```
scala> case class Foo(s: String, i: Int)
scala> implicit val fooEqual: Equal[Foo] =
         Divide[Equal].divide2(Equal[String], Equal[Int]) {
           (foo: Foo) => (foo.s, foo.i)
         }
scala> Foo("foo", 1) === Foo("bar", 1)
res: Boolean = false
```

Mirroring `Apply`, `Divide` also has terse syntax for tuples. A softer *divide so that we may reign* approach to world domination:

```
...
  def tuple2[A1, A2](a1: F[A1], a2: F[A2]): F[(A1, A2)] = ...
  ...
  def tuple22[...] = ...
}
```

Generally, if encoder typeclasses can provide an instance of `Divide`, rather than stopping at `Contravariant`, it makes it possible to derive instances for any `case class`. Similarly, decoder typeclasses can provide an `Apply` instance. We will explore this in a dedicated chapter on Typeclass Derivation.

`Divisible` is the `Contravariant` analogue of `Applicative` and introduces `.conquer`, the equivalent of `.pure`

```
@typeclass trait Divisible[F[_]] extends Divide[F] {
  def conquer[A]: F[A]
}
```

`.conquer` allows creating trivial implementations where the type parameter is ignored. Such values are called *universally quantified*. For example, the `Divisible[Equal].conquer[INil[String]]` returns an implementation of `Equal` for an empty list of `String` which is always `true`.

```
trait Twitter[F[_]] {
  def getUser(name: String): F[Maybe[User]]
  def getStars(user: User): F[Int]
}
def T[F[_]](implicit t: Twitter[F]): Twitter[F] = t
```

We need to call `getUser` followed by `getStars`. If we use `Monad` as our context, our function is difficult because we have to handle the `Empty` case:

```
def stars[F[_]: Monad: Twitter](name: String): F[Maybe[Int]] = for {
  maybeUser  <- T.getUser(name)
  maybeStars <- maybeUser.traverse(T.getStars)
} yield maybeStars
```

However, if we have a `MonadPlus` as our context, we can suck `Maybe` into the `F[_]` with `.orEmpty`, and forget about it:

```
def stars[F[_]: MonadPlus: Twitter](name: String): F[Int] = for {
  user  <- T.getUser(name) >>= (_.orEmpty[F])
  stars <- T.getStars(user)
} yield stars
```

However adding a `MonadPlus` requirement can cause problems downstream if the context does not have one. The solution is to either change the context of the program to `MaybeT[F, ?]` (lifting the `Monad[F]` into a `MonadPlus`), or to explicitly use `MaybeT` in the return type, at the cost of slightly more code:

```
def stars[F[_]: Monad: Twitter](name: String): MaybeT[F, Int] = for {
  user  <- MaybeT(T.getUser(name))
  stars <- T.getStars(user).liftM[MaybeT]
} yield stars
```

The decision to require a more powerful `Monad` vs returning a transformer is something that each team can decide for themselves based on the interpreters that they plan on using for their program.

7.4.3 `EitherT`

An optional value is a special case of a value that may be an error, but we don't know anything about the error. `EitherT` (and the lazy variant `LazyEitherT`) allows us to use any type we want as the error value, providing contextual information about what went wrong.

`EitherT` is a wrapper around an `F[A \/ B]`

7.4.10 `ContT`

Continuation Passing Style (CPS) is a style of programming where functions never return, instead *continuing* to the next computation. CPS is popular in Javascript and Lisp as they allow non-blocking I/O via callbacks when data is available. A direct translation of the pattern into impure Scala looks like

```
def foo[I, A](input: I)(next: A => Unit): Unit = next(doSomeStuff(input))
```

We can make this pure by introducing an `F[_]` context

```
def foo[F[_], I, A](input: I)(next: A => F[Unit]): F[Unit]
```

and refactor to return a function for the provided input

```
def foo[F[_], I, A](input: I): (A => F[Unit]) => F[Unit]
```

`ContT` is just a container for this signature, with a `Monad` instance

```
final case class ContT[F[_], B, A](_run: (A => F[B]) => F[B]) {
  def run(f: A => F[B]): F[B] = _run(f)
}
object IndexedContT {
  implicit def monad[F[_], B] = new Monad[ContT[F, B, ?]] {
    def point[A](a: =>A) = ContT(_(a))
    def bind[A, C](fa: ContT[F, B, A])(f: A => ContT[F, B, C]) =
      ContT(c_fb => fa.run(a => f(a).run(c_fb)))
  }
}
```

and convenient syntax to create a `ContT` from a monadic value:

```
implicit class ContTOps[F[_]: Monad, A](self: F[A]) {
  def cps[B]: ContT[F, B, A] = ContT(a_fb => self >>= a_fb)
}
```

However, the simple callback use of continuations brings nothing to pure functional programming because we already know how to sequence non-blocking, potentially distributed, computations: that is what `Monad` is for and we can do this with `.bind` or a `Kleisli` arrow. To see why continuations are useful we need to consider a more complex example under a rigid design constraint.

7.4.10.1 Control Flow

Say we have modularised our application into components that can perform I/O, with each component owned by a different development team:

```
trait ForComprehensible[C[_]] {
  def map[A, B](f: A => B): C[B]
  def flatMap[A, B](f: A => C[B]): C[B]
  def withFilter[A](p: A => Boolean): C[A]
  def foreach[A](f: A => Unit): Unit
}
```

If the context (`C[_]`) of a `for` comprehension doesn't provide its own `map` and `flatMap`, all is not lost. If an implicit `scalaz.Bind[T]` is available for `T`, it will provide `map` and `flatMap`.

> It often surprises developers when inline `Future` calculations in a `for` comprehension do not run in parallel:
>
> ```
> import scala.concurrent._
> import ExecutionContext.Implicits.global
>
> for {
>   i <- Future { expensiveCalc() }
>   j <- Future { anotherExpensiveCalc() }
> } yield (i + j)
> ```
>
> This is because the `flatMap` spawning `anotherExpensiveCalc` is strictly **after** `expensiveCalc`. To ensure that two `Future` calculations begin in parallel, start them outside the `for` comprehension.
>
> ```
> val a = Future { expensiveCalc() }
> val b = Future { anotherExpensiveCalc() }
> for { i <- a ; j <- b } yield (i + j)
> ```
>
> `for` comprehensions are fundamentally for defining sequential programs. We will show a far superior way of defining parallel computations in a later chapter. Spoiler: don't use `Future`.

```
    new DerivedJsEncoder[A, HNil, HNil] {
      def toJsFields(h: HNil, a: HNil) = IList.empty
    }

  implicit def cnil[A]: DerivedJsEncoder[A, CNil, HNil] =
    new DerivedJsEncoder[A, CNil, HNil] {
      def toJsFields(c: CNil, a: HNil) = sys.error("impossible")
    }
}
private[jsonformat] trait DerivedJsEncoder1 {
  implicit def hcons[A, K <: Symbol, H, T <: HList, J <: HList](
    implicit
    K: Witness.Aux[K],
    H: Lazy[JsEncoder[H]],
    T: DerivedJsEncoder[A, T, J]
  ): DerivedJsEncoder[A, FieldType[K, H] :: T, None.type :: J] =
    new DerivedJsEncoder[A, FieldType[K, H] :: T, None.type :: J] {
      private val field = K.value.name
      def toJsFields(ht: FieldType[K, H] :: T, anns: None.type :: J) =
        ht match {
          case head :: tail =>
            val rest = T.toJsFields(tail, anns.tail)
            H.value.toJson(head) match {
              case JsNull => rest
              case value  => (field -> value) :: rest
            }
        }
    }

  implicit def ccons[A, K <: Symbol, H, T <: Coproduct, J <: HList](
    implicit
    K: Witness.Aux[K],
    H: Lazy[JsEncoder[H]],
    T: DerivedJsEncoder[A, T, J]
  ): DerivedJsEncoder[A, FieldType[K, H] :+: T, None.type :: J] =
    new DerivedJsEncoder[A, FieldType[K, H] :+: T, None.type :: J] {
      private val hint = ("type" -> JsString(K.value.name))
      def toJsFields(ht: FieldType[K, H] :+: T, anns: None.type :: J) =
        ht match {
          case Inl(head) =>
            H.value.toJson(head) match {
              case JsObject(fields) => hint :: fields
              case v                => IList.single("xvalue" -> v)
            }
          case Inr(tail) => T.toJsFields(tail, anns.tail)
        }
    }
```

3. Application Design

In this chapter we will write the business logic and tests for a purely functional server application. The source code for this application is included under the `example` directory along with the book's source, however it is recommended not to read the source code until the final chapter as there will be significant refactors as we learn more about FP.

```
def get[A: JsDecoder](
  uri: String Refined Url,
  headers: IList[(String, String)]
): F[A] =
  I.liftIO(
      H.fetch(
        http4s.Request[Task](
          uri = convert(uri),
          headers = convert(headers)
        )
       )(handler[A])
    )
    .emap(identity)
```

`.get` is all plumbing: we convert our input types into the `http4s.Request`, then call `.fetch` on the `Client` with our `handler`. This gives us back a `Task[Error \/ A]`, but we need to return a `F[A]`. Therefore we use the `MonadIO.liftIO` to create a `F[Error \/ A]` and then `.emap` to push the error into the `F`.

Unfortunately, if we try to compile this code it will fail. The error will look something like

```
[error] BlazeJsonClient.scala:95:64: could not find implicit value for parameter
[error]  F: cats.effect.Sync[scalaz.ioeffect.Task]
```

Basically, something about a missing cat.

The reason for this failure is that http4s is using a different core FP library, not Scalaz. Thankfully, `scalaz-ioeffect` provides a compatibility layer and the shims[3] project provides seamless (until it isn't) implicit conversions. We can get our code to compile with these dependencies:

```
libraryDependencies ++= Seq(
  "com.codecommit" %% "shims"                % "1.4.0",
  "org.scalaz"     %% "scalaz-ioeffect-cats" % "2.10.1"
)
```

and these imports

```
import shims._
import scalaz.ioeffect.catz._
```

The implementation of `.post` is similar but we must also provide an instance of

[3] https://github.com/djspiewak/shims

```
private val dsl = new Http4sDsl[Task] {}
import dsl._
```

Now we can use the http4s dsl[4] to create HTTP endpoints. Rather than describe everything that can be done, we will simply implement the endpoint which is similar to any of other HTTP DSLs

```
private object Code extends QueryParamDecoderMatcher[String]("code")
private val service: HttpService[Task] = HttpService[Task] {
  case GET -> Root :? Code(code) => ...
}
```

The return type of each pattern match is a `Task[Response[Task]]`. In our implementation we want to take the `code` and put it into the `ptoken` promise:

```
final class BlazeUserInteraction private (
  pserver: Promise[Throwable, Server[Task]],
  ptoken: Promise[Throwable, String]
) extends UserInteraction[Task] {
  ...
  private val service: HttpService[Task] = HttpService[Task] {
    case GET -> Root :? Code(code) =>
      ptoken.complete(code) >> Ok(
        "That seems to have worked, go back to the console."
      )
  }
  ...
}
```

but the definition of our services routes is not enough, we need to launch a server, which we do with `BlazeBuilder`

```
private val launch: Task[Server[Task]] =
  BlazeBuilder[Task].bindHttp(0, "localhost").mountService(service, "/").start
```

Binding to port `0` makes the operating system assign an ephemeral port. We can discover which port it is actually running on by querying the `server.address` field.

Our implementation of the `.start` and `.stop` methods is now straightforward

[4] https://http4s.org/v0.18/dsl/

```
data [] a = [] | a : [a]
infixr 5 :
```

and a convenient multi-argument value constructor: `[1, 2, 3]` instead of `1 : 2 : 3 : []`.

Ultimately our ADTs need to hold primitive values. The most common primitive data types are:

- `Char` a unicode character
- `Text` for blocks of unicode text
- `Int` a machine dependent, fixed precision signed integer
- `Word` an unsigned `Int`, and fixed size `Word8` / `Word16` / `Word32` / `Word64`
- `Float` / `Double` IEEE single and double precision numbers
- `Integer` / `Natural` arbitrary precision signed / non-negative integers
- `(,)` tuples, from 0 (also known as *unit*) to 62 fields
- `IO` the inspiration for Scalaz's `IO`, implemented in the runtime.

with honorary mentions for

```
data Bool       = True | False
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
data Ordering   = LT | EQ | GT
```

Like Scala, Haskell has type aliases: an alias or its expanded form can be used interchangeably. For legacy reasons, `String` is defined as a linked list of `Char`

```
type String = [Char]
```

which is very inefficient and we always want to use `Text` instead.

Finally we can define field names on ADTs using *record syntax*, which means we contain the data constructors in curly brackets and use double colon *type annotations* to indicate the types

```
-- raw ADT
data Resource = Human Int String
data Company  = Company String [Resource]

-- with record syntax
data Resource = Human
                { serial    :: Int
                , humanName :: String
                }
data Company  = Company
                { companyName :: String
                , employees   :: [Resource]
                }
```

Note that the `Human` data constructor and `Resource` type do not have the same name. Record syntax generates the equivalent of a field accessor and a copy method.

3.3 Business Logic

Now we write the business logic that defines the application's behaviour, considering only the happy path.

We need a `WorldView` class to hold a snapshot of our knowledge of the world. If we were designing this application in Akka, `WorldView` would probably be a `var` in a stateful `Actor`.

`WorldView` aggregates the return values of all the methods in the algebras, and adds a *pending* field to track unfulfilled requests.

```
final case class WorldView(
  backlog: Int,
  agents: Int,
  managed: NonEmptyList[MachineNode],
  alive: Map[MachineNode, Epoch],
  pending: Map[MachineNode, Epoch],
  time: Epoch
)
```

Now we are ready to write our business logic, but we need to indicate that we depend on `Drone` and `Machines`.

We can write the interface for the business logic

```
trait DynAgents[F[_]] {
  def initial: F[WorldView]
  def update(old: WorldView): F[WorldView]
  def act(world: WorldView): F[WorldView]
}
```

and implement it with a *module*. A module depends only on other modules, algebras and pure functions, and can be abstracted over `F`. If an implementation of an algebraic interface is tied to a specific type, e.g. `IO`, it is called an *interpreter*.

```
final class DynAgentsModule[F[_]: Monad](D: Drone[F], M: Machines[F])
  extends DynAgents[F] {
```

The `Monad` context bound means that `F` is *monadic*, allowing us to use `map`, `pure` and, of course, `flatMap` via `for` comprehensions.

We have access to the algebra of `Drone` and `Machines` as `D` and `M`, respectively. Using a single capital letter name is a common naming convention for monad and algebra implementations.

Our business logic will run in an infinite loop (pseudocode)