Explorations in Variance

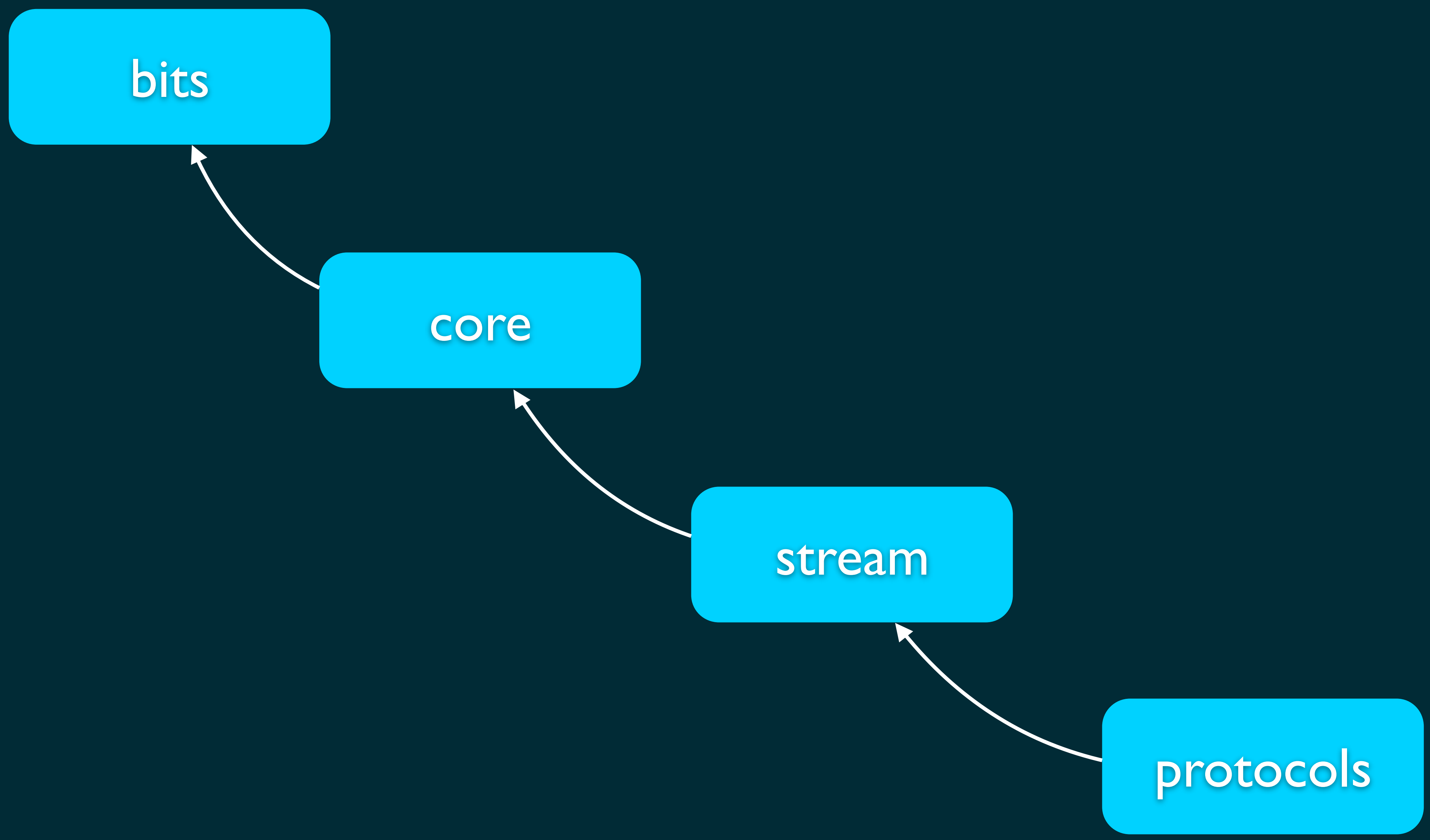
```scala
/** Boxed newtype for F[G[A]]. */
case class Nested[F[_], G[_], A](value: F[G[A]])

/** Nested covariant functors yield a covariant functor. */
implicit def `+[+] = +`[F[_]: Functor, G[_]: Functor]: Functor[({type l[a] = Nested[F, G, a]})#l] =
  new Functor[({type l[a] = Nested[F, G, a]})#l] {
    def map[A, B](nested: Nested[F, G, A])(f: A => B): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.map((ga: G[A]) => ga.map(f))
      Nested(fgb)
    }
  }

/** Contravariant functor in a covariant functor yields a contravariant functor. */
implicit def `+[-] = -`[F[_]: Functor, G[_]: Contravariant]: Contravariant[({type l[a] = Nested[F, G, a]})#l] =
  new Contravariant[({type l[a] = Nested[F, G, a]})#l] {
    def contramap[A, B](nested: Nested[F, G, A])(f: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.map((ga: G[A]) => ga.contramap(f))
      Nested(fgb)
    }
  }

/** Covariant functor in a contravariant functor yields a contravariant functor. */
implicit def `-[+] = -`[F[_]: Contravariant, G[_]: Functor]: Contravariant[({type l[a] = Nested[F, G, a]})#l] =
  new Contravariant[({type l[a] = Nested[F, G, a]})#l] {
    def contramap[A, B](nested: Nested[F, G, A])(f: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.contramap((gb: G[B]) => gb.map(f))
      Nested(fgb)
    }
  }

/** Nested contravariant functors yield a covariant functor. */
implicit def `-[-] = +`[F[_]: Contravariant, G[_]: Contravariant]: Functor[({type l[a] = Nested[F, G, a]})#l] =
  new Functor[({type l[a] = Nested[F, G, a]})#l] {
    def map[A, B](nested: Nested[F, G, A])(f: A => B): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.contramap((gb: G[B]) => gb.contramap(f))
      Nested(fgb)
    }
  }

/** Covariant functor in an invariant functor yields an invariant functor. */
implicit def `i[+] = i`[F[_]: InvariantFunctor, G[_]: Functor]: InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] =
  new InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] {
    def xmap[A, B](nested: Nested[F, G, A], f: A => B, g: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb: F[G[B]] = fga.xmap((ga: G[A]) => ga.map(f), (gb: G[B]) => gb.map(g))
      Nested(fgb)
    }
  }

/** Contravariant functor in an invariant functor yields an invariant functor. */
implicit def `i[-] = i`[F[_]: InvariantFunctor, G[_]: Contravariant]: InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] =
  new InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] {
    def xmap[A, B](nested: Nested[F, G, A], f: A => B, g: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb: F[G[B]] = fga.xmap((ga: G[A]) => ga.contramap(g), (gb: G[B]) => gb.contramap(f))
      Nested(fgb)
    }
  }

/** Invariant functor in a covariant functor yields an invariant functor. */
implicit def `+[i] = i`[F[_]: Functor, G[_]: InvariantFunctor]: InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] =
  new InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] {
    def xmap[A, B](nested: Nested[F, G, A], f: A => B, g: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb: F[G[B]] = fga.map((ga: G[A]) => ga.xmap(f, g))
      Nested(fgb)
    }
  }

/** Invariant functor in a contravariant functor yields an invariant functor. */
implicit def `-[i] = i`[F[_]: Contravariant, G[_]: InvariantFunctor]: InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] =
  new InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] {
    def xmap[A, B](nested: Nested[F, G, A], f: A => B, g: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb: F[G[B]] = fga.contramap((gb: G[B]) => gb.xmap(g, f))
      Nested(fgb)
    }
  }

/** Invariant functor in an invariant functor yields an invariant functor. */
implicit def `i[i] = i`[F[_]: InvariantFunctor, G[_]: InvariantFunctor]: InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] =
  new InvariantFunctor[({type l[a] = Nested[F, G, a]})#l] {
    def xmap[A, B](nested: Nested[F, G, A], f: A => B, g: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb: F[G[B]] = fga.xmap((ga: G[A]) => ga.xmap(f, g), (gb: G[B]) => gb.xmap(g, f))
      Nested(fgb)
    }
  }
```

What is scodec?

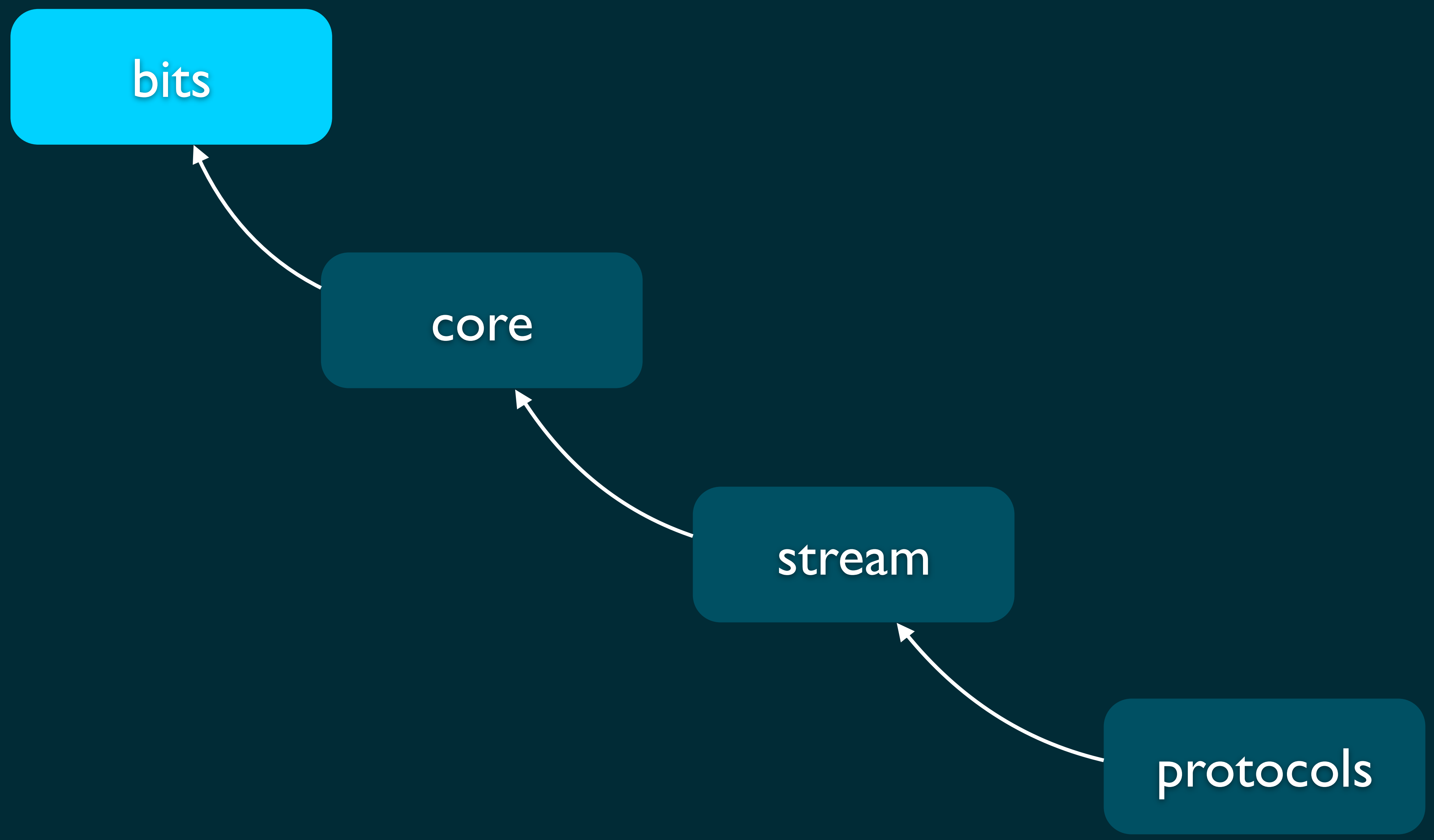
Scala combinator library that supports contract-first and pure functional encoding and decoding of binary data

http://typelevel.org/projects/scodec

Modules



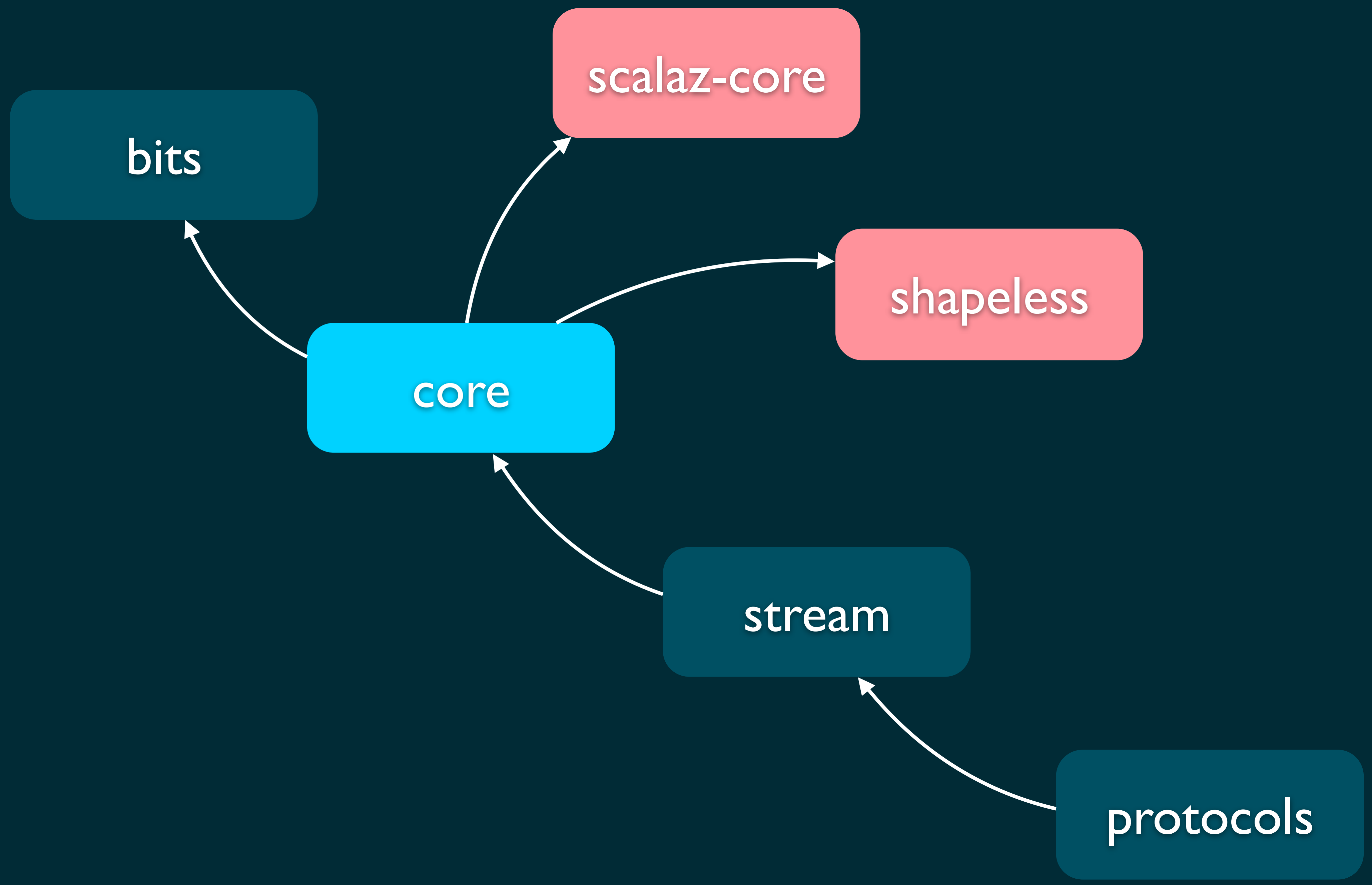
Modules



scodec-bits

- Zero dependencies
- Performant immutable data structures for working with bits (BitVector) and bytes (ByteVector)
- Base conversions (bin, hex, base64)
- CRCs (arbitrary! want a 93-bit CRC with a reflected output register?)
- Lots and lots of methods (>90 on BitVector)

Modules



Design Constraints

- Binary structure should mirror protocol definitions and be self-evident under casual reading
- Mapping of binary structures to types should be statically verified
- Encoding and decoding should be purely functional
- Failures in encoding and decoding should provide descriptive errors
- Compiler plugin should not be used

Example Usage

```scala
case class EthernetFrameHeader(
  destination: MacAddress,
  source: MacAddress,
  ethertypeOrLength: Int
) {
  def length: Option[Int] =
    (ethertypeOrLength <= 1500).option(ethertypeOrLength)
  def ethertype: Option[Int] =
    (ethertypeOrLength > 1500).option(ethertypeOrLength)
}

object EthernetFrameHeader {
  implicit val codec: Codec[EthernetFrameHeader] = {
    val macAddress = Codec[MacAddress]
    ("destination" | macAddress) ::
    ("source"      | macAddress) ::
    ("ethertype"   | uint16)
  }.as[EthernetFrameHeader]
}
```

```scala
trait Decoder[+A] {
  def decode(bits: BitVector): String \/ (BitVector, A)
}
```

Input

Error

Decoded Value

Remainder

```scala
trait Decoder[+A] {
  def decode(bits: BitVector): String \/ (BitVector, A)
}
```

- Decoder is a *type constructor*
  - applying a *proper type* to Decoder yields a *proper type*
  - e.g., applying Int to Decoder yields Decoder[Int]

```
trait Decoder[+A] {
  def decode(bits: BitVector): String \/ (BitVector, A)
}
```

- Decoder is *covariant* in its only type parameter
  ⟹ Decoder[X] a subtype of Decoder[Y] if X is a subtype of Y
- `<:<` shorthand for "is a subtype of"
- e.g., `Decoder[Int] <:< Decoder[AnyVal] <:< Decoder[Any]`

```scala
trait Encoder[-A] {
  def encode(value: A): String \/ BitVector
}
```

Input   Error   Encoded Value

```scala
trait Encoder[-A] {
  def encode(value: A): String \/ BitVector
}
```

- Encoder is *contravariant* in its only type parameter
- Encoder[X] <:< Encoder[Y] if Y <:< X
- e.g., Encoder[Animal] <:< Encoder[Cat]

Higher-Order Subtyping Variance

Subtyping Transitivity

```
trait Coll[+A]

trait List[+A] extends Coll[A]
```

`List[Int] <:< Coll[AnyVal]` ?

Given type constructors F and G and proper types X and Y:

$$F <:< G \land X <:< Y \Rightarrow F[X] <:< G[Y]$$

Variance Annotations

- Type constructor parameter variance is specified via *variance annotations*
  - + for covariance, - for contravariance
- If not annotated, parameter is *invariant*, meaning there is no subtyping relationship between F[X] and F[Y] for given distinct proper types X and Y
- +/- has origins in literature - see papers on Higher Order Subtyping Polarization by Cardelli, Steffen, Pierce, et al

Bivariance

- Bivariance defines a subtyping relationship where F[X] < : < F[Y] for all X, Y
- Scala does not support bivariance

n-ary Type Constrs

- Subtype relationship for type constructors with multiple arguments follows from single arg
- Intuition: consider each parameter in isolation and logical-and the result
- Example:
  - given F[+A, +B] and proper types X_1, Y_1, X_2, Y_2
    $X_1 <:< X_2 \wedge Y_1 <:< Y_2 \Rightarrow F[X_1, Y_1] <:< F[X_2, Y_2]$

n-ary Type Constrs

Given:

```
Dog <:< Pet
Person <:< Owner
```

Which are true?

✗ `Dog => Owner <:< Pet => Owner` `f(cat)`

✔ `Pet => Owner <:< Dog => Owner`

✔ `Dog => Person <:< Dog => Owner`

✗ `Dog => Owner <:< Dog => Person` `f(dog).height`

Function Types

Scala models a single argument function as a binary type constructor

```
trait Function1[-A, +B] {
  def apply(a: A): B
}
```

Variance Positions

- Covariant params cannot appear in argument lists of methods

```
scala> trait Foo[+A] { def f(a: A): Unit }

<console>:7: error: covariant type A occurs in contravariant position
in type A of value a
 trait Foo[+A] { def f(a: A): Unit }
                       ^
```

- Contravariant params cannot appear as a return type of methods

```
scala> trait Bar[-A] { def g(): A }

<console>:7: error: contravariant type A occurs in covariant position
in type ()A of method g
 trait Bar[-A] { def g(): A }
                     ^
```

Declaration Site

- Scala supports *declaration-site* (or definition-site) variance annotations
  - Parameters of type constructors are annotated
- C# also has declaration-site variance and uses *out* for covariant and *in* for contravariant
- Contrast with *use-site* annotations, where the type is not annotated
  - e.g., Java generics + wildcards

Use Site

```
public interface List<A> {
  void add(A a);
  A head();
  A set(int idx, A a);
  void clear();
}
```

Use Site

```
public interface List<A> {
  void add(A a);
  A head();
  A set(int idx, A a);
  void clear();
}
```

```
public <A> void covariant(List<? extends A> list) {
  // can call head or clear
  // cannot call add or set
}
```

```
App.java:4: add(capture#772 of ? extends A) in List<capture#772 of ? extends A> cannot be
applied to (A)
    list.add(list.head());
        ^
```

Use Site

```java
public interface List<A> {
  void add(A a);
  A head();
  A set(int idx, A a);
  void clear();
}
```

```java
public <A> void contravariant(List<? super A> list) {
  // can call add or clear
  // cannot call head or set
}
```

Use Site

```
public interface List<A> {
  void add(A a);
  A head();
  A set(int idx, A a);
  void clear();
}
```

```
public <A> void bivariant(List<?> list) {
  // can only call clear
}
```

Use Site

```java
public interface List<A> {
  void add(A a);
  A head();
  A set(int idx, A a);
  void clear();
}
```

```java
public <A> void invariant(List<A> list) {
  // can call all methods
}
```

Use Site

- Use site variance is generally harder to work with but is more flexible in some circumstances
- Recent research by John Altidor et al incorporates declaration site variance in to Java, without removing use site variance
    - See "Taming the Wildcards: Combining Definition- and Use-Site Variance"
    - May end up as JEP (http://mail.openjdk.java.net/pipermail/compiler-dev/2014-April/008745.html)

Cheating Decl Variance

Consider the beginnings of an immutable list:

```scala
trait List[+A] {
  def head: A
}
```

Challenge: add a cons method

✗

```scala
trait List[+A] {
  def head: A
  def cons(a: A): List[A]
}
```

✔

```scala
trait List[+A] {
  def head: A
  def cons[AA >: A](a: AA): List[AA]
}
```

Composition

```
trait List[+A] {
  def head: A
}
```

```
trait Ord[-A] {
  def compare(x: A, y: A): Order
}
```

What is the variance of:

```
type Foo[A] = List[List[A]]

type Bar[A] = Ord[Ord[A]]

type Baz[A] = List[Ord[A]]

type Qux[A] = Ord[List[A]]
```

Composition

- Scala rules:
  - covariance preserves variance
  - contravariant flips variance
  - invariant makes invariance

Composition

More generally…

Let ⊗ be an associative binary relation between type params to a higher kinded type

⊗	bi	+	-	inv
bi	bi	bi	bi	inv
+	bi	+	-	inv
-	bi	-	+	inv
inv	inv	inv	inv	inv

Composition

```scala
trait List[+A] {
  def head: A
}
```

```scala
trait Ord[-A] {
  def compare(x: A, y: A): Order
}
```

What is the variance of:

`type Foo[A] = List[List[A]]`	covariant
`type Bar[A] = Ord[Ord[A]]`	covariant
`type Baz[A] = List[Ord[A]]`	contravariant
`type Qux[A] = Ord[List[A]]`	contravariant

Composition

```scala
trait Function1[-A, +B] {
  def apply(a: A): B
}
```

What is the variance of A, B, and C in:

```scala
type F[A, B, C] = (A => B) => C

type F[A, B, C] = Function1[Function1[A, B], C]

type F[A, B, C] = Function1[Function1[A, B], C]
```

contravariant

A: $- \otimes - = +$

B: $- \otimes + = -$

C: $+$

```
scala> implicitly[F[Int, AnyVal, Int] <:< F[AnyVal, Int, AnyVal]]
res0: <:<[(Int => AnyVal) => Int,(AnyVal => Int) => AnyVal] = <function1>
```

…back to scodec

Mapping over a Decoder

```
trait Decoder[+A] {
  def decode(bits: BitVector): String \/ (BitVector, A)

  def map[B](f: A => B): Decoder[B] = ???
}
```

Mapping over a Decoder

```scala
trait Decoder[+A] { self =>
  def decode(bits: BitVector): String \/ (BitVector, A)

  def map[B](f: A => B): Decoder[B] = new Decoder[B] {
    def decode(bits: BitVector) =
      self.decode(bits) map { case (rem, a) => (rem, f(a)) }
  }
}
```

Abstracting over mappability

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

Laws:

- identity:
  map fa identity = fa
- composition:
  f: A => B, g: B => C
  map (map fa f) g = map fa (g compose f)

Decoder Functor

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


object Decoder {
  implicit val functorInstance: Functor[Decoder] =
    new Functor[Decoder] {
      def map[A, B](decoder: Decoder[A])(f: A => B) =
        decoder.map(f)
    }
}
```

Note that Decoder has more structure than just Functor…

```scala
trait Decoder[+A] { self =>
  def decode(bits: BitVector): String \/ (BitVector, A)

  def map[B](f: A => B): Decoder[B] = new Decoder[B] {
    def decode(bits: BitVector) =
      self.decode(bits) map { case (rem, a) => (rem, f(a)) }
  }

  def flatMap[B](f: A => Decoder[B]): Decoder[B] =
    new Decoder[B] {
      def decode(bits: BitVector) =
        self.decode(bits) flatMap { case (rem, a) =>
          f(a).decode(rem)
        }
    }

}
```

Models a dependency, where the next decoder is based on the previously decoded value

Decoder Monad

```scala
object Decoder {

  def point[A](a: => A): Decoder[A] = new Decoder[A] {
    private lazy val value = a
    def decode(bits: BitVector) = \/.right((bits, value))
  }

  implicit val monadInstance: Monad[Decoder] =
    new Monad[Decoder] {
      def point[A](a: => A) = Decoder.point(a)
      def bind[A, B](d: Decoder[A])(f: A => Decoder[B]) =
        d.flatMap(f)
    }
}
```

Won't be discussing monads in this talk, but note that a monad gives rise to a functor via
map f = bind(point compose f)

Encoder Functor?

```
trait Encoder[-A] {
  def encode(value: A): String \/ BitVector

  def map[B](f: A => B): Encoder[B] = ???
}
```

Encoder Functor?

```scala
trait Encoder[-A] {
  def encode(value: A): String \/ BitVector

  def map[B](f: A => B): Encoder[B] = new Encoder[B] {
    def encode(value: B) = self.encode(???)
  }
}
```

- We need a value of type A
- We have:
  - value of type B
  - function from A to B

Let's reverse the arrow on the function…

Encoder ~~Functor~~

```
trait Encoder[-A] {
  def encode(value: A): String \/ BitVector

  def map[B](f: B => A): Encoder[B] = new Encoder[B] {
    def encode(value: B) = self.encode(???)
  }
}
```

Encoder ~~Functor~~

```
trait Encoder[-A] {
  def encode(value: A): String \/ BitVector

  def map[B](f: B => A): Encoder[B] = new Encoder[B] {
    def encode(value: B) = self.encode(f(value))
  }
}
```

Encoder ???

```
trait Encoder[-A] {
  def encode(value: A): String \/ BitVector

  def contramap[B](f: B => A): Encoder[B] = new Encoder[B] {
    def encode(value: B) = self.encode(f(value))
  }
}
```

Contravariant Functor

Abstracting over the ability to contramap gives:

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

Laws:

- identity:
  contramap fa identity = fa
- composition:
  f: B => A, g: C => B
  contramap (contramap fa f) g =
  contramap fa (f compose g)

Contravariant Functor

- Contravariant Functor is just a Functor with "arrows" reversed
- Functor is also known as Covariant Functor
  - This abbreviation is common in both the programming community and in category theory

Contravariant Encoder

```scala
trait Contravariant[F[_]] {
  def contramap[A, B](fa: F[A])(f: B => A): F[B]
}

object Encoder extends EncoderFunctions {

  implicit val contraInstance: Contravariant[Encoder] =
    new Contravariant[Encoder] {
      def contramap[A, B](e: Encoder[A])(f: B => A) = e contramap f
    }
}
```

Codec

```
trait Codec[A] extends Decoder[A] with Encoder[A] {
  // Lots of combinators for building new codecs
}
```

- A codec support both encoding and decoding a value of type A
- Must be defined invariant in type A due to Decoder being covariant and encoder being contravariant

Codec

```
trait Codec[A] extends Decoder[A] with Encoder[A] {
  // Lots of combinators for building new codecs
}
```

- Let’s add a map-like operation for converting a Codec[A] to a Codec[B]
  - Note that given c: Codec[A] and f: A => B, calling c map f yields a Decoder[B]
  - Similarly, with g: B => A, c contramap g yields an Encoder[B]
  - We want map-like to retain ability to encode and decode

Let the types guide us…

```
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def mapLike[B](???): Codec[B] = new Codec[B] {
    def encode(b: B): String \/ BitVector = ???
    def decode(bv: BitVector): String \/ (BitVector, B) = ???
  }
}
```

We want encoding to behave like the original codec, so we probably should reuse the original encode…

```
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def mapLike[B](???): Codec[B] = new Codec[B] {
    def encode(b: B): String \/ BitVector = self.encode(???)
    def decode(bv: BitVector): String \/ (BitVector, B) = ???
  }
}
```

Need an A to pass to self.encode but we only have a B

Let's "dependency inject" the conversion

```
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def mapLike[B](g: B => A): Codec[B] = new Codec[B] {
    def encode(b: B): String \/ BitVector = self.encode(g(b))
    def decode(bv: BitVector): String \/ (BitVector, B) = ???
  }
}
```

Decode should behave like original decode but decoded value should be converted from a B to an A

```
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def mapLike[B](g: B => A): Codec[B] = new Codec[B] {
    def encode(b: B): String \/ BitVector = self.encode(g(b))
    def decode(bv: BitVector): String \/ (BitVector, B) =
      self.decode(bv).map { case (rem, a) => (rem, ???) }
  }
}
```

More dependency injection!

```scala
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def mapLike[B](f: A => B, g: B => A): Codec[B] = new Codec[B] {
    def encode(b: B): String \/ BitVector = self.encode(g(b))
    def decode(bv: BitVector): String \/ (BitVector, B) =
      self.decode(bv).map { case (rem, a) => (rem, f(a)) }
  }
}
```

```scala
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def xmap[B](f: A => B, g: B => A): Codec[B] = new Codec[B] {
    def encode(b: B): String \/ BitVector = self.encode(g(b))
    def decode(bv: BitVector): String \/ (BitVector, B) =
      self.decode(bv).map { case (rem, a) => (rem, f(a)) }
  }
}
```

- To convert from Codec[A] to Codec[B], we need a pair of functions A => B and B => A
- Let’s call this operation xmap

Invariant Functor

Abstracting over the ability to xmap gives:

```
trait InvariantFunctor[F[_]]  {
  def xmap[A, B](ma: F[A], f: A => B, g: B => A): F[B]
}
```

Laws:

- identity:
  xmap fa identity identity = fa
- composition:
  f_1: A => B, g_1: B => A, f_2: B => C, g_2: C => B
  xmap (xmap fa f_1 g_1) f_2 g_2 =
  xmap fa (f_2 compose f_1) (g_1 compose g_2)

Invariant Functor

- Invariant Functor is also known as Exponential Functor
- xmap is also known as invmap
- Every covariant functor is an invariant functor
  xmap f g = map f
- Every contravariant functor is an invariant functor
  xmap f g = contramap g

Codec

```
trait Codec[A] extends Decoder[A] with Encoder[A] {
  def xmap[B](f: A => B, g: B => A): Codec[B] = …
}
```

What about dependent codecs? Recall:

```
trait Decoder[+A] { self =>
  def flatMap[B](f: A => Decoder[B]): Decoder[B] = …
  …
}
```

Codec#flatMap “forgets” how to encode

Codec

Can we define flatMap directly?

```
trait Codec[A] extends Decoder[A] with Encoder[A] {
  def xmap[B](f: A => B, g: B => A): Codec[B] = …
  def flatMap[B](f: A => Codec[B]): Codec[B] = ???
}
```

Codec

Straightforward to define decode:

```
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def xmap[B](f: A => B, g: B => A): Codec[B] = …
  def flatMap[B](f: A => Codec[B]): Codec[B] = new Codec[B] {
    def encode(a: B): BitVector = ???
    def decode(bv: BitVector): String \/ (BitVector, B) = (for {
        a <- DecodingContext(self.decode)
        b <- DecodingContext(f(a).decode)
      } yield (a, b)).run(buffer)
  }
}
```

But impossible to define encode…

Codec

- We are trying to solve this domain specific problem:
  - when decoding, first decode using a Decoder[A] and then use the decoded A value to determine how to decode the remaining bits in to a B
  - when encoding, first encode using an Encoder[A] and then use that A to generate an Encoder[B] and encode that

Codec

We can implement this explicitly:

```
trait Codec[A] extends Decoder[A] with Encoder[A] { self =>
  def flatZip[B](f: A => Codec[B]): Codec[(A, B)] =
    new Codec[(A, B)] {
      override def encode(t: (A, B)) =
        Codec.encodeBoth(self, f(t._1))(t._1, t._2)
      override def decode(buffer: BitVector) = (for {
        a <- DecodingContext(self.decode)
        b <- DecodingContext(f(a).decode)
      } yield (a, b)).run(buffer)
    }
}
```

There are many other domain specific combinators e.g., combining a Codec[A] and Codec[B] in to a Codec[(A, B)]

Codec

```
trait Codec[A] extends Decoder[A] with Encoder[A] { … }
```

- Syntactically, transforming codecs can be difficult
  - Can’t use map/contramap/flatMap without forgetting behavior
- Can we incrementally convert a Codec[A] in to a Codec[B]?
- Intuition: don’t forget stuff

GenCodec

```scala
trait GenCodec[-A, +B] extends Encoder[A] with Decoder[B] { … }

trait Codec[A] extends GenCodec[A, A] { … }
```

- GenCodec lets the encoding type vary from the decoding type
- We want it to remember encoding behavior when transforming decoding behavior and vice-versa

GenCodec

```scala
trait GenCodec[-A, +B] extends Encoder[A] with Decoder[B] { self =>

  override def map[C](f: B => C): GenCodec[A, C] =
    new GenCodec[A, C] {
      def encode(a: A) = self.encode(a)
      def decode(bits: BitVector) =
        self.decode(bits).map { case (rem, b) => (rem, f(b)) }
    }

  override def contramap[C](f: C => A): GenCodec[C, B] =
    new GenCodec[C, B] {
      def encode(c: C) = self.encode(f(c))
      def decode(bits: BitVector) = self.decode(bits)
    }
}
```

GenCodec

Once transformations are done, we need a way to convert back to a Codec

```
trait GenCodec[-A, +B] extends Encoder[A] with Decoder[B] { self =>

  override def map[C](f: B => C): GenCodec[A, C] = …

  override def contramap[C](f: C => A): GenCodec[C, B] = …

  def fuse[AA <: A, BB >: B](implicit ev: BB =:= AA): Codec[BB] =
    new Codec[BB] {
      def encode(c: BB) = self.encode(ev(c))
      def decode(bits: BitVector) = self.decode(bits)
    }
}
```

Profunctor

Can we abstract out some form of xmap?

```
trait Profunctor[F[_, _]] { self =>

  def mapfst[A, B, C](fab: F[A, B])(f: C => A): F[C, B]

  def mapsnd[A, B, C](fab: F[A, B])(f: B => C): F[A, C]

  def dimap[A, B, C, D](fab: F[A, B])(f: C => A)(g: B => D): F[C, D] =
    mapsnd(mapfst(fab)(f))(g)
}
```

- dimap is a generalization of xmap
- Note similarities of mapfst/contramap and mapsnd/map

Profunctor

- Profunctors abstract binary type constructors that are contravariant in first parameter and covariant in second parameter
- Also known as Difunctor (due to Meijer/Hutton)
- Most famous Profunctor is the single argument function

Correspondence?

Correspondence

- Disclaimer: IANACT (I am not a category theorist)
- Subtyping covariance and contravariance come from category theory
  - from a specific category where objects are types and morphisms are “is a subtype of”
- Functor typeclasses come from category theory
  - from specific categories where objects are types and morphisms are functions

Postulate

The subtyping transform binary relation manifests in functor typeclasses by considering derived functors from nested functors *

⊗	bi	+	-	inv
bi	bi	bi	bi	inv
+	bi	+	-	inv
-	bi	-	+	inv
inv	inv	inv	inv	inv

* This may be completely false! IANACT

```scala
/** Boxed newtype for F[G[A]]. */
case class Nested[F[_], G[_], A](value: F[G[A]])
```

Nested covariant functors yield a covariant functor

```
implicit def `+[+] = +`[
  F[_]: Functor,
  G[_]: Functor
]: Functor[({type l[a] = Nested[F, G, a]})#l] =
  new Functor[({type l[a] = Nested[F, G, a]})#l] {
    def map[A, B](nested: Nested[F, G, A])(f: A => B): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.map((ga: G[A]) => ga.map(f))
      Nested(fgb)
    }
  }
```

Contravariant functor in a covariant functor yields a contravariant functor

```scala
implicit def `+[-] = -`[
  F[_]: Functor,
  G[_]: Contravariant
]: Contravariant[({type l[a] = Nested[F, G, a]})#l] =
  new Contravariant[({type l[a] = Nested[F, G, a]})#l] {
    def contramap[A, B](nested: Nested[F, G, A])(f: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.map((ga: G[A]) => ga.contramap(f))
      Nested(fgb)
    }
  }
```

Covariant functor in a contravariant functor yields a contravariant functor

```
implicit def `-[+] = -`[
  F[_]: Contravariant,
  G[_]: Functor
]: Contravariant[({type l[a] = Nested[F, G, a]})#l] =
  new Contravariant[({type l[a] = Nested[F, G, a]})#l] {
    def contramap[A, B](nested: Nested[F, G, A])(f: B => A): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.contramap((gb: G[B]) => gb.map(f))
      Nested(fgb)
    }
  }
```

Nested contravariant functors yield a covariant functor

```
implicit def `-[-] = +`[
  F[_]: Contravariant,
  G[_]: Contravariant
]: Functor[({type l[a] = Nested[F, G, a]})#l] =
  new Functor[({type l[a] = Nested[F, G, a]})#l] {
    def map[A, B](nested: Nested[F, G, A])(f: A => B): Nested[F, G, B] = {
      val fga = nested.value
      val fgb = fga.contramap((gb: G[B]) => gb.contramap(f))
      Nested(fgb)
    }
  }
```

More at https://github.com/mpilquist/variance-explorations/blob/master/variance.scala

Further Reading

- “Taming the Wildcards: Combining Definition- and Use-Site Variance” by Altidor http://cgi.di.uoa.gr/~smaragd/variance-pldi11.pdf
- “Polarized Higher-Order Subtyping” by Steffen http://home.ifi.uio.no/msteffen/download/diss/diss.pdf
- “Higher-Order Subtyping for Dependent Types” by Abel http://cs.ioc.ee/~tarmo/tsem11/abel-slides.pdf

Further Reading

- "Rotten Bananas" by Kmett
  http://comonad.com/reader/2008/rotten-bananas/
- "I love profunctors. They're so easy." by HU
  https://www.fpcomplete.com/school/to-infinity-and-beyond/pick-of-the-week/profunctors
- "What's up with Contravariant?"
  http://www.reddit.com/r/haskell/comments/1vc0mp/whats_up_with_contravariant/

Questions?