Code-follows-Type Programming in Scala

Adriaan Moors

June 26, 2007

This article introduces Scala programmers to what I call "Code-follows-Type Programming" (CfT). CfT's goal is scrapping repetitive code ("boilerplate") that is fully described by the structure of types. As our "killer app", we'll implement a serialisation function that can turn data of any (representable) type into a list of bits.

1 Introduction

The main goal of this article is to introduce Scala programmers to "Code-follows-Type Programming" (CfT), so all the examples will be in Scala. There's no need for a background in Haskell or the research on "Datatype-Generic Programming", as CfT is called in functional programming languages.

As our "killer app", we'll be implementing type-safe persistence once and for all! By definition, persistence (solely) uses the information already present in the definition of the class hierarchy to construct a certain representation of objects that are instances of these classes. We already defined the classes, they contain all the information necessary, why should we ever have to write another serialisation method again?

There are many CfT approaches (most of them in Haskell) that allow persistence to be defined once and for all. The code in this article is essentially a Scala port of Hinze's Generics for the Masses. Because of its relative simplicity, it makes for a great introduction to CfT programming in Scala. If you're interested in exploring the rich literature on Datatype-Generic Programming, see e.g., the work of Gibbons, Hinze, Jeuring, and Lämmel.

2 Reasoning about Types

If you want your functions to be derived from the structure of the data-types they manipulate, you better make sure you can reason about those types. One way of doing that, is choosing a small vocabulary in which we can express all the types we want to represent. This is the part where the difficult choices must be made, and the ones we make in this article probably aren't optimal for a language like Scala. They do work out pretty well for the example at hand.

An easy way of decomposing types into a small vocabulary is the sum-of-products view. In Haskell, a data type has a fixed number of associated data constructors: it's a

disjoint sum (a "tagged union") of data constructors. These data constructors are the only way to construct values of that type, and they are determined completely by the values of their arguments: a data constructor definition thus corresponds to a product (a tuple) of argument types.

In Scala, a Haskell data-type corresponds to a sealed abstract class with case classes as subclasses that play the role of the data constructors. The fact that these case classes are types too is an important difference between Scala and Haskell. It is one of the complications that must be dealt with in a full-blown CfT library.

Besides sums and products, we need a number of "core" ingredients. Some representable types cannot be decomposed further (e.g., numbers and characters). They have too many (or too few) values to represent them as a disjoint sum of a finite (non-zero) number of alternatives, and they lack structure that could be decomposed further. We simply add them to our core vocabulary. Note that there are also types that aren't representable, such as function types.

There is one more important piece of the puzzle missing. Let's try to express a simple type using our vocabulary. Consider a model of a simplistic view of people's opinions on the fabric for, say, a couch:

```
sealed trait FabricOpinion
  case class BoyOpinion extends FabricOpinion // undecided
    about spots
  case class GirlOpinion(opinion: String, ferocity: Int)
    extends FabricOpinion
```

`FabricOpinion` is the type of interest. Encoding it yields `<<FabricOpinion>> = () + (<<String>>, Int)`: it's either a `BoyOpinion`, which does not convey any additional information ("()" is the empty tuple or product), or a `GirlOpinion`, which is fully characterised by the comment and its ferocity. Therefore, our encoding is sound. Finally, a note on the similarities with Haskell: only `FabricOpinion` would be considered a type, with two data constructors to construct either a man or a woman's opinion on fabric.

I didn't go all the way in the reduction, though. `String` is not one of our core types. It can be reduced to a list of characters, and a list can be expressed as either the empty list or a concatenation of an element and a list. Thus: `<<List[a]>>=()+(<<a>>, <<List[a]>>)` and `<<String>>=<<List[char]>>`. This demonstrates the last core "type": recursion. An encoding must be able to reference itself. We'll reuse Scala's support for recursion, to avoid dealing with it explicitly using a fixed-point operator.

3 Functions that work on any Representable Type

Now we have determined the vocabulary for expressing representable types, we can fix the shape of a function that deals with any representable type:

```
trait GenFun[Result] {
  def apply[t](x: t): Result = ...
```

```
// these "cases" specify the function's behaviour for every
    core type
  def unit: Unit ⇒ Result
  def char: Char ⇒ Result
  def int:  Int  ⇒ Result
  def either[a, b]: Either[a, b] ⇒ Result
  def both[a, b]:   Pair[a, b]  ⇒ Result
}
```

A `GenFun` only specifies its result type (`Result`), since it is applicable to any representable type anyway. This motivates `apply`'s signature (remember that Scala models functions as objects with an `apply` method). The other methods define the function in terms of its behaviour for each core type. For example, the `unit` method returns a function that takes *any* empty tuple (ahem) and produces the result of the function. More interestingly, the `char` case is described by a function that takes a `Char` and returns a `Result`. Finally, `either` and `both` define the result for a sum or a product, respectively. We use methods that return functions to deal with recursion.

It couldn't be this simple, of course. The types `Either[a, b]` and `Pair[a, b]` are too general. Nothing stops the user from calling the last two methods with an arbitrary type, maybe even one with values that can't be examined, such as a function type. Thus, we must somehow constrain the type parameters to types we can represent.

Here's where it gets a little mind-bending. Keep in mind that our end goal is invoking a function on a representable type, where that function is defined in terms of a case analysis that deals with the core types.

Thus, `either` receives a value that contains an `a` or a `b`. We must constrain these types so that we can do the case analysis. First, the `Rep[t]` below declares the required functionality:

```
trait Rep[t]{ def rep[R](implicit fun: GenFun[R]): t ⇒ R }
```

An instance of `Rep[t]` will serve as a witness that a value of type `t` is representable. The witness object must provide a method that, given a `GenFun[R]`, selects the right case of this function. E.g., if `t` is `char`, the `rep` method simply calls `fun.char`.

For the actual constraint, Scala provides a view bound construct, `a <% b`, that expresses that it must be possible to transform any value of type `a` into a value of type `b`. This transformation is typically supplied by the programmer as an implicit method (or in fact any implicit value with the right type). See the Scala reference for more information. This way, we can refine `either` and `both` as:

```
def either[a <% Rep[a], b <% Rep[b]]: Either[a, b] ⇒ Result
def both[a <% Rep[a], b <% Rep[b]] : Pair[a, b]  ⇒ Result
```

These are the final signatures for these methods.

Now we can implement the `apply` method (with a signature that's refined similarly to the `either` and `both` methods):

```
def apply[t <% Rep[t]](x: t): Result = {
  val caseFun = x.rep(this)
  caseFun(x)
}
```

Our function can be applied to an `x` of a representable type `t` by retrieving the witness for the fact that `x` is representable: **`val`** `caseFun = x.rep(`**`this`**`)` performs the case analysis on `x` and calls the corresponding method on **`this`**. For example, when `x` is a character, **`this`**`.char` will be asked to supply the function that implements our function for this case. The actual invocation then becomes `caseFun(x)`. (To save some keystrokes, `apply`'s body can be abbreviated to `x.rep(`**`this`**`)(x)`.)

4 Making Types Representable

Now we decided on the structure of a CfT function, let's look at how we can satisfy the `t <% Rep[t]` view bound. The simplest case is to make the `Unit` type representable:

```
implicit def unit2rep(u: Unit) = new Rep[Unit] {
  def rep[R](implicit gen: GenFun[R]): Unit ⇒ R = gen.unit
}
```

The compiler automatically inserts a call to this method whenever a value of type `Unit` is found while `Rep[Unit]` was expected. It simply calls the `unit` method on `gen`, since we know that the value under scrutiny is of type `Unit`. The main functionality of this function is thus realised by the Scala compiler's selection of implicit values.

Surprisingly, making `Either[t, u]`, representable is equally simple:

```
implicit def plus2rep[t <% Rep[t], u <% Rep[u]](p: Either[t,
    u])
  = new Rep[Either[t, u]]{
      def rep[R](implicit gen: GenFun[R]): Either[t, u] ⇒ R
    = gen.either
     }
```

4.1 Representing User-defined Types

To facilitate making user-defined types representable, we complete `GenFun`'s definition with one last method:

```
def datatype[a <% Rep[a], b](iso: Iso[a,b]): b  ⇒ Result
  = (x: b) ⇒ this(iso.fromData(x))
```

That is, if we have an isomorphism between a representable type `a` and some arbitrary type `b`, we can use the isomorphism to transform values of `b` to a value of the representable type `a` and then simply use one of the other cases on the result.

(Where `Iso` is a simple helper, defined as **`case class`** `Iso[t,dt](fromData : dt⇒t, toData: t⇒dt)`.)

So, `List[a]` is made representable using:

```
implicit def list2rep[t <% Rep[t]](l: List[t]) = new Rep[List
    [t]] {
  def rep[R](implicit gen: GenFun[R]): List[t] ⇒ R
```

```
    = gen.datatype(Iso(fromList[t], toList[t]))
}
```

Where `fromList[t]` expresses a `List[t]` in terms a sum of products:

```
  def fromList[t](l: List[t]): Either[Unit, Pair[t, List[t]]]
     = l match {
    case Nil ⇒ Inl(())
    case x :: xs ⇒ Inr(Pair(x, xs))
  }
```

To convert our universal representation of `List[t]` back to its native form:

```
  def toList[t](l: Either[Unit, Pair[t, List[t]]]): List[t] =
     l match {
    case Inl(())⇒ Nil
    case Inr(Pair(x, xs)) ⇒ x :: xs
  }
```

5 Example: from any Representable Type to Bits

Finally, to bring it all together, here's how we can define a function that turns a value of any representable type into a list of `Bits`:

```
object showBin extends GenFun[List[Bit]] {
  def either[a <% Rep[a], b <% Rep[b]] = (x: Either[a, b]) ⇒
     x match {
    case Inl(el) ⇒ _0 :: this(el)
    case Inr(el) ⇒ _1 :: this(el) }
  def both[a <% Rep[a], b <% Rep[b]] = (x: Pair[a, b]) ⇒
    this(x._1) ::: this(x._2)

  def unit = ((x: Unit) ⇒ Nil)
  def char = (x: Char) ⇒ bits(16,x)
  def int = (x: Int) ⇒ bits(32,x)
}
```

To encode a disjoint sum, we must remember whether we encountered the left or the right alternative, and then we can simply encode the chosen alternative. Encoding a pair is the concatenation of the encoding of the first and the second entry. The case for `Unit` returns the empty list and the final two core types rely on a simple helper method (defined below).

Invoking the `showBin` function is as simple as `showBin(List(1,2,3))`.

We use

```
sealed trait Bit
  case object _0 extends Bit
  case object _1 extends Bit
```

to model our "bits", and the following helper function completes the implementation:

```
def bits(n: Int, x: Int): List[Bit]
  = if(n==0) Nil
    else (if((x&1)==1) _1 else _0) :: bits(n-1,x>>1)
```

6 Conclusion

The approach we implemented in this article uses divide and conquer to define functions that are applicable to any representable type. On the one hand, types are made representable by reducing them to a small set of core types: disjoint sums (tagged unions), products (tuples), characters, integers, and the unit type (Java's `void`). On the other hand, to make a function applicable to any representable type, it must be expressed as a case analysis that deals with these 5 core types. We illustrated the approach with a rudimentary type-safe persistence function.

The approach obviously needs to be made more robust before it can be applied in practice, but this certainly seems feasible. Remaining challenges include dealing with subtyping and encapsulation. An important extension of the example would be to add support for de-serialisation. I hope to deal with this in a follow-up article in the near future.

7 Acknowledgements

Burak Emir and Stéphane Micheloud provided helpful comments that improved the presentation of this article.

8 About this article

This article is a work in progress. I look forward to informal peer-review. Suggestions and comments are greatly appreciated! Please email me or register and post a comment.

PDF version also available. | Download the full source code for this example.