From Function1#apply to Kleisli

Different Ways of Function Composition

https://github.com/hermannhueck/composing-functions

Abstract (1/2)

Fine-grained composability of functions is one of the core advantages of FP.

Treating "Functions as Data" means that we can store, manipulate, pass functions around and compose them in much the same way we do with data.

This talk demonstrates different ways of function composition in Scala.

The focus lies on *scala.Function1*, because due to tupling and currying we can regard any FunctionN (except *Function0*) as a *Function1*. Curried functions are easier to compose.

Starting with the composition methods of *scala.Function1*: *apply*, *compose* and *andThen*, we will investigate folding a *Seq* of functions.

We can also define a pipe operator |> as in F# in order to 'pipe' values through a pipeline of functions.

Abstract (2/2)

Defining a Monoid for *Function1* allows us to combine two or more functions into a new one.

A function can also be seen as a Functor and a Monad. That means: Functions can be mapped and flatMapped over. And we can write for-comprehensions in a *Function1* context just as we do with *List*, *Option*, *Future*, *Either* etc.

Being Monads, we can use functions in any monadic context. We will see that *Function1* **is** the Reader Monad.

The most powerful way of function composition is *Kleisli* (also known as *ReaderT*). We will see that *Kleisli* (defined with the *Id* context) **is** the Reader Monad again.

Agenda

1. Compiler Settings
    - Kind Projector
    - Partial Unification
2. Functions as Data
3. Tupling and Currying Functions
4. *Function1*: *apply*, *compose*, *andThen*
5. Piping as in F#
6. Monoidal Function Composition
7. *Function1* as Functor
8. *Function1* as Monad
9. Kleisli Composition - hand-weaved
10. *case class Kleisli*
11. Reader Monad
12. Resources

1. Compiler Settings

1.1 Kind Projector

1.2 Partial Unification

To compile the subsequent code examples, kind-projector and partial-unification must be enabled in *build.sbt*:

```
scalacOptions += "-Ypartial-unification"

addCompilerPlugin("org.spire-math" %% "kind-projector" % "0.9.7")
```

1.1 Kind Projector - Compiler Plugin

Enable *kind-projector* in *build.sbt*:

```
addCompilerPlugin("org.spire-math" %% "kind-projector" % "0.9.7")
```

Kind projector on Github:
https://github.com/non/kind-projector

See: *demo.Demo01aKindProjector*

The Problem:

How to define a Functor or a Monad for an ADT with two type parameters?

The Problem:

How to define a Functor or a Monad for an ADT with two type parameters?

A Functor is defined like this:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

It is parameterized with a generic type constructor F[_] which "has one hole", i.e. F[_] needs another type parameter when reified.

The Problem:

How to define a Functor or a Monad for an ADT with two type parameters?

A Functor is defined like this:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

It is parameterized with a generic type constructor F[_] which "has one hole", i.e. F[_] needs another type parameter when reified.

Hence it is easy to define a Functor instance for *List*, *Option* or *Future*, ADTs which expect <u>exactly one type parameter</u> that "fills that hole".

The Problem:

How to define a Functor or a Monad for an ADT with two type parameters?

A Functor is defined like this:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

It is parameterized with a generic type constructor F[_] which "has one hole", i.e. F[_] needs another type parameter when reified.

Hence it is easy to define a Functor instance for *List*, *Option* or *Future*, ADTs which expect exactly one type parameter that "fills that hole".

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

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

The Problem:

How to define a Functor or a Monad for an ADT with two type parameters?

A Functor is defined like this:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

It is parameterized with a generic type constructor F[_] which "has one hole", i.e. F[_] needs another type parameter when reified.

Hence it is easy to define a Functor instance for *List*, *Option* or *Future*, ADTs which expect exactly one type parameter that "fills that hole".

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

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

What if we want to define a Functor for *Either*?

What if we want to define a Functor for *Either*?

Either has kind: * --> * --> *. It has two holes and hence cannot be used easily to define a Functor or a Monad instance.

What if we want to define a Functor for *Either*?

Either has kind: * --> * --> *. It has two holes and hence cannot be used easily to define a Functor or a Monad instance.

But we can fix one of the type parameters and leave the other one open.

```
// Code compiles without kind-projector.
// It uses a type alias within a structural type.
implicit def eitherFunctor[L]: Functor[({type f[x] = Either[L, x]})#f] =
  new Functor[({type f[x] = Either[L, x]})#f] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

What if we want to define a Functor for *Either*?

Either has kind: * --> * --> *. It has two holes and hence cannot be used easily to define a Functor or a Monad instance.

But we can fix one of the type parameters and leave the other one open.

```
// Code compiles without kind-projector.
// It uses a type alias within a structural type.
implicit def eitherFunctor[L]: Functor[({type f[x] = Either[L, x]})#f] =
  new Functor[({type f[x] = Either[L, x]})#f] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

This is a type lambda (analogous to a partially applied function on the value level).

The type alias must be defined inside *def eitherFunctor[L]* because the type parameter *L* is used in the type alias. This is done inside a structural type where *f* is returned through a type projection.

```
Functor[({type f[x] = Either[L, x]})#f]
```

This code is ugly but can be improved if *kind-projector* is enabled.

Without *kind-projector*:

```
implicit def eitherFunctor[L]: Functor[({type f[x] = Either[L, x]})#f] =
  new Functor[({type f[x] = Either[L, x]})#f] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

Without *kind-projector*:

```
implicit def eitherFunctor[L]: Functor[({type f[x] = Either[L, x]})#f] =
  new Functor[({type f[x] = Either[L, x]})#f] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

With *kind-projector*:

```
implicit def eitherFunctor[L]: Functor[Lambda[x => Either[L, x]]] =
  new Functor[Lambda[x => Either[L, x]]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

Without *kind-projector*:

```
implicit def eitherFunctor[L]: Functor[({type f[x] = Either[L, x]})#f] =
  new Functor[({type f[x] = Either[L, x]})#f] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

With *kind-projector*:

```
implicit def eitherFunctor[L]: Functor[Lambda[x => Either[L, x]]] =
  new Functor[Lambda[x => Either[L, x]]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

```
implicit def eitherFunctor[L]: Functor[λ[x => Either[L, x]]] =
  new Functor[λ[x => Either[L, x]]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

Without *kind-projector*:

```
implicit def eitherFunctor[L]: Functor[({type f[x] = Either[L, x]})#f] =
  new Functor[({type f[x] = Either[L, x]})#f] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

With *kind-projector*:

```
implicit def eitherFunctor[L]: Functor[Lambda[x => Either[L, x]]] =
  new Functor[Lambda[x => Either[L, x]]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

```
implicit def eitherFunctor[L]: Functor[λ[x => Either[L, x]]] =
  new Functor[λ[x => Either[L, x]]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

```
implicit def eitherFunctor[L]: Functor[Either[L, ?]] =
  new Functor[Either[L, ?]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] = fa map f
  }
```

1.2 Compiler Flag *-Ypartial-unification*

Enable partial unification in *build.sbt*:

```
scalacOptions += "-Ypartial-unification"
```

See: *demo.Demo01bPartialUnification*

The Problem:

```
def foo[F[_], A](fa: F[A]): String = fa.toString

foo { x: Int => x * 2 }



-------
[error] no type parameters for method foo: (fa: F[A])String exist
        so that it can be applied to arguments (Int => Int)
[error]  --- because ---
[error] argument expression's type is not compatible with formal parameter type;
[error]  found   : Int => Int
[error]  required: ?F[?A]
[error]   foo { x: Int => x * 2 }
[error]   ^
[error] type mismatch;
[error]  found   : Int => Int
[error]  required: F[A]
[error]   foo((x: Int) => x * 2)
```

The Problem:

```
def foo[F[_], A](fa: F[A]): String = fa.toString

foo { x: Int => x * 2 }


-------
[error] no type parameters for method foo: (fa: F[A])String exist
        so that it can be applied to arguments (Int => Int)
[error]  --- because ---
[error] argument expression's type is not compatible with formal parameter type;
[error]  found   : Int => Int
[error]  required: ?F[?A]
[error]   foo { x: Int => x * 2 }
[error]   ^
[error] type mismatch;
[error]  found   : Int => Int
[error]  required: F[A]
[error]   foo((x: Int) => x * 2)
```

This code doesn't compile without -*Ypartial-unification*. **Why?**

The Problem:

```
def foo[F[_], A](fa: F[A]): String = fa.toString

foo { x: Int => x * 2 }


-------
[error] no type parameters for method foo: (fa: F[A])String exist
        so that it can be applied to arguments (Int => Int)
[error]  --- because ---
[error] argument expression's type is not compatible with formal parameter type;
[error]  found   : Int => Int
[error]  required: ?F[?A]
[error]   foo { x: Int => x * 2 }
[error]   ^
[error] type mismatch;
[error]  found   : Int => Int
[error]  required: F[A]
[error]   foo((x: Int) => x * 2)
```

This code doesn't compile without *-Ypartial-unification*. **Why?**

def foo requires a type constructor F[_] with "one hole". It's kind is: * --> *.

foo is invoked with a function *Int* => *Int*. *Int* => *Int* is syntactic sugar for *Function1[Int, Int]*. *Function1* like *Either* has two holes. It's kind is: * --> * --> *.

The Solution:

-*Ypartial-unification* solves the problem by partially fixing (unifying) the type parameters <u>from left to right</u> until it fits to the number of holes required by the definition of *foo*.

The Solution:

-Ypartial-unification solves the problem by partially fixing (unifying) the type parameters from left to right until it fits to the number of holes required by the definition of *foo*.

Imagine this flag turns *Function1[A, B]* into *Function1Int[B]*. With this fix on the fly *Function1Int* has kind * --> * which is the kind required by F[_]. What the compiler transforms the invocation to would look something like this:

```
def foo[F[_], A](fa: F[A]): String = fa.toString

foo[Function1Int, Int] { x: Int => x * 2 }
```

The Solution:

-*Ypartial-unification* solves the problem by partially fixing (unifying) the type parameters from left to right until it fits to the number of holes required by the definition of *foo*.

Imagine this flag turns *Function1[A, B]* into *Function1Int[B]*. With this fix on the fly *Function1Int* has kind * --> * which is the kind required by F[_]. What the compiler transforms the invocation to would look something like this:

```
def foo[F[_], A](fa: F[A]): String = fa.toString

foo[Function1Int, Int] { x: Int => x * 2 }
```

Note that the partial unification fixes the types always in a left-to-right order which is a good fit for most cases where we have right-biased types like *Either*, *Tuple2* or *Function1*. (It is not a good fit for the very rare cases when you use a left-biased type like Scalactic's *Or* data type (a left-biased *Either*).)

When to use *kind-projector* and *-Ypartial-unification*?

When to use *kind-projector* and *-Ypartial-unification*?

Enable plugin and the flag when using a Functor, Applicative or a Monad instance for higher-kinded types which take more than one type parameter. *Either*, *Tuple2* and *Function1* are the best known representatives of this kind of types.

When programming on the type level regard both as your friends and keep them enabled.

When to use *kind-projector* and *-Ypartial-unification*?

Enable plugin and the flag when using a Functor, Applicative or a Monad instance for higher-kinded types which take more than one type parameter. *Either*, *Tuple2* and *Function1* are the best known representatives of this kind of types.

When programming on the type level regard both as your friends and keep them enabled.

Very good explanations of partial unification by Miles Sabin and Daniel Spiewak can be found at these links:

https://github.com/scala/scala/pull/5102
https://gist.github.com/djspiewak/7a81a395c461fd3a09a6941d4cd040f2

2. Functions as Data

Treating Functions as Data (1/2)

allows us to …

Treating Functions as Data (1/2)

allows us to ...

- store a function in a val

```
val str2Int: String => Int = str => str.toInt    // Function1[String, Int]
```

Treating Functions as Data (1/2)

allows us to ...

- store a function in a val

```
val str2Int: String => Int = str => str.toInt    // Function1[String, Int]
```

- pass it as arg to other (higher order) functions (HOFs)

```
val mapped = List("1", "2", "3").map(str2Int)
```

Treating Functions as Data (1/2)

allows us to ...

- store a function in a val

```
val str2Int: String => Int = str => str.toInt    // Function1[String, Int]
```

- pass it as arg to other (higher order) functions (HOFs)

```
val mapped = List("1", "2", "3").map(str2Int)
```

- return a function from other (higher order) functions (HOFs)

```
val plus100: Int => Int = { i =>
  i + 100
}
```

Treating Functions as Data (1/2)

allows us to ...

- store a function in a val

```
val str2Int: String => Int = str => str.toInt    // Function1[String, Int]
```

- pass it as arg to other (higher order) functions (HOFs)

```
val mapped = List("1", "2", "3").map(str2Int)
```

- return a function from other (higher order) functions (HOFs)

```
val plus100: Int => Int = { i =>
  i + 100
}
```

- process/manipulate a function like data

```
Option(5) map plus100
str2Int   map plus100     // Functor instance for Function1 must be defined
```

Treating Functions as Data (2/2)

allows us to ...

Treating Functions as Data (2/2)

allows us to ...

- organize functions (like data) in data structures like List, Option etc.

```
val functions: List[Int => Int] = List(_ + 1, _ + 2, _ + 3)
val f[Int => Int] = functions.foldLeft(...)(...)
```

Treating Functions as Data (2/2)

allows us to ...

- organize functions (like data) in data structures like List, Option etc.

```
val functions: List[Int => Int] = List(_ + 1, _ + 2, _ + 3)
val f[Int => Int] = functions.foldLeft(...)(...)
```

- wrap a function in a class / case class *Wrapper*

We can define methods on *Wrapper* which transform the wrapped function and return the transformation result again wrapped in a new instance of *case class Wrapper*.

```
case class Wrapper[A, B](run: A => B) {
  def transform[C](f: B => C): Wrapper[A, C] = Wrapper { a => f(run(a)) }
  // more methods ...
}

val f1: String => Int = _.toInt
val wrapper = Wrapper(f1)
val f2: Int => Int = _ * 2
val wrapper2 = wrapper.transform(f2)
wrapper2.run("5") // 10
```

3. Tupling and Currying Functions

See: *demo.Demo03aTupledFunctions*
demo.Demo03bCurriedFunctions

Every function is a *Function1* ...

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int          = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
val res0 = sum3Ints(1,2,3) // 6
```

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int         = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
val res0 = sum3Ints(1,2,3) // 6
```

... if you tuple up it's parameters.

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int        = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
val res0 = sum3Ints(1,2,3) // 6
```

... if you tuple up it's parameters.

```
val sumTupled: ((Int, Int, Int)) => Int = sum3Ints.tupled
// sumTupled: ((Int, Int, Int)) => Int = scala.Function3$$Lambda$1018/1492801385@4
val sumTupled2: Function1[(Int, Int, Int), Int] = sum3Ints.tupled
// sumTupled2: ((Int, Int, Int)) => Int = scala.Function3$$Lambda$1018/1492801385@
val resTupled = sumTupled((1,2,3)) // 6
val resTupled2 = sumTupled2((1,2,3)) // 6
```

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int         = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
val res0 = sum3Ints(1,2,3) // 6
```

... if you tuple up it's parameters.

```
val sumTupled: ((Int, Int, Int)) => Int = sum3Ints.tupled
// sumTupled: ((Int, Int, Int)) => Int = scala.Function3$$Lambda$1018/1492801385@4
val sumTupled2: Function1[(Int, Int, Int), Int] = sum3Ints.tupled
// sumTupled2: ((Int, Int, Int)) => Int = scala.Function3$$Lambda$1018/1492801385@
val resTupled = sumTupled((1,2,3)) // 6
val resTupled2 = sumTupled2((1,2,3)) // 6
```

If untupled again, we get the original function back.

```
val sumUnTupled: (Int, Int, Int) => Int = Function.untupled(sumTupled)
// sumUnTupled: (Int, Int, Int) => Int = scala.Function$$$Lambda$3583/1573807728@7
val resUnTupled = sumUnTupled(1,2,3) // 6
```

Every function is a *Function1* …

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int          = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
```

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int         = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
```

... if you curry it.

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int          = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int         = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int         = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

The type arrow (=>) is syntactic sugar for *Function1*.
A => *B* is equivalent to *Function1[A, B]*.

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int         = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

The type arrow (=>) is syntactic sugar for *Function1*.
A => *B* is equivalent to *Function1[A, B]*.

```
val sumCurried2: Function1[Int, Function1[Int, Function1[Int, Int]]] = sumCurried
// sumCurried2: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/991430
```

Every function is a *Function1* ...

```
val sum3Ints : (Int, Int, Int) => Int        = _ + _ + _
val sum3Ints2: Function3[Int, Int, Int, Int] = _ + _ + _
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

The type arrow (=>) is syntactic sugar for *Function1*.
A => *B* is equivalent to *Function1[A, B]*.

```
val sumCurried2: Function1[Int, Function1[Int, Function1[Int, Int]]] = sumCurried
// sumCurried2: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/991430
```

If uncurried again, we get the original function back.

```
val sumUncurried: (Int, Int, Int) => Int = Function.uncurried(sumCurried)
// sumUncurried: (Int, Int, Int) => Int = scala.Function$$$Lambda$6605/301079867@1
```

Partial application of curried functions

Partial application of curried functions

```scala
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```
val applied1st: Int => Int => Int = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```
val applied1st: Int => Int => Int = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

```
val applied2nd: Int => Int = applied1st(2)
// applied2nd: Int => Int = scala.Function3$$Lambda$4349/402963549@117e96fb
```

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```
val applied1st: Int => Int => Int = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

```
val applied2nd: Int => Int = applied1st(2)
// applied2nd: Int => Int = scala.Function3$$Lambda$4349/402963549@117e96fb
```

```
val applied3rd: Int = applied2nd(3)
// applied3rd: Int = 6
```

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```
val applied1st: Int => Int => Int = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

```
val applied2nd: Int => Int = applied1st(2)
// applied2nd: Int => Int = scala.Function3$$Lambda$4349/402963549@117e96fb
```

```
val applied3rd: Int = applied2nd(3)
// applied3rd: Int = 6
```

```
val appliedAllAtOnce: Int = sumCurried(1)(2)(3)
// appliedAllAtOnce: Int = 6
```

Advantages of curried functions

Advantages of curried functions

1. Curried functions can be partially applied.
   They are <u>easier to compose</u> than their uncurried counterparts.
   They allow for a more fine-grained composition. (very useful when working with Appicatives, which is not part of this talk.)

Advantages of curried functions

1. Curried functions can be partially applied.
   They are easier to compose than their uncurried counterparts.
   They allow for a more fine-grained composition. (very useful when working with Appicatives, which is not part of this talk.)

2. Curried functions (and methods) help the compiler with type inference.
   The compiler infers types by argument lists from left to right.

Advantages of curried functions

1. Curried functions can be partially applied.
   They are easier to compose than their uncurried counterparts.
   They allow for a more fine-grained composition. (very useful when working with Appicatives, which is not part of this talk.)

2. Curried functions (and methods) help the compiler with type inference.
   The compiler infers types by argument lists from left to right.

```
def filter1[A](la: List[A], p: A => Boolean) = ??? // uncurried

scala> filter1(List(0,1,2), _ < 2)
<console>:50: error: missing parameter type for expanded function ((x$1: <error>) 
       filter1(List(0,1,2), _ < 2)
                            ^
```

Advantages of curried functions

1. Curried functions can be partially applied.
   They are easier to compose than their uncurried counterparts.
   They allow for a more fine-grained composition. (very useful when working with Appicatives, which is not part of this talk.)

2. Curried functions (and methods) help the compiler with type inference.
   The compiler infers types by argument lists from left to right.

```scala
def filter1[A](la: List[A], p: A => Boolean) = ??? // uncurried

scala> filter1(List(0,1,2), _ < 2)
<console>:50: error: missing parameter type for expanded function ((x$1: <error>)
       filter1(List(0,1,2), _ < 2)
                            ^
```

```scala
def filter2[A](la: List[A])(p: A => Boolean) = ??? // curried

scala> filter2(List(0,1,2))(_ < 2)
res5: List[Int] = List(0,1)
```

4. *Function1: apply, compose, andThen*

See: *demo.Demo04ComposingFunctions*

Function1#apply

- The basic form of function composition is function application.
- Two functions *f* and *g* can be composed by applying *g* to the result of the application of *f*.

Function1#apply

- The basic form of function composition is function application.
- Two functions *f* and *g* can be composed by applying *g* to the result of the application of *f*.

```
val f: Int => Int = _ + 10
val g: Int => Int = _ * 2

val h: Int => Int = x => g(f(x))

h(1) // 22
```

Function1#apply

- The basic form of function composition is function application.
- Two functions *f* and *g* can be composed by applying *g* to the result of the application of *f*.

```
val f: Int => Int = _ + 10
val g: Int => Int = _ * 2

val h: Int => Int = x => g(f(x))

h(1) // 22
```

```
val f: A => B = ???
val g: B => C = ???

val h: A => C = x => g(f(x))   // x => g.apply(f.apply(x))
```

- The parameter type of *g* must be compliant with the result type of *f*.

Trait *Function1*

```scala
trait Function1[-T, +R] {

  def apply(a: T): R
  def compose[A](g: A => T): A => R = { x => apply(g(x)) }
  def andThen[A](g: R => A): T => A = { x => g(apply(x)) }
  override def toString = "<function1>"
}
```

Trait *Function1*

```scala
trait Function1[-T, +R] {

  def apply(a: T): R
  def compose[A](g: A => T): A => R = { x => apply(g(x)) }
  def andThen[A](g: R => A): T => A = { x => g(apply(x)) }
  override def toString = "<function1>"
}
```

```scala
val f: Int => Int = _ + 10
val g: Int => Int = _ * 2

(f compose g)(1) == f(g(1))            // true
(f andThen g)(1) == g(f(1))            // true
(f andThen g)(1) == (g compose f)(1)   // true
```

A pipeline of functions

A pipeline of functions

```scala
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString
```

A pipeline of functions

```scala
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString
```

```scala
// Function1#apply
val fComposed1: String => String = str => d2s(div10By(plus2(s2i(str))))
val res1 = fComposed1("3") // 2.0 !!!
```

A pipeline of functions

```scala
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString
```

```scala
// Function1#apply
val fComposed1: String => String = str => d2s(div10By(plus2(s2i(str))))
val res1 = fComposed1("3") // 2.0 !!!
```

```scala
// Function1#compose
val fComposed2: String => String = d2s compose div10By compose plus2 compose s2i
val res2 = fComposed2("3") // 2.0 !!!
```

A pipeline of functions

```scala
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString
```

```scala
// Function1#apply
val fComposed1: String => String = str => d2s(div10By(plus2(s2i(str))))
val res1 = fComposed1("3") // 2.0 !!!
```

```scala
// Function1#compose
val fComposed2: String => String = d2s compose div10By compose plus2 compose s2i
val res2 = fComposed2("3") // 2.0 !!!
```

```scala
// Function1#andThen
val fComposed3: String => String = s2i andThen plus2 andThen div10By andThen d2s
val res3 = fComposed3("3") // 2.0 !!!
```

Folding/Chaining a Seq of Functions

Folding/Chaining a Seq of Functions

```
val fs: Seq[Int => Int] = Seq(_*2, _+10, _+100)
```

Folding/Chaining a Seq of Functions

```
val fs: Seq[Int => Int] = Seq(_*2, _+10, _+100)
```

```
val fFolded = fs.foldLeft(identity[Int] _) { (acc, f) => acc andThen f }
println(fFolded(1)) // 112
```

Folding/Chaining a Seq of Functions

```
val fs: Seq[Int => Int] = Seq(_*2, _+10, _+100)
```

```
val fFolded = fs.foldLeft(identity[Int] _) { (acc, f) => acc andThen f }
println(fFolded(1)) // 112
```

The *identity* function is the no-op of function composition.

Folding/Chaining a Seq of Functions

```
val fs: Seq[Int => Int] = Seq(_*2, _+10, _+100)
```

```
val fFolded = fs.foldLeft(identity[Int] _) { (acc, f) => acc andThen f }
println(fFolded(1)) // 112
```

The *identity* function is the no-op of function composition.

This functionality is already provided in the Scala standard library in:
Function.chain[a](fs: Seq[a => a]): a => a

Folding/Chaining a Seq of Functions

```
val fs: Seq[Int => Int] = Seq(_*2, _+10, _+100)
```

```
val fFolded = fs.foldLeft(identity[Int] _) { (acc, f) => acc andThen f }
println(fFolded(1)) // 112
```

The *identity* function is the no-op of function composition.

This functionality is already provided in the Scala standard library in:
Function.chain[a](fs: Seq[a => a]): a => a

```
val fChained = Function.chain(fs)
println(fChained(1)) // 112
```

5. Piping as in F

See: *demo.Demo05PipeApp*

Piping as in F

When you apply a function f to an argument x you write: ***f(x)***

In F# it is very common to pipe the value into the function.
Then you write: ***x |> f***

Piping as in F

When you apply a function f to an argument x you write: ***f(x)***

In F# it is very common to pipe the value into the function.
Then you write: ***x |> f***

We can provide this functionality in Scala (using an implicit conversion):

Piping as in F

When you apply a function f to an argument x you write: ***f(x)***

In F# it is very common to pipe the value into the function.
Then you write: ***x |> f***

We can provide this functionality in Scala (using an implicit conversion):

```
object Pipe {
  implicit class PipeForA[A](a: A) {
    def pipe[B](f: A => B): B = f(a)
    def |>[B](f: A => B): B = f(a) // F#'s |> operator
  }
}
```

```
import Pipe._

val squared: Int => Int = x => x * x

println(5.pipe(squared)) // 25
println(5 pipe squared)  // 25
println(5.|>(squared))   // 25
println(5 |> squared)    // 25
```

Building a pipeline

```scala
import Pipe._

val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString + " !!!"


// Using regular Scala function invocation
val res1 = d2s(div10By(plus2(s2i("3")))) // 2.0 !!!


// Using pipe
val res2a = "3".pipe(s2i).pipe(plus2).pipe(div10By).pipe(d2s) // 2.0 !!!

val res2b = "3" pipe s2i pipe plus2 pipe div10By pipe d2s // 2.0 !!!

// Using F#'s pipe idiom
val res2c = "3" |> s2i |> plus2 |> div10By |> d2s // 2.0 !!!


// Using Function1#andThen
val res3 = (s2i andThen plus2 andThen div10By andThen d2s)("3") // 2.0 !!!

// Using Function1#compose
val res4 = (d2s compose div10By compose plus2 compose s2i)("3") // 2.0 !!!
```

Recommendation

F#'s pipe idiom is quite elegant.

Using it intensely would change your Scala code a lot.

Recommendation

F#'s pipe idiom is quite elegant.

Using it intensely would change your Scala code a lot.

Don't use it!

It is idiomatic F#, not idiomatic Scala.

Other Scala programmers (not used to it) would probably not recognize what your code is doing.

6. Monoidal Function Composition

See: *demo.Demo06ComposingWithMonoid*

Trait Monoid

```
trait Monoid[A] {

  def empty: A
  def combine(x: A, y: A): A

  def combineAll(as: List[A]): A =    // combines all functions in a List of funct
    as.foldLeft(empty)(combine)
}
```

Default Monoid Instance for *Function1*

```scala
// This one is the default Function1-Monoid in Cats.
// It requires the result type B to be a Monoid too.
//
implicit def function1Monoid[A, B: Monoid]: Monoid[A => B] = new Monoid[A => B] {

  override def empty: A => B =
    _ => Monoid[B].empty
  override def combine(f: A => B, g: A => B): A => B =
    a => Monoid[B].combine(f(a), g(a))
}
```

Default Monoid Instance for *Function1*

```
// This one is the default Function1-Monoid in Cats.
// It requires the result type B to be a Monoid too.
//
implicit def function1Monoid[A, B: Monoid]: Monoid[A => B] = new Monoid[A => B] {

  override def empty: A => B =
    _ => Monoid[B].empty
  override def combine(f: A => B, g: A => B): A => B =
    a => Monoid[B].combine(f(a), g(a))
}
```

- This instance defines a Monoid for Functions of type *A* => *B*.
- It requires type *B* to be a Monoid too.
- *empty* defines a function which ignores it's input and returns *Monoid[B].empty*.
- *combine* takes two functions *f* and *g*, invokes *f(a)* and *g(a)* on it's input and returns the combined result.

Composition with Monoid

```
val f: Int => Int = _ + 1
val g: Int => Int = _ * 2
val h: Int => Int = _ + 100
```

Composition with Monoid

```
val f: Int => Int = _ + 1
val g: Int => Int = _ * 2
val h: Int => Int = _ + 100
```

```
import Monoid.function1Monoid

(f combine g)(4)                                // 13
(f |+| g)(4)                                    // 13

(f |+| g |+| h)(4)                              // 117
Monoid[Int => Int].combineAll(List(f, g, h))(4)  // 117
```

Composition with Monoid

```
val f: Int => Int = _ + 1
val g: Int => Int = _ * 2
val h: Int => Int = _ + 100
```

```
import Monoid.function1Monoid

(f combine g)(4)                                 // 13
(f |+| g)(4)                                     // 13

(f |+| g |+| h)(4)                               // 117
Monoid[Int => Int].combineAll(List(f, g, h))(4)  // 117
```

Other Monoid instances for *Function1* yield different results.

Instance for *function1ComposeMonoid*

```
implicit def function1ComposeMonoid[A]: Monoid[A => A] = new Monoid[A => A] {
  override def empty: A => A = identity
  override def combine(f: A => A, g: A => A): A => A = f compose g
}
```

Instance for *function1ComposeMonoid*

```
implicit def function1ComposeMonoid[A]: Monoid[A => A] = new Monoid[A => A] {
  override def empty: A => A = identity
  override def combine(f: A => A, g: A => A): A => A = f compose g
}
```

- This instance composes the Functions with *Function1#compose*.
- It works only for functions of type *A* => *A*. (Input and output type are the same type.)

Instance for *function1ComposeMonoid*

```
implicit def function1ComposeMonoid[A]: Monoid[A => A] = new Monoid[A => A] {
  override def empty: A => A = identity
  override def combine(f: A => A, g: A => A): A => A = f compose g
}
```

- This instance composes the Functions with *Function1#compose*.
- It works only for functions of type *A* => *A*. (Input and output type are the same type.)

```
import Monoid.function1ComposeMonoid

(f combine g)(4)                               // 9
(f |+| g)(4)                                   // 9

(f |+| g |+| h)(4)                             // 209
Monoid[Int => Int].combineAll(List(f, g, h))(4)  // 209
```

Instance for *function1AndThenMonoid*

```scala
implicit def function1AndThenMonoid[A]: Monoid[A => A] = new Monoid[A => A] {
  override def empty: A => A = identity
  override def combine(f: A => A, g: A => A): A => A = f andThen g
}
```

Instance for *function1AndThenMonoid*

```
implicit def function1AndThenMonoid[A]: Monoid[A => A] = new Monoid[A => A] {
  override def empty: A => A = identity
  override def combine(f: A => A, g: A => A): A => A = f andThen g
}
```

- This instance composes the Functions with *Function1#andThen*.
- It works only for functions of type *A* => *A*. (Input and output type are the same type.)

Instance for *function1AndThenMonoid*

```scala
implicit def function1AndThenMonoid[A]: Monoid[A => A] = new Monoid[A => A] {
  override def empty: A => A = identity
  override def combine(f: A => A, g: A => A): A => A = f andThen g
}
```

- This instance composes the Functions with *Function1#andThen*.
- It works only for functions of type *A* => *A*. (Input and output type are the same type.)

```scala
import Monoid.function1AndThenMonoid

(f combine g)(4)                                // 10
(f |+| g)(4)                                    // 10

(f |+| g |+| h)(4)                              // 110
Monoid[Int => Int].combineAll(List(f, g, h))(4)  // 110
```

7. *Function1* as Functor

See: *demo.Demo07Functor*

Functor

A Functor is any Context F[_] that provides a function *map* ...
and abides by the Functor laws (not presented here).

```
trait Functor[F[_]] {

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

Functor instance

A Functor instance for *Functions* (found automatically by the compiler if defined in implicit scope, i.e inside the *Functor* companion object):

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

Functor instance

A Functor instance for *Functions* (found automatically by the compiler if defined in implicit scope, i.e inside the *Functor* companion object):

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

A Functor instance for *Either* (just for comparison):

```
implicit def eitherFunctor[L]: Functor[Either[L, ?]] =
  new Functor[Either[L, ?]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] =
      fa map f
  }
```

Functor instance

A Functor instance for *Functions* (found automatically by the compiler if defined in implicit scope, i.e inside the *Functor* companion object):

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

A Functor instance for *Either* (just for comparison):

```
implicit def eitherFunctor[L]: Functor[Either[L, ?]] =
  new Functor[Either[L, ?]] {
    override def map[A, B](fa: Either[L, A])(f: A => B): Either[L, B] =
      fa map f
  }
```

Using the *Function1* Functor:

```
val f: Int => Int = _ + 3
val g: Int => Int = _ * 2
val h = Functor[Function1[Int, ?]].map(f)(g)
val res = h(2)  // 10
```

Functor syntax

defined as implicit conversion …

Functor syntax

defined as implicit conversion ...

in a specific way for *Functor[Function1]*:

```
implicit class FunctorSyntaxFunction1[P, A](fa: Function1[P, A]) {
  def map[B](f: A => B): Function1[P, B] = Functor[Function1[P, ?]].map(fa)(f)
}
```

Functor syntax

defined as implicit conversion ...

in a specific way for *Functor[Function1]*:

```
implicit class FunctorSyntaxFunction1[P, A](fa: Function1[P, A]) {
  def map[B](f: A => B): Function1[P, B] = Functor[Function1[P, ?]].map(fa)(f)
}
```

or in a generic way for any *Functor[F[_]]*:

```
implicit class FunctorSyntax[F[_]: Functor, A](fa: F[A]) {
  def map[B](f: A => B): F[B] = Functor[F].map(fa)(f)
}
```

Functor syntax

defined as implicit conversion ...

in a specific way for *Functor[Function1]*:

```
implicit class FunctorSyntaxFunction1[P, A](fa: Function1[P, A]) {
  def map[B](f: A => B): Function1[P, B] = Functor[Function1[P, ?]].map(fa)(f)
}
```

or in a generic way for any *Functor[F[_]]*:

```
implicit class FunctorSyntax[F[_]: Functor, A](fa: F[A]) {
  def map[B](f: A => B): F[B] = Functor[F].map(fa)(f)
}
```

This allows for convenient invocation of *map*
as if *map* were a method of *Function1*:

```
val f: Int => Int = _ + 3
val g: Int => Int = _ * 2
val h = f map g
val res = h(2)  // 10
```

A pipeline of functions

A pipeline of functions

```
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString
```

A pipeline of functions

```scala
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString
```

composed with *map*:

```scala
val fMapped = s2i map plus2 map div10By map d2s // requires -Ypartial-unification
val res1 = fMapped("3") // 2.0 !!!
```

A pipeline of functions

```
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString
```

composed with *map*:

```
val fMapped = s2i map plus2 map div10By map d2s // requires -Ypartial-unification
val res1 = fMapped("3") // 2.0 !!!
```

Function1 can also be seen as a Monad ...

8. *Function1* as Monad

See: *demo.Demo08aMonad*

Monad

A Monad is any Context F[_] that provides the functions *pure* and *flatMap* ...
and abides by the Monad laws (not presented here).

```
trait Monad[F[_]] extends Functor[F] {

  def pure[A](a: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
```

Monad

A Monad is any Context F[_] that provides the functions *pure* and *flatMap* ...
and abides by the Monad laws (not presented here).

```
trait Monad[F[_]] extends Functor[F] {

  def pure[A](a: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
```

```
  // map in terms of flatMap and pure
  override def map[A, B](fa: F[A])(f: A => B): F[B] =
          flatMap(fa)(a => pure(f(a)))

  // flatten in terms of flatMap and identity
  def flatten[A](ffa: F[F[A]]): F[A] =
          flatMap(ffa)(identity)     // flatMap(ffa)(x => x)
}
```

Monad instance

A Monad instance for *Functions* (found automatically by the compiler if defined in implicit scope, i.e inside the *Monad* companion object):

```
implicit def function1Monad[P]: Monad[P => ?] = new Monad[P => ?] {
  override def pure[A](r: A): P => A = _ => r
  override def flatMap[A, B](f: P => A)(g: A => P => B)
    : P => B = p => g(f(p))(p)
}
```

Monad instance

A Monad instance for *Functions* (found automatically by the compiler if defined in implicit scope, i.e inside the *Monad* companion object):

```
implicit def function1Monad[P]: Monad[P => ?] = new Monad[P => ?] {
  override def pure[A](r: A): P => A = _ => r
  override def flatMap[A, B](f: P => A)(g: A => P => B)
    : P => B = p => g(f(p))(p)
}
```

Alternative instance definition:

```
implicit def function1Monad[P]: Monad[Function1[P, ?]] = new Monad[Function1[P, ?]
  override def pure[A](r: A): Function1[P, A] = _ => r
  override def flatMap[A, B](f: Function1[P, A])(g: A => Function1[P, B])
    : Function1[P, B] = p => g(f(p))(p)
}
```

Monad instance

A Monad instance for *Functions* (found automatically by the compiler if defined in implicit scope, i.e inside the *Monad* companion object):

```
implicit def function1Monad[P]: Monad[P => ?] = new Monad[P => ?] {
  override def pure[A](r: A): P => A = _ => r
  override def flatMap[A, B](f: P => A)(g: A => P => B)
    : P => B = p => g(f(p))(p)
}
```

Alternative instance definition:

```
implicit def function1Monad[P]: Monad[Function1[P, ?]] = new Monad[Function1[P, ?]
  override def pure[A](r: A): Function1[P, A] = _ => r
  override def flatMap[A, B](f: Function1[P, A])(g: A => Function1[P, B])
    : Function1[P, B] = p => g(f(p))(p)
}
```

A Monad instance for *Either* (just for comparison):

```
implicit def eitherMonad[L]: Monad[Either[L, ?]] = new Monad[Either[L, ?]] {
  override def pure[A](r: A): Either[L, A] = Right(r)
  override def flatMap[A, B](fa: Either[L, A])(f: A => Either[L, B])
    : Either[L, B] = fa flatMap f
}
```

FlatMap syntax

defined as implicit conversion …

FlatMap syntax

defined as implicit conversion ...

in a specific way for *Monad[Function1]*:

```
implicit class FlatMapSyntaxForFunction1[P, A](f: Function1[P, A]) {
  def flatMap[B](g: A => P => B): P => B = Monad[Function1[P, ?]].flatMap(f)(g)
}
```

FlatMap syntax

defined as implicit conversion ...

in a specific way for *Monad[Function1]*:

```
implicit class FlatMapSyntaxForFunction1[P, A](f: Function1[P, A]) {
  def flatMap[B](g: A => P => B): P => B = Monad[Function1[P, ?]].flatMap(f)(g)
}
```

or in a generic way for any *Monad[F[_]]*:

```
implicit class FlatMapSyntax[F[_]: Monad, A](fa: F[A]) {
  def flatMap[B](f: A => F[B]): F[B] = Monad[F].flatMap(fa)(f)
}
```

FlatMap syntax

defined as implicit conversion ...

in a specific way for *Monad[Function1]*:

```
implicit class FlatMapSyntaxForFunction1[P, A](f: Function1[P, A]) {
  def flatMap[B](g: A => P => B): P => B = Monad[Function1[P, ?]].flatMap(f)(g)
}
```

or in a generic way for any *Monad[F[_]]*:

```
implicit class FlatMapSyntax[F[_]: Monad, A](fa: F[A]) {
  def flatMap[B](f: A => F[B]): F[B] = Monad[F].flatMap(fa)(f)
}
```

This allows for convenient invocation of *flatMap*
as if *flatMap* were a method of *Function1*:

```
val h = f flatMap g
```

instead of:

```
val h = Monad[Function1[Int, ?]].flatMap(f)(g)
```

A pipeline of functions (Reader Monad)

A pipeline of functions (Reader Monad)

```
val countLines: String => Int = text => text.split("\n").length
val countWords: String => Int = text => text.split("\\W+").length
val countChars: String => Int = text => text.length
```

A pipeline of functions (Reader Monad)

```scala
val countLines: String => Int = text => text.split("\n").length
val countWords: String => Int = text => text.split("\\W+").length
val countChars: String => Int = text => text.length
```

FlatMapping over *Function1*:

```scala
val computeStatistics1: String => (Int, Int, Int) =
  countLines flatMap { nLines =>      // define a pure program which does nothing
    countWords flatMap { nWords =>
      countChars map { nChars =>
        (nLines, nWords, nChars)
      } } }
val stat1: (Int, Int, Int) = computeStatistics1(getInput) // exec program (impure)
```

A pipeline of functions (Reader Monad)

```
val countLines: String => Int = text => text.split("\n").length
val countWords: String => Int = text => text.split("\\W+").length
val countChars: String => Int = text => text.length
```

FlatMapping over *Function1*:

```
val computeStatistics1: String => (Int, Int, Int) =
  countLines flatMap { nLines =>      // define a pure program which does nothing
    countWords flatMap { nWords =>
      countChars map { nChars =>
        (nLines, nWords, nChars)
      } } }
val stat1: (Int, Int, Int) = computeStatistics1(getInput) // exec program (impure)
```

alternatively with a for-comprehension:

```
val computeStatistics2: String => (Int, Int, Int) =
  for {                               // define a pure program which does nothing
    nLines <- countLines // uses Function1#flatMap
    nWords <- countWords
    nChars <- countChars
  } yield (nLines, nWords, nChars)
val stat2: (Int, Int, Int) = computeStatistics2(getInput) // exec program (impure)
```

Another Reader Monad example - a bit more realistic ...

```scala
val users: Map[Int, String] = Map(
          1 -> "dade", 2 -> "kate", 3 -> "margo")
val passwords: Map[String, String] = Map(
          "dade" -> "zerocool", "kate" -> "acidburn", "margo" -> "secret")

case class Db(usernames: Map[Int, String], passwords: Map[String, String])
val db = Db(users, passwords)

type DbReader[A] = Db => A    // ^= Function1[Db, A]

def findUsername(userId: Int): DbReader[Option[String]] =
  db => db.usernames.get(userId)

def checkPassword(optUsername: Option[String], password: String): DbReader[Boolean
  def checkPw(db: Db, username: String): Boolean =
    db.passwords.get(username).contains(password)
  db => optUsername.exists(name => checkPw(db, name))
}

def checkLogin(userId: Int, password: String): DbReader[Boolean] = for {
  optUsername <- findUsername(userId)
  passwordOk <- checkPassword(optUsername, password)
} yield passwordOk

val loginOk1 = checkLogin(1, "zerocool")(db) // true
val loginOk2 = checkLogin(4, "davinci")(db)  // false
```

See: *demo.Demo08bDbReader*

Example taken from "Scala with Cats" (see chapter Resources for link)

Flattening Curried Functions (1/3)

```scala
// Flattening nested Option (has 1 type parameter)
val oooi: Option[Option[Option[Int]]] = Some(Some(Some(1)))
val oi: Option[Int] = oooi.flatten.flatten
println(oi) // Some(1)
```

Flattening Curried Functions (1/3)

```scala
// Flattening nested Option (has 1 type parameter)
val oooi: Option[Option[Option[Int]]] = Some(Some(Some(1)))
val oi: Option[Int] = oooi.flatten.flatten
println(oi) // Some(1)
```

```scala
// Flattening nested Either (has 2 type parameters)
val eeei: Either[String, Either[String, Either[String, Int]]] =
                              Right(Right(Right(1)))
val ei: Either[String, Int] = eeei.flatten.flatten
println(ei) // Right(1)
```

Flattening Curried Functions (1/3)

```scala
// Flattening nested Option (has 1 type parameter)
val oooi: Option[Option[Option[Int]]] = Some(Some(Some(1)))
val oi: Option[Int] = oooi.flatten.flatten
println(oi) // Some(1)
```

```scala
// Flattening nested Either (has 2 type parameters)
val eeei: Either[String, Either[String, Either[String, Int]]] =
                              Right(Right(Right(1)))
val ei: Either[String, Int] = eeei.flatten.flatten
println(ei) // Right(1)
```

```scala
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Function1[Int, Function1[Int, Function1[Int, Int]]] =
                              sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = ...
```

Can *Function1* be flattened?

Flattening Curried Functions (1/3)

```
// Flattening nested Option (has 1 type parameter)
val oooi: Option[Option[Option[Int]]] = Some(Some(Some(1)))
val oi: Option[Int] = oooi.flatten.flatten
println(oi) // Some(1)
```

```
// Flattening nested Either (has 2 type parameters)
val eeei: Either[String, Either[String, Either[String, Int]]] =
                              Right(Right(Right(1)))
val ei: Either[String, Int] = eeei.flatten.flatten
println(ei) // Right(1)
```

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Function1[Int, Function1[Int, Function1[Int, Int]]] =
                              sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = ...
```

Can *Function1* be flattened?

```
// Flattening nested Function1 (has 2 type parameters)
val fFlattened: Int => Int = sumCurried.flatten.flatten
println(fFlattened(5)) // => 15
```

Flattening Curried Functions (2/3)

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Function1[Int, Function1[Int, Function1[Int, Int]]] =
                              sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = ...
```

Flattening Curried Functions (2/3)

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Function1[Int, Function1[Int, Function1[Int, Int]]] =
                              sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = ...
```

```
val fFlattened: Int => Int = sumCurried.flatten.flatten
println(fFlattened(5)) // => 15
```

Flattening Curried Functions (2/3)

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Function1[Int, Function1[Int, Function1[Int, Int]]] =
                              sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = ...
```

```
val fFlattened: Int => Int = sumCurried.flatten.flatten
println(fFlattened(5)) // => 15
```

flatten is *flatMap* with *identity*.

```
val fFlatMapped: Int => Int = sumCurried.flatMap(identity).flatMap(identity)
println(fFlatMapped(5)) // => 15
```

Flattening Curried Functions (2/3)

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Function1[Int, Function1[Int, Function1[Int, Int]]] =
                              sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = ...
```

```
val fFlattened: Int => Int = sumCurried.flatten.flatten
println(fFlattened(5)) // => 15
```

flatten is *flatMap* with *identity*.

```
val fFlatMapped: Int => Int = sumCurried.flatMap(identity).flatMap(identity)
println(fFlatMapped(5)) // => 15
```

flatMap and *map* can be replaced with a for-comnprehension.

```
val fBuiltWithFor: Int => Int = for {
  f1 <- sumCurried
  f2 <- f1
  int <- f2
} yield int
println(fBuiltWithFor(5)) // => 15
```

See: *demo.Demo08cFlattenCurriedFunctions*

Flattening Curried Functions (3/3)

```
implicit class FlattenSyntax[F[_]: Monad, A](ffa: F[F[A]]) {
  def flatten: F[A] = Monad[F].flatten(ffa)
}
```

Flattening Curried Functions (3/3)

```
implicit class FlattenSyntax[F[_]: Monad, A](ffa: F[F[A]]) {
  def flatten: F[A] = Monad[F].flatten(ffa)
}
```

With this implicit conversion we can invoke *flatten* on any context *F[F[A]]* … such as a nested *List*, *Option*, *Either[L, ?]* or *Function1[P, ?]*, where *F[_]* is contrained to be a Monad.

Flattening Curried Functions (3/3)

```
implicit class FlattenSyntax[F[_]: Monad, A](ffa: F[F[A]]) {
  def flatten: F[A] = Monad[F].flatten(ffa)
}
```

With this implicit conversion we can invoke *flatten* on any context *F[F[A]]* ... such as a nested *List*, *Option*, *Either[L, ?]* or *Function1[P, ?]*, where *F[_]* is contrained to be a Monad.

This allows for convenient invocation of *Function1#flatten*
as if *flatten* were an intrinsic method of *Function1*:

```
val f: Function1[A, Function1[A, B]] = ???
```

```
val g: Function1[A, B] = f.flatten
```

Flattening Curried Functions (3/3)

```
implicit class FlattenSyntax[F[_]: Monad, A](ffa: F[F[A]]) {
  def flatten: F[A] = Monad[F].flatten(ffa)
}
```

With this implicit conversion we can invoke *flatten* on any context *F[F[A]]* ... such as a nested *List*, *Option*, *Either[L, ?]* or *Function1[P, ?]*, where *F[_]* is contrained to be a Monad.

This allows for convenient invocation of *Function1#flatten*
as if *flatten* were an intrinsic method of *Function1*:

```
val f: Function1[A, Function1[A, B]] = ???
```

```
val g: Function1[A, B] = f.flatten
```

instead of:

```
val g: Function1[A, B] = Monad[Function1[A, ?]].flatten(f)
```

9. Kleisli composition - done manually

See: *demo.Demo09KleisliDoneManually*

The Problem:

The Problem:

```
// Functions: A => F[B], where F is Option in this case
val s2iOpt: String => Option[Int] = s => Option(s.toInt)
val plus2Opt: Int => Option[Int] = i => Option(i + 2)
val div10ByOpt: Int => Option[Double] = i => Option(10.0 / i)
val d2sOpt: Double => Option[String] = d => Option(d.toString + " !!!")
```

These functions take an *A* and return a *B* inside of a context *F[_]: A => F[B]*
In our case *F[_]* is *Option*, but could be *List*, *Future* etc.

The Problem:

```scala
// Functions: A => F[B], where F is Option in this case
val s2iOpt: String => Option[Int] = s => Option(s.toInt)
val plus2Opt: Int => Option[Int] = i => Option(i + 2)
val div10ByOpt: Int => Option[Double] = i => Option(10.0 / i)
val d2sOpt: Double => Option[String] = d => Option(d.toString + " !!!")
```

These functions take an *A* and return a *B* inside of a context *F[_]: A => F[B]*
In our case *F[_]* is *Option*, but could be *List*, *Future* etc.

We want to compose these functions to a single function which is then fed with some input string.

The Problem:

```
// Functions: A => F[B], where F is Option in this case
val s2iOpt: String => Option[Int] = s => Option(s.toInt)
val plus2Opt: Int => Option[Int] = i => Option(i + 2)
val div10ByOpt: Int => Option[Double] = i => Option(10.0 / i)
val d2sOpt: Double => Option[String] = d => Option(d.toString + " !!!")
```

These functions take an *A* and return a *B* inside of a context *F[_]: A => F[B]*
In our case *F[_]* is *Option*, but could be *List*, *Future* etc.

We want to compose these functions to a single function which is then fed with some input string.

Let's try *map*.

```
val fMapped: String => Option[Option[Option[Option[String]]]] = str =>
  s2iOpt(str) map { i1 =>
    plus2Opt(i1) map { i2 =>
      div10ByOpt(i2) map {
        d => d2sOpt(d)
      }}}
```

We get nested *Options*. So lets try *flatMap* on the *Option* context.

FlatMapping on the *Option* context

FlatMapping on the *Option* context

with *flatMap* (this works):

```
val flatMappedOnOpt1: String => Option[String] = input =>
  s2iOpt(input) flatMap { i1 =>
    plus2Opt(i1) flatMap { i2 =>
      div10ByOpt(i2) flatMap { d =>
        d2sOpt(d)
      }}}
val res1: Option[String] = flatMappedOnOpt1("3") // Some(2.0)
```

FlatMapping on the *Option* context

with *flatMap* (this works):

```scala
val flatMappedOnOpt1: String => Option[String] = input =>
  s2iOpt(input) flatMap { i1 =>
    plus2Opt(i1) flatMap { i2 =>
      div10ByOpt(i2) flatMap { d =>
        d2sOpt(d)
      }}}
val res1: Option[String] = flatMappedOnOpt1("3") // Some(2.0)
```

or with a for-comprehension (this looks nicer):

```scala
val flatMappedOnOpt2: String => Option[String] = input => for {
  i1 <- s2iOpt(input)
  i2 <- plus2Opt(i1)
  d <- div10ByOpt(i2)
  s <- d2sOpt(d)
} yield s
val res2: Option[String] = flatMappedOnOpt2("3") // Some(2.0)
```

FlatMapping on the *Option* context

with *flatMap* (this works):

```
val flatMappedOnOpt1: String => Option[String] = input =>
  s2iOpt(input) flatMap { i1 =>
    plus2Opt(i1) flatMap { i2 =>
      div10ByOpt(i2) flatMap { d =>
        d2sOpt(d)
      }}}
val res1: Option[String] = flatMappedOnOpt1("3") // Some(2.0)
```

or with a for-comprehension (this looks nicer):

```
val flatMappedOnOpt2: String => Option[String] = input => for {
  i1 <- s2iOpt(input)
  i2 <- plus2Opt(i1)
  d <- div10ByOpt(i2)
  s <- d2sOpt(d)
} yield s
val res2: Option[String] = flatMappedOnOpt2("3") // Some(2.0)
```

But: We still have to bind the variables *i1*, *i2*, *d* and *s* to names.
We would like to build a function pipeline with some kind of *andThenF*.
Wanted: something like ...
s2iOpt andThenF plus2Opt andThenF div10ByOpt andThenF d2sOpt

Kleisli Composition

Kleisli composition takes two functions *A* => *F[B]* and *B* => *F[C]* and yields a new function *A* => *F[C]* where the context *F[_]* is constrained to be a Monad.

Kleisli Composition

Kleisli composition takes two functions *A => F[B]* and *B => F[C]* and yields a new function *A => F[C]* where the context *F[_]* is constrained to be a Monad.

Let's define (our hand-weaved) *kleisli*:

Kleisli Composition

Kleisli composition takes two functions *A* => *F[B]* and *B* => *F[C]* and yields a new function *A* => *F[C]* where the context *F[_]* is constrained to be a Monad.

Let's define (our hand-weaved) *kleisli*:

```
def kleisli[F[_]: Monad, A, B, C](f: A => F[B], g: B => F[C]): A => F[C] =
  a => Monad[F].flatMap(f(a))(g)
```

Kleisli Composition

Kleisli composition takes two functions *A => F[B]* and *B => F[C]* and yields a new function *A => F[C]* where the context *F[_]* is constrained to be a Monad.

Let's define (our hand-weaved) *kleisli*:

```
def kleisli[F[_]: Monad, A, B, C](f: A => F[B], g: B => F[C]): A => F[C] =
  a => Monad[F].flatMap(f(a))(g)
```

Using *kleisli*:

```
val kleisliComposed1: String => Option[String] =
  kleisli(kleisli(kleisli(s2iOpt, plus2Opt), div10ByOpt), d2sOpt)

val resKleisli1 = kleisliComposed1("3")    // 2.0 !!!
```

Kleisli Composition

Kleisli composition takes two functions *A => F[B]* and *B => F[C]* and yields a new function *A => F[C]* where the context *F[_]* is constrained to be a Monad.

Let's define (our hand-weaved) *kleisli*:

```
def kleisli[F[_]: Monad, A, B, C](f: A => F[B], g: B => F[C]): A => F[C] =
  a => Monad[F].flatMap(f(a))(g)
```

Using *kleisli*:

```
val kleisliComposed1: String => Option[String] =
  kleisli(kleisli(kleisli(s2iOpt, plus2Opt), div10ByOpt), d2sOpt)

val resKleisli1 = kleisliComposed1("3")    // 2.0 !!!
```

This works, but is still not exactly what we want.
kleisli should behave like a method of *Function1*.

kleisli defined on *Function1*

kleisli defined on *Function1*

with an implicit conversion:

```
implicit class Function1WithKleisli[F[_]: Monad, A, B](f: A => F[B]) {
  def kleisli[C](g: B => F[C]): A => F[C] = a => Monad[F].flatMap(f(a))(g)
  def andThenF[C](g: B => F[C]): A => F[C] = f kleisli g
  def >=>[C](g: B => F[C]): A => F[C] = f kleisli g     // Haskell's fish operator
}
```

kleisli defined on *Function1*

with an implicit conversion:

```
implicit class Function1WithKleisli[F[_]: Monad, A, B](f: A => F[B]) {
  def kleisli[C](g: B => F[C]): A => F[C] = a => Monad[F].flatMap(f(a))(g)
  def andThenF[C](g: B => F[C]): A => F[C] = f kleisli g
  def >=>[C](g: B => F[C]): A => F[C] = f kleisli g     // Haskell's fish operator
}
```

Using it:

```
val kleisliComposed2: String => Option[String] =
  s2iOpt kleisli plus2Opt kleisli div10ByOpt kleisli d2sOpt
kleisliComposed2("3") foreach println    // 2.0 !!!
```

kleisli defined on *Function1*

with an implicit conversion:

```
implicit class Function1WithKleisli[F[_]: Monad, A, B](f: A => F[B]) {
  def kleisli[C](g: B => F[C]): A => F[C] = a => Monad[F].flatMap(f(a))(g)
  def andThenF[C](g: B => F[C]): A => F[C] = f kleisli g
  def >=>[C](g: B => F[C]): A => F[C] = f kleisli g     // Haskell's fish operator
}
```

Using it:

```
val kleisliComposed2: String => Option[String] =
  s2iOpt kleisli plus2Opt kleisli div10ByOpt kleisli d2sOpt
kleisliComposed2("3") foreach println    // 2.0 !!!
```

```
(s2iOpt andThenF plus2Opt andThenF div10ByOpt andThenF d2sOpt)("3") foreach printl
```

kleisli defined on *Function1*

with an implicit conversion:

```
implicit class Function1WithKleisli[F[_]: Monad, A, B](f: A => F[B]) {
  def kleisli[C](g: B => F[C]): A => F[C] = a => Monad[F].flatMap(f(a))(g)
  def andThenF[C](g: B => F[C]): A => F[C] = f kleisli g
  def >=>[C](g: B => F[C]): A => F[C] = f kleisli g     // Haskell's fish operator
}
```

Using it:

```
val kleisliComposed2: String => Option[String] =
  s2iOpt kleisli plus2Opt kleisli div10ByOpt kleisli d2sOpt
kleisliComposed2("3") foreach println    // 2.0 !!!
```

```
(s2iOpt andThenF plus2Opt andThenF div10ByOpt andThenF d2sOpt)("3") foreach printl
```

```
(s2iOpt >=> plus2Opt >=> div10ByOpt >=> d2sOpt)("3") foreach println
```

10. *case class Kleisli*

See: *demo.Demo10KleisliCaseClass*

case class Kleisli

The previous *kleisli* impl was my personal artefact. Cats does not provide a *kleisli* method on *Function1*. Cats instead provides a *case class Kleisli* with the functionality shown above and more.

case class Kleisli

The previous *kleisli* impl was my personal artefact. Cats does not provide a *kleisli* method on *Function1*. Cats instead provides a *case class Kleisli* with the functionality shown above and more.

I tinkered my own case class impl in *mycats.Kleisli* which works much like the Cats impl: see next slide.

case class Kleisli

```scala
case class Kleisli[F[_], A, B](run: A => F[B]) {

  def apply(a: A): F[B] = run(a)

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

  def flatMap[C](f: B => Kleisli[F, A, C])(implicit M: Monad[F]): Kleisli[F, A, C]
    Kleisli { a => M.flatMap(run(a))(b => f(b).run(a)) }

  def flatMapF[C](f: B => F[C])(implicit M: Monad[F]): Kleisli[F, A, C] =
    Kleisli { a => M.flatMap(run(a))(f) }

  def andThen[C](f: B => F[C])(implicit M: Monad[F]): Kleisli[F, A, C] =
    flatMapF(f)

  def andThen[C](that: Kleisli[F, B, C])(implicit M: Monad[F]): Kleisli[F, A, C] =
    this andThen that.run

  def compose[Z](f: Z => F[A])(implicit M: Monad[F]): Kleisli[F, Z, B] =
    Kleisli(f) andThen this.run

  def compose[Z](that: Kleisli[F, Z, A])(implicit M: Monad[F]): Kleisli[F, Z, B] =
    that andThen this
}
```

The case class methods delegate to the wrapped *run* function and return the resulting *run* function rewrapped in a *Kleisli* instance.

Kleisli#flatMap

```
def flatMap[C](f: B => Kleisli[F, A, C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  M.flatMap(run(a))(b => f(b).run(a))
}
```

flatMap composes *this.run: A => F[B]* with the function *f: B => Kleisli[F, A, C]* yielding a new *Kleisli[F, A, C]* wrapping a new function *run: A => F[C]*.

Kleisli#flatMap

```scala
def flatMap[C](f: B => Kleisli[F, A, C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  M.flatMap(run(a))(b => f(b).run(a))
}
```

flatMap composes *this.run: A => F[B]* with the function *f: B => Kleisli[F, A, C]* yielding a new *Kleisli[F, A, C]* wrapping a new function *run: A => F[C]*.

```scala
val kleisli1: String => Option[String] = input =>
  Kleisli(s2iOpt).run(input) flatMap { i1 =>
    Kleisli(plus2Opt).run(i1) flatMap { i2 =>
      Kleisli(div10ByOpt).run(i2) flatMap { d =>
        Kleisli(d2sOpt).run(d)
      } } }
kleisli1("3") foreach println
```

Kleisli#flatMap

```
def flatMap[C](f: B => Kleisli[F, A, C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  M.flatMap(run(a))(b => f(b).run(a))
}
```

flatMap composes *this.run: A => F[B]* with the function *f: B => Kleisli[F, A, C]* yielding a new *Kleisli[F, A, C]* wrapping a new function *run: A => F[C]*.

```
val kleisli1: String => Option[String] = input =>
  Kleisli(s2iOpt).run(input) flatMap { i1 =>
    Kleisli(plus2Opt).run(i1) flatMap { i2 =>
      Kleisli(div10ByOpt).run(i2) flatMap { d =>
        Kleisli(d2sOpt).run(d)
      } } }
kleisli1("3") foreach println
```

```
val kleisli2: String => Option[String] = input => for {
  i1 <- Kleisli(s2iOpt).run(input)
  i2 <- Kleisli(plus2Opt).run(i1)
  d <- Kleisli(div10ByOpt).run(i2)
  s <- Kleisli(d2sOpt).run(d)
} yield s
kleisli2("3") foreach println
```

Kleisli#flatMapF

As we saw *Kleisli#flatMap* is not very convenient (even when using a for-comprehension). We have to bind values to variables and thread them through the for-comprehension. *flatMapF* is easier to use.

Kleisli#flatMapF

As we saw *Kleisli#flatMap* is not very convenient (even when using a for-comprehension). We have to bind values to variables and thread them through the for-comprehension. *flatMapF* is easier to use.

```
def flatMapF[C](f: B => F[C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  Kleisli { a => M.flatMap(run(a))(f) }
```

flatMapF composes *this.run: A => F[B]* with the function *f: B => F[C]* yielding a new *Kleisli[F, A, C]* wrapping a new function *run: A => F[C]*.

Kleisli#flatMapF

As we saw *Kleisli#flatMap* is not very convenient (even when using a for-comprehension). We have to bind values to variables and thread them through the for-comprehension. *flatMapF* is easier to use.

```
def flatMapF[C](f: B => F[C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  Kleisli { a => M.flatMap(run(a))(f) }
```

flatMapF composes *this.run: A => F[B]* with the function *f: B => F[C]* yielding a new *Kleisli[F, A, C]* wrapping a new function *run: A => F[C]*.

```
val kleisli3: Kleisli[Option, String, String] =
  Kleisli(s2iOpt) flatMapF plus2Opt flatMapF div10ByOpt flatMapF d2sOpt

kleisli3.run("3") foreach println
```

Kleisli#andThen

The behaviour of *flatMapF* is exactly what we expect from *andThen* (according to *Function1#andThen*).

Kleisli#andThen

The behaviour of *flatMapF* is exactly what we expect from *andThen* (according to *Function1#andThen*).

```
def andThen[C](f: B => F[C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  flatMapF(f)
```

Kleisli#andThen

The behaviour of *flatMapF* is exactly what we expect from *andThen* (according to *Function1#andThen*).

```
def andThen[C](f: B => F[C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  flatMapF(f)
```

The first version of *andThen* is an alias for *flatMapF*.

Kleisli#andThen

The behaviour of *flatMapF* is exactly what we expect from *andThen* (according to *Function1#andThen*).

```
def andThen[C](f: B => F[C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  flatMapF(f)
```

The first version of *andThen* is an alias for *flatMapF*.

```
val kleisli4: Kleisli[Option, String, String] =
  Kleisli(s2iOpt) andThen plus2Opt andThen div10ByOpt andThen d2sOpt

kleisli4.run("3") foreach println
```

Kleisli#andThen

Kleisli#andThen

```
def andThen[C](that: Kleisli[F, B, C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  this andThen that.run
```

This overloaded version of *andThen* doesn't take a function *f: B => F[C]*. Instead it takes another *Kleisli[F, B, C]* wrapping such a function. This version allows us to concatenate several Kleislis to a pipeline.

Kleisli#andThen

```
def andThen[C](that: Kleisli[F, B, C])(implicit M: Monad[F]): Kleisli[F, A, C] =
  this andThen that.run
```

This overloaded version of *andThen* doesn't take a function *f: B => F[C]*. Instead it takes another *Kleisli[F, B, C]* wrapping such a function. This version allows us to concatenate several Kleislis to a pipeline.

```
val kleisli5: Kleisli[Option, String, String] =
  Kleisli(s2iOpt) andThen Kleisli(plus2Opt) andThen
              Kleisli(div10ByOpt) andThen Kleisli(d2sOpt)

kleisli5.run("3") foreach println
```

Kleisli#compose

As with *andThen* there are two versions of *compose*. They work like *andThen* with the arguments flipped.

Kleisli#compose

As with *andThen* there are two versions of *compose*. They work like *andThen* with the arguments flipped.

```
def compose[Z](f: Z => F[A])(implicit M: Monad[F]): Kleisli[F, Z, B] =
  Kleisli(f) andThen this.run

def compose[Z](that: Kleisli[F, Z, A])(implicit M: Monad[F]): Kleisli[F, Z, B] =
  that andThen this
```

Kleisli#compose

As with *andThen* there are two versions of *compose*. They work like *andThen* with the arguments flipped.

```
def compose[Z](f: Z => F[A])(implicit M: Monad[F]): Kleisli[F, Z, B] =
  Kleisli(f) andThen this.run

def compose[Z](that: Kleisli[F, Z, A])(implicit M: Monad[F]): Kleisli[F, Z, B] =
  that andThen this
```

```
(Kleisli(d2sOpt) compose div10ByOpt compose
    plus2Opt compose s2iOpt).run("3") foreach println

(Kleisli(d2sOpt) compose Kleisli(div10ByOpt) compose
    Kleisli(plus2Opt) compose Kleisli(s2iOpt)).run("3") foreach println
```

Kleisli companion object with Monad instance

```
object Kleisli {

  // Kleisli Monad instance defined in companion object is in
  // 'implicit scope' (i.e. found by the compiler without import).

  implicit def kleisliMonad[F[_], A](implicit M: Monad[F]): Monad[Kleisli[F, A, ?]

    new Monad[Kleisli[F, A, ?]] {

      override def pure[B](b: B): Kleisli[F, A, B] =
          Kleisli { _ => M.pure(b) }

      override def flatMap[B, C](kl: Kleisli[F, A, B])(f: B => Kleisli[F, A, C])
          : Kleisli[F, A, C] = kl flatMap f
    }
}
```

11. Reader Monad

See: *demo.Demo11aReader*

Reader again

We already saw, that the *Function1* Monad is the *Reader* Monad.

But *Kleisli* can also be used as *Reader*, if *F[_]* is fixed to the *Id* context.

Reader again

We already saw, that the *Function1* Monad is the *Reader* Monad.

But *Kleisli* can also be used as *Reader*, if *F[_]* is fixed to the *Id* context.

```
type Id[A] = A

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

type Reader[A, B] = ReaderT[Id, A, B]

object Reader {
  def apply[A, B](f: A => B): Reader[A, B] = ReaderT[Id, A, B](f)
}
```

Reader#flatMap

Reader#flatMap

```scala
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString + " !!!"
```

Reader#flatMap

```scala
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString + " !!!"
```

```scala
val reader1: String => String = input => for {
  i1 <- Reader(s2i).run(input)
  i2 <- Reader(plus2).run(i1)
  d <- Reader(div10By).run(i2)
  s <- Reader(d2s).run(d)
} yield s

println(reader1("3"))     // 2.0 !!!
```

Reader#flatMap

```
val s2i: String => Int = _.toInt
val plus2: Int => Int = _ + 2
val div10By: Int => Double = 10.0 / _
val d2s: Double => String = _.toString + " !!!"
```

```
val reader1: String => String = input => for {
  i1 <- Reader(s2i).run(input)
  i2 <- Reader(plus2).run(i1)
  d <- Reader(div10By).run(i2)
  s <- Reader(d2s).run(d)
} yield s

println(reader1("3"))     // 2.0 !!!
```

Again *flatMap* is not the best choice as we have to declare all these intermediate identifiers in the for-comprehension.

Reader#andThen (two versions)

Reader#andThen (two versions)

```
val reader2 =
  Reader(s2i) andThen Reader(plus2) andThen Reader(div10By) andThen Reader(d2s)

println(reader2("3"))     // 2.0 !!!
```

Reader#andThen (two versions)

```
val reader2 =
  Reader(s2i) andThen Reader(plus2) andThen Reader(div10By) andThen Reader(d2s)

println(reader2("3"))     // 2.0 !!!
```

```
val reader3 =
  Reader(s2i) andThen plus2 andThen div10By andThen d2s

println(reader3("3"))     // 2.0 !!!
```

Reader#andThen (two versions)

```
val reader2 =
  Reader(s2i) andThen Reader(plus2) andThen Reader(div10By) andThen Reader(d2s)

println(reader2("3"))     // 2.0 !!!
```

```
val reader3 =
  Reader(s2i) andThen plus2 andThen div10By andThen d2s

println(reader3("3"))     // 2.0 !!!
```

All methods of *Kleisli* are available for Reader,
because *Kleisli* **is** *Reader* with *Id*.

The Reader example from before runs with minor changes.

```
val users: Map[Int, String] = Map(
              1 -> "dade", 2 -> "kate", 3 -> "margo" )
val passwords: Map[String, String] = Map(
              "dade" -> "zerocool", "kate" -> "acidburn", "margo" -> "secret" )

case class Db(usernames: Map[Int, String], passwords: Map[String, String])
val db = Db(users, passwords)

type DbReader[A] = Reader[Db, A]      // ^= Kleisli[Id, Db, Boolean]

def findUsername(userId: Int): DbReader[Option[String]] =
  Reader { db => db.usernames.get(userId) }

def checkPassword(optUsername: Option[String], password: String): DbReader[Boolean
  def checkPw(db: Db, username: String): Boolean =
    db.passwords.get(username).contains(password)
  Reader { db => optUsername.exists(name => checkPw(db, name)) }
}

def checkLogin(userId: Int, password: String): DbReader[Boolean] = for {
  optUsername <- findUsername(userId)
  passwordOk <- checkPassword(optUsername, password)
} yield passwordOk

val loginOk1 = checkLogin(1, "zerocool").run(db) // true
val loginOk2 = checkLogin(4, "davinci").run(db)  // false
```

See: *demo.Demo11bDbReader*
Example taken from "Scala with Cats" (see chapter Resources for link)

12. Resources

Resources (1/2)

- Code and Slides of this Talk:
  https://github.com/hermannhueck/composing-functions
- "Scala with Cats"
  Book by Noel Welsh and Dave Gurnell
  https://underscore.io/books/scala-with-cats/
- "Functional Programming in Scala"
  Book by Paul Chiusano and Runar Bjarnason
  https://www.manning.com/books/functional-programming-in-scala
- Cats documentation for *Kleisli*:
  https://typelevel.org/cats/datatypes/kleisli.html

Resources (2/2)

- Miles Sabin's pull request for partial unification:
  https://github.com/scala/scala/pull/5102
- "Explaining Miles's Magic:
  Gist of Daniel Spiewak on partial unification
  https://gist.github.com/djspiewak/7a81a395c461fd3a09a6941d4cd040f2
- "Kind Projector Compiler Plugin:
  https://github.com/non/kind-projector

Thanks for Listening

Q & A

https://github.com/hermannhueck/composing-functions