. Even this is just a sampling; a more complete list can be found on the Haskell wiki [54]. (All these monad tutorials have prompted some parodies [55] as well as other kinds of backlash [56, 1].) Other good monad references which are not necessarily tutorials include Henk-Jan van Tuyl’s tour of the functions in Control.Monad [33], Dan Piponi’s “field guide” [57], and Tim Newsham’s What’s a Monad? [58]. There are also many blog articles which have been written on various aspects of monads; a collection of links can be found on the Haskell wiki [59].

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 [60], and later made into a library by Ganesh Sittampalam and Peter Gavin [61].

There are many good reasons for eschewing do notation; some have gone so far as to consider it harmful [62].

Monads can be generalized in various ways; for an exposition of one possibility, parameterized monads, see Robert Atkey’s paper on the subject [63], or Dan Piponi’s exposition [64].

For the categorically inclined, monads can be viewed as monoids [65] and also as closure operators [66]. Derek Elkins’s article in this issue of the Monad.Reader [67] contains an exposition of the category-theoretic underpinnings of some of the standard Monad instances, such as State and Cont. There is also an alternative way to compose monads, using coproducts, as described by Lüth and Ghani [68], although this method has not (yet?) seen widespread use.

Links to many more research papers related to monads can be found on the Haskell wiki [69].

**Monoid**

A monoid is a set S together with a binary operation ⊕ which combines elements from S. The ⊕ operator is required to be associative (that is, (a⊕ b)⊕ c = a⊕ (b⊕ c), for any a, b, c which are elements of S), and there must be some element of S which is the identity with respect to ⊕. (If you are familiar with group theory, a monoid is
like a group without the requirement that inverses exist.) For example, the natural numbers under addition form a monoid: the sum of any two natural numbers is a natural number; \((a + b) + c = a + (b + c)\) for any natural numbers \(a\), \(b\), and \(c\); and zero is the additive identity. The integers under multiplication also form a monoid, as do natural numbers under max, Boolean values under conjunction and disjunction, lists under concatenation, functions from a set to itself under composition. ... Monoids show up all over the place, once you know to look for them.

**Definition**

The definition of the `Monoid` type class (defined in `Data.Monoid`) [70] is shown in Listing 24.

```haskell
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
    mconcat = foldr mappend mempty
```

**Listing 24:** The `Monoid` type class

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 [71].

**Laws**

Of course, every `Monoid` instance should actually be a monoid in the mathematical sense, which implies the laws shown in Listing 25.

**Instances**

There are quite a few interesting `Monoid` instances defined in `Data.Monoid`. 
mempty 'mappend' x = x
x 'mappend' mempty = x
(x 'mappend' y) 'mappend' z = x 'mappend' (y 'mappend' z)

Listing 25: The Monoid laws

- [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.
- 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 discussion of Foldable (page 44).
- Any and All are newtype wrappers providing Monoid instances for Bool (under disjunction and conjunction, respectively).
- 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.
- Endo a is a newtype wrapper for functions a -> a, which form a monoid under composition.
- 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 result using the underlying Monoid instance for a. This can be quite useful and elegant [72].
- 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. 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 [73].

There are also Monoid instances for several standard data structures in the
containers library [8], including Map, Set, and Sequence.
Monoid is also used to enable several other type class instances. As noted previ-
ously, we can use Monoid to make ((,) e) an instance of Applicative, as shown
in Listing 26.

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

Listing 26: An Applicative instance for ((,) e) using Monoid

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 (page 44).

Other monoidal classes: Alternative, MonadPlus, ArrowPlus

The Alternative type class [74], shown in Listing 27, is for Applicative functors
which also have a monoid structure.

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

Listing 27: The Alternative type class

Of course, instances of Alternative should satisfy the monoid laws.
Likewise, MonadPlus [75], shown in Listing 28, is for Monads with a monoid
structure.
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

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

42
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

Listing 28: The MonadPlus type class

which explains the sense in which \texttt{mzero} denotes failure. Since \texttt{mzero} should be the identity for \texttt{mplus}, the computation \texttt{m1 \texttt{mplus} m2} succeeds (evaluates to something other than \texttt{mzero}) if either \texttt{m1} or \texttt{m2} does; so \texttt{mplus} represents choice. The \texttt{guard} function can also be used with instances of \texttt{MonadPlus}; it requires a condition to be satisfied and fails (using \texttt{mzero}) if it is not. A simple example of a \texttt{MonadPlus} instance is \texttt{[]}, which is exactly the same as the \texttt{Monoid} instance for \texttt{[]}: the empty list represents failure, and list concatenation represents choice. In general, however, a \texttt{MonadPlus} instance for a type need not be the same as its \texttt{Monoid} instance; \texttt{Maybe} is an example of such a type. A great introduction to the \texttt{MonadPlus} type class, with interesting examples of its use, is Doug Auclair’s Monad.Reader article [76].

There used to be a type class called \texttt{MonadZero} containing only \texttt{mzero}, representing monads with failure. The do-notation requires some notion of failure to deal with failing pattern matches. Unfortunately, \texttt{MonadZero} was scrapped in favor of adding the \texttt{fail} method to the \texttt{Monad} class. If we are lucky, someday \texttt{MonadZero} will be restored, and \texttt{fail} will be banished to the bit bucket where it belongs [77]. The idea is that any do-block which uses pattern matching (and hence may fail) would require a \texttt{MonadZero} constraint; otherwise, only a \texttt{Monad} constraint would be required.

Finally, \texttt{ArrowZero} and \texttt{ArrowPlus} [78], shown in Listing 29, represent Arrows (page 51) with a monoid structure.

class Arrow (\texttt{~>}) => ArrowZero (\texttt{~>}) where
  zeroArrow :: b \texttt{~>} c

class ArrowZero (\texttt{~>}) => ArrowPlus (\texttt{~>}) where
  (\texttt{<+>}) :: (b \texttt{~>} c) -> (b \texttt{~>} c) -> (b \texttt{~>} c)

Listing 29: The ArrowZero and ArrowPlus type classes
Further reading

Monoids have gotten a fair bit of attention recently, ultimately due to a blog post by Brian Hurt [79], in which he complained about the fact that the names of many Haskell type classes (\texttt{Monoid} in particular) are taken from abstract mathematics. This resulted in a long haskell-cafe thread [71] arguing the point and discussing monoids in general.

However, this was quickly followed by several blog posts about \texttt{Monoid}. First, Dan Piponi wrote a great introductory post, “Haskell Monoids and their Uses” [80]. This was quickly followed by Heinrich Apfelmus’s “Monoids and Finger Trees” [81], an accessible exposition of Hinze and Paterson’s classic paper on 2-3 finger trees [82], which makes very clever use of \texttt{Monoid} to implement an elegant and generic data structure. Dan Piponi then wrote two fascinating articles about using \texttt{Monoids} (and finger trees) to perform fast incremental regular expression matching [83, 84].

In a similar vein, David Place’s article on improving \texttt{Data.Map} in order to compute incremental folds [85] is also a good example of using \texttt{Monoid} to generalize a data structure.

Some other interesting examples of \texttt{Monoid} use include building elegant list sorting combinators [73], collecting unstructured information [86], and a brilliant series of posts by Chung-Chieh Shan and Dylan Thurston using \texttt{Monoids} to elegantly solve a difficult combinatorial puzzle [87, 88, 89, 90].

As unlikely as it sounds, monads can actually be viewed as a sort of monoid, with \texttt{join} playing the role of the binary operation and \texttt{return} the role of the identity; see Dan Piponi’s blog post [65].

Foldable

The \texttt{Foldable} class, defined in the \texttt{Data.Foldable} module [91], 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 \texttt{Foldable} type class is shown in Listing 30.

This may look complicated, but in fact, to make a \texttt{Foldable} instance you only need to implement one method: your choice of \texttt{foldMap} or \texttt{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.

\footnote{May its name live forever.}
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

Listing 30: The Foldable type class

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. Listing 31 shows two examples: a simple implementation of foldMap for lists, and a binary tree example provided by the Foldable documentation.

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 ++ f k ++ foldMap f r
  where (++) = mappend

Listing 31: Two foldMap examples

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 [8] (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 the examples shown in Listing 32.

---

```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')
```

Listing 32: Foldable examples

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 reader may enjoy coming up with elegant implementations of these functions using `fold` or `foldMap` and appropriate `Monoid` instances.

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`. 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 [91] 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.

**Further reading**

The **Foldable** class had its genesis in McBride and Paterson’s paper introducing **Applicative** [12], 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!** [92].

**Traversable**

**Definition**

The **Traversable** type class, defined in the **Data.Traversable** module [93], is shown in Listing 33.

```haskell
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)
```

**Listing 33:** The **Traversable** type class

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? (Answer: yes, in two ways. Figuring out what they are, and why, is left as an exercise. A much more challenging question is whether a list of trees can be turned into a tree of lists.)

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 \( \text{join} :: M (M a) \to M a \), that is, \( \text{join} :: m (n (m (n a))) \to m (n a) \), 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 \( \text{joins} \) for \( m \) and \( n \) to produce something of type \( m (n a) \). See Mark Jones’s paper for more details [42].

Instances and examples

What’s an example of a \text{Traversable} instance? Listing 34 shows an example instance for the same \text{Tree} type used as an example in the previous \text{Foldable} section. It is instructive to compare this instance with a \text{Functor} instance for \text{Tree}, which is also shown.

```haskell
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)
```

Listing 34: An example Tree instance of Traversable

It should be clear that the \text{Traversable} and \text{Functor} instances for \text{Tree} are almost identical; the only difference is that the \text{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. A good exercise is to implement `fmap` and `foldMap` using only the `Traversable` methods; the implementations are surprisingly elegant. The `Traversable` module provides these implementations as `fmapDefault` and `foldMapDefault`.

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`.

Further reading

The `Traversable` class also had its genesis in McBride and Paterson’s `Applicative` paper [12], and is described in more detail in Gibbons and Oliveira, *The Essence of the Iterator Pattern* [94], which also contains a wealth of references to related work.

Category

Category is another fairly new addition to the Haskell standard libraries; you may or may not have it installed depending on the version of your `base` package. It generalizes the notion of function composition to general “morpisms.”

The definition of the `Category` type class (from `Control.Category` [95]) is shown in Listing 35. For ease of reading, note that I have used an infix type constructor (~>), much like the infix function type constructor (->). This syntax is not part of Haskell 98. The second definition shown is the one used in the standard libraries. For the remainder of the article, I will use the infix type constructor (~>) for `Category` as well as `Arrow`.

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 in Listing 36, 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.
class Category (~>) where
  id :: a ~> a
  (.) :: (b ~> c) -> (a ~> b) -> (a ~> 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

Listing 35: The Category type class

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)

Listing 36: The Kleisli Category instance

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 [2]. For more about category theory in general, see the excellent Haskell wikibook page [9], Steve Awodey’s new book [96], Benjamin Pierce’s Basic category theory for computer scientists [97], or Barr and Wells’s category theory lecture notes [98]. Benjamin Russell’s blog post [99] 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 (~>) is an instance of *Arrow*, a value of type b ~> 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* [100], is shown in Listing 37.

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

Listing 37: The *Arrow* type class

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 ~> c and h :: c ~> d, we can form their composition g >>> h :: b ~> d.\(^{14}\)

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.

\(^{14}\)In versions of the *base* package prior to version 4, there is no *Category* class, and the *Arrow* class includes the arrow composition operator (>>>). It also includes *pure* as a synonym for *arr*, but this was removed since it conflicts with the *pure* from *Applicative.*
Intuition

Let’s look at each of the arrow methods in turn. Ross Paterson’s web page on arrows [101] has nice diagrams which can help build intuition.

▶ The \texttt{arr} function takes any function \( b \rightarrow c \) and turns it into a generalized arrow \( b \Rightarrow c \). The \texttt{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 \( \texttt{arr} \ g \) is “pure” in the sense that it only computes \( g \) and has no “effects” (whatever that might mean for any particular arrow type).

▶ The \texttt{first} method turns any arrow from \( b \) to \( c \) into an arrow from \( (b,d) \) to \( (c,d) \). The idea is that \texttt{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 \texttt{Arrow}, of course, \texttt{first} \( g \ (x,y) = (g \ x, \ y) \).

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

▶ The \texttt{(***)} 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 \texttt{product} (hence \* ) of \( g \) and \( h \). For the function instance of \texttt{Arrow}, we define \( (g \ (***) \ h) \ (x,y) = (g \ x, \ h \ y) \). The default implementation of \texttt{(***)} is in terms of \texttt{first}, \texttt{second}, and sequential arrow composition (\texttt{>>>}) . The reader may also wish to think about how to implement \texttt{first} and \texttt{second} in terms of \texttt{(***)}.

▶ The \texttt{(&&&)} 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 \texttt{g and h} (hence \& & ) on its input. For functions, we define \( (g \ &\&\& \ h) \ x = (g \ x, \ h \ x) \).

Instances

The \texttt{Arrow} library itself only provides two \texttt{Arrow} instances, both of which we have already seen: \texttt{(-->)}, the normal function constructor, and \texttt{Kleisli m}, which makes functions of type \texttt{a -> m b} into \texttt{Arrows} for any \texttt{Monad m}. These instances are shown in Listing 38.
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) )

Listing 38: The (->) and Kleisli m instances of Arrow

Laws

There are quite a few laws that instances of Arrow should satisfy [102, 103, 104];
ythey are shown in Listing 39. Note that the version of the laws shown in Listing 39
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 [104], which uses the Category class.

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))

Listing 39: The Arrow laws

The reader is advised not to lose too much sleep over the Arrow laws,15 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 [104].

15Unless category-theory-induced insomnolence is your cup of tea.
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; it is shown in Listing 40.

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

Listing 40: The ArrowChoice type class

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 \( \text{left} \ g \) is an arrow from \( \text{Either} \ b \ d \) to \( \text{Either} \ c \ d \). On inputs tagged with Left, the \( \text{left} \ g \) arrow has the behavior of \( g \); on inputs tagged with Right, it behaves as the identity.
- The \( \text{right} \) function, of course, is the mirror image of \( \text{left} \). The arrow \( \text{right} \ g \) 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 as \( h \) 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 shown in Listing 41.

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

**Listing 41**: The *ArrowApply* type class

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 ~> ((b ~> c) ~> 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, shown in Listing 42.

```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
```

**Listing 42**: Equivalence of *ArrowApply* and *Monad*

**ArrowLoop**

The *ArrowLoop* type class is shown in Listing 43; 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 \texttt{loop g} is obtained by recursively “fixing” the second component of the input to \texttt{g}.

\begin{alltt}
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
\end{alltt}

\textbf{Listing 43:} The \texttt{ArrowLoop} type class

It can be a bit difficult to grok what the \texttt{trace} function is doing. How can \texttt{d} appear on the left and right sides of the \texttt{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 \texttt{fix} function, and to read Paterson’s arrow tutorial [104].

\section*{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 \texttt{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 [104], is shown in Listing 44. 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 [105], or his later tutorial which presents a simplified version [104].

\section*{Further reading}

An excellent starting place for the student of arrows is the web page put together by Paterson [101], which contains an introduction and many references. Some key papers on arrows include Hughes’s original paper introducing arrows, \textit{Generalising Monads to Arrows} [102], and Paterson’s paper on arrow notation [105].
class ArrowLoop (~>) => ArrowCircuit (~>) where
delay :: b -> (b ~> b)

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

Listing 44: An example arrow using arrow notation

Both Hughes and Paterson later wrote accessible tutorials intended for a broader audience [104, 106]. 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 [107], who also invented a new calculus of arrows, based on the lambda calculus, which considerably simplifies the presentation of the arrow laws [103].

Some examples of Arrows include Yampa [108], the Haskell XML Toolkit [109], and the functional GUI library Grapefruit [110].

Some extensions to arrows have been explored; for example, the BiArrows of Alimarine et al., for two-way instead of one-way computation [111].

Links to many additional research papers relating Arrows can be found on the Haskell wiki [69].

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 category-extras library [2], is shown in Listing 45.

As you can see, extract is the dual of return, duplicate is the dual of join, and extend is the dual of (>>=) (although its arguments are in a different order).
class Functor f => Copointed f where  
extract :: f a -> a

class Copointed w => Comonad w where  
duplicate :: w a -> w (w a)  
extend :: (w a -> b) -> w a -> w b

---

**Listing 45:** The Comonad type class

The definition of Comonad is a bit redundant (after all, the Monad class does not need join), but this is so that a Comonad can be defined by fmap, extract, and either duplicate or extend. Each has a default implementation in terms of the other.

A prototypical example of a Comonad instance is shown in Listing 46.

---

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

instance Functor Stream where  
fmap g (Cons x xs) = Cons (g x) (fmap g xs)

instance Copointed Stream where  
extract (Cons x _) = x

-- '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  
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)
```

**Listing 46:** A Comonad instance for Stream
Further reading

Dan Piponi explains in a blog post what cellular automata have to do with comonads [112]. In another blog post, Conal Elliott has examined a comonadic formulation of functional reactive programming [113]. Sterling Clover’s blog post Comonads in everyday life [114] 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 [115, 116, 117, 118, 119].

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 this article: 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 [120, 121] is a first-year Ph.D. student in the programming languages group at the University of Pennsylvania [122]. He enjoys teaching, creating EDSLs, playing Bach fugues, musing upon category theory, and cooking tasty lambd-treats for the denizens of #haskell.
References


3. Dan Piponi. The Mother of all Monads. 


http://members.chello.nl/hjgtuyl/tourdemonad.html.


7. Monad laws. 
http://haskell.org/haskellwiki/Monad_laws.


15. Jeff Newbern. All About Monads. 
http://www.haskell.org/all_about_monads/html/.
http://haskell.org/haskellwiki/Monads_as_Containers.


http://haskell.org/haskellwiki/Monads_as_computation.

http://metafoo.co.uk/practical-monads.txt.

[53] Cale Gibbard. Introduction to IO.  
http://www.haskell.org/haskellwiki/Introduction_to_IO.

[54] Monad tutorials timeline.  
http://haskell.org/haskellwiki/Monad_tutorials_timeline.

[55] Don Stewart and Eric Kow. think of a monad...  


[57] Dan Piponi. Monads, a Field Guide.  

[58] Tim Newsham. What’s a Monad?  

[59] Blog articles/Monads.  
http://haskell.org/haskellwiki/Blog_articles/Monads.


http://www.soi.city.ac.uk/%7Eross/papers/FingerTree.html.


http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers1/.

http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers2/.

http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers3/.
Chung-Chieh Shan and Dylan Thurston. Word numbers, part 4: Sort the words, sum the numbers. 
http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers4/.


Yampa:FunctionalReactiveProgrammingwithArrows.


http://conal.net/blog/posts/functional-interactive-behavior/.


