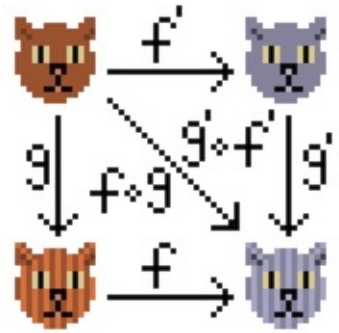
Table of Contents

Cats

Overview

Cats is a library which provides abstractions for functional programming in Scala.

The name is a playful shortening of the word *category*.



Getting Started

Cats is currently available for Scala 2.10 and 2.11.

To get started with SBT, simply add the following to your `build.sbt` file:

```
libraryDependencies += "org.typelevel" %% "cats" % "0.6.0"
```

This will pull in all of Cats' modules. If you only require some functionality, you can pick-and-choose from amongst these modules (used in place of `"cats"`):

- `cats-macros` : Macros used by Cats syntax (*required*).
- `cats-kernel` : Small set of basic type classes (*required*).
- `cats-core` : Most core type classes and functionality (*required*).
- `cats-laws` : Laws for testing type class instances.
- `cats-free` : Free structures such as the free monad, and supporting type classes.

Release notes for Cats are available in CHANGES.md.

Cats is still under active development. While we don't anticipate any major redesigns, changes that are neither source nor binary compatibility are to be expected in upcoming cats releases. We will update the minor version of cats accordingly for such changes. Once cats 1.0 is released (ETA: Q3 2016), there will be an increased focus on making changes in compatible ways.

Documentation

Documentation is here.

Among the goals of Cats is to provide approachable and useful documentation. Documentation is available in the form of tutorials on the Cats website, as ePub, Mobi or PDF, as well as through Scaladoc (also reachable through the website).

Building Cats

To build Cats you should have sbt and Node.js installed. Run `sbt`, and then use any of the following commands:

- `compile` : compile the code

- `console` : launch a REPL
- `test` : run the tests
- `unidoc` : generate the documentation
- `scalastyle` : run the style-checker on the code
- `validate` : run tests, style-checker, and doc generation

Scala and Scala-js

Cats cross-compiles to both JVM and Javascript(JS). If you are not used to working with cross-compiling builds, the first things that you will notice is that builds:

- Will take longer: To build JVM only, just use the `catsJVM`, or `catsJS` for JS only. And if you want the default project to be `catsJVM`, just copy the file `scripts/sbtrc-JVM` to `.sbtrc` in the root directory.
- May run out of memory: We suggest you use Paul Philips's sbt script that will use the settings from Cats.

Design

The goal is to provide a lightweight, modular, and extensible library that is approachable and powerful. We will also provide useful documentation and examples which are type-checked by the compiler to ensure correctness.

Cats will be designed to use modern *best practices*:

- simulacrum for minimizing type class boilerplate
- machinist for optimizing implicit operators
- scalacheck for property-based testing
- discipline for encoding and testing laws
- kind-projector for type lambda syntax
- algebra for shared algebraic structures
- ...and of course a pure functional subset of the Scala language.

(We also plan to support Miniboxing in a branch.)

Currently Cats is experimenting with providing laziness via a type constructor (`Eval[_]`), rather than via ad-hoc by-name parameters.This design may change if it ends up being impractical.

The goal is to make Cats as efficient as possible for both strict and lazy evaluation. There are also issues around by-name parameters that mean they are not well-suited to all situations where laziness is desirable.

Modules

Cats will be split into modules, both to keep the size of the artifacts down and also to avoid unnecessarily tight coupling between type classes and data types.

Initially Cats will support the following modules:

- `macros` : Macro definitions needed for `core` and other projects.
- `core` : Definitions for widely-used type classes and data types.
- `laws` : The encoded laws for type classes, exported to assist third-party testing.
- `cats-free` : Free structures such as the free monad, and supporting type classes.
- `tests` : Verifies the laws, and runs any other tests. Not published.

As the type class families grow, it's possible that additional modules will be added as well. Modules which depend on other libraries (e.g. Shapeless-based type class derivation) may be added as well.

How can I contribute to Cats?

There are many ways to support Cats' development:

- Fix bugs: Despite using static types, law-checking, and property-based testing bugs can happen. Reporting problems you encounter (with the documentation, code, or anything else) helps us to improve. Look for issues labelled "ready" as good targets, but **please add a comment to the issue** if you start working on one. We want to avoid any duplicated effort.
- Write ScalaDoc comments: One of our goals is to have ScalaDoc comments for all types in Cats. The documentation should describe the type and give a basic usage (it may also link to relevant papers).
- Write tutorials and examples: In addition to inline ScalaDoc comments, we hope to provide Markdown-based tutorials which can demonstrate how to use all the provided types. These should be *literate programs* i.e. narrative text interspersed with code.
- Improve the laws and tests: Cats' type classes rely on laws (and law-checking) to make type classes useful and reliable. If you notice laws or tests which are missing (or could be improved) you can open an issue (or send a pull request).
- Help with code review: Most of our design decisions are made through conversations on issues and pull requests. You can participate in these conversations to help guide the future of Cats.

  We will be using the **meta** label for large design decisions, and your input on these is especially appreciated.
- Contribute new code: Cats is growing! If there are type classes (or concrete data types) which you need, feel free to contribute! You can open an issue to discuss your idea, or start hacking and submit a pull request. One advantage of opening an issue is that it may save you time to get other opinions on your approach.
- Ask questions: we are hoping to make Cats (and functional programming in Scala) accessible to the largest number of people. If you have questions it is likely many other people do as well, and as a community this is how we can grow and improve.

Maintainers

The current maintainers (people who can merge pull requests) are:

- ceedubs Cody Allen
- rossabaker Ross Baker
- travisbrown Travis Brown
- adelbertc Adelbert Chang
- tpolecat Rob Norris
- stew Mike O'Connor
- non Erik Osheim
- mpilquist Michael Pilquist
- milessabin Miles Sabin
- fthomas Frank Thomas
- julien-truffaut Julien Truffaut
- kailuowang Kailuo Wang

We are currently following a practice of requiring at least two sign-offs to merge PRs (and for large or contentious issues we may wait for more). For typos or other small fixes to documentation we relax this to a single sign-off.

Contributing

Discussion around Cats is currently happening in the Gitter channel as well as on Github issue and PR pages. You can get an overview of who is working on what via Waffle.io.

Feel free to open an issue if you notice a bug, have an idea for a feature, or have a question about the code. Pull requests are also gladly accepted. For more information, check out the contributor guide. You can also see a list of past contributors in AUTHORS.md.

People are expected to follow the Typelevel Code of Conduct when discussing Cats on the Github page, Gitter channel, or other venues.

We hope that our community will be respectful, helpful, and kind. If you find yourself embroiled in a situation that becomes heated, or that fails to live up to our expectations, you should disengage and contact one of the project maintainers in private. We hope to avoid letting minor aggressions and misunderstandings escalate into larger problems.

If you are being harassed, please contact one of us immediately so that we can support you.

Related Projects

Cats is closely-related to Structures; both projects are descended from Scalaz.

There are many related Haskell libraries, for example:

- semigroupoids
- profunctors
- contravariant
- ...and so on.

Copyright and License

Type classes

The type class pattern is a ubiquitous pattern in Scala, its function is to provide a behavior for some type. You think of it as an "interface" in the Java sense. Here's an example.

```
/**
 * A type class to provide textual representation
 */
trait Show[A] {
  def show(f: A): String
}
```

This class says that a value of type `Show[A]` has a way to turn `A` s into `String` s. Now we can write a function which is polymorphic on some `A` , as long as we have some value of `Show[A]` , so that our function can have a way of producing a `String` :

```
def log[A](a: A)(implicit s: Show[A]) = println(s.show(a))
```

If we now try to call log, without supplying a `Show` instance, we will get a compilation error:

```
scala> log("a string")
<console>:15: error: could not find implicit value for parameter s: Show[String]
       log("a string")
          ^
```

It is trivial to supply a `Show` instance for `String` :

```
implicit val stringShow = new Show[String] {
  def show(s: String) = s
}
```

and now our call to Log succeeds

```
log("a string")
// a string
```

This example demonstrates a powerful property of the type class pattern. We have been able to provide an implementation of `Show` for `String` , without needing to change the definition of `java.lang.String` to extend a new Java-style interface; something we couldn't have done even if we wanted to, since we don't control the implementation of `java.lang.String` . We use this pattern to retrofit existing types with new behaviors. This is usually referred to as "ad-hoc polymorphism".

For some types, providing a `Show` instance might depend on having some implicit `Show` instance of some other type, for instance, we could implement `Show` for `Option` :

```
implicit def optionShow[A](implicit sa: Show[A]) = new Show[Option[A]] {
  def show(oa: Option[A]): String = oa match {
    case None => "None"
    case Some(a) => "Some("+ sa.show(a) + ")"
  }
}
```

Now we can call our log function with a `Option[String]` or a `Option[Option[String]]` :

```
log(Option(Option("hello")))
// Some(Some(hello))
```

Scala has syntax just for this pattern that we use frequently:

```
def log[A: Show](a: A) = println(implicitly[Show[A]].show(a))
```

is the same as

```
def log[A](a: A)(implicit s: Show[A]) = println(s.show(a))
```

That is that declaring the type parameter as `A : Show`, it will add an implicit parameter to the method signature (with a name we do not know).

Applicative

`Applicative` extends `Apply` by adding a single method, `pure`:

```
    def pure[A](x: A): F[A]
`
```

This method takes any value and returns the value in the context of the functor. For many familiar functors, how to do this is obvious. For `Option`, the `pure` operation wraps the value in `Some`. For `List`, the `pure` operation returns a single element `List`:

```
import cats._
// import cats._

import cats.implicits._
// import cats.implicits._

Applicative[Option].pure(1)
// res0: Option[Int] = Some(1)

Applicative[List].pure(1)
// res1: List[Int] = List(1)
```

Like `Functor` and `Apply`, `Applicative` functors also compose naturally with each other. When you compose one `Applicative` with another, the resulting `pure` operation will lift the passed value into one context, and the result into the other context:

```
import cats.data.Nested
// import cats.data.Nested

val nested = Applicative[Nested[List, Option, ?]].pure(1)
// nested: cats.data.Nested[[+A]List[A],Option,Int] = Nested(List(Some(1)))

val unwrapped = nested.value
// unwrapped: List[Option[Int]] = List(Some(1))
```

Applicative Functors & Monads

`Applicative` is a generalization of `Monad`, allowing expression of effectful computations in a pure functional way.

`Applicative` is generally preferred to `Monad` when the structure of a computation is fixed a priori. That makes it possible to perform certain kinds of static analysis on applicative values.

Apply

`Apply` extends the `Functor` type class (which features the familiar `map` function) with a new function `ap`. The `ap` function is similar to `map` in that we are transforming a value in a context (a context being the `F` in `F[A]`; a context can be `Option`, `List` or `Future` for example). However, the difference between `ap` and `map` is that for `ap` the function that takes care of the transformation is of type `F[A => B]`, whereas for `map` it is `A => B`:

```
import cats._

val intToString: Int => String = _.toString
val double: Int => Int = _ * 2
val addTwo: Int => Int = _ + 2

implicit val optionApply: Apply[Option] = new Apply[Option] {
  def ap[A, B](f: Option[A => B])(fa: Option[A]): Option[B] =
    fa.flatMap (a => f.map (ff => ff(a)))

  def map[A,B](fa: Option[A])(f: A => B): Option[B] = fa map f
}

implicit val listApply: Apply[List] = new Apply[List] {
  def ap[A, B](f: List[A => B])(fa: List[A]): List[B] =
    fa.flatMap (a => f.map (ff => ff(a)))

  def map[A,B](fa: List[A])(f: A => B): List[B] = fa map f
}
```

map

Since `Apply` extends `Functor`, we can use the `map` method from `Functor`:

```
Apply[Option].map(Some(1))(intToString)
// res3: Option[String] = Some(1)

Apply[Option].map(Some(1))(double)
// res4: Option[Int] = Some(2)

Apply[Option].map(None)(double)
// res5: Option[Int] = None
```

compose

And like functors, `Apply` instances also compose (via the `Nested` data type):

```
import cats.data.Nested
// import cats.data.Nested

val listOpt = Nested[List, Option, Int](List(Some(1), None, Some(3)))
// listOpt: cats.data.Nested[List,Option,Int] = Nested(List(Some(1), None, Some(3)))

val plusOne = (x:Int) => x + 1
// plusOne: Int => Int = <function1>

val f = Nested[List, Option, Int => Int](List(Some(plusOne)))
// f: cats.data.Nested[List,Option,Int => Int] = Nested(List(Some(<function1>)))

Apply[Nested[List, Option, ?]].ap(f)(listOpt)
// res6: cats.data.Nested[[+A]List[A],Option,Int] = Nested(List(Some(2), None, Some(4)))
```

ap

The `ap` method is a method that `Functor` does not have:

```
Apply[Option].ap(Some(intToString))(Some(1))
// res7: Option[String] = Some(1)

Apply[Option].ap(Some(double))(Some(1))
// res8: Option[Int] = Some(2)

Apply[Option].ap(Some(double))(None)
// res9: Option[Int] = None

Apply[Option].ap(None)(Some(1))
// res10: Option[Nothing] = None

Apply[Option].ap(None)(None)
// res11: Option[Nothing] = None
```

ap2, ap3, etc

`Apply` also offers variants of `ap`. The functions `apN` (for `N` between `2` and `22`) accept `N` arguments where `ap` accepts `1`:

For example:

```
val addArity2 = (a: Int, b: Int) => a + b
// addArity2: (Int, Int) => Int = <function2>

Apply[Option].ap2(Some(addArity2))(Some(1), Some(2))
// res12: Option[Int] = Some(3)

val addArity3 = (a: Int, b: Int, c: Int) => a + b + c
// addArity3: (Int, Int, Int) => Int = <function3>

Apply[Option].ap3(Some(addArity3))(Some(1), Some(2), Some(3))
// res13: Option[Int] = Some(6)
```

Note that if any of the arguments of this example is `None`, the final result is `None` as well. The effects of the context we are operating on are carried through the entire computation:

```
Apply[Option].ap2(Some(addArity2))(Some(1), None)
// res14: Option[Int] = None

Apply[Option].ap4(None)(Some(1), Some(2), Some(3), Some(4))
// res15: Option[Nothing] = None
```

map2, map3, etc

Similarly, `mapN` functions are available:

```
Apply[Option].map2(Some(1), Some(2))(addArity2)
// res16: Option[Int] = Some(3)

Apply[Option].map3(Some(1), Some(2), Some(3))(addArity3)
// res17: Option[Int] = Some(6)
```

tuple2, tuple3, etc

And `tupleN`:

```
Apply[Option].tuple2(Some(1), Some(2))
// res18: Option[(Int, Int)] = Some((1,2))

Apply[Option].tuple3(Some(1), Some(2), Some(3))
// res19: Option[(Int, Int, Int)] = Some((1,2,3))
```

apply builder syntax

The `|@|` operator offers an alternative syntax for the higher-arity `Apply` functions (`apN` , `mapN` and `tupleN`). In order to use it, first import `cats.syntax.all._` or `cats.syntax.cartesian._` . Here we see that the following two functions, `f1` and `f2` , are equivalent:

```
import cats.implicits._
// import cats.implicits._

def f1(a: Option[Int], b: Option[Int], c: Option[Int]) =
  (a |@| b |@| c) map { _ * _ * _ }
// f1: (a: Option[Int], b: Option[Int], c: Option[Int])Option[Int]

def f2(a: Option[Int], b: Option[Int], c: Option[Int]) =
  Apply[Option].map3(a, b, c)(_ * _ * _)
// f2: (a: Option[Int], b: Option[Int], c: Option[Int])Option[Int]

f1(Some(1), Some(2), Some(3))
// res20: Option[Int] = Some(6)

f2(Some(1), Some(2), Some(3))
// res21: Option[Int] = Some(6)
```

All instances created by `|@|` have `map` , `ap` , and `tupled` methods of the appropriate arity:

```
val option2 = Option(1) |@| Option(2)
// option2: cats.syntax.CartesianBuilder[Option]#CartesianBuilder2[Int,Int] = cats.syntax.CartesianBuilder$CartesianB
uilder2@b01b8ad

val option3 = option2 |@| Option.empty[Int]
// option3: cats.syntax.CartesianBuilder[Option]#CartesianBuilder3[Int,Int,Int] = cats.syntax.CartesianBuilder$Cartes
ianBuilder3@1d2cd142

option2 map addArity2
// res22: Option[Int] = Some(3)

option3 map addArity3
// res23: Option[Int] = None

option2 apWith Some(addArity2)
// res24: Option[Int] = Some(3)

option3 apWith Some(addArity3)
// res25: Option[Int] = None

option2.tupled
// res26: Option[(Int, Int)] = Some((1,2))

option3.tupled
// res27: Option[(Int, Int, Int)] = None
```

Contravariant

The `Contravariant` type class is for functors that define a `contramap` function with the following type:

```
def contramap[A, B](fa: F[A])(f: B => A): F[B]
```

It looks like regular (also called `Covariant`) `Functor`'s `map`, but with the `f` transformation reversed.

Generally speaking, if you have some context `F[A]` for type `A`, and you can get an `A` value out of a `B` value — `Contravariant` allows you to get the `F[B]` context for `B`.

Examples of `Contravariant` instances are `Show` and `scala.math.Ordering` (along with `algebra.Order`).

Contravariant instance for Show.

Say we have class `Money` with a `Show` instance, and `Salary` class.

```
import cats._
import cats.functor._
import cats.implicits._

case class Money(amount: Int)
case class Salary(size: Money)

implicit val showMoney: Show[Money] = Show.show(m => s"$$${m.amount}")
```

If we want to show a `Salary` instance, we can just convert it to a `Money` instance and show it instead.

Let's use `Show`'s `Contravariant`:

```
implicit val showSalary: Show[Salary] = showMoney.contramap(_.size)
// showSalary: cats.Show[Salary] = cats.Show$$anon$2@29458bb3

Salary(Money(1000)).show
// res2: String = $1000
```

Contravariant instance for scala.math.Ordering.

`Show` example is trivial and quite far-fetched, let's see how `Contravariant` can help with orderings.

`scala.math.Ordering` typeclass defines comparison operations, e.g. `compare`:

```
Ordering.Int.compare(2, 1)
// res3: Int = 1

Ordering.Int.compare(1, 2)
// res4: Int = -1
```

There's also a method, called `by`, that creates new `Orderings` out of existing ones:

```
def by[T, S](f: T => S)(implicit ord: Ordering[S]): Ordering[T]
```

In fact, it is just `contramap`, defined in a slightly different way! We supply `T => S` to receive `F[S] => F[T]` back.

So let's use it in our advantage and get `Ordering[Money]` for free:

```
// we need this for `<` to work
import scala.math.Ordered._
// import scala.math.Ordered._

implicit val moneyOrdering: Ordering[Money] = Ordering.by(_.amount)
// moneyOrdering: Ordering[Money] = scala.math.Ordering$$anon$9@533b08ad

Money(100) < Money(200)
// res6: Boolean = true
```

Subtyping

Contravariant functors have a natural relationship with subtyping, dual to that of covariant functors:

```
class A
// defined class A

class B extends A
// defined class B

val b: B = new B
// b: B = B@ddd7233

val a: A = b
// a: A = B@ddd7233

val showA: Show[A] = Show.show(a => "a!")
// showA: cats.Show[A] = cats.Show$$anon$2@5c988a7a

val showB1: Show[B] = showA.contramap(b => b: A)
// showB1: cats.Show[B] = cats.Show$$anon$2@468682a9

val showB2: Show[B] = showA.contramap(identity[A])
// showB2: cats.Show[B] = cats.Show$$anon$2@22650093

val showB3: Show[B] = Contravariant[Show].narrow[A, B](showA)
// showB3: cats.Show[B] = cats.Show$$anon$2@5c988a7a
```

Subtyping relationships are "lifted backwards" by contravariant functors, such that if `F` is a lawful contravariant functor and `A <: B` then `F[B] <: F[A]`, which is expressed by `Contravariant.narrow`.

Frequently Asked Questions

What imports do I need?

The easiest approach to cats imports is to import everything that's commonly needed:

```
import cats._
import cats.data._
import cats.implicits._
```

The `cats._` import brings in quite a few typeclasses (similar to interfaces) such as Monad, Semigroup, and Foldable. Instead of the entire `cats` package, you can import only the types that you need, for example:

```
import cats.Monad
import cats.Semigroup
import cats.Foldable
```

The `cats.data._`, import brings in data structures such as Xor, Validated, and State. Instead of the entire `cats.data` package, you can import only the types that you need, for example:

```
import cats.data.Xor
import cats.data.Validated
import cats.data.State
```

The `cats.implicits._` import does a couple of things. Firstly, it brings in implicit type class instances for standard library types - so after this import you will have `Monad[List]` and `Semigroup[Int]` instances in implicit scope. Secondly, it adds syntax enrichment onto certain types to provide some handy methods, for example:

```
// cats adds a toXor method to the standard library's Either
val e: Either[String, Int] = Right(3)
// e: Either[String,Int] = Right(3)

e.toXor
// res1: cats.data.Xor[String,Int] = Right(3)

// cats adds an orEmpty method to the standard library's Option
val o: Option[String] = None
// o: Option[String] = None

o.orEmpty
// res3: String = ""
```

Note: if you import `cats.implicits._` (the preferred method), you should *not* also use imports like `cats.syntax.option._` or `cats.std.either._`. This can result in ambiguous implicit values that cause bewildering compile errors.

Why can't the compiler find implicit instances for Future?

If you have already followed the imports advice but are still getting error messages like `could not find implicit value for parameter e: cats.Monad[scala.concurrent.Future]` or `value |+| is not a member of scala.concurrent.Future[Int]`, then make sure that you have an implicit `scala.concurrent.ExecutionContext` in scope. The easiest way to do this is to `import scala.concurrent.ExecutionContext.Implicits.global`, but note that you may want to use a different execution context for your production application.

How can I turn my List of `<something>` into a `<something>` of a list?

It's really common to have a `List` of values with types like `Option`, `Xor`, or `Validated` that you would like to turn "inside out" into an `Option` (or `Xor` or `Validated`) of a `List`. The `sequence`, `sequenceU`, `traverse`, and `traverseU` methods are *really* handy for this. You can read more about them in the Traverse documentation.

How can I help?

The cats community welcomes and encourages contributions, even if you are completely new to cats and functional programming. Here are a few ways to help out:

- Find an undocumented method and write a ScalaDoc entry for it. See Arrow.scala for some examples of ScalaDoc entries that use sbt-doctest.
- Look at the code coverage report, find some untested code, and write a test for it. Even simple helper methods and syntax enrichment should be tested.
- Find an open issue, leave a comment on it to let people know you are working on it, and submit a pull request. If you are new to cats, you may want to look for items with the low-hanging-fruit label.

See the contributing guide for more information.

Foldable

Foldable type class instances can be defined for data structures that can be folded to a summary value.

In the case of a collection (such as `List` or `Set`), these methods will fold together (combine) the values contained in the collection to produce a single result. Most collection types have `foldLeft` methods, which will usually be used by the associated `Foldable[_]` instance.

`Foldable[F]` is implemented in terms of two basic methods:

- `foldLeft(fa, b)(f)` eagerly folds `fa` from left-to-right.
- `foldRight(fa, b)(f)` lazily folds `fa` from right-to-left.

These form the basis for many other operations, see also: A tutorial on the universality and expressiveness of fold

First some standard imports.

```
import cats._
import cats.implicits._
```

And examples.

```
Foldable[List].fold(List("a", "b", "c"))
// res0: String = abc

Foldable[List].foldMap(List(1, 2, 4))(_.toString)
// res1: String = 124

Foldable[List].foldK(List(List(1,2,3), List(2,3,4)))
// res2: List[Int] = List(1, 2, 3, 2, 3, 4)

Foldable[List].reduceLeftToOption(List[Int]())(_.toString)((s,i) => s + i)
// res3: Option[String] = None

Foldable[List].reduceLeftToOption(List(1,2,3,4))(_.toString)((s,i) => s + i)
// res4: Option[String] = Some(1234)

Foldable[List].reduceRightToOption(List(1,2,3,4))(_.toString)((i,s) => Later(s.value + i)).value
// res5: Option[String] = Some(4321)

Foldable[List].reduceRightToOption(List[Int]())(_.toString)((i,s) => Later(s.value + i)).value
// res6: Option[String] = None

Foldable[Set].find(Set(1,2,3))(_ > 2)
// res7: Option[Int] = Some(3)

Foldable[Set].exists(Set(1,2,3))(_ > 2)
// res8: Boolean = true

Foldable[Set].forall(Set(1,2,3))(_ > 2)
// res9: Boolean = false

Foldable[Set].forall(Set(1,2,3))(_ < 4)
// res10: Boolean = true

Foldable[Vector].filter_(Vector(1,2,3))(_ < 3)
// res11: List[Int] = List(1, 2)

Foldable[List].isEmpty(List(1,2))
// res12: Boolean = false

Foldable[Option].isEmpty(None)
// res13: Boolean = true

Foldable[List].nonEmpty(List(1,2))
```

```
// res14: Boolean = true

Foldable[Option].toList(Option(1))
// res15: List[Int] = List(1)

Foldable[Option].toList(None)
// res16: List[Nothing] = List()

def parseInt(s: String): Option[Int] = scala.util.Try(Integer.parseInt(s)).toOption
// parseInt: (s: String)Option[Int]

Foldable[List].traverse_(List("1", "2"))(parseInt)
// res17: Option[Unit] = Some(())

Foldable[List].traverse_(List("1", "A"))(parseInt)
// res18: Option[Unit] = None

Foldable[List].sequence_(List(Option(1), Option(2)))
// res19: Option[Unit] = Some(())

Foldable[List].sequence_(List(Option(1), None))
// res20: Option[Unit] = None

val prints: Eval[Unit] = List(Eval.always(println(1)), Eval.always(println(2))).sequence_
// prints: cats.Eval[Unit] = cats.Eval$$anon$5@4fd63e6c

prints.value
// 1
// 2

Foldable[List].dropWhile_(List[Int](2,4,5,6,7))(_ % 2 == 0)
// res22: List[Int] = List(5, 6, 7)

Foldable[List].dropWhile_(List[Int](1,2,4,5,6,7))(_ % 2 == 0)
// res23: List[Int] = List(1, 2, 4, 5, 6, 7)

import cats.data.Nested
// import cats.data.Nested

val listOption0 = Nested(List(Option(1), Option(2), Option(3)))
// listOption0: cats.data.Nested[List,Option,Int] = Nested(List(Some(1), Some(2), Some(3)))

val listOption1 = Nested(List(Option(1), Option(2), None))
// listOption1: cats.data.Nested[List,Option,Int] = Nested(List(Some(1), Some(2), None))

Foldable[Nested[List, Option, ?]].fold(listOption0)
// res24: Int = 6

Foldable[Nested[List, Option, ?]].fold(listOption1)
// res25: Int = 3
```

Hence when defining some new data structure, if we can define a `foldLeft` and `foldRight` we are able to provide many other useful operations, if not always the most efficient implementations, over the structure without further implementation.

Note that, in order to support laziness, the signature of `Foldable`'s `foldRight` is

```
def foldRight[A, B](fa: F[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B]
```

as opposed to

```
def foldRight[A, B](fa: F[A], z: B)(f: (A, B) => B): B
```

which someone familiar with the `foldRight` from the collections in Scala's standard library might expect. This will prevent operations which are lazy in their right hand argument to traverse the entire structure unnecessarily. For example, if you have:

```
val allFalse = Stream.continually(false)
// allFalse: scala.collection.immutable.Stream[Boolean] = Stream(false, ?)
```

which is an infinite stream of `false` values, and if you wanted to reduce this to a single false value using the logical and (`&&`). You intuitively know that the result of this operation should be `false` . It is not necessary to consider the entire stream in order to determine this result, you only need to consider the first value. Using `foldRight` from the standard library *will* try to consider the entire stream, and thus will eventually cause a stack overflow:

```
try {
  allFalse.foldRight(true)(_ && _)
} catch {
  case e:StackOverflowError => println(e)
}
// java.lang.StackOverflowError
// res26: AnyVal = ()
```

With the lazy `foldRight` on `Foldable` , the calculation terminates after looking at only one value:

```
Foldable[Stream].foldRight(allFalse, Eval.True)((a,b) => if (a) b else Eval.now(false)).value
// res27: Boolean = false
```

Functor

A `Functor` is a ubiquitous type class involving types that have one "hole", i.e. types which have the shape `F[?]`, such as `Option`, `List` and `Future`. (This is in contrast to a type like `Int` which has no hole, or `Tuple2` which has two holes (`Tuple2[?,?]`)).

The `Functor` category involves a single operation, named `map`:

```
def map[A, B](fa: F[A])(f: A => B): F[B]
```

This method takes a function `A => B` and turns an `F[A]` into an `F[B]`. The name of the method `map` should remind you of the `map` method that exists on many classes in the Scala standard library, for example:

```
Option(1).map(_ + 1)
// res0: Option[Int] = Some(2)

List(1,2,3).map(_ + 1)
// res1: List[Int] = List(2, 3, 4)

Vector(1,2,3).map(_.toString)
// res2: scala.collection.immutable.Vector[String] = Vector(1, 2, 3)
```

Creating Functor instances

We can trivially create a `Functor` instance for a type which has a well behaved `map` method:

```
import cats._

implicit val optionFunctor: Functor[Option] = new Functor[Option] {
  def map[A,B](fa: Option[A])(f: A => B) = fa map f
}

implicit val listFunctor: Functor[List] = new Functor[List] {
  def map[A,B](fa: List[A])(f: A => B) = fa map f
}
```

However, functors can also be created for types which don't have a `map` method. For example, if we create a `Functor` for `Function1[In, ?]` we can use `andThen` to implement `map`:

```
implicit def function1Functor[In]: Functor[Function1[In, ?]] =
  new Functor[Function1[In, ?]] {
    def map[A,B](fa: In => A)(f: A => B): Function1[In,B] = fa andThen f
  }
```

This example demonstrates the use of the kind-projector compiler plugin. This compiler plugin can help us when we need to change the number of type holes. In the example above, we took a type which normally has two type holes, `Function1[?,?]` and constrained one of the holes to be the `In` type, leaving just one hole for the return type, resulting in `Function1[In,?]`. Without kind-projector, we'd have to write this as something like `({type F[A] = Function1[In,A]})#F`, which is much harder to read and understand.

Using Functor

map

`List` is a functor which applies the function to each element of the list:

```
val len: String => Int = _.length
// len: String => Int = <function1>

Functor[List].map(List("qwer", "adsfg"))(len)
// res5: List[Int] = List(4, 5)
```

`Option` is a functor which only applies the function when the `Option` value is a `Some`:

```
Functor[Option].map(Some("adsf"))(len) // Some(x) case: function is applied to x; result is wrapped in Some
// res6: Option[Int] = Some(4)

Functor[Option].map(None)(len) // None case: simply returns None (function is not applied)
// res7: Option[Int] = None
```

Derived methods

lift

We can use `Functor` to "lift" a function from `A => B` to `F[A] => F[B]`:

```
val lenOption: Option[String] => Option[Int] = Functor[Option].lift(len)
// lenOption: Option[String] => Option[Int] = <function1>

lenOption(Some("abcd"))
// res8: Option[Int] = Some(4)
```

fproduct

`Functor` provides an `fproduct` function which pairs a value with the result of applying a function to that value.

```
val source = List("a", "aa", "b", "ccccc")
// source: List[String] = List(a, aa, b, ccccc)

Functor[List].fproduct(source)(len).toMap
// res9: scala.collection.immutable.Map[String,Int] = Map(a -> 1, aa -> 2, b -> 1, ccccc -> 5)
```

compose

Functors compose! Given any functor `F[_]` and any functor `G[_]` we can create a new functor `F[G[_]]` by composing them via the `Nested` data type:

```
import cats.data.Nested
// import cats.data.Nested

val listOpt = Nested[List, Option, Int](List(Some(1), None, Some(3)))
// listOpt: cats.data.Nested[List,Option,Int] = Nested(List(Some(1), None, Some(3)))

Functor[Nested[List, Option, ?]].map(listOpt)(_ + 1)
// res10: cats.data.Nested[[+A]List[A],Option,Int] = Nested(List(Some(2), None, Some(4)))

val optList = Nested[Option, List, Int](Some(List(1, 2, 3)))
// optList: cats.data.Nested[Option,List,Int] = Nested(Some(List(1, 2, 3)))

Functor[Nested[Option, List, ?]].map(optList)(_ + 1)
// res11: cats.data.Nested[Option,[+A]List[A],Int] = Nested(Some(List(2, 3, 4)))
```

Subtyping

Functors have a natural relationship with subtyping:

```
class A
// defined class A

class B extends A
// defined class B

val b: B = new B
// b: B = B@518ea869

val a: A = b
// a: A = B@518ea869

val listB: List[B] = List(new B)
// listB: List[B] = List(B@52475ef6)

val listA1: List[A] = listB.map(b => b: A)
// listA1: List[A] = List(B@52475ef6)

val listA2: List[A] = listB.map(identity[A])
// listA2: List[A] = List(B@52475ef6)

val listA3: List[A] = Functor[List].widen[B, A](listB)
// listA3: List[A] = List(B@52475ef6)
```

Subtyping relationships are "lifted" by functors, such that if `F` is a lawful functor and `A <: B` then `F[A] <: F[B]` - almost. Almost, because to convert an `F[B]` to an `F[A]` a call to `map(identity[A])` is needed (provided as `widen` for convenience). The functor laws guarantee that `fa map identity == fa`, however.

Id

The identity monad can be seen as the ambient monad that encodes the effect of having no effect. It is ambient in the sense that plain pure values are values of `Id`.

It is encoded as:

```
type Id[A] = A
```

That is to say that the type Id[A] is just a synonym for A. We can freely treat values of type `A` as values of type `Id[A]`, and vice-versa.

```
import cats._
// import cats._

val x: Id[Int] = 1
// x: cats.Id[Int] = 1

val y: Int = x
// y: Int = 1
```

Using this type declaration, we can treat our Id type constructor as a `Monad` and as a `Comonad`. The `pure` method, which has type `A => Id[A]` just becomes the identity function. The `map` method from `Functor` just becomes function application:

```
import cats.Functor
// import cats.Functor

val one: Int = 1
// one: Int = 1

Functor[Id].map(one)(_ + 1)
// res0: cats.Id[Int] = 2
```

Compare the signatures of `map` and `flatMap` and `coflatMap`:

```
  def map[A, B](fa: Id[A])(f: A => B): Id[B]
  def flatMap[A, B](fa: Id[A])(f: A => Id[B]): Id[B]
  def coflatMap[A, B](a: Id[A])(f: Id[A] => B): Id[B]
```

You'll notice that in the flatMap signature, since `Id[B]` is the same as `B` for all B, we can rewrite the type of the `f` parameter to be `A => B` instead of `A => Id[B]`, and this makes the signatures of the two functions the same, and, in fact, they can have the same implementation, meaning that for `Id`, `flatMap` is also just function application:

```
import cats.Monad
// import cats.Monad

val one: Int = 1
// one: Int = 1

Monad[Id].map(one)(_ + 1)
// res1: cats.Id[Int] = 2

Monad[Id].flatMap(one)(_ + 1)
// res2: cats.Id[Int] = 2
```

And that similarly, coflatMap is just function application:

```
import cats.Comonad
// import cats.Comonad

Comonad[Id].coflatMap(one)(_ + 1)
// res3: cats.Id[Int] = 2
```

Invariant

The `Invariant` type class is for functors that define an `imap` function with the following type:

```
def imap[A, B](fa: F[A])(f: A => B)(g: B => A): F[B]
```

Every covariant (as well as contravariant) functor gives rise to an invariant functor, by ignoring the `g` (or in case of contravariance, `f`) function.

Examples for instances of `Invariant` are `Semigroup` and `Monoid`, in the following we will explain why this is the case using `Semigroup`, the reasoning for `Monoid` is analogous.

Invariant instance for Semigroup

Pretend that we have a `Semigroup[Long]` representing a standard UNIX timestamp. Let's say that we want to create a `Semigroup[Date]`, by *reusing* `Semigroup[Long]`.

Semigroup does not form a covariant functor

If `Semigroup` had an instance for the standard covariant `Functor` typeclass, we could use `map` to apply a function `longToDate`:

```
import java.util.Date
def longToDate: Long => Date = new Date(_)
```

But is this enough to give us a `Semigroup[Date]`? The answer is no, unfortunately. A `Semigroup[Date]` should be able to combine two values of type `Date`, given a `Semigroup` that only knows how to combine `Long`s! The `longToDate` function does not help at all, because it only allows us to convert a `Long` into a `Date`. Seems like we can't have an `Functor` instance for `Semigroup`.

Semigroup does not form a contravariant functor

On the other side, if `Semigroup` would form a *contravariant* functor by having an instance for `Contravariant`, we could make use of `contramap` to apply a function `dateToLong`:

```
import java.util.Date
def dateToLong: Date => Long = _.getTime
```

Again we are faced with a problem when trying to get a `Semigroup[Date]` based on a `Semigroup[Long]`. As before consider the case where we have two values of `Date` at hand. Using `dateToLong` we can turn them into `Long`s and use `Semigroup[Long]` to combine the two values. We are left with a value of type `Long`, but we can't turn it back into a `Date` using only `contramap`!

Semigroup does form an invariant functor

From the previous discussion we conclude that we need both the `map` from (covariant) `Functor` and `contramap` from `Contravariant`. There already is a type class for this and it is called `Invariant`. Instances of the `Invariant` type class provide the `imap` function:

```
def imap[A, B](fa: F[A])(f: A => B)(g: B => A): F[B]
```

Reusing the example of turning `Semigroup[Long]` into `Semigroup[Date]`, we can use the `g` parameter to turn `Date` into a `Long`, combine our two values using `Semigroup[Long]` and then convert the result back into a `Date` using the `f` parameter of `imap`:

```
import java.util.Date

// import everything for simplicity:
import cats._
import cats.implicits._

// or only import what's actually required:
// import cats.Semigroup
// import cats.std.long._
// import cats.syntax.semigroup._
// import cats.syntax.invariant._

def longToDate: Long => Date = new Date(_)
def dateToLong: Date => Long = _.getTime

implicit val semigroupDate: Semigroup[Date] =
  Semigroup[Long].imap(longToDate)(dateToLong)

val today: Date = longToDate(1449088684104l)
val timeLeft: Date = longToDate(1900918893l)
```

```
today |+| timeLeft
// res11: java.util.Date = Thu Dec 24 21:40:02 CET 2015
```

Monad

`Monad` extends the `Applicative` type class with a new function `flatten`. Flatten takes a value in a nested context (eg. `F[F[A]]` where F is the context) and "joins" the contexts together so that we have a single context (ie. `F[A]`).

The name `flatten` should remind you of the functions of the same name on many classes in the standard library.

```
Option(Option(1)).flatten
// res0: Option[Int] = Some(1)

Option(None).flatten
// res1: Option[Nothing] = None

List(List(1),List(2,3)).flatten
// res2: List[Int] = List(1, 2, 3)
```

Monad instances

If `Applicative` is already present and `flatten` is well-behaved, extending the `Applicative` to a `Monad` is trivial. To provide evidence that a type belongs in the `Monad` type class, cats' implementation requires us to provide an implementation of `pure` (which can be reused from `Applicative`) and `flatMap`.

We can use `flatten` to define `flatMap`: `flatMap` is just `map` followed by `flatten`. Conversely, `flatten` is just `flatMap` using the identity function `x => x` (i.e. `flatMap(_)(x => x)`).

```
import cats._

implicit def optionMonad(implicit app: Applicative[Option]) =
  new Monad[Option] {
    // Define flatMap using Option's flatten method
    override def flatMap[A, B](fa: Option[A])(f: A => Option[B]): Option[B] =
      app.map(fa)(f).flatten
    // Reuse this definition from Applicative.
    override def pure[A](a: A): Option[A] = app.pure(a)
  }
```

flatMap

`flatMap` is often considered to be the core function of `Monad`, and cats follows this tradition by providing implementations of `flatten` and `map` derived from `flatMap` and `pure`.

```
implicit val listMonad = new Monad[List] {
  def flatMap[A, B](fa: List[A])(f: A => List[B]): List[B] = fa.flatMap(f)
  def pure[A](a: A): List[A] = List(a)
}
```

Part of the reason for this is that name `flatMap` has special significance in scala, as for-comprehensions rely on this method to chain together operations in a monadic context.

```
import scala.reflect.runtime.universe
// import scala.reflect.runtime.universe

universe.reify(
  for {
    x <- Some(1)
    y <- Some(2)
  } yield x + y
).tree
// res4: reflect.runtime.universe.Tree = Some.apply(1).flatMap(((x) => Some.apply(2).map(((y) => x.$plus(y)))))
```

ifM

`Monad` provides the ability to choose later operations in a sequence based on the results of earlier ones. This is embodied in `ifM`, which lifts an `if` statement into the monadic context.

```
Monad[List].ifM(List(true, false, true))(List(1, 2), List(3, 4))
// res5: List[Int] = List(1, 2, 3, 4, 1, 2)
```

Composition

Unlike `Functor`s and `Applicative`s, not all `Monad`s compose. This means that even if `M[_]` and `N[_]` are both `Monad`s, `M[N[_]]` is not guaranteed to be a `Monad`.

However, many common cases do. One way of expressing this is to provide instructions on how to compose any outer monad (`F` in the following example) with a specific inner monad (`Option` in the following example).

Note: the example below assumes usage of the kind-projector compiler plugin and will not compile if it is not being used in a project.

```
case class OptionT[F[_], A](value: F[Option[A]])

implicit def optionTMonad[F[_]](implicit F : Monad[F]) = {
  new Monad[OptionT[F, ?]] {
    def pure[A](a: A): OptionT[F, A] = OptionT(F.pure(Some(a)))
    def flatMap[A, B](fa: OptionT[F, A])(f: A => OptionT[F, B]): OptionT[F, B] =
      OptionT {
        F.flatMap(fa.value) {
          case None => F.pure(None)
          case Some(a) => f(a).value
        }
      }
  }
}
```

This sort of construction is called a monad transformer.

Cats has an `OptionT` monad transformer, which adds a lot of useful functions to the simple implementation above.

Monoid

`Monoid` extends the `Semigroup` type class, adding an `empty` method to semigroup's `combine`. The `empty` method must return a value that when combined with any other instance of that type returns the other instance, i.e.

```
(combine(x, empty) == combine(empty, x) == x)
```

For example, if we have a `Monoid[String]` with `combine` defined as string concatenation, then `empty = ""`.

Having an `empty` defined allows us to combine all the elements of some potentially empty collection of `T` for which a `Monoid[T]` is defined and return a `T`, rather than an `Option[T]` as we have a sensible default to fall back to.

First some imports.

```
import cats._
import cats.implicits._
```

Examples.

```
Monoid[String].empty
// res0: String = ""

Monoid[String].combineAll(List("a", "b", "c"))
// res1: String = abc

Monoid[String].combineAll(List())
// res2: String = ""
```

The advantage of using these type class provided methods, rather than the specific ones for each type, is that we can compose monoids to allow us to operate on more complex types, e.g.

```
Monoid[Map[String,Int]].combineAll(List(Map("a" -> 1, "b" -> 2), Map("a" -> 3)))
// res3: Map[String,Int] = Map(b -> 2, a -> 4)

Monoid[Map[String,Int]].combineAll(List())
// res4: Map[String,Int] = Map()
```

This is also true if we define our own instances. As an example, let's use `Foldable`'s `foldMap`, which maps over values accumulating the results, using the available `Monoid` for the type mapped onto.

```
val l = List(1, 2, 3, 4, 5)
// l: List[Int] = List(1, 2, 3, 4, 5)

l.foldMap(identity)
// res5: Int = 15

l.foldMap(i => i.toString)
// res6: String = 12345
```

To use this with a function that produces a tuple, cats also provides a `Monoid` for a tuple that will be valid for any tuple where the types it contains also have a `Monoid` available, thus.

```
l.foldMap(i => (i, i.toString)) // do both of the above in one pass, hurrah!
// res7: (Int, String) = (15,12345)
```

N.B. Cats does not define a `Monoid` type class itself, it uses the `Monoid` trait which is defined in the algebra project on which it depends. The `cats` package object defines type aliases to the `Monoid` from algebra, so that you can `import cats.Monoid`. Also the `Monoid` instance for tuple is also implemented in algebra, cats merely provides it through inheritance.

Semigroup

A semigroup for some given type A has a single operation (which we will call `combine`), which takes two values of type A, and returns a value of type A. This operation must be guaranteed to be associative. That is to say that:

```
((a combine b) combine c)
```

must be the same as

```
(a combine (b combine c))
```

for all possible values of a,b,c.

There are instances of `Semigroup` defined for many types found in the scala common library:

First some imports.

```
import cats._
import cats.implicits._
```

Examples.

```
Semigroup[Int].combine(1, 2)
// res0: Int = 3

Semigroup[List[Int]].combine(List(1,2,3), List(4,5,6))
// res1: List[Int] = List(1, 2, 3, 4, 5, 6)

Semigroup[Option[Int]].combine(Option(1), Option(2))
// res2: Option[Int] = Some(3)

Semigroup[Option[Int]].combine(Option(1), None)
// res3: Option[Int] = Some(1)

Semigroup[Int => Int].combine({(x: Int) => x + 1},{(x: Int) => x * 10}).apply(6)
// res4: Int = 67
```

Many of these types have methods defined directly on them, which allow for such combining, e.g. `++` on List, but the value of having a `Semigroup` type class available is that these compose, so for instance, we can say

```
Map("foo" -> Map("bar" -> 5)).combine(Map("foo" -> Map("bar" -> 6), "baz" -> Map()))
// res5: Map[String,scala.collection.immutable.Map[String,Int]] = Map(baz -> Map(), foo -> Map(bar -> 11))

Map("foo" -> List(1, 2)).combine(Map("foo" -> List(3,4), "bar" -> List(42)))
// res6: Map[String,List[Int]] = Map(foo -> List(1, 2, 3, 4), bar -> List(42))
```

which is far more likely to be useful than

```
Map("foo" -> Map("bar" -> 5)) ++  Map("foo" -> Map("bar" -> 6), "baz" -> Map())
// res7: scala.collection.immutable.Map[String,scala.collection.immutable.Map[_ <: String, Int]] = Map(foo -> Map(bar
 -> 6), baz -> Map())

Map("foo" -> List(1, 2)) ++ Map("foo" -> List(3,4), "bar" -> List(42))
// res8: scala.collection.immutable.Map[String,List[Int]] = Map(foo -> List(3, 4), bar -> List(42))
```

There is inline syntax available for `Semigroup`. Here we are following the convention from scalaz, that `|+|` is the operator from `Semigroup`.

```
import cats.implicits._

val one = Option(1)
val two = Option(2)
val n: Option[Int] = None
```

Thus.

```
one |+| two
// res10: Option[Int] = Some(3)

n |+| two
// res11: Option[Int] = Some(2)

n |+| n
// res12: Option[Int] = None

two |+| n
// res13: Option[Int] = Some(2)
```

You'll notice that instead of declaring `one` as `Some(1)`, I chose `Option(1)`, and I added an explicit type declaration for `n`. This is because there aren't type class instances for Some or None, but for Option. If we try to use Some and None, we'll get errors:

```
scala> Some(1) |+| None
<console>:22: error: value |+| is not a member of Some[Int]
       Some(1) |+| None
               ^

scala> None |+| Some(1)
<console>:22: error: value |+| is not a member of object None
       None |+| Some(1)
            ^
```

N.B. Cats does not define a `Semigroup` type class itself, it uses the `Semigroup` trait which is defined in the algebra project on which it depends. The `cats` package object defines type aliases to the `Semigroup` from algebra, so that you can `import cats.Semigroup`.

SemigroupK

Before introducing a `SemigroupK`, it makes sense to talk about what a `Semigroup` is. A semigroup for some given type `A` has a single operation (which we will call `combine`), which takes two values of type `A`, and returns a value of type `A`. This operation must be guaranteed to be associative. That is to say that:

```
((a combine b) combine c)
```

must be the same as

```
(a combine (b combine c))
```

for all possible values of `a`, `b`, `c`.

Cats does not define a `Semigroup` type class itself. Instead, we use the `Semigroup` trait which is defined in the algebra project. The `cats` package object defines type aliases to the `Semigroup` from algebra, so that you can `import cats.semigroup`.

There are instances of `Semigroup` defined for many types found in the scala common library:

```
import cats._
import cats.implicits._
```

Examples.

```
Semigroup[Int].combine(1, 2)
// res0: Int = 3

Semigroup[List[Int]].combine(List(1,2,3), List(4,5,6))
// res1: List[Int] = List(1, 2, 3, 4, 5, 6)

Semigroup[Option[Int]].combine(Option(1), Option(2))
// res2: Option[Int] = Some(3)

Semigroup[Option[Int]].combine(Option(1), None)
// res3: Option[Int] = Some(1)

Semigroup[Int => Int].combine({(x: Int) => x + 1},{(x: Int) => x * 10}).apply(6)
// res4: Int = 67
```

`SemigroupK` has a very similar structure to `Semigroup`, the difference is that `SemigroupK` operates on type constructors of one argument. So, for example, whereas you can find a `Semigroup` for types which are fully specified like `Int` or `List[Int]` or `Option[Int]`, you will find `SemigroupK` for type constructors like `List` and `Option`. These types are type constructors in that you can think of them as "functions" in the type space. You can think of the `List` type as a function which takes a concrete type, like `Int`, and returns a concrete type: `List[Int]`. This pattern would also be referred to having `kind: * -> *`, whereas `Int` would have kind `*` and `Map` would have kind `*,* -> *`, and, in fact, the `K` in `SemigroupK` stands for `Kind`.

For `List`, the `Semigroup` instance's `combine` operation and the `SemigroupK` instance's `combineK` operation are both list concatenation:

```
SemigroupK[List].combineK(List(1,2,3), List(4,5,6)) == Semigroup[List[Int]].combine(List(1,2,3), List(4,5,6))
// res5: Boolean = true
```

However for `Option`, the `Semigroup`'s `combine` and the `SemigroupK`'s `combineK` operation differ. Since `Semigroup` operates on fully specified types, a `Semigroup[Option[A]]` knows the concrete type of `A` and will use `Semigroup[A].combine` to combine the inner `A`s. Consequently, `Semigroup[Option[A]].combine` requires an implicit `Semigroup[A]`.

In contrast, since `SemigroupK[Option]` operates on `Option` where the inner type is not fully specified and can be anything (i.e. is "universally quantified"). Thus, we cannot know how to `combine` two of them. Therefore, in the case of `Option` the `SemigroupK[Option].combineK` method has no choice but to use the `orElse` method of Option:

```
Semigroup[Option[Int]].combine(Some(1), Some(2))
// res6: Option[Int] = Some(3)

SemigroupK[Option].combineK(Some(1), Some(2))
// res7: Option[Int] = Some(1)

SemigroupK[Option].combineK(Some(1), None)
// res8: Option[Int] = Some(1)

SemigroupK[Option].combineK(None, Some(2))
// res9: Option[Int] = Some(2)
```

There is inline syntax available for both `Semigroup` and `SemigroupK`. Here we are following the convention from scalaz, that `|+|` is the operator from semigroup and that `<+>` is the operator from `SemigroupK` (called `Plus` in scalaz).

```
import cats.implicits._

val one = Option(1)
val two = Option(2)
val n: Option[Int] = None
```

Thus.

```
one |+| two
// res11: Option[Int] = Some(3)

one <+> two
// res12: Option[Int] = Some(1)

n |+| two
// res13: Option[Int] = Some(2)

n <+> two
// res14: Option[Int] = Some(2)

n |+| n
// res15: Option[Int] = None

n <+> n
// res16: Option[Int] = None

two |+| n
// res17: Option[Int] = Some(2)

two <+> n
// res18: Option[Int] = Some(2)
```

You'll notice that instead of declaring `one` as `Some(1)`, we chose `Option(1)`, and we added an explicit type declaration for `n`. This is because the `SemigroupK` type class instances is defined for `Option`, not `Some` or `None`. If we try to use `Some` or `None`, we'll get errors:

```
scala> Some(1) <+> None
<console>:22: error: value <+> is not a member of Some[Int]
       Some(1) <+> None
               ^

scala> None <+> Some(1)
res20: Option[Int] = Some(1)
```

Traverse

In functional programming it is very common to encode "effects" as data types - common effects include `Option` for possibly missing values, `Xor` and `Validated` for possible errors, and `Future` for asynchronous computations.

These effects tend to show up in functions working on a single piece of data - for instance parsing a single `String` into an `Int`, validating a login, or asynchronously fetching website information for a user.

```
import cats.data.Xor
import scala.concurrent.Future

def parseInt(s: String): Option[Int] = ???

trait SecurityError
trait Credentials

def validateLogin(cred: Credentials): Xor[SecurityError, Unit] = ???

trait Profile
trait User

def userInfo(user: User): Future[Profile] = ???
```

Each function asks only for the data it actually needs; in the case of `userInfo`, a single `User`. We certainly could write one that takes a `List[User]` and fetch profile for all of them, would be a bit strange. If we just wanted to fetch the profile of just one user, we would either have to wrap it in a `List` or write a separate function that takes in a single user anyways. More fundamentally, functional programming is about building lots of small, independent pieces and composing them to make larger and larger pieces - does this hold true in this case?

Given just `User => Future[Profile]`, what should we do if we want to fetch profiles for a `List[User]`? We could try familiar combinators like `map`.

```
def profilesFor(users: List[User]) = users.map(userInfo)
// profilesFor: (users: List[User])List[scala.concurrent.Future[Profile]]
```

Note the return type `List[Future[Profile]]`. This makes sense given the type signatures, but seems unwieldy. We now have a list of asynchronous values, and to work with those values we must then use the combinators on `Future` for every single one. It would be nicer instead if we could get the aggregate result in a single `Future`, say a `Future[List[Profile]]`.

As it turns out, the `Future` companion object has a `traverse` method on it. However, that method is specialized to standard library collections and `Future`s - there exists a much more generalized form that would allow us to parse a `List[String]` or validate credentials for a `List[User]`.

Enter `Traverse`.

The type class

At center stage of `Traverse` is the `traverse` method.

```
trait Traverse[F[_]] {
  def traverse[G[_] : Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
}
```

In our above example, `F` is `List`, and `G` is `Option`, `Xor`, or `Future`. For the profile example, `traverse` says given a `List[User]` and a function `User => Future[Profile]`, it can give you a `Future[List[Profile]]`.

Abstracting away the `G` (still imagining `F` to be `List`), `traverse` says given a collection of data, and a function that takes a piece of data and returns an effectful value, it will traverse the collection, applying the function and aggregating the effectful values (in a `List`) as it goes.

In the most general form, `F[_]` is some sort of context which may contain a value (or several). While `List` tends to be among the most general cases, there also exist `Traverse` instances for `Option`, `Xor`, and `Validated` (among others).

Choose your effect

The type signature of `Traverse` appears highly abstract, and indeed it is - what `traverse` does as it walks the `F[A]` depends on the effect of the function. Let's see some examples where `F` is taken to be `List`.

Note in the following code snippet we are using `traverseU` instead of `traverse`. `traverseU` is for all intents and purposes the same as `traverse`, but with some type-level trickery to allow it to infer the `Applicative[Xor[A, ?]]` and `Applicative[Validated[A, ?]]` instances - `scalac` has issues inferring the instances for data types that do not trivially satisfy the `F[_]` shape required by `Applicative`.

```
import cats.Semigroup
import cats.data.{NonEmptyList, OneAnd, Validated, ValidatedNel, Xor}
import cats.implicits._

def parseIntXor(s: String): Xor[NumberFormatException, Int] =
  Xor.catchOnly[NumberFormatException](s.toInt)

def parseIntValidated(s: String): ValidatedNel[NumberFormatException, Int] =
  Validated.catchOnly[NumberFormatException](s.toInt).toValidatedNel
```

Examples.

```
val x1 = List("1", "2", "3").traverseU(parseIntXor)
// x1: cats.data.Xor[NumberFormatException,List[Int]] = Right(List(1, 2, 3))

val x2 = List("1", "abc", "3").traverseU(parseIntXor)
// x2: cats.data.Xor[NumberFormatException,List[Int]] = Left(java.lang.NumberFormatException: For input string: "abc")


val x3 = List("1", "abc", "def").traverseU(parseIntXor)
// x3: cats.data.Xor[NumberFormatException,List[Int]] = Left(java.lang.NumberFormatException: For input string: "abc")


val v1 = List("1", "2", "3").traverseU(parseIntValidated)
// v1: cats.data.Validated[cats.data.OneAnd[[+A]List[A],NumberFormatException],List[Int]] = Valid(List(1, 2, 3))

val v2 = List("1", "abc", "3").traverseU(parseIntValidated)
// v2: cats.data.Validated[cats.data.OneAnd[[+A]List[A],NumberFormatException],List[Int]] = Invalid(OneAnd(java.lang.
NumberFormatException: For input string: "abc",List()))

val v3 = List("1", "abc", "def").traverseU(parseIntValidated)
// v3: cats.data.Validated[cats.data.OneAnd[[+A]List[A],NumberFormatException],List[Int]] = Invalid(OneAnd(java.lang.
NumberFormatException: For input string: "abc",List(java.lang.NumberFormatException: For input string: "def")))
```

Notice that in the `Xor` case, should any string fail to parse the entire traversal is considered a failure. Moreover, once it hits its first bad parse, it will not attempt to parse any others down the line (similar behavior would be found with using `Option` as the effect). Contrast this with `Validated` where even if one bad parse is hit, it will continue trying to parse the others, accumulating any and all errors as it goes. The behavior of traversal is closely tied with the `Applicative` behavior of the data type.

Going back to our `Future` example, we can write an `Applicative` instance for `Future` that runs each `Future` concurrently. Then when we traverse a `List[A]` with an `A => Future[B]`, we can imagine the traversal as a scatter-gather. Each `A` creates a concurrent computation that will produce a `B` (the scatter), and as the `Future`s complete they

will be gathered back into a `List`.

Playing with `Reader`

Another interesting effect we can use is `Reader`. Recall that a `Reader[E, A]` is a type alias for `Kleisli[Id, E, A]` which is a wrapper around `E => A`.

If we fix `E` to be some sort of environment or configuration, we can use the `Reader` applicative in our traverse.

```
import cats.data.Reader

trait Context
trait Topic
trait Result

type Job[A] = Reader[Context, A]

def processTopic(topic: Topic): Job[Result] = ???
```

We can imagine we have a data pipeline that processes a bunch of data, each piece of data being categorized by a topic. Given a specific topic, we produce a `Job` that processes that topic. (Note that since a `Job` is just a `Reader` / `Kleisli`, one could write many small `Job`s and compose them together into one `Job` that is used/returned by `processTopic`.)

Corresponding to our bunches of data are bunches of topics, a `List[Topic]` if you will. Since `Reader` has an `Applicative` instance, we can `traverse` over this list with `processTopic`.

```
def processTopics(topics: List[Topic]) =
  topics.traverse(processTopic)
```

Note the nice return type - `Job[List[Result]]`. We now have one aggregate `Job` that when run, will go through each topic and run the topic-specific job, collecting results as it goes. We say "when run" because a `Job` is some function that requires a `Context` before producing the value we want.

One example of a "context" can be found in the Spark project. In Spark, information needed to run a Spark job (where the master node is, memory allocated, etc.) resides in a `SparkContext`. Going back to the above example, we can see how one may define topic-specific Spark jobs (`type Job[A] = Reader[SparkContext, A]`) and then run several Spark jobs on a collection of topics via `traverse`. We then get back a `Job[List[Result]]`, which is equivalent to `SparkContext => List[Result]`. When finally passed a `SparkContext`, we can run the job and get our results back.

Moreover, the fact that our aggregate job is not tied to any specific `SparkContext` allows us to pass in a `SparkContext` pointing to a production cluster, or (using the exact same job) pass in a test `SparkContext` that just runs locally across threads. This makes testing our large job nice and easy.

Finally, this encoding ensures that all the jobs for each topic run on the exact same cluster. At no point do we manually pass in or thread a `SparkContext` through - that is taken care for us by the (applicative) effect of `Reader` and therefore by `traverse`.

Sequencing

Sometimes you may find yourself with a collection of data, each of which is already in an effect, for instance a `List[Option[A]]`. To make this easier to work with, you want a `Option[List[A]]`. Given `Option` has an `Applicative` instance, we can traverse over the list with the identity function.

```
import cats.implicits._
// import cats.implicits._

val l1 = List(Option(1), Option(2), Option(3)).traverse(identity)
// l1: Option[List[Int]] = Some(List(1, 2, 3))

val l2 = List(Option(1), None, Option(3)).traverse(identity)
// l2: Option[List[Int]] = None
```

`Traverse` provides a convenience method `sequence` that does exactly this.

```
val l1 = List(Option(1), Option(2), Option(3)).sequence
// l1: Option[List[Int]] = Some(List(1, 2, 3))

val l2 = List(Option(1), None, Option(3)).sequence
// l2: Option[List[Int]] = None
```

Traversing for effect

Sometimes our effectful functions return a `Unit` value in cases where there is no interesting value to return (e.g. writing to some sort of store).

```
trait Data
def writeToStore(data: Data): Future[Unit] = ???
```

If we traverse using this, we end up with a funny type.

```
import cats.implicits._
import scala.concurrent.ExecutionContext.Implicits.global

def writeManyToStore(data: List[Data]) =
  data.traverse(writeToStore)
```

We end up with a `Future[List[Unit]]` ! A `List[Unit]` is not of any use to us, and communicates the same amount of information as a single `Unit` does.

Traversing solely for the sake of the effect (ignoring any values that may be produced, `Unit` or otherwise) is common, so `Foldable` (superclass of `Traverse`) provides `traverse_` and `sequence_` methods that do the same thing as `traverse` and `sequence` but ignores any value produced along the way, returning `Unit` at the end.

```
import cats.implicits._

def writeManyToStore(data: List[Data]) =
  data.traverse_(writeToStore)

// Int values are ignored with traverse_
def writeToStoreV2(data: Data): Future[Int] =
  ???

def writeManyToStoreV2(data: List[Data]) =
  data.traverse_(writeToStoreV2)
```

Const

At first glance `Const` seems like a strange data type - it has two type parameters, yet only stores a value of the first type. What possible use is it? As it turns out, it does have its uses, which serve as a nice example of the consistency and elegance of functional programming.

Thinking about `Const`

The `Const` data type can be thought of similarly to the `const` function, but as a data type.

```
def const[A, B](a: A)(b: => B): A = a
```

The `const` function takes two arguments and simply returns the first argument, ignoring the second.

```
final case class Const[A, B](getConst: A)
```

The `Const` data type takes two type parameters, but only ever stores a value of the first type parameter. Because the second type parameter is not used in the data type, the type parameter is referred to as a "phantom type".

Why do we care?

It would seem `Const` gives us no benefit over a data type that would simply not have the second type parameter. However, while we don't directly use the second type parameter, it's existence becomes useful in certain contexts.

Example 1: Lens

The following is heavily inspired by Julien Truffaut's blog post on Monocle, a fully-fledged optics library in Scala.

Types that contain other types are common across many programming paradigms. It is of course desirable in many cases to get out members of other types, or to set them. In traditional object-oriented programming this is handled by getter and setter methods on the outer object. In functional programming, a popular solution is to use a lens.

A lens can be thought of as a first class getter/setter. A `Lens[S, A]` is a data type that knows how to get an `A` out of an `S`, or set an `A` in an `S`.

```
trait Lens[S, A] {
  def get(s: S): A

  def set(s: S, a: A): S

  def modify(s: S)(f: A => A): S =
    set(s, f(get(s)))
}
```

It can be useful to have effectful modifications as well - perhaps our modification can fail (`Option`) or can return several values (`List`).

```
trait Lens[S, A] {
  def get(s: S): A

  def set(s: S, a: A): S

  def modify(s: S)(f: A => A): S =
    set(s, f(get(s)))

  def modifyOption(s: S)(f: A => Option[A]): Option[S] =
    f(get(s)).map(a => set(s, a))

  def modifyList(s: S)(f: A => List[A]): List[S] =
    f(get(s)).map(a => set(s, a))
}
```

Note that both `modifyOption` and `modifyList` share the *exact* same implementation. If we look closely, the only thing we need is a `map` operation on the data type. Being good functional programmers, we abstract.

```
import cats.Functor
import cats.implicits._

trait Lens[S, A] {
  def get(s: S): A

  def set(s: S, a: A): S

  def modify(s: S)(f: A => A): S =
    set(s, f(get(s)))

  def modifyF[F[_] : Functor](s: S)(f: A => F[A]): F[S] =
    f(get(s)).map(a => set(s, a))
}
```

We can redefine `modify` in terms of `modifyF` by using `cats.Id`. We can also treat `set` as a modification that simply ignores the current value. Due to these modifications however, we must leave `modifyF` abstract since having it defined in terms of `set` would lead to infinite circular calls.

```
import cats.Id

trait Lens[S, A] {
  def modifyF[F[_] : Functor](s: S)(f: A => F[A]): F[S]

  def set(s: S, a: A): S = modify(s)(_ => a)

  def modify(s: S)(f: A => A): S = modifyF[Id](s)(f)

  def get(s: S): A
}
```

What about `get`? Certainly we can't define `get` in terms of the others.. the others are to modify an existing value, whereas `get` is to retrieve it. Let's give it a shot anyways.

Looking at `modifyF`, we have an `S` we can pass in. The tricky part will be the `A => F[A]`, and then somehow getting an `A` out of `F[S]`. If we imagine `F` to be a type-level constant function however, we could imagine it would simply take any type and return some other constant type, an `A` perhaps. This suggests our `F` is a `Const`.

We then take a look at the fact that `modifyF` takes an `F[_]`, a type constructor that takes a single type parameter. `Const` takes two, so we must fix one. The function returns an `F[S]`, but we want an `A`, which implies we have the first type parameter fixed to `A` and leave the second one free for the function to fill in as it wants.

Substituting in `Const[A, _]` wherever we see `F[_]`, the function wants an `A => Const[A, A]` and will give us back a `Const[A, S]`. Looking at the definition of `Const`, we see that we only ever have a value of the first type parameter and completely ignore the second. Therefore, we can treat any `Const[X, Y]` value as equivalent to `X` (plus or minus some

wrapping into `Const`). This leaves us with needing a function `A => A`. Given the type, the only thing we can do is to take an `A` and return it right back (lifted into `Const`).

Before we plug and play however, note that `modifyF` has a `Functor` constraint on `F[_]`. This means we need to define a `Functor` instance for `Const`, where the first type parameter is fixed.

Note: the example below assumes usage of the kind-projector compiler plugin and will not compile if it is not being used in a project.

```
import cats.data.Const

implicit def constFunctor[X]: Functor[Const[X, ?]] =
  new Functor[Const[X, ?]] {
    // Recall Const[X, A] ~= X, so the function is not of any use to us
    def map[A, B](fa: Const[X, A])(f: A => B): Const[X, B] =
      Const(fa.getConst)
  }
```

Now that that's taken care of, let's substitute and see what happens.

```
trait Lens[S, A] {
  def modifyF[F[_] : Functor](s: S)(f: A => F[A]): F[S]

  def set(s: S, a: A): S = modify(s)(_ => a)

  def modify(s: S)(f: A => A): S = modifyF[Id](s)(f)

  def get(s: S): A = {
    val storedValue = modifyF[Const[A, ?]](s)(a => Const(a))
    storedValue.getConst
  }
}
```

It works! We get a `Const[A, S]` out on the other side, and we simply just retrieve the `A` value stored inside.

What's going on here? We can treat the effectful "modification" we are doing as a store operation - we take an `A` and store it inside a `Const`. Knowing only `F[_]` has a `Functor` instance, it can only `map` over the `Const` which will do nothing to the stored value. After `modifyF` is done getting the new `S`, we retrieve the stored `A` value and we're done!

Example 2: Traverse

In the popular The Essence of the Iterator Pattern paper, Jeremy Gibbons and Bruno C. d. S. Oliveria describe a functional approach to iterating over a collection of data. Among the abstractions presented are `Foldable` and `Traverse`, replicated below (also available in Cats).

```
import cats.{Applicative, Monoid}

trait Foldable[F[_]] {
  // Given a collection of data F[A], and a function mapping each A to a B where B has a Monoid instance,
  // reduce the collection down to a single B value using the monoidal behavior of B
  def foldMap[A, B : Monoid](fa: F[A])(f: A => B): B
}

trait Traverse[F[_]] {
  // Given a collection of data F[A], for each value apply the function f which returns an effectful
  // value. The result of traverse is the composition of all these effectful values.
  def traverse[G[_] : Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
}
```

These two type classes seem unrelated - one reduces a collection down to a single value, the other traverses a collection with an effectful function, collecting results. It may be surprising to see that in fact `Traverse` subsumes `Foldable`.

```
trait Traverse[F[_]] extends Foldable[F] {
  def traverse[G[_] : Applicative, A, X](fa: F[A])(f: A => G[X]): G[F[X]]

  def foldMap[A, B : Monoid](fa: F[A])(f: A => B): B
}
```

To start, we observe that if we are to implement `foldMap` in terms of `traverse`, we will want a `B` out at some point. However, `traverse` returns a `G[F[X]]`. It would seem there is no way to unify these two. However, if we imagine `G[_]` to be a sort of type-level constant function, where the fact that it's taking a `F[X]` is irrelevant to the true underlying value, we can begin to see how we might be able to pull this off.

`traverse` however wants `G[_]` to have an `Applicative` instance, so let's define one for `Const`. Since `F[X]` is the value we want to ignore, we treat it as the second type parameter and hence, leave it as the free one.

```
import cats.data.Const

implicit def constApplicative[Z]: Applicative[Const[Z, ?]] =
  new Applicative[Const[Z, ?]] {
    def pure[A](a: A): Const[Z, A] = ???

    def ap[A, B](f: Const[Z, A => B])(fa: Const[Z, A]): Const[Z, B] = ???
  }
```

Recall that `Const[Z, A]` means we have a `Z` value in hand, and don't really care about the `A` type parameter. Therefore we can more or less treat the type `Const[Z, A]` as just `Z`.

In functions `pure` and `ap` we have a problem. In `pure`, we have an `A` value, but want to return a `Z` value. We have no function `A => Z`, so our only option is to completely ignore the `A` value. But we still don't have a `Z`! Let's put that aside for now, but still keep it in the back of our minds.

In `ap` we have two `Z` values, and want to return a `Z` value. We could certainly return one or the other, but we should try to do something more useful. This suggests composition of `Z`s, which we don't know how to do.

So now we need a constant `Z` value, and a binary function that takes two `Z`s and produces a `Z`. Sound familiar? We want `Z` to have a `Monoid` instance!

```
implicit def constApplicative[Z : Monoid]: Applicative[Const[Z, ?]] =
  new Applicative[Const[Z, ?]] {
    def pure[A](a: A): Const[Z, A] = Const(Monoid[Z].empty)

    def ap[A, B](f: Const[Z, A => B])(fa: Const[Z, A]): Const[Z, B] =
      Const(Monoid[Z].combine(fa.getConst, f.getConst))
  }
```

We have our `Applicative`!

Going back to `Traverse`, we fill in the first parameter of `traverse` with `fa` since that's the only value that fits.

Now we need a `A => G[B]`. We have an `A => B`, and we've decided to use `Const` for our `G[_]`. We need to fix the first parameter of `Const` since `Const` takes two type parameters and `traverse` wants a type constructor which only takes one. The first type parameter which will be the type of the actual values we store, and therefore will be the type of the value we get out at the end, so we leave the second one free, similar to the `Applicative` instance. We don't care about the second type parameter and there are no restrictions on it, so we can just use `Nothing`, the type that has no values.

So to summarize, what we want is a function `A => Const[B, Nothing]`, and we have a function `A => B`. Recall that `Const[B, Z]` (for any `Z`) is the moral equivalent of just `B`, so `A => Const[B, Nothing]` is equivalent to `A => B`, which is exactly what we have, we just need to wrap it.

```
trait Traverse[F[_]] extends Foldable[F] {
  def traverse[G[_] : Applicative, A, X](fa: F[A])(f: A => G[X]): G[F[X]]

  def foldMap[A, B : Monoid](fa: F[A])(f: A => B): B = {
    val const: Const[B, F[Nothing]] = traverse[Const[B, ?], A, Nothing](fa)(a => Const(f(a)))
    const.getConst
  }
}
```

Hurrah!

What's happening here? We can see `traverse` is a function that goes over a collection, applying an effectful function to each value, and combining all of these effectful values. In our case, the effect is mapping each value to a value of type `B`, where we know how to combine `B`s via its `Monoid` instance. The `Monoid` instance is exactly what is used when `traverse` goes to collect the effectful values together. Should the `F[A]` be "empty", it can use `Monoid#empty` as a value to return back.

Pretty nifty. `traverse`-ing over a collection with an effectful function is more general than traversing over a collection to reduce it down to a single value.

Free Applicative

`FreeApplicative`s are similar to `Free` (monads) in that they provide a nice way to represent computations as data and are useful for building embedded DSLs (EDSLs). However, they differ from `Free` in that the kinds of operations they support are limited, much like the distinction between `Applicative` and `Monad`.

Dependency

If you'd like to use cats' free applicative, you'll need to add a library dependency for the `cats-free` module.

Example

Consider building an EDSL for validating strings - to keep things simple we'll just have a way to check a string is at least a certain size and to ensure the string contains numbers.

```
sealed abstract class ValidationOp[A]
case class Size(size: Int) extends ValidationOp[Boolean]
case object HasNumber extends ValidationOp[Boolean]
```

Much like the `Free` monad tutorial, we use smart constructors to lift our algebra into the `FreeApplicative`.

```
import cats.free.FreeApplicative
import cats.free.FreeApplicative.lift

type Validation[A] = FreeApplicative[ValidationOp, A]

def size(size: Int): Validation[Boolean] = lift(Size(size))

val hasNumber: Validation[Boolean] = lift(HasNumber)
```

Because a `FreeApplicative` only supports the operations of `Applicative`, we do not get the nicety of a for-comprehension. We can however still use `Applicative` syntax provided by Cats.

```
import cats.implicits._

val prog: Validation[Boolean] = (size(5) |@| hasNumber).map { case (l, r) => l && r}
```

As it stands, our program is just an instance of a data structure - nothing has happened at this point. To make our program useful we need to interpret it.

```
import cats.Id
import cats.arrow.FunctionK
import cats.implicits._

// a function that takes a string as input
type FromString[A] = String => A

val compiler =
  new FunctionK[ValidationOp, FromString] {
    def apply[A](fa: ValidationOp[A]): String => A =
      str =>
        fa match {
          case Size(size) => str.size >= size
          case HasNumber  => str.exists(c => "0123456789".contains(c))
        }
  }
```

```
val validator = prog.foldMap[FromString](compiler)
// validator: FromString[Boolean] = <function1>

validator("1234")
// res7: Boolean = false

validator("12345")
// res8: Boolean = true
```

Differences from `Free`

So far everything we've been doing has been not much different from `Free` - we've built an algebra and interpreted it. However, there are some things `FreeApplicative` can do that `Free` cannot.

Recall a key distinction between the type classes `Applicative` and `Monad` - `Applicative` captures the idea of independent computations, whereas `Monad` captures that of dependent computations. Put differently `Applicative`s cannot branch based on the value of an existing/prior computation. Therefore when using `Applicative`s, we must hand in all our data in one go.

In the context of `FreeApplicative`s, we can leverage this static knowledge in our interpreter.

Parallelism

Because we have everything we need up front and know there can be no branching, we can easily write a validator that validates in parallel.

```
import cats.data.Kleisli
import cats.implicits._
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

// recall Kleisli[Future, String, A] is the same as String => Future[A]
type ParValidator[A] = Kleisli[Future, String, A]

val parCompiler =
  new FunctionK[ValidationOp, ParValidator] {
    def apply[A](fa: ValidationOp[A]): ParValidator[A] =
      Kleisli { str =>
        fa match {
          case Size(size) => Future { str.size >= size }
          case HasNumber  => Future { str.exists(c => "0123456789".contains(c)) }
        }
      }
  }

val parValidation = prog.foldMap[ParValidator](parCompiler)
```

Logging

We can also write an interpreter that simply creates a list of strings indicating the filters that have been used - this could be useful for logging purposes. Note that we need not actually evaluate the rules against a string for this, we simply need to map each rule to some identifier. Therefore we can completely ignore the return type of the operation and return just a `List[String]` - the `Const` data type is useful for this.

```
import cats.data.Const
import cats.implicits._

type Log[A] = Const[List[String], A]

val logCompiler =
  new FunctionK[ValidationOp, Log] {
    def apply[A](fa: ValidationOp[A]): Log[A] =
      fa match {
        case Size(size) => Const(List(s"size >= $size"))
        case HasNumber  => Const(List("has number"))
      }
  }

def logValidation[A](validation: Validation[A]): List[String] =
  validation.foldMap[Log](logCompiler).getConst
```

```
logValidation(prog)
// res16: List[String] = List(size >= 5, has number)

logValidation(size(5) *> hasNumber *> size(10))
// res17: List[String] = List(size >= 5, has number, size >= 10)

logValidation((hasNumber |@| size(3)).map(_ || _))
// res18: List[String] = List(has number, size >= 3)
```

Why not both?

It is perhaps more plausible and useful to have both the actual validation function and the logging strings. While we could easily compile our program twice, once for each interpreter as we have above, we could also do it in one go - this would avoid multiple traversals of the same structure.

Another useful property `Applicative`s have over `Monad`s is that given two `Applicative`s `F[_]` and `G[_]`, their product `type FG[A] = (F[A], G[A])` is also an `Applicative`. This is not true in the general case for monads.

Therefore, we can write an interpreter that uses the product of the `ParValidator` and `Log` `Applicative`s to interpret our program in one go.

```
import cats.data.Prod

type ValidateAndLog[A] = Prod[ParValidator, Log, A]

val prodCompiler =
  new FunctionK[ValidationOp, ValidateAndLog] {
    def apply[A](fa: ValidationOp[A]): ValidateAndLog[A] = {
      fa match {
        case Size(size) =>
          val f: ParValidator[Boolean] = Kleisli(str => Future { str.size >= size })
          val l: Log[Boolean] = Const(List(s"size > $size"))
          Prod[ParValidator, Log, Boolean](f, l)
        case HasNumber  =>
          val f: ParValidator[Boolean] = Kleisli(str => Future(str.exists(c => "0123456789".contains(c))))
          val l: Log[Boolean] = Const(List("has number"))
          Prod[ParValidator, Log, Boolean](f, l)
      }
    }
  }

val prodValidation = prog.foldMap[ValidateAndLog](prodCompiler)
```

References

Deeper explanations can be found in this paper Free Applicative Functors by Paolo Capriotti

Free Monad

What is it?

A *free monad* is a construction which allows you to build a *monad* from any *Functor*. Like other *monads*, it is a pure way to represent and manipulate computations.

In particular, *free monads* provide a practical way to:

- represent stateful computations as data, and run them
- run recursive computations in a stack-safe way
- build an embedded DSL (domain-specific language)
- retarget a computation to another interpreter using natural transformations

> (In cats, the type representing a *free monad* is abbreviated as `Free[_]`.)

Using Free Monads

If you'd like to use cats' free monad, you'll need to add a library dependency for the `cats-free` module.

A good way to get a sense for how *free monads* work is to see them in action. The next section uses `Free[_]` to create an embedded DSL (Domain Specific Language).

If you're interested in the theory behind *free monads*, the What is Free in theory? section discusses free monads in terms of category theory.

Study your topic

Let's imagine that we want to create a DSL for a key-value store. We want to be able to do three things with keys:

- *put* a `value` into the store, associated with its `key`.
- *get* a `value` from the store given its `key`.
- *delete* a `value` from the store given its `key`.

The idea is to write a sequence of these operations in the embedded DSL as a "program", compile the "program", and finally execute the "program" to interact with the actual key-value store.

For example:

```
put("toto", 3)
get("toto") // returns 3
delete("toto")
```

But we want:

- the computation to be represented as a pure, immutable value
- to separate the creation and execution of the program
- to be able to support many different methods of execution

Study your grammar

We have 3 commands to interact with our KeyValue store:

- `Put` a value associated with a key into the store
- `Get` a value associated with a key out of the store

- `Delete` a value associated with a key from the store

Create an ADT representing your grammar

ADT stands for *Algebraic Data Type*. In this context, it refers to a closed set of types which can be combined to build up complex, recursive values.

We need to create an ADT to represent our key-value operations:

```
sealed trait KVStoreA[A]
case class Put[T](key: String, value: T) extends KVStoreA[Unit]
case class Get[T](key: String) extends KVStoreA[Option[T]]
case class Delete(key: String) extends KVStoreA[Unit]
```

Free your ADT

There are six basic steps to "freeing" the ADT:

1. Create a type based on `Free[_]` and `KVStoreA[_]`.
2. Create smart constructors for `KVStore[_]` using `liftF`.
3. Build a program out of key-value DSL operations.
4. Build a compiler for programs of DSL operations.
5. Execute our compiled program.

1. Create a `Free` type based on your ADT

```
import cats.free.Free

type KVStore[A] = Free[KVStoreA, A]
```

2. Create smart constructors using `liftF`

These methods will make working with our DSL a lot nicer, and will lift `KVStoreA[_]` values into our `KVStore[_]` monad (note the missing "A" in the second type).

```
import cats.free.Free.liftF

// Put returns nothing (i.e. Unit).
def put[T](key: String, value: T): KVStore[Unit] =
  liftF[KVStoreA, Unit](Put[T](key, value))

// Get returns a T value.
def get[T](key: String): KVStore[Option[T]] =
  liftF[KVStoreA, Option[T]](Get[T](key))

// Delete returns nothing (i.e. Unit).
def delete(key: String): KVStore[Unit] =
  liftF(Delete(key))

// Update composes get and set, and returns nothing.
def update[T](key: String, f: T => T): KVStore[Unit] =
  for {
    vMaybe <- get[T](key)
    _ <- vMaybe.map(v => put[T](key, f(v))).getOrElse(Free.pure(()))
  } yield ()
```

3. Build a program

Now that we can construct `KVStore[_]` values we can use our DSL to write "programs" using a *for-comprehension*:

```
def program: KVStore[Option[Int]] =
  for {
    _ <- put("wild-cats", 2)
    _ <- update[Int]("wild-cats", (_ + 12))
    _ <- put("tame-cats", 5)
    n <- get[Int]("wild-cats")
    _ <- delete("tame-cats")
  } yield n
```

This looks like a monadic flow. However, it just builds a recursive data structure representing the sequence of operations.

4. Write a compiler for your program

As you may have understood now, `Free[_]` is used to create an embedded DSL. By itself, this DSL only represents a sequence of operations (defined by a recursive data structure); it doesn't produce anything.

`Free[_]` is a programming language inside your programming language!

So, like any other programming language, we need to compile our abstract language into an *effective* language and then run it.

To do this, we will use a *natural transformation* between type containers. Natural transformations go between types like `F[_]` and `G[_]` (this particular transformation would be written as `FunctionK[F,G]` or as done here using the symbolic alternative as `F ~> G`).

In our case, we will use a simple mutable map to represent our key value store:

```
import cats.arrow.FunctionK
import cats.{Id, ~>}
import scala.collection.mutable

// the program will crash if a key is not found,
// or if a type is incorrectly specified.
def impureCompiler: KVStoreA ~> Id  =
  new (KVStoreA ~> Id) {

    // a very simple (and imprecise) key-value store
    val kvs = mutable.Map.empty[String, Any]

    def apply[A](fa: KVStoreA[A]): Id[A] =
      fa match {
        case Put(key, value) =>
          println(s"put($key, $value)")
          kvs(key) = value
          ()
        case Get(key) =>
          println(s"get($key)")
          kvs.get(key).map(_.asInstanceOf[A])
        case Delete(key) =>
          println(s"delete($key)")
          kvs.remove(key)
          ()
      }
  }
```

Please note this `impureCompiler` is impure -- it mutates `kvs` and also produces logging output using `println`. The whole purpose of functional programming isn't to prevent side-effects, it is just to push side-effects to the boundaries of your system in a well-known and controlled way.

`Id[_]` represents the simplest type container: the type itself. Thus, `Id[Int]` is just `Int`. This means that our program will execute immediately, and block until the final value can be returned.

However, we could easily use other type containers for different behavior, such as:

- `Future[_]` for asynchronous computation

- `List[_]` for gathering multiple results
- `Option[_]` to support optional results
- `Xor[E, ?]` to support failure
- a pseudo-random monad to support non-determinism
- and so on...

5. Run your program

The final step is naturally running your program after compiling it.

`Free[_]` is just a recursive structure that can be seen as sequence of operations producing other operations. In this way it is similar to `List[_]`. We often use folds (e.g. `foldRight`) to obtain a single value from a list; this recurses over the structure, combining its contents.

The idea behind running a `Free[_]` is exactly the same. We fold the recursive structure by:

- consuming each operation.
- compiling the operation into our effective language using `impureCompiler` (applying its effects if any).
- computing next operation.
- continue recursively until reaching a `Pure` state, and returning it.

This operation is called `Free.foldMap`:

```
final def foldMap[M[_]](f: FunctionK[S,M])(M: Monad[M]): M[A] = ...
```

`M` must be a `Monad` to be flattenable (the famous monoid aspect under `Monad`). As `Id` is a `Monad`, we can use `foldMap`.

To run your `Free` with previous `impureCompiler`:

```
val result: Option[Int] = program.foldMap(impureCompiler)
// put(wild-cats, 2)
// get(wild-cats)
// put(wild-cats, 14)
// put(tame-cats, 5)
// get(wild-cats)
// delete(tame-cats)
// result: Option[Int] = Some(14)
```

An important aspect of `foldMap` is its **stack-safety**. It evaluates each step of computation on the stack then unstack and restart. This process is known as trampolining.

As long as your natural transformation is stack-safe, `foldMap` will never overflow your stack. Trampolining is heap-intensive but stack-safety provides the reliability required to use `Free[_]` for data-intensive tasks, as well as infinite processes such as streams.

7. Use a pure compiler (optional)

The previous examples used a effectful natural transformation. This works, but you might prefer folding your `Free` in a "purer" way. The State data structure can be used to keep track of the program state in an immutable map, avoiding mutation altogether.

```
import cats.data.State

type KVStoreState[A] = State[Map[String, Any], A]
val pureCompiler: KVStoreA ~> KVStoreState = new (KVStoreA ~> KVStoreState) {
  def apply[A](fa: KVStoreA[A]): KVStoreState[A] =
    fa match {
      case Put(key, value) => State.modify(_.updated(key, value))
      case Get(key) =>
        State.inspect(_.get(key).map(_.asInstanceOf[A]))
      case Delete(key) => State.modify(_ - key)
    }
}
```

(You can see that we are again running into some places where Scala's support for pattern matching is limited by the JVM's type erasure, but it's not too hard to get around.)

```
val result: (Map[String, Any], Option[Int]) = program.foldMap(pureCompiler).run(Map.empty).value
// result: (Map[String,Any], Option[Int]) = (Map(wild-cats -> 14),Some(14))
```

Composing Free monads ADTs.

Real world applications often time combine different algebras. The `Inject` type class described by Swierstra in Data types à la carte lets us compose different algebras in the context of `Free`.

Let's see a trivial example of unrelated ADT's getting composed as a `Coproduct` that can form a more complex program.

```
import cats.data.{Xor, Coproduct}
import cats.free.{Inject, Free}
import cats.{Id, ~>}
import scala.collection.mutable.ListBuffer
```

```
/* Handles user interaction */
sealed trait Interact[A]
case class Ask(prompt: String) extends Interact[String]
case class Tell(msg: String) extends Interact[Unit]

/* Represents persistence operations */
sealed trait DataOp[A]
case class AddCat(a: String) extends DataOp[Unit]
case class GetAllCats() extends DataOp[List[String]]
```

Once the ADTs are defined we can formally state that a `Free` program is the Coproduct of it's Algebras.

```
type CatsApp[A] = Coproduct[DataOp, Interact, A]
```

In order to take advantage of monadic composition we use smart constructors to lift our Algebra to the `Free` context.

```
class Interacts[F[_]](implicit I: Inject[Interact, F]) {
  def tell(msg: String): Free[F, Unit] = Free.inject[Interact, F](Tell(msg))
  def ask(prompt: String): Free[F, String] = Free.inject[Interact, F](Ask(prompt))
}

object Interacts {
  implicit def interacts[F[_]](implicit I: Inject[Interact, F]): Interacts[F] = new Interacts[F]
}

class DataSource[F[_]](implicit I: Inject[DataOp, F]) {
  def addCat(a: String): Free[F, Unit] = Free.inject[DataOp, F](AddCat(a))
  def getAllCats: Free[F, List[String]] = Free.inject[DataOp, F](GetAllCats())
}

object DataSource {
  implicit def dataSource[F[_]](implicit I: Inject[DataOp, F]): DataSource[F] = new DataSource[F]
}
```

ADTs are now easily composed and trivially intertwined inside monadic contexts.

```
def program(implicit I : Interacts[CatsApp], D : DataSource[CatsApp]): Free[CatsApp, Unit] = {

  import I._, D._

  for {
    cat <- ask("What's the kitty's name?")
    _ <- addCat(cat)
    cats <- getAllCats
    _ <- tell(cats.toString)
  } yield ()
}
```

Finally we write one interpreter per ADT and combine them with a `FunctionK` to `Coproduct` so they can be compiled and applied to our `Free` program.

```
object ConsoleCatsInterpreter extends (Interact ~> Id) {
  def apply[A](i: Interact[A]) = i match {
    case Ask(prompt) =>
      println(prompt)
      readLine()
    case Tell(msg) =>
      println(msg)
  }
}

object InMemoryDatasourceInterpreter extends (DataOp ~> Id) {

  private[this] val memDataSet = new ListBuffer[String]

  def apply[A](fa: DataOp[A]) = fa match {
    case AddCat(a) => memDataSet.append(a); ()
    case GetAllCats() => memDataSet.toList
  }
}

val interpreter: CatsApp ~> Id = InMemoryDatasourceInterpreter or ConsoleCatsInterpreter
```

Now if we run our program and type in "snuggles" when prompted, we see something like this:

```
import DataSource._, Interacts._
```

```
val evaled: Unit = program.foldMap(interpreter)
// What's the kitty's name?
// List(snuggles)
// evaled: Unit = ()
```

For the curious ones: what is Free in theory?

Mathematically-speaking, a *free monad* (at least in the programming language context) is a construction that is left adjoint to a forgetful functor whose domain is the category of Monads and whose co-domain is the category of Endofunctors. Huh?

Concretely, **it is just a clever construction that allows us to build a *very simple* Monad from any *functor***.

The above forgetful functor takes a `Monad` and:

- forgets its *monadic* part (e.g. the `flatMap` function)
- forgets its *pointed* part (e.g. the `pure` function)
- finally keeps the *functor* part (e.g. the `map` function)

By reversing all arrows to build the left-adjoint, we deduce that the free monad is basically a construction that:

- takes a *functor*
- adds the *pointed* part (e.g. `pure`)
- adds the *monadic* behavior (e.g. `flatMap`)

In terms of implementation, to build a *monad* from a *functor* we use the following classic inductive definition:

```
sealed abstract class Free[F[_], A]
case class Pure[F[_], A](a: A) extends Free[F, A]
case class Suspend[F[_], A](a: F[Free[F, A]]) extends Free[F, A]
```

(*This generalizes the concept of fixed point functor*.)

In this representation:

- `Pure` builds a `Free` instance from an `A` value (it *reifies* the `pure` function)
- `Suspend` build a new `Free` by applying `F` to a previous `Free` (it *reifies* the `flatMap` function)

So a typical `Free` structure might look like:

```
Suspend(F(Suspend(F(Suspend(F(....(Pure(a))))))))
```

`Free` is a recursive structure. It uses `A` in `F[A]` as the recursion "carrier" with a terminal element `Pure` .

From a computational point of view, `Free` recursive structure can be seen as a sequence of operations.

- `Pure` returns an `A` value and ends the entire computation.
- `Suspend` is a continuation; it suspends the current computation with the suspension functor `F` (which can represent a command for example) and hands control to the caller. `A` represents a value bound to this computation.

Please note this `Free` construction has the interesting quality of *encoding* the recursion on the heap instead of the stack as classic function calls would. This provides the stack-safety we heard about earlier, allowing very large `Free` structures to be evaluated safely.

For the very curious ones

If you look at implementation in cats, you will see another member of the `Free[_]` ADT:

```
sealed abstract case class Gosub[S[_], B]() extends Free[S, B] {
  type C
  val a: () => Free[S, C]
  val f: C => Free[S, B]
}
```

`Gosub` represents a call to a subroutine `a` and when `a` is finished, it continues the computation by calling the function `f` with the result of `a`.

It is actually an optimization of `Free` structure allowing to solve a problem of quadratic complexity implied by very deep recursive `Free` computations.

It is exactly the same problem as repeatedly appending to a `List[_]`. As the sequence of operations becomes longer, the slower a `flatMap` "through" the structure will be. With `Gosub`, `Free` becomes a right-associated structure not subject to quadratic complexity.

Future Work (TODO)

There are many remarkable uses of `Free[_]`. In the future, we will include some here, such as:

- Trampoline
- Option
- Iteratee
- Source
- etc...

We will also discuss the *Coyoneda Trick*.

Credits

This article was written by Pascal Voitot and edited by other members of the Cats community.

Kleisli

Kleisli enables composition of functions that return a monadic value, for instance an `Option[Int]` or a `Xor[String, List[Double]]`, without having functions take an `Option` or `Xor` as a parameter, which can be strange and unwieldy.

We may also have several functions which depend on some environment and want a nice way to compose these functions to ensure they all receive the same environment. Or perhaps we have functions which depend on their own "local" configuration and all the configurations together make up a "global" application configuration. How do we have these functions play nice with each other despite each only knowing about their own local requirements?

These situations are where `Kleisli` is immensely helpful.

Functions

One of the most useful properties of functions is that they **compose**. That is, given a function `A => B` and a function `B => C`, we can combine them to create a new function `A => C`. It is through this compositional property that we are able to write many small functions and compose them together to create a larger one that suits our needs.

```
val twice: Int => Int =
  x => x * 2

val countCats: Int => String =
  x => if (x == 1) "1 cat" else s"$x cats"

val twiceAsManyCats: Int => String =
  twice andThen countCats // equivalent to: countCats compose twice
```

Thus.

```
twiceAsManyCats(1) // "2 cats"
// res2: String = 2 cats
```

Sometimes, our functions will need to return monadic values. For instance, consider the following set of functions.

```
val parse: String => Option[Int] =
  s => if (s.matches("-?[0-9]+")) Some(s.toInt) else None

val reciprocal: Int => Option[Double] =
  i => if (i != 0) Some(1.0 / i) else None
```

As it stands we cannot use `Function1.compose` (or `Function1.andThen`) to compose these two functions. The output type of `parse` is `Option[Int]` whereas the input type of `reciprocal` is `Int`.

This is where `Kleisli` comes into play.

Kleisli

At it's core, `Kleisli[F[_], A, B]` is just a wrapper around the function `A => F[B]`. Depending on the properties of the `F[_]`, we can do different things with `Kleisli`s. For instance, if `F[_]` has a `FlatMap[F]` instance (we can call `flatMap` on `F[A]` values), we can compose two `Kleisli`s much like we can two functions.

```
import cats.FlatMap
import cats.implicits._

final case class Kleisli[F[_], A, B](run: A => F[B]) {
  def compose[Z](k: Kleisli[F, Z, A])(implicit F: FlatMap[F]): Kleisli[F, Z, B] =
    Kleisli[F, Z, B](z => k.run(z).flatMap(run))
}
```

Returning to our earlier example:

```
// Bring in cats.FlatMap[Option] instance
import cats.implicits._

val parse = Kleisli((s: String) => try { Some(s.toInt) } catch { case _: NumberFormatException => None })

val reciprocal = Kleisli((i: Int) => if (i == 0) None else Some(1.0 / i))

val parseAndReciprocal = reciprocal.compose(parse)
```

`Kleisli#andThen` can be defined similarly.

It is important to note that the `F[_]` having a `FlatMap` (or a `Monad`) instance is not a hard requirement - we can do useful things with weaker requirements. Such an example would be `Kleisli#map`, which only requires that `F[_]` have a `Functor` instance (e.g. is equipped with `map: F[A] => (A => B) => F[B]`).

```
import cats.Functor

final case class Kleisli[F[_], A, B](run: A => F[B]) {
  def map[C](f: B => C)(implicit F: Functor[F]): Kleisli[F, A, C] =
    Kleisli[F, A, C](a => F.map(run(a))(f))
}
```

Below are some more methods on `Kleisli` that can be used so long as the constraint on `F[_]` is satisfied.

```
Method    | Constraint on `F[_]`
--------- | -------------------
andThen   | FlatMap
compose   | FlatMap
flatMap   | FlatMap
lower     | Monad
map       | Functor
traverse  | Applicative
```

Type class instances

The type class instances for `Kleisli`, like that for functions, often fix the input type (and the `F[_]`) and leave the output type free. What type class instances it has tends to depend on what instances the `F[_]` has. For instance, `Kleisli[F, A, B]` has a `Functor` instance so long as the chosen `F[_]` does. It has a `Monad` instance so long as the chosen `F[_]` does. The instances in Cats are laid out in a way such that implicit resolution will pick up the most specific instance it can (depending on the `F[_]`).

An example of a `Monad` instance for `Kleisli` is shown below.

Note: the example below assumes usage of the kind-projector compiler plugin and will not compile if it is not being used in a project.

```
import cats.implicits._

// We can define a FlatMap instance for Kleisli if the F[_] we chose has a FlatMap instance
// Note the input type and F are fixed, with the output type left free
implicit def kleisliFlatMap[F[_], Z](implicit F: FlatMap[F]): FlatMap[Kleisli[F, Z, ?]] =
  new FlatMap[Kleisli[F, Z, ?]] {
    def flatMap[A, B](fa: Kleisli[F, Z, A])(f: A => Kleisli[F, Z, B]): Kleisli[F, Z, B] =
      Kleisli(z => fa.run(z).flatMap(a => f(a).run(z)))

    def map[A, B](fa: Kleisli[F, Z, A])(f: A => B): Kleisli[F, Z, B] =
      Kleisli(z => fa.run(z).map(f))
  }
```

Below is a table of some of the type class instances `Kleisli` can have depending on what instances `F[_]` has.

```
Type class     | Constraint on `F[_]`
-------------- | -------------------
Functor        | Functor
Apply          | Apply
Applicative    | Applicative
FlatMap        | FlatMap
Monad          | Monad
Arrow          | Monad
Split          | FlatMap
Strong         | Functor
SemigroupK*    | FlatMap
MonoidK*       | Monad
```

*These instances only exist for Kleisli arrows with identical input and output types; that is, `Kleisli[F, A, A]` for some type A. These instances use Kleisli composition as the `combine` operation, and `Monad.pure` as the `empty` value.

Also, there is an instance of `Monoid[Kleisli[F, A, B]]` if there is an instance of `Monoid[F[B]]`. `Monoid.combine` here creates a new Kleisli arrow which takes an `A` value and feeds it into each of the combined Kleisli arrows, which together return two `F[B]` values. Then, they are combined into one using the `Monoid[F[B]]` instance.

Other uses

Monad Transformers

Many data types have a monad transformer equivalent that allows us to compose the `Monad` instance of the data type with any other `Monad` instance. For instance, `OptionT[F[_], A]` allows us to compose the monadic properties of `Option` with any other `F[_]`, such as a `List`. This allows us to work with nested contexts/effects in a nice way (for example, in for-comprehensions).

`Kleisli` can be viewed as the monad transformer for functions. Recall that at its essence, `Kleisli[F, A, B]` is just a function `A => F[B]`, with niceties to make working with the value we actually care about, the `B`, easy. `Kleisli` allows us to take the effects of functions and have them play nice with the effects of any other `F[_]`.

This may raise the question, what exactly is the "effect" of a function?

Well, if we take a look at any function, we can see it takes some input and produces some output with it, without having touched the input (assuming the function is pure, i.e. referentially transparent). That is, we take a read-only value, and produce some value with it. For this reason, the type class instances for functions often refer to the function as a `Reader`. For instance, it is common to hear about the `Reader` monad. In the same spirit, Cats defines a `Reader` type alias along the lines of:

```
// We want A => B, but Kleisli provides A => F[B]. To make the types/shapes match,
// we need an F[_] such that providing it a type A is equivalent to A
// This can be thought of as the type-level equivalent of the identity function
type Id[A] = A

type Reader[A, B] = Kleisli[Id, A, B]
object Reader {
  // Lifts a plain function A => B into a Kleisli, giving us access
  // to all the useful methods and type class instances
  def apply[A, B](f: A => B): Reader[A, B] = Kleisli[Id, A, B](f)
}

type ReaderT[F[_], A, B] = Kleisli[F, A, B]
val ReaderT = Kleisli
```

The `ReaderT` value alias exists to allow users to use the `Kleisli` companion object as if it were `ReaderT`, if they were so inclined.

The topic of functions as a read-only environment brings us to our next common use case of `Kleisli` - configuration.

Configuration

Functional programming advocates the creation of programs and modules by composing smaller, simpler modules. This philosophy intentionally mirrors that of function composition - write many small functions, and compose them to build larger ones. After all, our programs are just functions.

Let's look at some example modules, where each module has it's own configuration that is validated by a function. If the configuration is good, we return a `Some` of the module, otherwise a `None`. This example uses `Option` for simplicity - if you want to provide error messages or other failure context, consider using `Xor` instead.

```
case class DbConfig(url: String, user: String, pass: String)
trait Db
object Db {
  val fromDbConfig: Kleisli[Option, DbConfig, Db] = ???
}

case class ServiceConfig(addr: String, port: Int)
trait Service
object Service {
  val fromServiceConfig: Kleisli[Option, ServiceConfig, Service] = ???
}
```

We have two independent modules, a `Db` (allowing access to a database) and a `Service` (supporting an API to provide data over the web). Both depend on their own configuration parameters. Neither know or care about the other, as it should be. However our application needs both of these modules to work. It is plausible we then have a more global application configuration.

```
case class AppConfig(dbConfig: DbConfig, serviceConfig: ServiceConfig)

class App(db: Db, service: Service)
```

As it stands, we cannot use both `Kleisli` validation functions together nicely - one takes a `DbConfig`, the other a `ServiceConfig`. That means the `FlatMap` (and by extension, the `Monad`) instances differ (recall the input type is fixed in the type class instances). However, there is a nice function on `Kleisli` called `local`.

```
final case class Kleisli[F[_], A, B](run: A => F[B]) {
  def local[AA](f: AA => A): Kleisli[F, AA, B] = Kleisli(f.andThen(run))
}
```

What `local` allows us to do is essentially "expand" our input type to a more "general" one. In our case, we can take a `Kleisli` that expects a `DbConfig` or `ServiceConfig` and turn it into one that expects an `AppConfig`, so long as we tell it how to go from an `AppConfig` to the other configs.

Now we can create our application config validator!

```
final case class Kleisli[F[_], Z, A](run: Z => F[A]) {
  def flatMap[B](f: A => Kleisli[F, Z, B])(implicit F: FlatMap[F]): Kleisli[F, Z, B] =
    Kleisli(z => F.flatMap(run(z))(a => f(a).run(z)))

  def map[B](f: A => B)(implicit F: Functor[F]): Kleisli[F, Z, B] =
    Kleisli(z => F.map(run(z))(f))

  def local[ZZ](f: ZZ => Z): Kleisli[F, ZZ, A] = Kleisli(f.andThen(run))
}

case class DbConfig(url: String, user: String, pass: String)
trait Db
object Db {
  val fromDbConfig: Kleisli[Option, DbConfig, Db] = ???
}

case class ServiceConfig(addr: String, port: Int)
trait Service
object Service {
  val fromServiceConfig: Kleisli[Option, ServiceConfig, Service] = ???
}

case class AppConfig(dbConfig: DbConfig, serviceConfig: ServiceConfig)

class App(db: Db, service: Service)

def appFromAppConfig: Kleisli[Option, AppConfig, App] =
  for {
    db <- Db.fromDbConfig.local[AppConfig](_.dbConfig)
    sv <- Service.fromServiceConfig.local[AppConfig](_.serviceConfig)
  } yield new App(db, sv)
```

What if we need a module that doesn't need any config validation, say a strategy to log events? We would have such a module be instantiated from a config directly, without an `Option` - we would have something like `Kleisli[Id, LogConfig, Log]` (alternatively, `Reader[LogConfig, Log]`). However, this won't play nice with our other `Kleisli`s since those use `Option` instead of `Id`.

We can define a `lift` method on `Kleisli` (available already on `Kleisli` in Cats) that takes a type parameter `G[_]` such that `G` has an `Applicative` instance and lifts a `Kleisli` value such that its output type is `G[F[B]]`. This allows us to then lift a `Reader[A, B]` into a `Kleisli[G, A, B]`. Note that lifting a `Reader[A, B]` into some `G[_]` is equivalent to having a `Kleisli[G, A, B]` since `Reader[A, B]` is just a type alias for `Kleisli[Id, A, B]`, and `type Id[A] = A` so `G[Id[A]]` is equivalent to `G[A]`.

OneAnd

The `OneAnd[F[_],A]` data type represents a single element of type `A` that is guaranteed to be present (`head`) and in addition to this a second part that is wrapped inside an higher kinded type constructor `F[_]` . By choosing the `F` parameter, you can model for example non-empty lists by choosing `List` for `F` , giving:

```
import cats.data.OneAnd

type NonEmptyList[A] = OneAnd[List, A]
```

which is the actual implementation of non-empty lists in cats. By having the higher kinded type parameter `F[_]` , `OneAnd` is also able to represent other "non-empty" data structures, e.g.

```
import cats.data.OneAnd

type NonEmptyVector[A] = OneAnd[Vector, A]
```

OptionT

`OptionT[F[_], A]` is a light wrapper on an `F[Option[A]]`. Speaking technically, it is a monad transformer for `Option`, but you don't need to know what that means for it to be useful. `OptionT` can be more convenient to work with than using `F[Option[A]]` directly.

Reduce map boilerplate

Consider the following scenario:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

val customGreeting: Future[Option[String]] = Future.successful(Some("welcome back, Lola"))
```

We want to try out various forms of our greetings.

```
val excitedGreeting: Future[Option[String]] = customGreeting.map(_.map(_ + "!"))

val hasWelcome: Future[Option[String]] = customGreeting.map(_.filter(_.contains("welcome")))

val noWelcome: Future[Option[String]] = customGreeting.map(_.filterNot(_.contains("welcome")))

val withFallback: Future[String] = customGreeting.map(_.getOrElse("hello, there!"))
```

As you can see, the implementations of all of these variations are very similar. We want to call the `Option` operation (`map`, `filter`, `filterNot`, `getOrElse`), but since our `Option` is wrapped in a `Future`, we first need to `map` over the `Future`.

`OptionT` can help remove some of this boilerplate. It exposes methods that look like those on `Option`, but it handles the outer `map` call on the `Future` so we don't have to:

```
import cats.data.OptionT
import cats.implicits._

val customGreetingT: OptionT[Future, String] = OptionT(customGreeting)

val excitedGreeting: OptionT[Future, String] = customGreetingT.map(_ + "!")

val withWelcome: OptionT[Future, String] = customGreetingT.filter(_.contains("welcome"))

val noWelcome: OptionT[Future, String] = customGreetingT.filterNot(_.contains("welcome"))

val withFallback: Future[String] = customGreetingT.getOrElse("hello, there!")
```

From `Option[A]` and/or `F[A]` to `OptionT[F, A]`

Sometimes you may have an `Option[A]` and/or `F[A]` and want to *lift* them into an `OptionT[F, A]`. For this purpose `OptionT` exposes two useful methods, namely `fromOption` and `liftF`, respectively. E.g.:

```
val greetingFO: Future[Option[String]] = Future.successful(Some("Hello"))

val firstnameF: Future[String] = Future.successful("Jane")

val lastnameO: Option[String] = Some("Doe")

val ot: OptionT[Future, String] = for {
  g <- OptionT(greetingFO)
  f <- OptionT.liftF(firstnameF)
  l <- OptionT.fromOption[Future](lastnameO)
} yield s"$g $f $l"

val result: Future[Option[String]] = ot.value // Future(Some("Hello Jane Doe"))
```

From `A` to `OptionT[F,A]`

If you have only an `A` and you wish to *lift* it into an `OptionT[F,A]` assuming you have an `Applicative` instance for `F` you can use `some` which is an alias for `pure`. There also exists a `none` method which can be used to create an `OptionT[F,A]`, where the `Option` wrapped `A` type is actually a `None`:

```
val greet: OptionT[Future,String] = OptionT.pure("Hola!")

val greetAlt: OptionT[Future,String] = OptionT.some("Hi!")

val failedGreet: OptionT[Future,String] = OptionT.none
```

Beyond map

Sometimes the operation you want to perform on an `Future[Option[String]]` might not be as simple as just wrapping the `Option` method in a `Future.map` call. For example, what if we want to greet the customer with their custom greeting if it exists but otherwise fall back to a default `Future[String]` greeting? Without `OptionT`, this implementation might look like:

```
val defaultGreeting: Future[String] = Future.successful("hello, there")

val greeting: Future[String] = customGreeting.flatMap(custom =>
  custom.map(Future.successful).getOrElse(defaultGreeting))
```

We can't quite turn to the `getOrElse` method on `OptionT`, because it takes a `default` value of type `A` instead of `Future[A]`. However, the `getOrElseF` method is exactly what we want:

```
val greeting: Future[String] = customGreetingT.getOrElseF(defaultGreeting)
```

Getting to the underlying instance

If you want to get the `F[Option[A]]` value (in this case `Future[Option[String]]`) out of an `OptionT` instance, you can simply call `value`:

```
val customGreeting: Future[Option[String]] = customGreetingT.value
```

State

`State` is a structure that provides a functional approach to handling application state. `State[S, A]` is basically a function `S => (S, A)`, where `S` is the type that represents your state and `A` is the result the function produces. In addition to returning the result of type `A`, the function returns a new `S` value, which is the updated state.

Robots

Let's try to make this more concrete with an example. We have this `Robot` model:

```
final case class Robot(
  id: Long,
  sentient: Boolean,
  name: String,
  model: String)
```

We would like to generate some random `Robot` instances for test data.

Pseudorandom values

Scala's standard library has a built-in `Random` class that provides a (pseudo)random number generator (RNG). Let's use it to write a method that creates robots.

```
val rng = new scala.util.Random(0L)

def createRobot(): Robot = {
  val id = rng.nextLong()
  val sentient = rng.nextBoolean()
  val isCatherine = rng.nextBoolean()
  val name = if (isCatherine) "Catherine" else "Carlos"
  val isReplicant = rng.nextBoolean()
  val model = if (isReplicant) "replicant" else "borg"
  Robot(id, sentient, name, model)
}
```

```
val robot = createRobot()
// robot: Robot = Robot(-4962768465676381896,false,Catherine,replicant)
```

We create a single `Random` instance, which is mutated as a side-effect each time that we call `nextLong` or `nextBoolean` on it. This mutation makes it more difficult to reason about our code. Someone might come along and see that we have `rng.nextBoolean` repeated three times within a single method. They might cleverly avoid repeated code and method invocations by extracting the common code into a variable:

```
val rng = new scala.util.Random(0L)

def createRobot(): Robot = {
  val id = rng.nextLong()
  val b = rng.nextBoolean()
  val sentient = b
  val isCatherine = b
  val name = if (isCatherine) "Catherine" else "Carlos"
  val isReplicant = b
  val model = if (isReplicant) "replicant" else "borg"
  Robot(id, sentient, name, model)
}
```

```
val robot = createRobot()
// robot: Robot = Robot(-4962768465676381896,false,Carlos,borg)
```

But now the output of our program has changed! We used to have a replicant robot named Catherine, but now we have a borg robot named Carlos. It might not have been obvious, but the `nextBoolean` calls we were making had the side effect of mutating internal RNG state, and we were depending on that behavior.

When we can't freely refactor identical code into a common variable, the code becomes harder to reason about. In functional programming lingo, one might say that such code lacks referential transparency).

Purely functional pseudorandom values

Since mutating state caused us trouble, let's create an RNG that is immutable.

We'll use a simple RNG that can generate pseudorandom `Long` values based only on the previous "seed" value and some carefully chosen constants. You don't need to understand the details of this implementation for the purposes of this example, but if you'd like to know more, this is Knuth's 64-bit linear congruential generator.

```
final case class Seed(long: Long) {
  def next = Seed(long * 6364136223846793005L + 1442695040888963407L)
}
```

Instead of mutating the existing `long` value, calling `next` returns a *new* `Seed` instance with an updated `long` value.

Since the RNG isn't updating state internally, we will need to keep track of state outside of the RNG. When we call `nextBoolean` we will want it to return a `Boolean` as it did before, but we will also want it to return an updated `Seed` that we can use to generate our next random value.

```
  def nextBoolean(seed: Seed): (Seed, Boolean) =
    (seed.next, seed.long >= 0L)
```

Similarly, `nextLong` will return an updated `Seed` along with a `Long` value.

```
  def nextLong(seed: Seed): (Seed, Long) =
    (seed.next, seed.long)
```

Now we need to explicitly pass in the updated state as we generate each new value.

```
def createRobot(seed: Seed): Robot = {
  val (seed1, id) = nextLong(seed)
  val (seed2, sentient) = nextBoolean(seed1)
  val (seed3, isCatherine) = nextBoolean(seed2)
  val name = if (isCatherine) "Catherine" else "Carlos"
  val (seed4, isReplicant) = nextBoolean(seed3)
  val model = if (isReplicant) "replicant" else "borg"
  Robot(id, sentient, name, model)
}

val initialSeed = Seed(13L)
```

```
val robot = createRobot(initialSeed)
// robot: Robot = Robot(13,false,Catherine,replicant)
```

Now it is a bit more obvious that we can't extract the three `nextBoolean` calls into a single variable, because we are passing each one a different seed value.

However, it is a bit cumbersome to explicitly pass around all of this intermediate state. It's also a bit error-prone. It would have been easy to accidentally call `nextBoolean(seed2)` for both the name generation and the model generation, instead of remembering to use `nextBoolean(seed3)` the second time.

Cleaning it up with State

State's special power is keeping track of state and passing it along. Recall the description of `State` at the beginning of this document. It is basically a function `S => (S, A)`, where `S` is a type representing state.

Our `nextLong` function takes a `Seed` and returns an updated `Seed` and a `Long`. It can be represented as `Seed => (Seed, Long)`, and therefore matches the pattern `S => (S, A)` where `S` is `Seed` and `A` is `Long`.

Let's write a new version of `nextLong` using `State`:

```
import cats.data.State

val nextLong: State[Seed, Long] = State(seed =>
  (seed.next, seed.long))
```

The `map` method on `State` allows us to transform the `A` value without affecting the `S` (state) value. This is perfect for implementing `nextBoolean` in terms of `nextLong`.

```
val nextBoolean: State[Seed, Boolean] = nextLong.map(long =>
  long > 0)
```

The `flatMap` method on `State[S, A]` lets you use the result of one `State` in a subsequent `State`. The updated state (`S`) after the first call is passed into the second call. These `flatMap` and `map` methods allow us to use `State` in for-comprehensions:

```
val createRobot: State[Seed, Robot] =
  for {
    id <- nextLong
    sentient <- nextBoolean
    isCatherine <- nextBoolean
    name = if (isCatherine) "Catherine" else "Carlos"
    isReplicant <- nextBoolean
    model = if (isReplicant) "replicant" else "borg"
  } yield Robot(id, sentient, name, model)
```

At this point, we have not yet created a robot; we have written instructions for creating a robot. We need to pass in an initial seed value, and then we can call `value` to actually create the robot:

```
val (finalState, robot) = createRobot.run(initialSeed).value
// finalState: Seed = Seed(2999987205171331217)
// robot: Robot = Robot(13,false,Catherine,replicant)
```

If we only care about the robot and not the final state, then we can use `runA`:

```
val robot = createRobot.runA(initialSeed).value
// robot: Robot = Robot(13,false,Catherine,replicant)
```

The `createRobot` implementation reads much like the imperative code we initially wrote for the mutable RNG. However, this implementation is free of mutation and side-effects. Since this code is referentially transparent, we can perform the refactoring that we tried earlier without affecting the result:

```
val createRobot: State[Seed, Robot] = {
  val b = nextBoolean

  for {
    id <- nextLong
    sentient <- b
    isCatherine <- b
    name = if (isCatherine) "Catherine" else "Carlos"
    isReplicant <- b
    model = if (isReplicant) "replicant" else "borg"
  } yield Robot(id, sentient, name, model)
}
```

```
val robot = createRobot.runA(initialSeed).value
// robot: Robot = Robot(13,false,Catherine,replicant)
```

This may seem surprising, but keep in mind that `b` isn't simply a `Boolean`. It is a function that takes a seed and *returns* a `Boolean`, threading state along the way. Since the seed that is being passed into `b` changes from line to line, so do the returned `Boolean` values.

Fine print

TODO explain StateT and the fact that State is an alias for StateT with Eval.

Validated

Imagine you are filling out a web form to signup for an account. You input your username and password and submit. Response comes back saying your username can't have dashes in it, so you make some changes and resubmit. Can't have special characters either. Change, resubmit. Passwords need to have at least one capital letter. Change, resubmit. Password needs to have at least one number.

Or perhaps you're reading from a configuration file. One could imagine the configuration library you're using returns a `scala.util.Try`, or maybe a `scala.util.Either` (or `cats.data.Xor`). Your parsing may look something like:

```
for {
  url  <- config[String]("url")
  port <- config[Int]("port")
} yield ConnectionParams(url, port)
```

You run your program and it says key "url" not found, turns out the key was "endpoint". So you change your code and re-run. Now it says the "port" key was not a well-formed integer.

It would be nice to have all of these errors be reported simultaneously. That the username can't have dashes can be validated separately from it not having special characters, as well as from the password needing to have certain requirements. A misspelled (or missing) field in a config can be validated separately from another field not being well-formed.

Enter `Validated`.

Parallel validation

Our goal is to report any and all errors across independent bits of data. For instance, when we ask for several pieces of configuration, each configuration field can be validated separately from one another. How then do we enforce that the data we are working with is independent? We ask for both of them up front.

As our running example, we will look at config parsing. Our config will be represented by a `Map[String, String]`. Parsing will be handled by a `Read` type class - we provide instances just for `String` and `Int` for brevity.

```
trait Read[A] {
  def read(s: String): Option[A]
}

object Read {
  def apply[A](implicit A: Read[A]): Read[A] = A

  implicit val stringRead: Read[String] =
    new Read[String] { def read(s: String): Option[String] = Some(s) }

  implicit val intRead: Read[Int] =
    new Read[Int] {
      def read(s: String): Option[Int] =
        if (s.matches("-?[0-9]+")) Some(s.toInt)
        else None
    }
}
```

Then we enumerate our errors - when asking for a config value, one of two things can go wrong: the field is missing, or it is not well-formed with regards to the expected type.

```
sealed abstract class ConfigError
final case class MissingConfig(field: String) extends ConfigError
final case class ParseError(field: String) extends ConfigError
```

We need a data type that can represent either a successful value (a parsed configuration), or an error.

```
sealed abstract class Validated[+E, +A]

object Validated {
  final case class Valid[+A](a: A) extends Validated[Nothing, A]
  final case class Invalid[+E](e: E) extends Validated[E, Nothing]
}
```

Now we are ready to write our parser.

```
import cats.data.Validated
import cats.data.Validated.{Invalid, Valid}

case class Config(map: Map[String, String]) {
  def parse[A : Read](key: String): Validated[ConfigError, A] =
    map.get(key) match {
      case None        => Invalid(MissingConfig(key))
      case Some(value) =>
        Read[A].read(value) match {
          case None    => Invalid(ParseError(key))
          case Some(a) => Valid(a)
        }
    }
}
```

Everything is in place to write the parallel validator. Recall that we can only do parallel validation if each piece is independent. How do we enforce the data is independent? By asking for all of it up front. Let's start with two pieces of data.

```
def parallelValidate[E, A, B, C](v1: Validated[E, A], v2: Validated[E, B])(f: (A, B) => C): Validated[E, C] =
  (v1, v2) match {
    case (Valid(a), Valid(b))       => Valid(f(a, b))
    case (Valid(_), i@Invalid(_))   => i
    case (i@Invalid(_), Valid(_))   => i
    case (Invalid(e1), Invalid(e2)) => ???
  }
```

We've run into a problem. In the case where both have errors, we want to report both. But we have no way of combining the two errors into one error! Perhaps we can put both errors into a `List`, but that seems needlessly specific - clients may want to define their own way of combining errors.

How then do we abstract over a binary operation? The `Semigroup` type class captures this idea.

```
import cats.Semigroup

def parallelValidate[E : Semigroup, A, B, C](v1: Validated[E, A], v2: Validated[E, B])(f: (A, B) => C): Validated[E, C
] =
  (v1, v2) match {
    case (Valid(a), Valid(b))       => Valid(f(a, b))
    case (Valid(_), i@Invalid(_))   => i
    case (i@Invalid(_), Valid(_))   => i
    case (Invalid(e1), Invalid(e2)) => Invalid(Semigroup[E].combine(e1, e2))
  }
```

Perfect! But.. going back to our example, we don't have a way to combine `ConfigError`s. But as clients, we can change our `Validated` values where the error can be combined, say, a `List[ConfigError]`. It is more common however to use a `NonEmptyList[ConfigError]` - the `NonEmptyList` statically guarantees we have at least one value, which aligns with the fact

that if we have an `Invalid`, then we most certainly have at least one error. This technique is so common there is a convenient method on `Validated` called `toValidatedNel` that turns any `Validated[E, A]` value to a `Validated[NonEmptyList[E], A]`. Additionally, the type alias `ValidatedNel[E, A]` is provided.

Time to parse.

```
import cats.SemigroupK
import cats.data.NonEmptyList
import cats.implicits._

case class ConnectionParams(url: String, port: Int)

val config = Config(Map(("endpoint", "127.0.0.1"), ("port", "not an int")))

implicit val nelSemigroup: Semigroup[NonEmptyList[ConfigError]] =
  SemigroupK[NonEmptyList].algebra[ConfigError]

implicit val readString: Read[String] = Read.stringRead
implicit val readInt: Read[Int] = Read.intRead
```

Any and all errors are reported!

```
val v1 = parallelValidate(config.parse[String]("url").toValidatedNel,
                          config.parse[Int]("port").toValidatedNel)(ConnectionParams.apply)
// v1: cats.data.Validated[cats.data.NonEmptyList[ConfigError],ConnectionParams] = Invalid(OneAnd(MissingConfig(url),
List(ParseError(port))))

val v2 = parallelValidate(config.parse[String]("endpoint").toValidatedNel,
                          config.parse[Int]("port").toValidatedNel)(ConnectionParams.apply)
// v2: cats.data.Validated[cats.data.NonEmptyList[ConfigError],ConnectionParams] = Invalid(OneAnd(ParseError(port),Li
st()))

val config = Config(Map(("endpoint", "127.0.0.1"), ("port", "1234")))
// config: Config = Config(Map(endpoint -> 127.0.0.1, port -> 1234))

val v3 = parallelValidate(config.parse[String]("endpoint").toValidatedNel,
                          config.parse[Int]("port").toValidatedNel)(ConnectionParams.apply)
// v3: cats.data.Validated[cats.data.NonEmptyList[ConfigError],ConnectionParams] = Valid(ConnectionParams(127.0.0.1,1
234))
```

Apply

Our `parallelValidate` function looks awfully like the `Apply#map2` function.

```
def map2[F[_], A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]
```

Which can be defined in terms of `Apply#ap` and `Apply#map`, the very functions needed to create an `Apply` instance.

Can we perhaps define an `Apply` instance for `Validated`? Better yet, can we define an `Applicative` instance?

Note: the example below assumes usage of the kind-projector compiler plugin and will not compile if it is not being used in a project.

```
import cats.Applicative

implicit def validatedApplicative[E : Semigroup]: Applicative[Validated[E, ?]] =
  new Applicative[Validated[E, ?]] {
    def ap[A, B](f: Validated[E, A => B])(fa: Validated[E, A]): Validated[E, B] =
      (fa, f) match {
        case (Valid(a), Valid(fab)) => Valid(fab(a))
        case (i@Invalid(_), Valid(_)) => i
        case (Valid(_), i@Invalid(_)) => i
        case (Invalid(e1), Invalid(e2)) => Invalid(Semigroup[E].combine(e1, e2))
      }

    def pure[A](x: A): Validated[E, A] = Validated.valid(x)
  }
```

Awesome! And now we also get access to all the goodness of `Applicative`, which includes `map{2-22}`, as well as the `Cartesian` syntax `|@|`.

We can now easily ask for several bits of configuration and get any and all errors returned back.

```
import cats.Apply
import cats.data.ValidatedNel

implicit val nelSemigroup: Semigroup[NonEmptyList[ConfigError]] =
  SemigroupK[NonEmptyList].algebra[ConfigError]

val config = Config(Map(("name", "cat"), ("age", "not a number"), ("houseNumber", "1234"), ("lane", "feline street"))
)

case class Address(houseNumber: Int, street: String)
case class Person(name: String, age: Int, address: Address)
```

Thus.

```
val personFromConfig: ValidatedNel[ConfigError, Person] =
  Apply[ValidatedNel[ConfigError, ?]].map4(config.parse[String]("name").toValidatedNel,
                                           config.parse[Int]("age").toValidatedNel,
                                           config.parse[Int]("house_number").toValidatedNel,
                                           config.parse[String]("street").toValidatedNel) {
    case (name, age, houseNumber, street) => Person(name, age, Address(houseNumber, street))
  }
// personFromConfig: cats.data.ValidatedNel[ConfigError,Person] = Invalid(OneAnd(MissingConfig(street),List(MissingCo
nfig(house_number), ParseError(age))))
```

Of `flatMap`s and `Xor`s

`Option` has `flatMap`, `Xor` has `flatMap`, where's `Validated`'s? Let's try to implement it - better yet, let's implement the `Monad` type class.

```
import cats.Monad

implicit def validatedMonad[E]: Monad[Validated[E, ?]] =
  new Monad[Validated[E, ?]] {
    def flatMap[A, B](fa: Validated[E, A])(f: A => Validated[E, B]): Validated[E, B] =
      fa match {
        case Valid(a)     => f(a)
        case i@Invalid(_) => i
      }

    def pure[A](x: A): Validated[E, A] = Valid(x)
  }
```

Note that all `Monad` instances are also `Applicative` instances, where `ap` is defined as

```
trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
  def pure[A](x: A): F[A]

  def map[A, B](fa: F[A])(f: A => B): F[B] =
    flatMap(fa)(f.andThen(pure))

  def ap[A, B](fa: F[A])(f: F[A => B]): F[B] =
    flatMap(fa)(a => map(f)(fab => fab(a)))
}
```

However, the `ap` behavior defined in terms of `flatMap` does not behave the same as that of our `ap` defined above. Observe:

```
val v = validatedMonad.tuple2(Validated.invalidNel[String, Int]("oops"), Validated.invalidNel[String, Double]("uh oh"
))
// v: cats.data.Validated[cats.data.NonEmptyList[String],(Int, Double)] = Invalid(OneAnd(oops,List()))
```

This one short circuits! Therefore, if we were to define a `Monad` (or `FlatMap`) instance for `Validated` we would have to override `ap` to get the behavior we want. But then the behavior of `flatMap` would be inconsistent with that of `ap`, not good. Therefore, `Validated` has only an `Applicative` instance.

`Validated` vs `Xor`

We've established that an error-accumulating data type such as `Validated` can't have a valid `Monad` instance. Sometimes the task at hand requires error-accumulation. However, sometimes we want a monadic structure that we can use for sequential validation (such as in a for-comprehension). This leaves us in a bit of a conundrum.

Cats has decided to solve this problem by using separate data structures for error-accumulation (`Validated`) and short-circuiting monadic behavior (`Xor`).

If you are trying to decide whether you want to use `Validated` or `Xor`, a simple heuristic is to use `Validated` if you want error-accumulation and to otherwise use `Xor`.

Sequential Validation

If you do want error accumulation but occasionally run into places where you sequential validation is needed, then `Validated` provides a couple methods that may be helpful.

`andThen`

The `andThen` method is similar to `flatMap` (such as `Xor.flatMap`). In the cause of success, it passes the valid value into a function that returns a new `Validated` instance.

```
val houseNumber = config.parse[Int]("house_number").andThen{ n =>
   if (n >= 0) Validated.valid(n)
   else Validated.invalid(ParseError("house_number"))
}
// houseNumber: cats.data.Validated[ConfigError,Int] = Invalid(MissingConfig(house_number))
```

`withXor`

The `withXor` method allows you to temporarily turn a `Validated` instance into an `Xor` instance and apply it to a function.

```
import cats.data.Xor

def positive(field: String, i: Int): ConfigError Xor Int = {
  if (i >= 0) Xor.right(i)
  else Xor.left(ParseError(field))
}
```

Thus.

```
val houseNumber = config.parse[Int]("house_number").withXor{ xor: ConfigError Xor Int =>
  xor.flatMap{ i =>
    positive("house_number", i)
  }
}
// houseNumber: cats.data.Validated[ConfigError,Int] = Invalid(MissingConfig(house_number))
```

Xor

In day-to-day programming, it is fairly common to find ourselves writing functions that can fail. For instance, querying a service may result in a connection issue, or some unexpected JSON response.

To communicate these errors it has become common practice to throw exceptions. However, exceptions are not tracked in any way, shape, or form by the Scala compiler. To see what kind of exceptions (if any) a function may throw, we have to dig through the source code. Then to handle these exceptions, we have to make sure we catch them at the call site. This all becomes even more unwieldy when we try to compose exception-throwing procedures.

```
val throwsSomeStuff: Int => Double = ???

val throwsOtherThings: Double => String = ???

val moreThrowing: String => List[Char] = ???

val magic = throwsSomeStuff.andThen(throwsOtherThings).andThen(moreThrowing)
```

Assume we happily throw exceptions in our code. Looking at the types, any of those functions can throw any number of exceptions, we don't know. When we compose, exceptions from any of the constituent functions can be thrown. Moreover, they may throw the same kind of exception (e.g. `IllegalArgumentException`) and thus it gets tricky tracking exactly where that exception came from.

How then do we communicate an error? By making it explicit in the data type we return.

Xor

`Xor` vs `Validated`

In general, `Validated` is used to accumulate errors, while `Xor` is used to short-circuit a computation upon the first error. For more information, see the `Validated` vs `Xor` section of the `Validated` documentation.

Why not `Either`

`Xor` is very similar to `scala.util.Either` - in fact, they are *isomorphic* (that is, any `Either` value can be rewritten as an `Xor` value, and vice versa).

```
sealed abstract class Xor[+A, +B]

object Xor {
  final case class Left[+A](a: A) extends Xor[A, Nothing]
  final case class Right[+B](b: B) extends Xor[Nothing, B]
}
```

Just like `Either`, it has two type parameters. Instances of `Xor` either hold a value of one type parameter, or the other. Why then does it exist at all?

Taking a look at `Either`, we notice it lacks `flatMap` and `map` methods. In order to map over an `Either[A, B]` value, we have to state which side we want to map over. For example, if we want to map `Either[A, B]` to `Either[A, C]` we would need to map over the right side. This can be accomplished by using the `Either#right` method, which returns a `RightProjection` instance. `RightProjection` does have `flatMap` and `map` on it, which acts on the right side and ignores the left - this property is referred to as "right-bias."

```
val e1: Either[String, Int] = Right(5)
// e1: Either[String,Int] = Right(5)

e1.right.map(_ + 1)
// res0: Product with Serializable with scala.util.Either[String,Int] = Right(6)

val e2: Either[String, Int] = Left("hello")
// e2: Either[String,Int] = Left(hello)

e2.right.map(_ + 1)
// res1: Product with Serializable with scala.util.Either[String,Int] = Left(hello)
```

Note the return types are themselves back to `Either`, so if we want to make more calls to `flatMap` or `map` then we again must call `right` or `left`.

More often than not we want to just bias towards one side and call it a day - by convention, the right side is most often chosen. This is the primary difference between `Xor` and `Either` - `Xor` is right-biased. `Xor` also has some more convenient methods on it, but the most crucial one is the right-biased being built-in.

```
import cats.data.Xor
// import cats.data.Xor

val xor1: Xor[String, Int] = Xor.right(5)
// xor1: cats.data.Xor[String,Int] = Right(5)

xor1.map(_ + 1)
// res2: cats.data.Xor[String,Int] = Right(6)

val xor2: Xor[String, Int] = Xor.left("hello")
// xor2: cats.data.Xor[String,Int] = Left(hello)

xor2.map(_ + 1)
// res3: cats.data.Xor[String,Int] = Left(hello)
```

Because `Xor` is right-biased, it is possible to define a `Monad` instance for it. You could also define one for `Either` but due to how it's encoded it may seem strange to fix a bias direction despite it intending to be flexible in that regard. The `Monad` instance for `Xor` is consistent with the behavior of the data type itself, whereas the one for `Either` would only introduce bias when `Either` is used in a generic context (a function abstracted over `M[_] : Monad`).

Since we only ever want the computation to continue in the case of `Xor.Right` (as captured by the right-bias nature), we fix the left type parameter and leave the right one free.

Note: the example below assumes usage of the kind-projector compiler plugin and will not compile if it is not being used in a project.

```
import cats.Monad

implicit def xorMonad[Err]: Monad[Xor[Err, ?]] =
  new Monad[Xor[Err, ?]] {
    def flatMap[A, B](fa: Xor[Err, A])(f: A => Xor[Err, B]): Xor[Err, B] =
      fa.flatMap(f)

    def pure[A](x: A): Xor[Err, A] = Xor.right(x)
  }
```

Example usage: Round 1

As a running example, we will have a series of functions that will parse a string into an integer, take the reciprocal, and then turn the reciprocal into a string.

In exception-throwing code, we would have something like this:

```
object ExceptionStyle {
  def parse(s: String): Int =
    if (s.matches("-?[0-9]+")) s.toInt
    else throw new NumberFormatException(s"${s} is not a valid integer.")

  def reciprocal(i: Int): Double =
    if (i == 0) throw new IllegalArgumentException("Cannot take reciprocal of 0.")
    else 1.0 / i

  def stringify(d: Double): String = d.toString
}
```

Instead, let's make the fact that some of our functions can fail explicit in the return type.

```
object XorStyle {
  def parse(s: String): Xor[NumberFormatException, Int] =
    if (s.matches("-?[0-9]+")) Xor.right(s.toInt)
    else Xor.left(new NumberFormatException(s"${s} is not a valid integer."))

  def reciprocal(i: Int): Xor[IllegalArgumentException, Double] =
    if (i == 0) Xor.left(new IllegalArgumentException("Cannot take reciprocal of 0."))
    else Xor.right(1.0 / i)

  def stringify(d: Double): String = d.toString
}
```

Now, using combinators like `flatMap` and `map`, we can compose our functions together.

```
import XorStyle._

def magic(s: String): Xor[Exception, String] =
  parse(s).flatMap(reciprocal).map(stringify)
```

With the composite function that we actually care about, we can pass in strings and then pattern match on the exception. Because `Xor` is a sealed type (often referred to as an algebraic data type, or ADT), the compiler will complain if we do not check both the `Left` and `Right` case.

```
magic("123") match {
  case Xor.Left(_: NumberFormatException) => println("not a number!")
  case Xor.Left(_: IllegalArgumentException) => println("can't take reciprocal of 0!")
  case Xor.Left(_) => println("got unknown exception")
  case Xor.Right(s) => println(s"Got reciprocal: ${s}")
}
// Got reciprocal: 0.008130081300813009
```

Not bad - if we leave out any of those clauses the compiler will yell at us, as it should. However, note the `Xor.Left(_)` clause - the compiler will complain if we leave that out because it knows that given the type `Xor[Exception, String]`, there can be inhabitants of `Xor.Left` that are not `NumberFormatException` or `IllegalArgumentException`. However, we "know" by inspection of the source that those will be the only exceptions thrown, so it seems strange to have to account for other exceptions. This implies that there is still room to improve.

Example usage: Round 2

Instead of using exceptions as our error value, let's instead enumerate explicitly the things that can go wrong in our program.

```
object XorStyle {
  sealed abstract class Error
  final case class NotANumber(string: String) extends Error
  final case object NoZeroReciprocal extends Error

  def parse(s: String): Xor[Error, Int] =
    if (s.matches("-?[0-9]+")) Xor.right(s.toInt)
    else Xor.left(NotANumber(s))

  def reciprocal(i: Int): Xor[Error, Double] =
    if (i == 0) Xor.left(NoZeroReciprocal)
    else Xor.right(1.0 / i)

  def stringify(d: Double): String = d.toString

  def magic(s: String): Xor[Error, String] =
    parse(s).flatMap(reciprocal).map(stringify)
}
```

For our little module, we enumerate any and all errors that can occur. Then, instead of using exception classes as error values, we use one of the enumerated cases. Now when we pattern match, we get much nicer matching. Moreover, since `Error` is `sealed`, no outside code can add additional subtypes which we might fail to handle.

```
import XorStyle._
// import XorStyle._

magic("123") match {
  case Xor.Left(NotANumber(_)) => println("not a number!")
  case Xor.Left(NoZeroReciprocal) => println("can't take reciprocal of 0!")
  case Xor.Right(s) => println(s"Got reciprocal: ${s}")
}
// Got reciprocal: 0.008130081300813009
```

Xor in the small, Xor in the large

Once you start using `Xor` for all your error-handling, you may quickly run into an issue where you need to call into two separate modules which give back separate kinds of errors.

```
sealed abstract class DatabaseError
trait DatabaseValue

object Database {
  def databaseThings(): Xor[DatabaseError, DatabaseValue] = ???
}

sealed abstract class ServiceError
trait ServiceValue

object Service {
  def serviceThings(v: DatabaseValue): Xor[ServiceError, ServiceValue] = ???
}
```

Let's say we have an application that wants to do database things, and then take database values and do service things. Glancing at the types, it looks like `flatMap` will do it.

```
def doApp = Database.databaseThings().flatMap(Service.serviceThings)
```

This doesn't work! Well, it does, but it gives us `Xor[Object, ServiceValue]` which isn't particularly useful for us. Now if we inspect the `Left`s, we have no clue what it could be. The reason this occurs is because the first type parameter in the two `Xor`s are different - `databaseThings()` can give us a `DatabaseError` whereas `serviceThings()` can give us a

`ServiceError` : two completely unrelated types. Recall that the type parameters of `Xor` are covariant, so when it sees an `Xor[E1, A1]` and an `Xor[E2, A2]` , it will happily try to unify the `E1` and `E2` in a `flatMap` call - in our case, the closest common supertype is `Object` , leaving us with practically no type information to use in our pattern match.

Solution 1: Application-wide errors

So clearly in order for us to easily compose `Xor` values, the left type parameter must be the same. We may then be tempted to make our entire application share an error data type.

```
sealed abstract class AppError
final case object DatabaseError1 extends AppError
final case object DatabaseError2 extends AppError
final case object ServiceError1 extends AppError
final case object ServiceError2 extends AppError

trait DatabaseValue

object Database {
  def databaseThings(): Xor[AppError, DatabaseValue] = ???
}

object Service {
  def serviceThings(v: DatabaseValue): Xor[AppError, ServiceValue] = ???
}

def doApp = Database.databaseThings().flatMap(Service.serviceThings)
```

This certainly works, or at least it compiles. But consider the case where another module wants to just use `Database` , and gets an `Xor[AppError, DatabaseValue]` back. Should it want to inspect the errors, it must inspect **all** the `AppError` cases, even though it was only intended for `Database` to use `DatabaseError1` or `DatabaseError2` .

Solution 2: ADTs all the way down

Instead of lumping all our errors into one big ADT, we can instead keep them local to each module, and have an application-wide error ADT that wraps each error ADT we need.

```
sealed abstract class DatabaseError
trait DatabaseValue

object Database {
  def databaseThings(): Xor[DatabaseError, DatabaseValue] = ???
}

sealed abstract class ServiceError
trait ServiceValue

object Service {
  def serviceThings(v: DatabaseValue): Xor[ServiceError, ServiceValue] = ???
}

sealed abstract class AppError
object AppError {
  final case class Database(error: DatabaseError) extends AppError
  final case class Service(error: ServiceError) extends AppError
}
```

Now in our outer application, we can wrap/lift each module-specific error into `AppError` and then call our combinators as usual. `Xor` provides a convenient method to assist with this, called `Xor.leftMap` - it can be thought of as the same as `map` , but for the `Left` side.

```
def doApp: Xor[AppError, ServiceValue] =
  Database.databaseThings().leftMap(AppError.Database).
  flatMap(dv => Service.serviceThings(dv).leftMap(AppError.Service))
```

Hurrah! Each module only cares about its own errors as it should be, and more composite modules have their own error ADT that encapsulates each constituent module's error ADT. Doing this also allows us to take action on entire classes of errors instead of having to pattern match on each individual one.

```
def awesome =
  doApp match {
    case Xor.Left(AppError.Database(_)) => "something in the database went wrong"
    case Xor.Left(AppError.Service(_))  => "something in the service went wrong"
    case Xor.Right(_)                   => "everything is alright!"
  }
```

Working with exception-y code

There will inevitably come a time when your nice `Xor` code will have to interact with exception-throwing code. Handling such situations is easy enough.

```
val xor: Xor[NumberFormatException, Int] =
  try {
    Xor.right("abc".toInt)
  } catch {
    case nfe: NumberFormatException => Xor.left(nfe)
  }
// xor: cats.data.Xor[NumberFormatException,Int] = Left(java.lang.NumberFormatException: For input string: "abc")
```

However, this can get tedious quickly. `Xor` provides a `catchOnly` method on its companion object that allows you to pass it a function, along with the type of exception you want to catch, and does the above for you.

```
val xor: Xor[NumberFormatException, Int] =
  Xor.catchOnly[NumberFormatException]("abc".toInt)
// xor: cats.data.Xor[NumberFormatException,Int] = Left(java.lang.NumberFormatException: For input string: "abc")
```

If you want to catch all (non-fatal) throwables, you can use `catchNonFatal`.

```
val xor: Xor[Throwable, Int] =
  Xor.catchNonFatal("abc".toInt)
// xor: cats.data.Xor[Throwable,Int] = Left(java.lang.NumberFormatException: For input string: "abc")
```

Additional syntax

```
import cats.implicits._
// import cats.implicits._

val xor3: Xor[String, Int] = 7.right[String]
// xor3: cats.data.Xor[String,Int] = Right(7)

val xor4: Xor[String, Int] = "hello s" .left[Int]
// xor4: cats.data.Xor[String,Int] = Left(hello s)
```

Contributor guide

Discussion around Cats is currently happening in the Gitter channel as well as on Github issue and PR pages. You can get an overview of who is working on what via Waffle.io.

Feel free to open an issue if you notice a bug, have an idea for a feature, or have a question about the code. Pull requests are also gladly accepted.

People are expected to follow the Typelevel Code of Conduct when discussing Cats on the Github page, Gitter channel, or other venues.

We hope that our community will be respectful, helpful, and kind. If you find yourself embroiled in a situation that becomes heated, or that fails to live up to our expectations, you should disengage and contact one of the project maintainers in private. We hope to avoid letting minor aggressions and misunderstandings escalate into larger problems.

If you are being harassed, please contact one of us immediately so that we can support you.

How can I help?

Cats follows a standard fork and pull model for contributions via GitHub pull requests.

Below is a list of the steps that might be involved in an ideal contribution. If you don't have the time to go through every step, contribute what you can, and someone else will probably be happy to follow up with any polishing that may need to be done.

If you want to touch up some documentation or fix typos, feel free to skip these steps and jump straight to submitting a pull request.

1. Find something that belongs in cats
2. Let us know you are working on it
3. Implement your contribution
4. Write tests
5. Write documentation
6. Write examples
7. Submit pull request

Find something that belongs in cats

Looking for a way that you can help out? Check out our Waffle.io page. Choose a card from the "Ready" column. Before you start working on it, make sure that it's not already assigned to someone and that nobody has left a comment saying that they are working on it!

(Of course, you can also comment on an issue someone is already working on and offer to collaborate.)

Have an idea for something new? That's great! We recommend that you make sure it belongs in cats before you put effort into creating a pull request. The preferred ways to do that are to either:

- create a GitHub issue describing your idea.
- get feedback in the cats Gitter room.

Things that belong in cats generally have the following characteristics:

- Their behavior is governed by well-defined laws.
- They provide general abstractions.

Laws help keep types consistent, and remove ambiguity or sensitivity about how particular instances can behave. We've found that types with laws are often more useful than *lawless* types

(In some cases, *lawless* type classes and instances are useful. We intend to support some of these in a future module.)

By staying general, Cats' abstractions are widely-applicable, and not tied to particular libraries or strategies. Rather than being a library to work with databases, HTTP requests, etc, Cats provides abstractions used to build those libraries.

Cats (and especially `cats-core`) is intended to be lean and modular. Some great ideas are not a great fit, either due to their size or their complexity. In these cases, creating your own library that depends on Cats is probably the best plan.

Let us know you are working on it

If there is already a GitHub issue for the task you are working on, leave a comment to let people know that you are working on it. If there isn't already an issue and it is a non-trivial task, it's a good idea to create one (and note that you're working on it). This prevents contributors from duplicating effort.

Write code

TODO

Should this just link to a separate doc? This might get large.

Write about implicit params as discussed in https://github.com/typelevel/cats/issues/27

Write about type class methods on data structures as described in https://github.com/typelevel/cats/issues/25

Write about https://github.com/typelevel/cats/pull/36#issuecomment-72892359

Write tests

- Tests for cats-core go into the tests module, under the `cats.tests` package.
- Tests for additional modules, such as 'jvm', go into the tests directory within that module.
- Cats tests should extend `CatsSuite`. `CatsSuite` integrates ScalaTest with Discipline for law checking, and imports all syntax and standard instances for convenience.
- The first parameter to the `checkAll` method provided by Discipline, is the name of the test and will be output to the console as part of the test execution. By convention:
  - When checking laws, this parameter generally takes a form that describes the data type being tested. For example the name *"Validated[String, Int]"* might be used when testing a type class instance that the `Validated` data type supports.
  - An exception to this is serializability tests, where the type class name is also included in the name. For example, in the case of `Validated`, the serializability test would take the form, *"Applicative[Validated[String, Int]"*, to indicate that this test is verifying that the `Applicative` type class instance for the `Validated` data type is serializable.
  - This convention helps to ensure clear and easy to understand output, with minimal duplication in the output.
- It is also a goal that, for every combination of data type and supported type class instance:
  - Appropriate law checks for that combination are included to ensure that the instance meets the laws for that type class.
  - A serializability test for that combination is also included, such that we know that frameworks which rely heavily on serialization, such as `Spark`, will have strong compatibility with `cats`.
  - Note that custom serialization tests are not required for instances of type classes which come from `algebra`, such as `Monoid`, because the `algebra` laws include a test for serialization.

TODO

Write about checking laws

Contributing documentation

Source for the documentation

The documentation for this website is stored alongside the source, in the docs subproject.

- The source for the static pages is in `docs/src/site`
- The source for the tut compiled pages is in `docs/src/main/tut`

Generating the Site

run `sbt docs/makeSite`

Previewing the site

1. Install jekyll locally, depending on your platform, you might do this with:

   yum install jekyll

   apt-get install jekyll

   gem install jekyll

2. In a shell, navigate to the generated site directory in `docs/target/site`
3. Start jekyll with `jekyll serve`
4. Navigate to http://localhost:4000/cats/ in your browser
5. Make changes to your site, and run `sbt makeSite` to regenerate the site. The changes should be reflected as soon as you run `makeSite` .

Generating the book

run `bash scripts/build-book.sh`

Compiler verified documentation

We use tut to compile source code which appears in the documentation, this ensures us that our examples should always compile, and our documentation has a better chance of staying up-to-date.

Publishing the site to github.

The `git.remoteRepo` variable in `docs/build.sbt` controls which repository you will push to. Ensure that this variable points to a repo you wish to push to, and that it is one for which you have push access, then run `sbt ghpagesPushSite`

Write examples

TODO

Submit a pull request

Before you open a pull request, you should make sure that `sbt validate` runs successfully. Travis will run this as well, but it may save you some time to be alerted to style problems earlier.

If your pull request addresses an existing issue, please tag that issue number in the body of your pull request or commit message. For example, if your pull request addresses issue number 52, please include "fixes #52".

If you make changes after you have opened your pull request, please add them as separate commits and avoid squashing or rebasing. Squashing and rebasing can lead to a tidier git history, but they can also be a hassle if somebody else has done work based on your branch.

How did we do?

Getting involved in an open source project can be tough. As a newcomer, you may not be familiar with coding style conventions, project layout, release cycles, etc. This document seeks to demystify the contribution process for the cats project.

It may take a while to familiarize yourself with this document, but if we are doing our job right, you shouldn't have to spend months poring over the project source code or lurking the Gitter room before you feel comfortable contributing. In fact, if you encounter any confusion or frustration during the contribution process, please create a GitHub issue and we'll do our best to improve the process.

Process

Introduction

This document is intended to help consolidate the practices we are using to develop and maintain Cats. Its primary audience is Cats maintainers who may need to close issues, merge PRs, do releases, or understand how these things work.

Merging pull requests

Pull requests currently require two sign-offs from Cats maintainers. Community member sign-offs are appreciated as votes of confidence but don't usually count toward this total unless the commenter has already been involved with the area of code in question.

When fixing typos or improving documentation only one sign-off is required (although for major edits waiting for two may be preferable).

For serious emergencies or work on the build which can't easily be reviewed or tested, pushing directly to master may be OK (but is definitely not encouraged). In these cases it's best to comment in Gitter or elsewhere about what happened and what was done.

Versioning

If a release is simply a bug fix, increment the patch version number (e.g. 1.2.3 becomes 1.2.4). These releases may happen quite quickly in response to reported bugs or problems, and should usually be source and binary compatible.

If the major version is 0, then the minor version should be updated (e.g. 0.2.3 becomes 0.3.0). There are no compatibility guarantees for this type of change.

If the major version is 1 or greater, then significant additions should increment the minor version number (e.g. 1.2.3 becomes 1.3.0) and breaking or incompatible changes should increment the major number (e.g. 1.2.3 becomes 2.0.0). These major version bumps should only occur after substantial review, warning, and with proper deprecation cycles.

Releasing

Before the release, the tests and other validation must be passing.

Currently, the best way to release cats is:

1. Run `+ clean` to ensure a clean start.
2. Run `+ package` to ensure everything builds.
3. Run `release`, which will validate tests/code, etc.

(Eventually `release` should do all of these things but for now the individual steps are necessary.)

You will need to enter a GPG password to sign the artifacts correctly, and you will also need to have Sonatype credentials to publish the artifacts. The release process should take care of everything, including releasing the artifacts on Sonatype.

If the Sonatype publishing fails, but the artifacts were uploaded, you can finish the release manually using the Sonatype web site. In that case you will also need to do `git push --tags origin master` to push the new version's tags.

(In the future we may want to create branches for major versions, e.g. `v1.x`. For now we don't have these.)

Post-release

After the release occurs, you will need to update the documentation. Here is a list of the places that will definitely need to be updated:

- `docs/src/site/index.md` : update version numbers
- `README.md` : update version numbers
- `AUTHORS.md` : add new contributors
- `CHANGES.md` : summarize changes since last release

(Other changes may be necessary, especially for large releases.)

You can get a list of changes between release tags `v0.1.2` and `v0.2.0` via `git log v0.1.2..v0.2.0`. Scanning this list of commit messages is a good way to get a summary of what happened, although it does not account for conversations that occured on Github.

Once the relevant documentation changes have been committed, new release notes should be added. You can add a release by clicking the "Draft a new release" button on that page, or if the relevant release already exists, you can click "Edit release".

The website should then be updated via `sbt docs/ghpagesPushSite`.

Conclusion

Ideally this document will evolve and grow as the project matures. Pull requests to add sections explaining how maintenance occurs (or should occur) are very welcome.

Authors

A successful open-source project relies upon the community to:

- discuss requirements and possible designs
- submit code and tests
- identify and fix bugs
- create documentation and examples
- provide feedback
- support each other

This file lists the people whose contributions have made Cats possible:

- 3rdLaw
- Aaron Levin
- Adelbert Chang
- Aldo Stracquadanio
- Alessandro Lacava
- Alexey Levan
- Alissa Pajer
- Alistair Johnson
- Amir Mohammad Saied
- Andrew Jones
- Angelo Genovese
- Antoine Comte
- Arya Irani
- Ash Pook
- Aλ
- Benjamin Thuillier
- Binh Nguyen
- Bobby Rauchenberg
- Brendan McAdams
- Cody Allen
- Colt Frederickson
- Dale Wijnand
- Daniel Spiewak
- Dave Gurnell
- Dave Rostron
- David Allsopp
- David Gregory
- Denis Mikhaylov
- Derek Wickern
- Edmund Noble
- Erik LaBianca
- Erik Osheim
- Eugene Burmako
- Eugene Yokota
- Feynman Liang
- Frank S. Thomas
- Ian McIntosh
- ImLiar
- Jean-Rémi Desjardins
- Jisoo Park

- Josh Marcus
- Juan Pedro Moreno
- Julien Richard-Foy
- Julien Truffaut
- Kailuo Wang
- Kenji Yoshida
- Long Cao
- Luis Angel Vicente Sanchez
- Luis Sanchez
- Luke Wyman
- Marc Siegel
- Markus Hauck
- Matt Martin
- Matthias Lüneberg
- Max Worgan
- Michael Pilquist
- Mike Curry
- Miles Sabin
- Olli Helenius
- Owen Parry
- Pascal Voitot
- Paul Phillips
- Pavkin Vladimir
- Pere Villega
- Peter Neyens
- Philip Wills
- Raúl Raja Martínez
- Rintcius Blok
- Rob Norris
- Romain Ruetschi
- Ross A. Baker
- Ryan Case
- Sarunas Valaskevicius
- Shunsuke Otani
- Sinisa Louc
- Stephen Carman
- Stephen Judkins
- Stew O'Connor
- Sumedh Mungee
- Tomas Mikula
- Travis Brown
- Wedens
- Yosef Fertel
- yilinwei
- Zach Abbott

We've tried to include everyone, but if you've made a contribution to Cats and are not listed, please feel free to open an issue or pull request with your name and contribution.

Thank you!

Version 0.6.0

> 2016 May 19

Version 0.6.0 is the sixth release.

Highlights of this release:

- #990: Separate free package into its own module
- #1001: Introduce cats-kernel and remove algebra dependency

This release also includes some API changes:

- #1046: summon `ApplicativeErrorSyntax` for `F[_]` instead of `F[_, _]`
- #1034: Don't combine lefts on `Xor` and `XorT` `combine`
- #1018: Remove blocking (JVM-only) Future instances
- #877: Remove required laziness in Prod, fixes #615

And additions:

- #1032: Added `Coproduct` `fold`
- #1028: Added `withFilter` for `OptionT`
- #1014: Added `Monoid` instance for `WriterT`
- #1029: Added an `ApplicativeError` instance for `Kleisli` and a `MonadError[Option, Unit]` to `std.option`
- #1023: Add `XorT#fromEither`
- #984: Add `Validated.ensure`
- #1020: Add `Traverse.traverseM`

And some code improvements:

- #1015: Add `Apply.map2Eval` and allow traverse laziness
- #1024: Override reverse on reversed `PartialOrder` to return original instance
- #880: Optimize `Eq[Vector[A]]` instance
- #1019: Use `Future#successful` in `pureEval` when possible

And bug fixes:

- #1011: Add missing type parameters.

And some other improvements to the organization documentation, tutorials, laws and tests, including:

- #1045: Add a link to the `OptionT` documentation from the monad docs.
- #1043: Add notes about kind-projector usage in docs
- #1042: Cats 0.5.0 no longer pre-release
- #1036: Add FPiS to the "Resources for Learners" section
- #1035: Run kernel-law tests for JS as part of build
- #991: Replace `~>` with `NaturalTransformation`
- #1027: Remove unnecessary `nelSemigroup` from `traverse` doc
- #1022: Add law-checking for `asMeetPartialOrder` and `asJoinPartialOrder`
- #990: Separate free package into its own module

Version 0.5.0

> 2016 April 28

Version 0.5.0 is the fifth release.

This release includes some API changes:

`cats.laws.discipline.eq` no longer provides `Eq` instances for `Tuple2` and `Tuple3`, these instances and together with some other new instances for `Tuple`s are now provided by `cats.std.tuple` (through inheriting the instance trait defined in algebra 0.4.2).

- #910: Remove `Streaming` and `StreamingT`
- #967: `product` and `map` can be implemented in terms of `ap`
- #970: Renamed `Kleisli#apply` to `ap`
- #994: updated to latest algebra (brought in all the new goodies)

And additions:

- #853: Adds a new `LiftTrans` typeclass
- #864: Add `Bifoldable`
- #875: Add `.get` method to `StateT`
- #884: Add `Applicative` syntax
- #886: Add `map` method to `OneAnd`
- #927: `XorT.ensure` method
- #925: Stack-safe `foldM`
- #922: Add `tell` and `writer` syntax for creating `Writers`.
- #903: Add `Bitraverse`
- #928: Add missing `Show` instances
- #940: More flexible `TransLift`
- #946: Added `OptionT.none`
- #947: Syntax for `ApplicativeError`
- #971: Add `toValidatedNel` to `Xor`
- #973: Add `flatMapF` for `StateT`
- #985: Add object `reducible` for reducible syntax
- #996: Add `SemigroupK` instance for `Xor`
- #998: Add `SemigroupK` instance for `Validated`
- #986: Add `Bitraverse` instances for `Validated` and `XorT`

And bug fixes:

- #873: Fix `OptionIdOps.some` to always return `Some`
- #958: Switch off scaladoc generation for Scala 2.10 due to macro problems
- #955: Rename `Id` instances to `idInstances` to make selective import easier

And removals:

- #910: Remove `Streaming` and `StreamingT`

And some other improvements to the documentation, tutorials, laws and tests, including:

- #880: Optimize `Eq[Vector[A]]` instance
- #878: Fix bug in freemonad doc
- #870: Fixed doc string for `StateT`'s `runEmptyA()`
- #866: Add some tests for `Coproduct` and `WriterT`
- #883: Delegate to `Traverse.sequence` in `Applicative.sequence`
- #893: Add `Reducible` laws
- #923: Make `Call.loop` `@tailrec` optimized
- #916: add `-P:scalajs:mapSourceURI` option
- #909: Make `Bifunctor` universal
- #905: make `Unapply` serializable
- #902: Make table in `Kleisli` readable
- #897: Add `Prod` tests
- #938: Onward to scala 2.11.8
- #941: Type class composition and `MonadState` tests

- #949: Add .ensime_cache to gitignore
- #954: Switch to use nodeJsEnv as default jsEnv to build scala.js
- #956: Upgrade scala.js from 0.6.7 -> 0.6.8
- #960: More `Reducible` tests
- #962: Improving test coverage
- #964: Clarify stabilty guarantees; drop 'proof of concept' and 'experimental'
- #972: Fix swapped f and g in `invariant` docs
- #979: Fix outdated import for `cats.syntax.apply._`
- #995: Move coverage away from bash
- #1002: Correct the URL for *Data types à la carte*
- #1005: fix broken link in foldable docs

As always thanks to everyone who filed issues, participated in the Cats Gitter channel, submitted code, or helped review pull requests.

Version 0.4.1

> 2016 February 4

Version 0.4.1 is a patch release in the 0.4 series and is binary compatible with version 0.4.0.

This patch fixes bugs with the `dropWhile` methods on `Streaming` and `StreamingT`.

This release corrects outdated build/POM metadata, which should fix API doc URLS.

Bug fixes:

- #856: Fix `Streaming` and `StreamingT` `dropWhile` functions

Build/publishing changes:

- #852 Update build with org change

Documentation and site improvements:

- #859 Add Contravariant documentation page
- #861 Docs: Revive useful links section. Update URLs

Version 0.4.0

> 2016 February 1

Version 0.4.0 is the fourth release of the Cats library, and the first release published under the `org.typelevel` group from the Typelevel organization on GitHub (previous releases had been published to `org.spire-math` from `non/cats`). This means that users will need to change the `groupId` for their Cats dependencies when updating. If you have a line like this in your SBT build configuration, for example:

```
libraryDependencies += "org.spire-math" %% "cats" % "0.3.0"
```

You will need to change it to the following:

```
libraryDependencies += "org.typelevel" %% "cats" % "0.4.0"
```

This release no longer includes `cats-state` or `cats-free` artifacts, since the `cats.state` and `cats.free` packages have been moved into `cats-core`.

If you've checked out the GitHub repository locally, it would be a good idea to update your remote to point to the new organization, which will typically look like this (note that you should confirm that `origin` is the appropriate remote name):

```
git remote set-url origin git@github.com:typelevel/cats.git
```

This release includes a large number of breaking changes, including most prominently the introduction of a new `Cartesian` type class that is a supertype of `Monad` (and many other types). If you use the `|@|` syntax that had previously been provided by `Apply`, you'll need to change your imports from `cats.syntax.apply._` to `cats.syntax.cartesian._`. For example:

```
scala> import cats.Eval, cats.syntax.cartesian._
import cats.Eval
import cats.syntax.cartesian._

scala> (Eval.now("v") |@| Eval.now(0.4)).tupled
res0: cats.Eval[(String, Double)] = cats.Eval$$anon$5@104f8bbd
```

Other changes in this release are described below.

This version includes API changes:

- #555: `|@|` syntax is now provided by `cats.syntax.cartesian`
- #835: `State` and `StateT` are now in the `cats.data` package
- #781: `combine` on `SemigroupK` is now `combineK`
- #821 and #833: The order of arguments for `ap` has been reversed (now function first)
- #833: `ap` on `CartesianBuilderN` is now `apWith`
- #782: `State` now uses `Eval` instead of `Trampoline` for stack safety
- #697: `or` for natural transformations is now an instance method
- #725: `orElse` on `XorT` and does not unnecessarily constrain the type of the left side of the result
- #648: Some types now extend `Product` and `Serializable` to improve type inference
- #647: `ProdInstancesN` names changed for consistency
- #636: `Eval` is now `Serializable`
- #685: Fixes for copy-paste errors in method names for instances for `Validated`
- #778: Unnecessary type parameter on `Foldable`'s `sequence_` has been removed

And additions:

- #555 and #795: `Cartesian`
- #671: `Coproduct` and `Inject`
- #812: `ApplicativeError`
- #765: `State` and `Free` (and related types) are now in the core module
- #611: `Validated` now has an `andThen` method that provides binding (but without the `for`-comprehension syntactic sugar that the name `flatMap` would bring)
- #796: `sequenceU_` and `traverseU_` on `Foldable`
- #780: `transformS` for `StateT`
- #807: `valueOr` for `XorT`
- #714: `orElse` for `XorT`
- #705: `getOrElseF` for `XorT`
- #731: `swap` for `Validated`
- #571: `transform` and `subflatMap` on `OptionT` and `XorT`
- #757 and #843: `compose` for `Alternative` and `composeK` for `MonoidK`
- #667: `OptionT.liftF`

And removals:

- #613: `Free` and `FreeApplicative` constructors are now private
- #605: `filter` on `Validated`

- #698: `MonadCombine` instances for `OptionT`
- #635: `Kleisli` 's redundant `lmap` , which was equivalent to `local`
- #752: `Cokleisli.cokleisli` , which was equivalent to `Cokleisli.apply`
- #687: Unused `XorTMonadCombine`
- #622: Many prioritization types are now private

And new type class instances:

- #644: `Traverse` and `Foldable` instances for `XorT`
- #691: Various instances for `Function1`
- #628 and #696: Various instances for `WriterT`
- #673: `Bifunctor` instances for `WriterT`
- #715 and #716: `Semigroup` and `Monoid` instances for `Validated`
- #717 and #718: `Semigroup` instances for `Xor` and `Const`
- #818: `CoflatMap` instance for `Vector`
- #626: `Contravariant` instances for `Const` and `Kleisli`
- #621: `Id` instances for `Kleisli`
- #772: `Reducible` instances for `OneAnd`
- #816: `Traverse` instances for `OneAnd`
- #639: `Traverse` instance for `Id`
- #774 and #775: `Show` instances for `Vector` and `Stream`

And bug fixes:

- #623 fixes #563, a bug in the behavior of `dropWhile_` on `Foldable`
- #665 fixes #662, a bug that resulted in re-evaluation after memoization in `Streaming`
- #683 fixes #677, a bug in `Streaming.thunk`
- #801: Fixes order effect bug in `foldMap` on `FreeApplicative`
- #798: Fixes bug in `filter` on `StreamingT`
- #656: Fixes bug in `drop` on `StreamingT`
- #769: Improved stack consumption for `Eval.Call`

And some dependency updates:

- #833: Update to Simulacrum 0.7.0
- #764: 2.10 version is now 2.10.6
- #643: Update to Catalysts 0.2.0
- #727: Update to Scalastyle 0.8.0

There are also many improvements to the documentation, tutorials, laws, tests, and benchmarks, including the following:

- #724: sbt-doctest is now used to validate Scaladoc examples
- #806: Various improvements to use of Simulacrum, which is now a compile-time-only dependency
- #734: Documentation on testing conventions
- #710: Documentation for `Invariant`
- #832: Updated `Free` documentation
- #824: New examples for `Foldable`
- #797: Scaladoc examples for methods on `Arrow`
- #783 and others: Scaladoc examples for syntax methods
- #720: Expanded documentation for `FreeApplicative`
- #636: Law checking for `Eval`
- #649 and #660: Better `Arbitrary` instances for `Streaming` and `StreamingT`
- #722: More consistent `toString` for `StreamingT`
- #672: Additional laws for `Profunctor`
- #668, #669, #679, #680, and #681: Additional law checking for `Xor` , `XorT` , and `Either`
- #707: Additional testing for `State` and `StateT`
- #736: `map` / `flatMap` coherence

- #748: Left and right identity laws for `Kleisli`
- #753: Consistency tests for `Cokleisli`
- #733: Associativity laws for `Kleisli` and `Cokleisli` composition
- #741: Tests for `Unapply` -supported syntax
- #690: Error reporting improvements for serializability tests
- #701: Better documentation for the Travis CI script
- #787: Support for cross-module Scaladoc links

Known issues:

- #702: This change identified and fixed a stack safety bug in `foldMap` on `Free`, but raised other issues (see #712) and was reverted in #713; #721 now tracks the non-stack safety of `Free`'s `foldMap`

As always thanks to everyone who filed issues, participated in the Cats Gitter channel, submitted code, or helped review pull requests.

Version 0.3.0

> 2015 November 8

Version 0.3.0 is the third release of the Cats library.

This version includes new type class instances:

- #545: `Semigroup` instances for `OneAnd`
- #521: `Monoid` instances for `Xor` when the left side has a `Semigroup` instance and the right side has a `Monoid`
- #497: `Monoid` instances for `Set`
- #559: `Bifunctor` instances for `Validated`, `Ior`, `Xor`, and `XorT`
- #569: `Functor` instances for `OptionT` when `F` has a `Functor` instance but not a `Monad`
- #600: `Show` instances for `Option` and `OptionT`
- #601: `Show` instances for `List`
- #602: `Show` instances for `Set`
- #568: Several new `Unapply` shapes

And API changes:

- #592: `fromTryCatch` on `Xor` and `Validated` is now `catchOnly`
- #553: `MonadError` now characterizes type constructors of kind `* -> *` instead of `(*, *) -> *`
- #598: `OneAnd`'s type constructor type parameter is now before the element type
- #610: `XorT`'s `toOption` returns an `OptionT[F, B]` instead of an `F[Option[B]]`
- #518: `Free`'s `resume` method now returns an `Xor` instead of an `Either`
- #575 and #606: `orElse` on `Xor` and `Validated` does not unnecessarily constrain the type of the left side of the result
- #577: `*Aux` helper classes have been renamed `*PartiallyApplied`

And additions:

- #542: `WriterT`
- #567: `Ior.fromOptions`
- #528: `OptionT.fromOption`
- #562: `handleErrorWith` and related helper methods on `MonadError`
- #520: `toNel` and `fromList` conversions from `List` to `NonEmptyList`
- #533: Conversions between types with `Foldable` instances and `Streaming`
- #507: `isJvm` and `isJs` macros in the new `cats.macros.Platform`
- #572: `analyze` on `FreeApplicative` for compilation into a `Monoid`
- #587: Syntax for lifting values (and optional values) into `Validated`

And several aliases:

- #492: `FlatMapSyntax` now includes `followedBy`, which is an alias for `>>`, together with a new `followedByEval`, which allows the caller to choose the evaluation strategy of the second action
- #523: `Foldable` now has a `combineAll` method that aliases `fold` and allows postfix usage via `FoldableSyntax`

And a few removals:

- #524: `FreeApplicative`'s redundant `hoist`, which was equivalent to `compile`
- #531: `Coyoneda`'s `by`
- #612: Many prioritization and instance traits are now private

And bug fixes:

- #547: The `empty` values for `Monoid[Double]` and `Monoid[Float]` are now `0` instead of `1`
- #530: `Streaming.take(n).toList` no longer evaluates the `n + 1`-st element
- #538: `OneAnd`'s instances are properly prioritized

There are also many improvements to the documentation, tutorials, laws, tests, and benchmarks:

- #522: ScalaTest's `===` now uses `Eq` instances
- #502: `Traverse`'s laws verify the consistency of `foldMap` and `traverse`
- #519: Benchmarks (and performance improvements) for `Eval`
- …and many others

Thanks to everyone who filed issues, participated in the Cats Gitter channel, submitted code, or helped review pull requests.

Version 0.2.0

> 2015 August 31

Version 0.2.0 is the second release of the Cats library.

The most exciting feature of this release is Scala.js support, which comes courtesy of much hard work by the Scala.js community (especially Alistair Johnson). The SBT build configuration and project layout were updated to support building for both the JVM and JS platforms.

Since the 0.1.2 release there was wide agreement that the split between `cats-core` and `cats-std` was awkward. The two projects have been combined into `cats-core`, meaning that type class instances for common types like `List` are now available in `cats-core`.

There was also a concerted effort to improve and add documentation to the project. Many people helped find typos, broken links, and places where the docs could be improved. In particular, the following tutorials were added or overhauled:

- `Applicative`
- `Const`
- `Foldable`
- `Free`
- `FreeApplicative`
- `Kleisli`
- `Monad`
- `Monoid`
- `Semigroup`
- `SemigroupK`
- `Traverse`
- `Validated`
- `Xor`

Several new type classes and data types were introduced:

- `Choice[F[_, _]]`
- `Group[A]`
- `MonadReader[F[_, _], R]`
- `Streaming[A]` and `StreamingT[F[_], A]`
- `Prod[F[_], G[_], A]` and `Func[F[_], A, B]`

Syntax tests were added to ensure that existing syntax worked, and there has been some movement to enrich existing types with syntax to make converting them to Cats types easier.

The previous `Fold[A]` type, which was used to support lazy folds, has been replaced with `Eval[A]`. This type supports both strict and lazy evaluation, supports lazy `map` and `flatMap`, and is trampolined for stack safety. The definition of `Foldable#foldRight` has been updated to something much more idiomatic and easier to reason about. The goal is to support laziness in Cats via the `Eval[A]` type wherever possible.

In addition to these specific changes there were numerous small bug fixes, additions, improvements, and updates. Thanks to everyone who filed issues, participated in the Cats Gitter channel, submitted code, or helped review pull requests.

Version 0.1.2

> 2015 July 17

(Due to problems with publishing 0.1.0 and 0.1.1 are incomplete.)

Version 0.1.2 is the first non-snapshot version of the Cats library! It is intended to assist the creation of dependent libraries and to be an early look at Cats' design.

Much of the library is quite mature, but there are no source- or binary-compatibility guarantees at this time. The overarching design of the library is still somewhat in flux, although mostly we expect there will be new type classes, instances, and syntax. Some package and module boundaries may also shift.

For complete credits, see AUTHORS.md for a list of people whose work has made this release possible.