@AndrzejWasowski
Andrzej Wąsowski
Florian Biermann

Advanced Programming

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

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

API for Property Based Testing

What problem we are solving today?

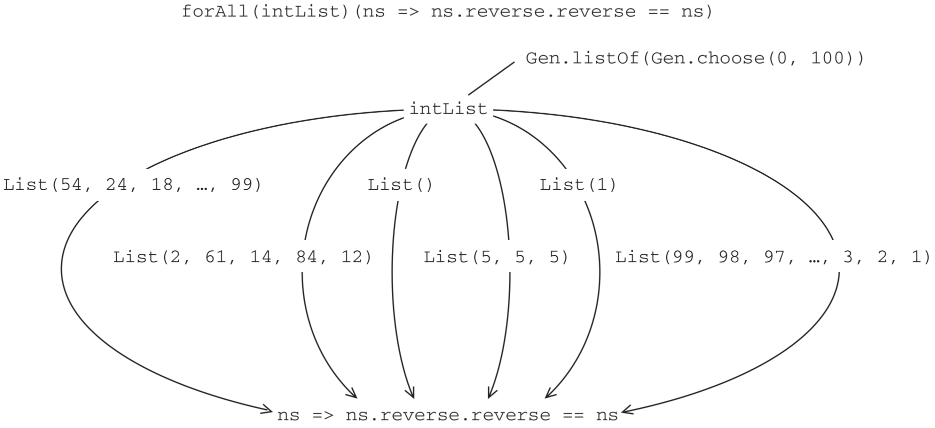
The Problem:

- Design a general property-based testing library like scalacheck
- With generation, property checking, test-case size control

The Solution:

- Implement a type to represent properties
- Implement composable generators (essentially like Rand)
- Minimize user effort to program generators by using type classes

Generators and properties



A `Gen` object generates a variety of different objects to pass to a Boolean expression, searching for one that will make it false.

Figure from Pilquist, Bjarnasson, Chiusano 2022

The Prop Type

Represents a property to test

```
1 opaque type TestCases = Int
2 opaque type MaxSize = Int

4 opaque type Prop = (MaxSize, TestCases, RNG) => Result

6 extension (self: Prop)
7   def && (that: Prop): Prop = ???
8   def || (that: Prop): Prop = ???

10 opaque type FailedCase = String
11 opaque type SuccessCount = Int

13 enum Result:
14   case Passed
15   case Falsified(failure: FailedCase, successes: SuccessCount)

17   def isFalsified: Boolean = this match
18     case Passed => false
19     case Falsified(_, _) => true
```

- Making `Prop` opaque allows to limit the extension just to this type
- Could have used a class, but this has a cost of Boxing
- For the types in lines 7–10 is annoying, as one cannot give these params as simply integers, but it increases safety
- Confusing parameters in a tuple is less likely

The Prop Type

A simple implementation of `forAll` (not sized)

```scala
def randomLazyList[A](g: Gen[A])(rng: RNG): LazyList[A] =
  LazyList.unfold(rng)(rng => Some(g.run(rng)))

def buildMsg[A](s: A, e: Exception): String =
  s"test case: $s\n" +
  s"generated an exception: ${e.getMessage}\n" +
  s"stack trace:\n ${e.getStackTrace.mkString("\n")}"

def forAll[A](as: Gen[A])(f: A => Boolean): Prop = (max, n, rng) =>
  randomLazyList(as)(rng)
    .zip(LazyList.from(0))
    .take(n)
    .map { (a, i) =>
           try if f(a) then Passed else Falsified(a.toString, i)
           catch case e: Exception => Falsified(buildMsg(a, e), i) }
    .find { _.isFalsified }
    .getOrElse(Passed)
```

The `max` parameter is not used in this variant (it is in the sized version)

QuickCheck and ScalaCheck Use Shrinking

We implement sized generation instead

```scala
opaque type MaxSize = Int
opaque type TestCases = Int
opaque type FailedCase = String
opaque type SuccessCount = Int
opaque type Prop = (MaxSize, TestCases, RNG) => Result

// The type of generators bounded by size
opaque type SGen[+A] = Int => Gen[A]

def forAll[A](g: SGen[A])(f: A => Boolean): Prop = (max, n, rng) =>
  val casesPerSize = (n.toInt - 1) / max.toInt + 1
  val props: LazyList[Prop] = LazyList.from(0)
    .take(min(n.toInt, max.toInt) + 1)
    .map { i => forAllNotSized(g(i))(f) } // call the other forAll
  val prop: Prop = props
    .map[Prop] { p => (max, n, rng) => p(max, casesPerSize, rng) }
    .toList
    .reduce { _ && _ }
  prop(max, n, rng)
```

Executing Tests

```scala
opaque type Prop = (MaxSize, TestCases, RNG) => Result

extension (self: Prop)
  def run(
    maxSize: MaxSize = 100,                        // by default objects up to 100 size
    testCases: TestCases = 100,                    // by default try 100 test cases
    rng: RNG = RNG.Simple(System.currentTimeMillis)   // by default use a different seed each time
  ): Boolean =

    self(maxSize, testCases, rng) match
      case Result.Falsified(msg, n) =>
        println(s"Falsified after $n passed tests:\n $msg [message from our Prop framework]")
        false

      case Result.Passed =>
        println(s"+ OK, passed $testCases tests. [message from our Prop framework]")
        true
```

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 `map2`ed to generate other values
- Recall the type `State`, implementing the **automaton abstraction** with state space `S` and outputs `A`:

```scala
opaque type State[S, +A] = S => (A, S)
```

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

```scala
opaque type Gen[+A] = State[RNG, A]
```

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

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

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

- **Mentimeter [6644 6761]:** How do I get an integer number out of `anyInteger`?

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` for `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)] =anyInteger.map2(anyInteger) (xy=>xy)
```

  (because `Gen` is opaque we need to provide a delegation to `State.map2`)

- Note how nicely composable are the libraries we build!
  (We use the code from chapter 6)
- **Question [6644 6761]**: What is the following generator creating ?

```
anyInteger.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 [6644 6761]:** What is the type of `G` in

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

- This needs a delegate for `flatMap` from `Gen` to `State` as well

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)
```

I assume that `map2` on Gen delegates to `State` again.

- 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 ScalaCheck; 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)]]]
```

  ScalaCheck 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 ...

Using arguments as type class constraints /1

- A **type class** is mechanism to add constraints on type variables in generic types
- `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 with **using constraints** and **given** values

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

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

  When you use it, in the context a given value of type `Gen[A]` must exist

```
given val anyStudent: Gen[Student] =... //the user provides this
```

  Then: `... listOfN[Student] (5) ...` will work without the last argument
- The compiler will find `genA` by searching for available given 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 given `Gen[A]` instance is found
- So `using genA: Gen[A]` **constrains** possible types `A`
- If you want to override the **using** 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 Given Values: Odds and Ends

- So `(using 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 `'summon[Gen[A]]'` to **access the unnamed using argument**:

  ```
  ... =sequence (List.fill (n, summon[Gen[A]]))
  ```

- Fun fact from Predef.scala, `summon` is just **identity with a constraint**

  ```
  def summon[T](using e: T): T =e
  ```

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

  ```
  given 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 `anyInteger`

- 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 givens in most cases, for standard types
- Note that **in scalacheck** the type is not `Gen[A]` but `Arbitrary[A]`, but the idea is the same

Using Arguments vs Default Argument Values

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

- Using arguments 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 given objects)

Type classes: History and Context

This is not only about Scala ...

- **Implicits** / **givens-using** (under this name) are a Scala-specific invention but other languages picked them up (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
- When reading blogs and Stack Overflow, it is useful to know **Scala 2** terminology
  - `given` was an `implicit` value
  - `using` was an `implicit` argument
  - `summon[A]` was `implicitly[A]`.

Type Classes and Givens: 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 the ’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/givens the extension is bound when **the caller (client) of our code is compiled** (the latest binding of all the discussed mechanism)

The Prop Type, forAll revisited [Menti]

This time with givens (6644 6761)

```
def randomLazyList[A](rng: RNG)(using g: Gen[A]): LazyList[A] =e
```

Adapted to better resemble scalacheck

The Prop Type, forAll revisited

with more concise syntax (6644 6761)

```scala
def randomLazyList[A: Gen](rng: RNG): LazyList[A] =
  LazyList.unfold(rng)(rng => Some(summon[Gen[A]].run(rng)))

def buildMsg[A](s: A, e: Exception): String =
  s"test case: $s\n" +
  s"generated an exception: ${e.getMessage}\n" +
  s"stack trace:\n ${e.getStackTrace.mkString("\n")}"

def forAll[A: Gen](f: A => Boolean): Prop = (max, n, rng) =>
  randomLazyList(as)(rng)
    .zip(LazyList.from(0))
    .take(n)
    .map { (a, i) =>
           try if f(a) then Passed else Falsified(a.toString, i)
           catch case e: Exception => Falsified(buildMsg(a, e), i) }
    .find { _.isFalsified }
    .getOrElse(Passed)
```

where is Waldo?