scodec / **scodec.github.io**

Branch: master ▾ **scodec.github.io** / guide / **Combined+Pages.md** Find file Copy path

mpilquist Revert "updated site" f773bd3 on Sep 9, 2016

1 contributor

749 lines (495 sloc) 39.6 KB

scodec

scodec is a suite of libraries for working with binary data. Support ranges from simple, performant data structures for working with bits and bytes to streaming encoding and decoding.

There are three primary modules:

- scodec-bits - Zero dependency library that provides persistent data structures, `BitVector` and `ByteVector`, for working with binary.
- scodec-core - Combinator based library for encoding/decoding values to/from binary.
- scodec-stream - Binding between scodec-core and scalaz-stream that enables streaming encoding/decoding.

There are a few secondary modules as well:

- scodec-scalaz - Binding between scodec-core and scalaz, which provides typeclass instances for the types from scodec-bits and scodec-core.
- scodec-spire - Binding between scodec-core and spire, mostly taking advantage of unsigned numeric types.
- scodec-protocols - Library of general purpose implementations of common protocols.

This guide goes over each of these modules in detail.

Getting Source, Binaries, and Docs

Each of the modules are available on GitHub under the scodec organization. Binaries are published to Maven Central under the group id `org.scodec`. ScalaDoc is available for online browsing at typelevel.org. The ScalaDoc has a lot of detail, especially in package level documentation.

All of the modules adhere to binary compatibility rules. In short, versions that share the same major.minor version number are forward binary compatible. An exception to this rule is major version 0, which indicates that no binary compatibility is guaranteed from version to version. For example, code that was compiled against scodec-bits 1.0.1 will function with 1.0.4 but not necessarily 1.0.0 or 1.1.0.

Getting Help

To get help with scodec, consider using the Typelevel mailing list, using the scodec tag on StackOverflow, or mentioning #scodec on Twitter.

scodec-bits

The scodec-bits library contains data structures for working with binary. It has no dependencies, which allows it to be used by other libraries without causing dependency conflicts.

There are two primary data structures in the library, `ByteVector` and `BitVector`. Both are immutable collections and have performance characteristics that are optimized for use in the other scodec modules. However, each type has been designed for general purpose usage, even when other scodec modules are not used. For instance, `ByteVector` can be safely used as a replacement for immutable byte arrays.

ByteVector

The `ByteVector` type is isomorphic to a `scala.collection.immutable.Vector[Byte]` but has much better performance characteristics. A `ByteVector` is represented as a balanced binary tree of chunks. Most operations have asymptotic performance that is logarithmic in the depth of this tree. There are also quite a number of convenience based features, like value based equality, a sensible `toString`, and many conversions to/from other data types.

It is important to note that `ByteVector` does not extend any types from the Scala collections framework. For instance, `ByteVector` is *not* a `scala.collection.immutable.Traversable[Byte]`. This allows some deviance, like `Long` based indexing instead of `Int` based indexing from standard collections. Additionally, it avoids a large category of bugs, especially as the standard library collections are refactored. Nonetheless, the methods on `ByteVector` are named to correspond with the methods in the standard library when possible.

Getting Started

Let's create a `ByteVector` from a literal hexadecimal string:

```
scala> import scodec.bits._
import scodec.bits._

scala> val x: ByteVector = hex"deadbeef"
x: scodec.bits.ByteVector = ByteVector(4 bytes, 0xdeadbeef)

scala> val y: ByteVector = hex"DEADBEEF"
y: scodec.bits.ByteVector = ByteVector(4 bytes, 0xdeadbeef)

scala> x == y
res0: Boolean = true
```

We first start by importing all members of the `scodec.bits` package, which contains the entirety of this library. We then create two byte vectors from hexadecimal literals, using the `hex` string interpolator. Finally, we compare them for equality, which returns true, because each vector contains the same bytes.

Constructing Byte Vectors

There are a variety of ways to construct byte vectors. The `hex` string interpolator is useful for testing and REPL experiments but often, it is necessary to construct byte vectors from other data types. Most commonly, a byte vector must be created from a standard library collection, like a `Vector[Byte]` or `Array[Byte]`, or a Java NIO `ByteBuffer`. This is accomplished with the `apply` method on the `ByteVector` companion.

When constructing a `ByteVector` from an array, the array contents are *copied*. This is the safest behavior, as any mutations to the original byte array do not cause problems with the immuable `ByteVector`. However, the cost of copying can be prohibitive in some situations. To address this, a byte array can be converted to a `ByteVector` with a constant time operation -- `ByteVector.view(array)`. Using `view` requires the byte array to never be modified after the vector is constructed.

`ByteVector`s can also be created from strings in various bases, like hexadecimal, binary, or base 64. For example, to convert a hexadecimal string to a `ByteVector`, use `ByteVector.fromHex(string)`. There are quite a number of methods related to base conversions -- explore the ScalaDoc of the `ByteVector` companion object for details.

BitVector

The `BitVector` type is similar to `ByteVector` with the exception of indexing bits instead of bytes. This allows access and update of specific bits (via `apply` and `update`) as well as storage of a bit count that is not evenly divisible by 8.

Getting Started

```
scala> val x: BitVector = bin"00110110101"
x: scodec.bits.BitVector = BitVector(11 bits, 0x36a)

scala> val y: BitVector = bin"00110110100"
y: scodec.bits.BitVector = BitVector(11 bits, 0x368)

scala> x == y
res0: Boolean = false

scala> val z = y.update(10, true)
z: scodec.bits.BitVector = BitVector(11 bits, 0x36a)

scala> x == z
res1: Boolean = true
```

In this example, we create two 10-bit vectors using the `bin` string interpolator that differ in only the last bit. We then create a third vector, `z` , by updating the 10th bit of `y` to true. Comparing `x` and `y` for equality returns false whereas comparing `x` and `z` returns true.

Constructing Bit Vectors

`BitVector` s are constructed in much the same way as `ByteVector` s. That is, typically via the `apply` and `view` methods in the `BitVector` companion object. Additionally, any `ByteVector` can be converted to a `BitVector` via the `bits` method (e.g., `myByteVector.bits`) and any `BitVector` can be converted to a `ByteVector` via the `bytes` method.

TODO unfold, nio

Transforms

Both `ByteVector` s and `BitVector` s support a number of different transformations.

Collection Like Transforms

TODO

Bitwise Transforms

TODO

Base Conversions

TODO

Cyclic Redundancy Checks

TODO

scodec-core

Getting Started

```
scala> import scodec.Codec

scala> import scodec.codecs.implicits._

scala> case class Point(x: Int, y: Int)

scala> case class Line(start: Point, end: Point)

scala> case class Arrangement(lines: Vector[Line])

scala> val arr = Arrangement(Vector(
  Line(Point(0, 0), Point(10, 10)),
  Line(Point(0, 10), Point(10, 0))))
arr: Arrangement = ...

scala> val arrBinary = Codec.encode(arr).require
arrBinary: scodec.bits.BitVector =
  BitVector(288 bits, 0x0000000200000000000000000000000a0000000a000000000000000a0000000a00000000)

scala> val decoded = Codec[Arrangement].decode(arrBinary).require.valid
decoded: Arrangement = Arrangement(Vector(Line(Point(0,0),Point(10,10)), Line(Point(0,10),Point(10,0))))
```

We start by importing the primary type in scodec-core, the `Codec` type, along with all implicit codecs defined in `scodec.codecs.implicits`. The latter provides commonly useful implicit codecs, but is opinionated -- that is, it decides that a `String` is represented as a 32-bit signed integer whose value is the string length in bytes, followed by the UTF-8 encoding of the stirng.

Aside: the predefined implicit codecs are useful at the REPL and when your application does not require a specific binary format. However, scodec-core is designed to support "contract-first" binary formats -- ones in which the format is fixed in stone. For binary serialization to arbitrary formats, consider tools like scala-pickling, Avro, and protobuf.

We then create three case classes followed by instantiating them all and assigning the result to the `arr` val. We encode `arr` to binary using `Codec.encode` followed by a call to 'require', then decode the resulting binary back to an `Arrangement`. In this example, both encoding and decoding rely on an implicitly available `Codec[Arrangement]`, which is automatically derived based on *compile time* reflection on the structure of the `Arrangement` class and its product types.

We use `encode(...).require`, which throws an `IllegalArgumentException` if encoding fails, because we know that our arrangement codec cannot fail to encode. To decode, we summon the implicit arrangement codec via `Codec[Arrangement]` and then use `decode(...).require.value` for REPL convenience -- which throws an `IllegalArgumentException` if decoding fails and throws away any bits left over after decoding finishes. In this case, we know that decoding will succeed and there will be no remaining bits, so this is safe. It is generally better to avoid use of `require`, as it is unsafe -- because it may throw.

Running the same code with a different implicit `Codec[Int]` in scope changes the output accordingly:

```
scala> import scodec.codecs.implicits.{ implicitIntCodec => _, _ }

scala> implicit val ci = scodec.codecs.uint8
ci: scodec.Codec[Int] = 8-bit unsigned integer

...

scala> val arrBinary = Codec.encode(arr).require
arrBinary: scodec.bits.BitVector = BitVector(72 bits, 0x0200000a0a000a0a00)

scala> val decoded = Codec.decode[Arrangement](arrBinary).require.value
decoded: Arrangement = Arrangement(Vector(Line(Point(0,0),Point(10,10)), Line(Point(0,10),Point(10,0))))
```

In this case, we import all predefined implicits except for the `Codec[Int]` and then we define an implicit `Int` codec for 8-bit unsigned big endian integers. The resulting encoded binary is 1/4 the size. However, our arrangement codec is no longer total in encoding -- that is, it may result in errors. Consider:

```
scala> val arr2 = Arrangement(Vector(
  Line(Point(0, 0), Point(10, 10)),
  Line(Point(0, 10), Point(10, -1))))
arr2: Arrangement = Arrangement(Vector(Line(Point(0,0),Point(10,10)), Line(Point(0,10),Point(10,-1))))

scala> val encoded = Codec.encode(arr2)
encoded: scodec.Attempt[scodec.bits.BitVector] =
  Failure(lines/1/end/y: -1 is less than minimum value 0 for 8-bit unsigned integer)

scala> val encoded = Codec.encode(arr2).require
java.lang.IllegalArgumentException: lines/1/end/y: -1 is less than minimum value 0 for 8-bit unsigned inte
```

Attempting to encode an arrangement that contains a point with a negative number resulted in an error being returned from `encode` and an exception being thrown from `require`. The error includes the path to the error -- `lines/1/end/y`. In this case, the `lines` field on `Arrangement`, the line at the first index of that vector, the `end` field on that line, and the `y` field on that point.

If you prefer to avoid using implicits, do not fret! The above example makes use of implicits and uses Shapeless for compile time reflection, but this is built as a layer on top of the core algebra of scodec-core. The library supports a usage model where implicits are not used.

With the first example under our belts, let's look at the core algebra in detail.

Core Algebra

We saw the `Codec` type when we used it to encode a value to binary and decode binary back to a value. The ability to decode and encode come from two fundamental traits, `Decoder` and `Encoder`. Let's look at these in turn.

Decoder

```
case class DecodeResult[A](value: A, remainder: BitVector) { ... }

trait Decoder[+A] {
  def decode(b: BitVector): Attempt[DecodeResult[A]]
}
```

A decoder defines a single abstract operation, `decode`, which attempts to convert a bit vector into a `DecodeResult`. A `DecodeResult` is a case class made up of a value of type `A` and a remainder -- bits that are left over, or unconsumed, after decoding has completed. For example, a decoder that decodes a 32-bit integer returns an error when the supplied vector has less than 32-bits, and returns the supplied vector less 32-bits otherwise.

The result type is an `Attempt[A]`, which is equivalent to an `Either[scodec.Err, A]`. `Err` is an open-for-subclassing data type that contains an error message and a context stack. The context stack contains strings that provide context on where the error occurred in a large structure. We saw an example of this earlier, where the context stack represented a path through the `Arrangement` class, into a `Vector`, and then into a `Line` and `Point`. The type is open-for-subclassing so that codecs can return domain specific error types and then pattern match on the received type. An `Err` is *not* a subtype of `Throwable`, so it cannot be used (directly) with `scala.util.Try`. Also note that codecs never throw exceptions (or should never!). All errors are communicated via the `Err` type.

map

A function can be mapped over a decoder, resulting in our first combinator:

```
trait Decoder[+A] { self =>
  def decode(b: BitVector): Attempt[DecodeResult[A]]
  def map[B](f: A => B): Decoder[B] = new Decoder[B] {
    def decode(b: BitVector): Attempt[DecodeResult[B]] =
      self.decode(b) map { result => result map f }
  }
}
```

Note that the *implementation* of the `map` method is not particularly important -- rather, the type signature is the focus.

As a first use case for `map`, consider creating a decoder for the following case class by reusing the built-in `int32` codec:

```
case class Foo(x: Int)
val fooDecoder: Decoder[Foo] = int32 map { i => Foo(i) }
```

emap

The `map` operation does not allow room for returning an error. We can define a variant of `map` that allows the supplied function to indicate error:

```
trait Decoder[+A] { self =>
  ...
  def emap[B](f: A => Attempt[B]): Decoder[B] = new Decoder[B] {
    def decode(b: BitVector): Attempt[DecodeResult[B]] =
      self.decode(bits) flatMap { result =>
        f(result.value).map { b => DecodeResult(b, result.remainder) }
      }
  }
}
```

flatMap

Further generalizing, we can `flatMap` a function over a decoder to express that the "next" codec is *dependent* on the decoded value from the current decoder:

```
trait Decoder[+A] { self =>
  ...
  def flatMap[B](f: A => Decoder[B]): Decoder[B] = new Decoder[B] {
    def decode(b: BitVector): Attempt[DecodeResult[B]] =
      self.decode(b) flatMap { result =>
        val next: Codec[B] = f(result.value)
        next.decode(result.remainder)
      }
  }
}
```

The resulting decoder first decodes a value of type `A` using the original decoder. If that's successful, it applies the decoded value to the supplied function to get a `Decoder[B]` and then decodes the bits remaining from decoding `A` using that decoder.

As mentioned previously, `flatMap` models a dependency between a decoded value and the decoder to use for the remaining bits. A good use case for this is a bit pattern that first encodes a count followed by a number of records. An implementation of this is not provided because we will see it later in a different context.

Encoder

```
trait Encoder[-A] {
  def encode(a: A): Attempt[BitVector]
  def sizeBound: SizeBound
}
```

An encoder defines two abstract operations, `encode` , which converts a value to binary or returns an error, and `sizeBound` , which provides an upper and lower bound on the size of the bit vectors created by the encoder. This design differs from other libraries by allowing `encode` to be defined partially over type `A` . For example, this allows an integer encoder to be defined on a subset of the integers without having to resort to newtypes or wrapper types.

contramap

A function can be mapped over an encoder, similar to `map` on decoder, but unlike `map` , the supplied function has its arrow reversed -- that is, we convert an `Encoder[A]` to an `Encoder[B]` with a function `B => A` . This may seem strange at first, but all we are doing is using the supplied function to convert a `B` to an `A` and then delegating the encoding logic to the original `Encoder[A]` . This operation is called `contramap` :

```
trait Encoder[-A] { self =>
  def encode(a: A): Attempt[BitVector]
  def contramap[B](f: B => A): Encoder[B] = new Encoder[B] {
    def encode(b: B): Attempt[BitVector] =
      self.encode(f(b))
    def sizeBound: SizeBound =
      self.sizeBound
  }
}
```

econtramap

Like decoder's `map` , `contramap` takes a total function. To use a partial function, there's `econtramap` :

```
trait Encoder[-A] { self =>
  ...
  def econtramap[B](f: B => Attempt[A]): Encoder[B] = new Encoder[B] {
    def encode(b: B): Attempt[BitVector] =
      f(b) flatMap self.encode
    def sizeBound: SizeBound =
      self.sizeBound
  }
}
```

flatMap?

Unlike decoder, there is no `flatMap` operation on encoder. Further, there's no "corresponding" operation -- in the way that `contramap` corresponds to `map` and `econtramap` corresponds to `emap` . To get a feel for why this is, try defining a `flatMap` -like method. For instance, you could try "reversing the arrows" and substituting `Encoder` for `Decoder` , yielding a method like `def flatMapLike[B](f: Encoder[B] => A): Encoder[B]` -- but you'll find there's no reasonable way to implement `encode` on the returned encoder.

Codec

We can now implement `Codec` as the combination of an encoder and decoder:

```
trait Codec[A] extends Encoder[A] with Decoder[A]
```

A codec has no further abstract operations -- leaving it with only `encode` and `decode` , along with a number of derived operations like `map` and `contramap` . However, at least as presented here, calling `map` on a codec results in a decoder and calling `contramap` on a codec results in an encoder -- effectively "forgetting" how to encode and decode respectively. We need a new set of combinators for working with codecs that does not result in forgetting information.

xmap

The codec equivalent to `map` and `contramap` is called `xmap` :

```
trait Codec[A] extends Encoder[A] with Decoder[A] { self =>
  def xmap[B](f: A => B, g: B => A): Codec[B] = new Codec[B] {
    def encode(b: B) = self.contramap(g).encode(b)
    def decode(b: BitVector) = self.map(f).decode(b)
    def sizeBound = self.sizeBound
  }
}
```

Here, we've defined `xmap` in terms of `map` and `contramap` . The `xmap` operation is one of the most commonly used operations in scodec-core. Consider this example:

```
case class Point(x: Int, y: Int)

val tupleCodec: Codec[(Int, Int)] = ...
val pointCodec: Codec[Point] = tupleCodec.xmap[Point](t => Point(t._1, t._2), pt => (pt.x, pt.y))
```

We convert a `Codec[(Int, Int)]` into a `Codec[Point]` using the `xmap` operation, passing two functions -- one that converts from a `Tuple2[Int, Int]` to a `Point` , and another that converts a `Point` to a `Tuple2[Int, Int]` . Note: there are a few simpler ways to define codecs for case classes that we'll see later.

exmap

In a similar fashion to `emap` and `econtramap` , the `exmap` operation is like `xmap` but allows both functions to be defined partially:

```
trait Codec[A] extends Encoder[A] with Decoder[A] { self =>
  ...
  def exmap[B](f: A => Attempt[B], g: B => Attempt[A]): Codec[B] = new Codec[B] {
    def encode(b: B) = self.econtramap(g).encode(b)
    def decode(b: BitVector) = self.emap(f).decode(b)
    def sizeBound = self.sizeBound
  }
}
```

Additional transforms

Unlike `map` , `emap` , `contramap` , and `econtramap` , `xmap` and `exmap` each take two functions. `xmap` takes two total functions and `exmap` takes two partial functions. There are two other operations that take two conversion functions -- where one of the functions is total and the other is partial.

```
trait Codec[A] extends Encoder[A] with Decoder[A] { self =>
  ...
  def narrow[B](f: A => Attempt[B], g: B => A): Codec[B] = exmap(f, right compose g)
  def widen[B](f: A => B, g: B => Attempt[A]): Codec[B] = exmap(right compose f, g)
}
```

Finally, there's a variant of `widen` where the partial function is represented as a `B => Option[A]` instead of a `B => Attempt[A]` .

```scala
trait Codec[A] extends Encoder[A] with Decoder[A] { self =>
  ...
  def widenOpt[B](f: A => B, g: B => Option[A]): Codec[B] = ...
}
```

The `widenOpt` operation is provided to make manual authored case class codecs simpler -- by passing the `apply` and `unapply` methods from a case class companion. For instance, the earlier example becomes:

```scala
case class Point(x: Int, y: Int)

val tupleCodec: Codec[(Int, Int)] = ...
val pointCodec: Codec[Point] = tupleCodec.widenOpt(Point.apply, Point.unapply)
```

GenCodec

The `xmap` and related operations allow us to transform a `Codec[A]` into a `Codec[B]`. Nonetheless, we can improve the definitions of the decoder and encoder specific methods (`map` , `contramap` , etc.). With the types as presented, we said that calling `map` on a codec forgot the encoding logic and returned a decoder, and that calling `contramap` on a codec forgot the decoding logic and returned an encoder.

We can remedy this somewhat by introducing a new type that is similar to `Codec` in that it is both an `Encoder` and a `Decoder` -- but dissimilar in that it allows the encoding type to differ from the decoding type.

```scala
trait GenCodec[-A, +B] extends Encoder[A] with Decoder[B] { self =>
  override def map[C](f: B => C): GenCodec[A, C] = GenCodec(this, super.map(f))
  override def contramap[C](f: C => A): GenCodec[C, B] = GenCodec(super.contramap(f), this)
  ...
  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)
    def sizeBound = self.sizeBound
  }
}

object GenCodec {
  def apply[A, B](encoder: Encoder[A], decoder: Decoder[B]): GenCodec[A, B] = new GenCodec[A, B] {
    override def encode(a: A) = encoder.encode(a)
    override def decode(bits: BitVector) = decoder.decode(bits)
    def sizeBound = encoder.sizeBound
  }
}

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

A `GenCodec` represents the pairing of an `Encoder` and a `Decoder` , with potentially different types. Each of the combinators from `Encoder` and `Decoder` are overridden such that they return `GenCodec` s that "remember" the behavior of the non-transformed type. For instance, the `map` operation on a `GenCodec` changes the decoding behavior while remembering the encoding behavior.

Hence, `GenCodec` has two type parameters -- the first is the encoding type and the second is the decoding type. Any time that the two types are equal, the `fuse` method can be used to convert the `GenCodec[A, A]` to a `Codec[A]` .

`GenCodec` is useful because it allows *incremental* transforms to be applied to a codec. Further, it plays an important role in the categorical view of codecs, which is discussed later. Still, direct usage of `GenCodec` is rare.

Variance

You may have noticed the variance annotations in `Encoder` , `Decoder` , and `GenCodec` , and the lack of a variance annotation in `Codec` . Specifically:

- `Encoder` is defined contravariantly in its type parameter
- `Decoder` is defined covariantly in its type parameter
- `GenCodec` is defined contravariantly in its first type parameter and covariantly in its second type parameter
- `Codec` is defined invariantly in its type parameter

The variance annotations -- specifically the contravariant ones -- can cause problems with implicit search. At the current time, the implicit search problems cannot be fixed without making `Encoder` invariant. The authors of scodec believe the utility provided by subtyping variance outweighs the inconvenience of the implicit search issues they cause. If you disagree, please weigh-in on the mailing list or the related pull request.

For the categorically minded

The core types have a number of type class instances. Note that this section assumes a strong familiarity with the major typeclasses of functional programming and can be safely skipped.

`Decoder` has a monad instance, where `flatMap` is defined as above and the point operation is defined as:

```
def point(a: A): Decoder[A] = new Decoder[A] {
  def decode(b: BitVector): Attempt[DecodeResult[A]] =
    Attempt.successful(DecodeResult(a, b))
}
```

`Encoder` has a contravariant functor instance, defined using the `contramap` operation from above. It also has a corepresentable instance with `Attempt[BitVector]` .

`GenCodec` has a profunctor instance, where `mapfst` is implemented using `contramap` and `mapsnd` is implemented using `map` .

`Codec` has an invariant (aka exponential) functor instance, using the `xmap` operation from above.

Instances for the Scalaz versions of each these type classes are located in the scodec-scalaz module, which is discussed later.

scodec-core defines one additional type class, `Transform` , which abstracts over the type constructor in the transform operations. It defines a single abstract operation -- `exmap` -- and defines concrete versions of `xmap` , `narrow` , `widen` , etc. in terms of `exmap` . This type class is unlikely to be useful outside of scodec libraries due to the use of `scodec.Err` . It exists in order to share transform API between `Codec` and another scodec-core type we'll see later.

Manually creating codecs

Codecs are typically created by transforming or combining other codecs. However, we can create a codec manually by writing a class that extends `Codec` and implements `sizeBound` , `encode` , and `decode` .

```
class BitVectorCodec(size: Long) extends Codec[BitVector] {
  def sizeBound = SizeBound.exact(size)
  def encode(b: BitVector) = {
    if (b.size == size) Attempt.successful(b)
    else Attempt.failure(Err(s"expected size ${size} but got ${b.size}"))
  }
  def decode(b: BitVector) = {
    val (result, remaining) = b.splitAt(size)
    if (result.size != size)
      Attempt.failure(new Err.InsufficientBits(size, result.size))
    else Attempt.successful(DecodeResult(result, remaining))
  }
}
```

Besides the fundamental types -- `Codec`, `Decoder`, and `Encoder` -- scodec-core is focused on providing *combinators*. That is, providing ways to combine two or more codecs in to a new codec, or transform a single codec in to another. We've seen a few examples of combinators (e.g., `xmap`) and we'll see many more.

The combinators exist to make codecs easier to read and write. They promote correctness by allowing a codec to be built from components that are known to be correct. They increase readability by encapsulating boilerplate and wiring logic, leaving the structure of the codec evident. However, it is easy to get distracted by searching for an elegant combinator based codec implementation when a manually authored codec is appropriate. As you work with scodec, you'll develop an intuition for when to write codecs manually.

Summary

The `Codec` type is the work horse of scodec-core, which combines a `Decoder` with an `Encoder`. Codecs can be transformed in a variety of ways -- and we'll see many more ways in later sections. However, we can always fall back to implementing `Codec` or even `Decoder` or `Encoder` directly if a combinator based approach proves inconvenient.

Simple Value Codecs

There are a number of pre-defined codecs for simple value types provided by the `scodec.codecs` object. In this section, we'll look at some of these.

BitVector and ByteVector

One of the simplest codecs is an identity for `BitVector`s. That is, a `Codec[BitVector]` that returns the supplied bit vector from `encode` and `decode`. This is provided by the `scodec.codecs.bits` method. This codec has some interesting properties -- it is both *total* and *greedy*. By total, we mean that it never returns an error from `encode` or `decode`. By greedy, we mean that the `decode` method always consumes the entire input bit vector and returns an empty bit vector as the remaining bits.

The greedy property may seem strange, or at least more specialized than codec for a fixed number of bits -- for instance, a constant width binary field. However, non-greedy codecs can often be built out of greedy codecs. We'll see a general combinator for doing so later, in the Framing section.

Nonetheless, constant width binary fields occur often enough to warrant their own built-in constructor. The `scodec.codecs.bits(size: Long)` method returns a `Codec[BitVector]` that decodes exactly `size` bits from the supplied vector, failing to decode with an `Err.InsufficientBits` error if there are less than `size` bits provided. If a bit vector less than `size` bits is supplied to `encode`, it is right-padded with 0s.

Similarly, the `scodec.codecs.bytes` and `scodec.codecs.bytes(size: Int)` methods return a greedy `Codec[ByteVector]` and a fixed-width `Codec[ByteVector]`, where the latter's size is specified in bytes instead of bits.

Booleans

The `bool` codec is a `Codec[Boolean]` which encodes a 1-bit vector where `0` represents `false` and `1` represents true.

There's an overload of `bool` which takes a bit count -- `bool(n)` -- which also is a `Codec[Boolean]`. When decoding, it treats `n` consecutive `0`s as `false` and all other vectors as `true`. When encoding, `true` is encoded as `n` consecutive `1`s.

Numerics

Codecs are also provided for various numeric types -- `Int`, `Long`, `Short`, `Float`, and `Double`. Let's consider first the integral types, `Int`, `Long`, and `Short`, followed by the non-integral types.

Integral Types

There are a number of predefined integral codecs defined by methods named according to the form:

```
[u]int${size}[L]
```

where `u` stands for unsigned, `size` is replaced by one of `8, 16, 24, 32, 64`, and `L` stands for little-endian. For each codec of that form, the type is `Codec[Int]` or `Codec[Long]` depending on the specified size. Signed integer codecs use the 2's complement encoding.

For example, `int32` supports 32-bit big-endian 2s complement signed integers, and `uint16L` supports 16-bit little-endian unsigned integers. Note: `uint64` and `uint64L` are not provided because a 64-bit unsigned integer does not fit in to a `Long`.

Additionally, methods of the form `[u]int[L](size: Int)` and `[u]long[L](size: Int)` exist to build arbitrarily sized codecs, within the limitations of `Int` and `Long`. Hence, a 13-bit unsigned integer codec is given by `uint(13)`.

Similarly, `Short` codecs are provided by `short16`, `short16L`, `short(size)`, `shortL(size)`, `ushort(size)`, and `ushortL(size)`. The signed methods take a size up to `16` and the unsigned methods take a size up to `15`.

Non-Integral Types

`Float` and `Double` codecs are provided by `float` and `double`. Both use IEEE754, with the former represented as 32-bits and the latter represented as 64-bits.

Strings

`String` s are supported by a variety of codecs.

The rawest form is the `string` method, which takes an implicit `java.nio.charset.Charset`. The resulting codec encodes strings using the supplied charset -- that is, all of the heavy lifting of converting each character to binary is handled directly by the charset. Hence, the `string` codec is nothing more than glue between the `Codec` type and `Charset`.

There are two convenience codecs defined as well, `utf8` and `ascii`, which are simply aliases for the `string(charset)`, passing the `UTF-8` and `US-ASCII` charsets.

Codecs returned from `string`, including `utf8` and `ascii`, are greedy. The byte size / character count is *not* encoded in the binary, and hence, it is not safe to decode a vector that has been concatenated with another vector. For example:

```
scala> val pair = utf8 ~ uint8
pair: scodec.Codec[(String, Int)] = (UTF-8, 8-bit unsigned integer)

scala> val enc = pair.encode(("Hello", 48))
enc: scodec.Attempt[scodec.bits.BitVector] =
  Successful(BitVector(48 bits, 0x48656c6c6f30))

scala> pair.decode(enc.require)
res1: scodec.Attempt[scodec.DecodeResult[(String, Int)]] =
  Failure(cannot acquire 8 bits from a vector that contains
0 bits)
```

Here, we create a `Codec[(String, Int)]` using the `utf8` codec and a `uint8` codec. We then encoded the pair `("Hello", 48)` and then tried to decode the result. However, we got a failure indicating there were not enough bits. Let's try decoding the resulting vector using the `utf8` codec directly:

```
scala> utf8.decode(enc.require)
res2: scodec.Attempt[scodec.DecodeResult[String]] =
    Successful(DecodeResult(Hello0,BitVector(empty)))
```

The result is `"Hello0"` , not `"Hello"` as expected. The `utf8` codec decoded the entire vector, including the `0x30` byte that originally was written by the `uint8` codec.

This greediness property is a feature of `string` -- often, the size of a string field in a binary protocol is provided by some external mechanism. For example, by a record size field that is defined in another part of the message.

There are alternatives to `string` , `utf8` , and `ascii` that encode the string's byte size in the binary -- `string32` , `utf8_32` , and `ascii32` . Executing the same example as above with `utf8_32` instead of `utf8` yields the expected result:

```
scala> val pair = utf8_32 ~ uint8
pair: scodec.Codec[(String, Int)] = (string32(UTF-8), 8-bit unsigned integer)

scala> val enc = pair.encode(("Hello", 48))
enc: scodec.Attempt[scodec.bits.BitVector] =
  Successful(BitVector(80 bits, 0x0000000548656c6c6f30))

scala> val dec = pair.decode(enc.require)
dec: scodec.Attempt[scodec.DecodeResult[(String, Int)]] =
  Successful(DecodeResult((Hello,48),BitVector(empty)))
```

By looking at the encoded binary, we can see that the byte size of the string was encoded in a 32-bit field the preceeded the encoded string.

In order to handle size delimited string fields, like the above, except with artibrary size fields, we can use the `variableSizeBytes` combinator along with `string` , `utf8` , or `ascii` . The `variableSizeBytes` combinator is covered in more detail in a later section. For now though, consider the following example, which encodes the byte size of the string in an unsigned little-endian 18-bit integer field.

```
scala> val str18 = variableSizeBytes(uintL(18), utf8)
str18: scodec.Codec[String] = variableSizeBytes(18-bit unsigned integer, UTF-8)

scala> str18.encode("Hello")
res0: scodec.Attempt[scodec.bits.BitVector] =
  Successful(BitVector(58 bits, 0x050012195b1b1bc))
```

This has the same benefits as `utf8_32` except it allows for an arbitrary size field, rather than being limited to a 32-bit size field.

Simple Constructors and Combinators

Constants

Constant codecs are codecs that always encode a specific bit pattern. For example, the `constant(bin"1110")` always encodes the bit pattern `1110` . When decoding, a constant codec consumes the same number of bits as its constant value, and then either validates that the consumed bits match the constant value or ignores the consumed bits. Codecs created by `constant(...)` perform validation when decoding and codecs created by `constantLenient(...)` ignore the consumed bits.

For example, decoding `0000` with each technique yields:

```
scala> val x = constant(bin"1110").decode(bin"0000")
x: scodec.Attempt[scodec.DecodeResult[Unit]] =
  Failure(expected constant BitVector(4 bits, 0xe) but got BitVector(4 bits, 0x0))

scala> val y = constantLenient(bin"1110").decode(bin"0000")
```

```
y: scodec.Attempt[scodec.DecodeResult[Unit]] =
  Successful(DecodeResult((),BitVector(empty)))
```

The `constant(...)` method returns a `Codec[Unit]`, which may seem odd at first glance. Unit codecs occur frequently in scodec. They are used for a variety of use cases, including encoding a predefined value, decoding a specific pattern, manipulating the remainder during decoding, or raising errors from encoding/decoding.

Literal Constants

When working with binary formats that make heavy use of constant values, manually wrapping each constant bit pattern with a `constant` method can be verbose. This verbosity can be avoided by importing implicit conversions that allow treating binary literals as codecs.

```
scala> import scodec.codecs.literals._
import scodec.codecs.literals._

scala> val c: Codec[Unit] = bin"1110"
c: scodec.Codec[Unit] = constant(BitVector(4 bits, 0xe))
```

Unit Codecs

Any codec can be turned in to a unit codec -- that is, a `Codec[Unit]` -- using the `unit` method on the `Codec` type. The resulting codec encodes and decodes using the original codec, but the decoded value is thrown away, and the value to encode is "fixed" at tht time the unit codec is generated.

For example:

```
scala> val c = int8.unit(-1)
c: scodec.Codec[Unit] = ...

scala> val enc = c.encode(())
enc: scodec.Attempt[scodec.bits.BitVector] =
  Successful(BitVector(8 bits, 0xff))

scala> val dec = c.decode(bin"00000000 00000001")
dec: scodec.Attempt[scodec.DecodeResult[Unit]] =
  Successful(DecodeResult((),BitVector(8 bits, 0x01)))
```

In this example, the value to encode is fixed to `-1` at the time `c` is created. Hence, every call to encode results in a call to `int8.encode(-1)`. Decoding is interesting in that it consumed 8 bits of the vector.

In general, converting a codec to a unit codec may seem like a useless operation. However, it plays an important role in both tuple codecs and heterogeneous list codecs, which are both covered at length later.

Context

Recall that `scodec.Err` includes a context stack along with a message and any error type specific data. The errors generated by the built-in codecs typically do not provide context in errors. Rather, context can be provided using the `withContext` method on `Codec`.

For example:

```
scala> val noContext = int8.decode(bin"1111")
noContext: scodec.Attempt[scodec.DecodeResult[Int]] =
  Failure(cannot acquire 8 bits from a vector that contains 4 bits)

scala> val oneContext = int8.withContext("x").decode(bin"1111")
oneContext: scodec.Attempt[scodec.DecodeResult[Int]] =
  Failure(x: cannot acquire 8 bits from a vector that contains 4 bits)
```

```
scala> val twoContext = int8.withContext("x").withContext("y").decode(bin"1111")
twoContext: scodec.Attempt[scodec.DecodeResult[Int]] =
  Failure(y/x: cannot acquire 8 bits from a vector that contains 4 bits)
```

Each of these examples fails with the same error, with the exception of the context stack.

Context stacking is typically used to provide the path in to a large structure, in the same way that lenses peer in to a large structure.

The `|` operator on a `String` is provided as an alias for `withContext` with the arguments reversed -- e.g., `"x" | codec` is equivalent to `codec.withContext("x")`. In standalone examples, `withContext` is typically clearer but in large structures, where each field in a record is labelled with its name, the `|` syntax may be clearer.

Miscellaneous

There are a number of other miscellaneous combinators that are useful.

Complete

TODO - complete, compact, withToString

Collections

TODO

Framing

TODO - fixed size, variable size, etc.

Tuple Codecs

TODO

HList Codecs

TODO

Case Class Codecs

TODO

Coproduct Codecs

TODO