



 @AndrzejWasowski

Andrzej Wąsowski

Advanced Programming

Type Classes and Implicits (on the example of a PBT library)

Generation for PBT as an Instance of State

- For property-based testing (PBT) we need to **implement generators**
- First, need random **number generators**, to generate arbitrary random data
- Random number generators can be mapped, flatMapped, and map2ed to generate other values
- Recall class State, implementing the **automaton abstraction** with state space S and outputs A:

```
case class State[S,+A] (run: S => (A,S))
```

- We define generators of A's as an automaton producing A's with RNG as a state space:

```
case class Gen[+A] (sample: State[RNG,A])
```

- **Question:** Why are generators covariant? What this will allow?
- Examples:

Recall: `_.nextInt: RNG => (Int,RNG)`

then `def anyInteger: Gen[Int] =Gen (State (_.nextInt))`

- **Mentimeter:** How do I get an integer number out of anyInteger? (the first one)

How do we create more complex generators?

- Let's begin with a generator of **pairs of integers**, so `Gen[(Int,Int)]`
- Recall the **sequential chaining of automata** with `map2`:

```
(this: State[S,A]) def map2[B,C] (that: State[S,B]) (f: (A,B) =>C): State[S,C] =...
```

- `Gen[A]` is a `State[RNG,A]`, so it has `map2` like above
- We use `map2` to create the **generator of pairs of integers**:

```
def intPair: Gen[(Int,Int)] =Gen (anyInteger.sample.map2 (anyInteger.sample) (xy=>xy))
```

- Note how nicely composable are the libraries we build!
(We use the code from chapter 6)
- This composition is powerful. In combination with implicits we can build a test case generation library for **arbitrary data shapes** (arbitrary data structures)
- **Question:** What is the following generator creating ?

```
Gen (anyInteger.sample.map (x =>x % 100 + 200)): Gen[Int]
```

Generating random lists of integers

- Assume that we have a generator of lists of random integers of length n

```
def listOfN (n: Int): Gen[List[Int]]
```

- **Question:** What is the type of G in

```
val G =Gen (anyInteger.sample.flatMap (n =>listOfN (n).sample))
```

- **Answer:** $\text{Gen}[\text{List}[\text{Int}]]$

- Once we implement `flatMap` for `Gen` (delegating to `State`), we can simplify:

```
val G: Gen[List[Int]] =anyInteger.flatMap (n =>listOfN (n))
```

- This is the first complex data structure we generated!!!
- **Question:** Well, what exactly is the generator G generating?

Generating instances of polymorphic types /1

- Let's return to generating random pairs. Can you do a `Gen[(A,B)]`?

```
def anyPair[A,B]: Gen[(A,B)] =???
```

Below `intPair` as a hint:

```
def intPair :Gen[(Int,Int)] =anyInteger.map2 (anyInteger) (xy=>xy)
```

-
- We seem to lack a way to generate A's and B's! So let's add them as arguments:

```
def anyPair[A,B] (genA: Gen[A], genB: Gen[B]): Gen[(A,B)] =genA.map2 (genB) ( ab =>ab )
```

For simplicity, I am assuming that we have `map2` on `Gen`. Otherwise:

```
def anyPair[A,B] (genA: Gen[A], genB: Gen[B]) =Gen (genA.sample.map2 (genB.sample) (ab =>ab))
```

-
- Similarly, if we wanted a polymorphic generator of lists:

```
def listOfN[A] (n: Int, anyA: Gen[A])=???
```

- Or if the list is to be of the random size:

```
def listOf[A] (anyInt: Gen[Int], anyA: Gen[A])=...
```

- Alternatively toss a coin to see whether the list is long enough:

```
def listOf[A] (anyBool: Gen[Bool], anyA: Gen[A])=...
```

Generating instances of polymorphic types /2

Actual test code from exercises in the prior weeks

- Now when we use `listOf[A]` we have to do something like:

```
listOf[Student] (anyInt, anyStudent)
```

We already have `anyInt`, we just need to implement `anyStudent` (not shown)

- A bit annoying to have to always parameterize all these calls
- We might be able to eliminate `anyInt` but `anyStudent` seems difficult. **Why?**

-
- Now think about the `forAll` function from `ScalaTest`; It could have type like

```
def forAll[A] (p: A => Boolean) (genA: Gen[A]): Prop
```

- In many cases, providing generators would feel **redundant** for the user, as the `forAll` **type parameter already specifies** that we are quantifying over `A`'s
- Particularly annoying if `A` is just a complex library type, like: `List[Stream[Option[(Double, Double)]]]`
Scalatest should know how to generate standard types!
- Should we now write generators **for any combinations of types that programmers imagine???**
- It would be nice for the **compiler to find a generator** for `A` in the library and just use it ...

Implicit arguments as type class constraints /1

- **Formally:** Gen is a **type class** and in order to generate instances of A we need **an instance of this type class** for A so a **value of type Gen[A]**
- In Scala type classes are implemented using **implicit arguments and values**

```
def listOfN[A] (n: Int) (implicit genA: Gen[A]): Gen[List[A]] =... //use genA for generate A's
```

For instance: `... =Gen (State.sequence (List.fill (n, genA.sample)))`

When you use it, in the context something like this must exist

```
implicit val anyStudent: Gen[Student] =...
```

Then: `... listOfN[Student] (5) ...` will work without the last argument

- The compiler will find genA by searching visible implicit values of type Gen[A].
- If there is a single such, it will be bound to genA, and you can use genA in the body
- The compiler **fails** if you call listOfN[A] for a type A for each no implicit Gen[A] instance is found
- `implicit genA: Gen[A]` **constrains** possible types A
- If you want to override the implicitly used argument, you can always **add it explicitly**, as if it was a normal argument: `listOfN[Int] (5) (anyInt): Gen[List[Int]]`

Type Classes and Implicit Values: Odds and Ends

- So `(implicit genA: Gen[A])` is a **type constraint** on A (it must be a type with Gen)
- This is why Scala provides an alternative syntax for this pattern, called **type bounds**:

```
def listOfN[A: Gen] (n: Int): Gen[List[A]] =...
```

Use `'implicitly[Gen[A]]'` to **access the unnamed implicit argument**:

```
... =Gen (State.sequence (List.fill (n, implicitly[Gen[A]].sample)))
```

- Fun fact from `Predef.scala`, `implicitly` is just **identity with an implicit argument**

```
def implicitly[T](implicit e: T): T = e
```

- Finally, type classes as functions (or type class instance generators) are very useful:

```
implicit def listOf[A: Gen] : Gen[List[A]] =...
```

The compiler **will automatically construct** a generator for list of anything that has a generator

E.g., `listOf[List[List[Int]]]` works automatically using the above generator and any `Integer`

- For `listOfN (5)` type inference will often fail, better **add the annotation**: `listOfN[Student] (5)`
- Not only to help the type checker, but to make the code **more self-explanatory**
- In practice you **import or inherit** the implicits in most cases, for standard types
- Note that in **scalatest** the type is not `Gen[A]` but `Arbitrary[A]`, but the idea is the same

Implicit Arguments vs Default Argument Values

```
def listOfN[A: Gen] (n: Int) (implicit genA: Gen[A]) =???  
def listOfN[A: Gen] (n: Int) (genA: Gen[A] =null) =???
```

- Implicit parameters are **more general than default parameter** values
- Unlike for default parameter values, the actual values of implicit parameters are **not known at the implementation and compilation time** of the function
- **For generic parameter types default values do not work**
- What default value should I give for genA, if we do not know what A is?
- Like with default parameters you can **override the value at call site**
- Unlike default parameter values you can also override them at call site **implicitly** (for instance by importing a different set of implicit objects)

Type classes: History and Context

This is not only about Scala ...

- **Implicits** (under this name) are a Scala-specific invention, but some other languages picked them up too (Idris, Agda, Coq, some logic programming languages)
- **Type classes** originally invented by Phil Wadler for Standard ML to allow adding implementation of equality test to new types,
- Type classes are the main extension mechanism in **Haskell**
- **Rust's traits** are a limited form of type class;
- In **F#** there is a neverending debate whether to add or not to add type classes.
- **C++** has recently introduced **Concepts**, which can be used to implement a form of type classes

Type Classes and Implicits: Key Points

- Type classes are a bit like traits and extension methods:
 - You define a type class as a generic class, an interface, a new '**skill**' for a type
 - You can add this new '**skill**', say generation,
 - to any type, like with extension methods,
 - after it has been implemented,
 - without recompiling or otherwise changing the type, and
 - any library that needs generation '**skill**' will recognize it
- Important: whatever code we write, we can constrain its users to provide an instance of the skill (an instance of the type class)
- The library using generation (or any other '**skill**') does not have to know about the type it operates on, it just gets the instance of the '**skill**', the instance is often called **evidence**
- Traits can only be mixed into objects at **creation time**, so they are **not good for extending objects created by legacy code** (for instance a factory method in a legacy library)
- An **extension method implementation must exist** when compiling the code that uses the extension
- For type classes/implicits the extension is bound when **the caller (client) of our code is compiled** (the latest binding of all the discussed mechanism)

A simple example of forAll [back to PBT]

```
1 def forAll[A] (f: A => Boolean) (implicit genA: Gen[A]): Prop = Prop (
2
3   (n, rng) =>                                // number of trials, random seed
4     randomStream (rng)                        // the implicit argument is propagated
5     .zip (Stream.from (0))                    // stream of random A's and indexes
6     .take (n)                                 // we only want to try n times
7     .map { case (a, i) =>                     // run the test and produce the result
8       try { if (f (a)) Passed
9         else Falsified (a.toString, i)
10      } catch { case e: Exception => Falsified (buildMsg (a, e), i) } }
11     .find (_.isFalsified)
12     .getOrElse (Passed)                      // report the first failure or pass
13 )
14
15 def randomStream[A] (rng: RNG) (implicit g: Gen[A]): Stream[A] =
16   Stream.unfold (rng) (rng => Some (g.sample.run (rng)))
17
18 def buildMsg[A] (s: A, e: Exception): String = // uninteresting
19   s"test case: $s\n" +
20   s"generated an exception: ${e.getMessage}\n" +
21   s"stack trace:\n ${e.getStackTrace.mkString("\n")}"
```

Adapted from the textbook, Listing 8.3, to resemble scalatest/scalacheck more