

# Tests de propriétés

---

AVEC SCALACHECK

# Hello, my name is...

---

Thomas Dufour

Programmeur à EPITECH : outils, automatisation

On recrute 😊 -> [thomas.dufour@epitech.eu](mailto:thomas.dufour@epitech.eu)

<3 Scala, Haskell, Elm

Slides&code : [github.com/chwthewke/lillefp8](https://github.com/chwthewke/lillefp8) (soon)

# Au programme

---

Introduction à ScalaCheck

Principes d'implémentation

Utilisation en pratique

# Historique

---

Origine : Haskell

“QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs” (ICFP 2000)

Introduit toutes les notions essentielles

- Propriétés
- Générateurs
- Réduction

Réimplémenté dans environ tous les langages du monde

# Introduction

---

# Introduction

---



# Une première propriété

---

```
val propLengthPositive = forall { s: String =>  
  s.length >= 0  
}
```

```
scala> propLengthPositive.check  
+ OK, passed 100 tests.
```

# Une deuxième propriété

---

```
val propAssoc = forall { (s: Int, t: Int, r: Int) =>
  (s + t) + r == s + (t + r)
}
```

```
scala> propAssoc.check
+ OK, passed 100 tests.
```



# Un autre exemple

---

```
val propAbs = forall { x: Int =>  
  math.abs(x) >= 0  
}
```

```
scala> propAbs.check  
! Falsified after 7 passed tests.  
> ARG_0: -2147483648
```

# Implication

---

```
val propAbs = forAll { x: Int =>  
  x > Int.MinValue ==>  
    math.abs(x) >= 0  
}
```

```
scala> propAbs.check  
+ OK, passed 100 tests.
```

# Implication

---

```
val propSqrtMono = forAll { (x: Double, y: Double) =>
  x < y ==>
    math.sqrt(x) < math.sqrt(y)
}
```

```
scala> propSqrtMono.check
! Falsified after 0 passed tests.
> ARG_0: -2.539250982004331E-5
> ARG_0_ORIGINAL: -3.651835246433264E304
> ARG_1: -9.578799231976415E-198
```

# Implication

---

```
val propSqrtMono = forall { (x: Double, y: Double) =>
  (x < y && x >= 0 && y >= 0) ==>
    math.sqrt(x) < math.sqrt(y)
}
```

```
scala> propSqrtMono.check
! Gave up after only 70 passed tests. 501 tests were
discarded.
```

# Générateurs explicites

---

Pour l'instant on a laissé ScalaCheck générer des valeurs

Mais on peut choisir son comportement

À l'aide de **générateurs**

**Gen[A]** fournit des valeurs de type A, pseudo-aléatoires

Scalacheck propose de nombreux générateurs pré-définis

Nous verrons plus tard comment écrire les notres

# Générateurs explicites

---

```
import org.scalacheck.Gen

val posDouble = Gen.choose(0d, Double.MaxValue)

val propSqrtMono = forall(posDouble, posDouble) {
  (x: Double, y: Double) =>
    x < y ==>
      math.sqrt(x) < math.sqrt(y)
}
```

```
scala> propSqrtMono.check
+ OK, passed 100 tests.
```

# Réduction

---

```
def sort(xs: List[Int]): List[Int] = xs match {  
  case Nil          => Nil  
  case x :: Nil      => x :: Nil  
  case x :: y :: Nil => (x min y) :: (x max y) :: Nil  
  case _            => xs // Trop compliqué  
}
```

```
val propSort = forAll { xs: List[Int] =>  
  sort(xs) == xs.sorted  
}
```

```
scala> propSort.check  
! Falsified after 6 passed tests.  
> Labels of failing property:  
Expected List("-1", "0", "0") but got List("0", "-1", "0")  
> ARG_0: List("0", "-1", "0")  
> ARG_0_ORIGINAL: List("-1", "2147483647", "-1965185375", "1",  
"1438799817")
```

# Récap

---

Scalacheck nous facilite la vie grâce à un processus “type => valeur” (un peu) magique

Ça peut déjà être un gain de temps

Un test de propriété doit nous donner plus confiance en notre code qu’un test d’exemple

Car il y a plus de cas testés

Car ScalaCheck est “malin” dans ses choix



# Implémentation d'un générateur

---

# Génération “aléatoire”

---

Pour un type de PRNG acceptable Rng

$$\text{Gen}[A] \approx \text{Rng} \Rightarrow A$$

Mais

- Valeur aléatoire = effet de bord
- Comment rejouer un test ?

Y a t’il une approche “fonctionnelle” du RNG ?

# Génération “aléatoire” fonctionnelle

---

Pour un type de PRNG **immutable** Rng

$$\text{Gen}[A] \approx \text{Rng} \Rightarrow (A, \text{Rng})$$

Ex : Générateur congruentiel linéaire

```
// MMIX by Donald Knuth
case class Rng(seed: Long) {
  def next: Rng =
    Rng(6364136223846793005L * seed + 1442695040888963407L)
}
```

# Générateurs simples

---

```
case class Gen[A](run: Rng => (A, Rng))

val long: Gen[Long] =
  Gen(r => (r.seed, r.next))

val int: Gen[Int] =
  Gen(r => (r.seed.toInt, r.next))

val boolean: Gen[Boolean] =
  Gen(r => (r.seed >= 0, r.next))

val char: Gen[Char] =
  Gen(r => (r.seed.toChar, r.next))

val double: Gen[Double] =
  Gen(r => ((r.seed >>> 11) * 1.1102230246251565e-16, r.next))
```

# Gen est un foncteur

---

```
case class Gen[A](run: Rng => (A, Rng)) {  
  def map[B](f: A => B): Gen[B] = Gen { r =>  
    val (a, next) = this.run(r)  
    (f(a), next)  
  }  
}  
  
val int: Gen[Int] = long.map(_.toInt)  
val boolean: Gen[Boolean] = long.map(_ >= 0)  
val char: Gen[Char] = long.map(_.toChar)  
val double: Gen[Double] = long.map(l => (l >>> 11) * 1.1102230246251565e-16)  
  
def const[A](a: A): Gen[A] = long.map(_ => a)  
def proba(p: Double): Gen[Boolean] = double.map(_ < p)  
def upTo(n: Int): Gen[Int] = double.map(x => (x * n).toInt)
```

# Générateurs avancés

---

Générer une liste ?

- Récursion
- Passage méticuleux de l'état du RNG à chaque étape

```
def fixedList[A](gen: Gen[A], n: Int): Gen[List[A]] =  
  if (n == 0) const(Nil)  
  else  
    Gen { r =>  
      val (head, r1) = gen.run(r)  
      val (tail, r2) = fixedList(gen, n - 1).run(r1)  
      (head :: tail, r2)  
    }
```

# Gen est une monade

---

```
case class Gen[A](run: Rng => (A, Rng)) {  
  ...  
  def flatMap[B](f: A => Gen[B]): Gen[B] = Gen { r =>  
    val (a, next) = this.run(r)  
    f(a).run(next)  
  }  
}  
  
def fixedList[A](gen: Gen[A], n: Int): Gen[List[A]] =  
  if (n == 0) const(Nil)  
  else gen.flatMap(a => fixedList(gen, n - 1).map(as => a :: as))  
  
def list[A](gen: Gen[A]): Gen[List[A]] =  
  for {  
    n <- upTo(32768)  
    l <- fixedList(gen, n)  
  } yield l
```

# Générateurs avancés

---

```
def tuple[A, B](ga: Gen[A], gb: Gen[B]): Gen[(A, B)] =  
  ga.flatMap(a => gb.map(b => (a, b)))  
  
def option[A](ga: Gen[A]): Gen[Option[A]] =  
  proba(0.1).flatMap(  
    isNone =>  
      if (isNone) const(None)  
      else ga.map(Some(_))  
  )  
  
def either[A, B](ga: Gen[A], gb: Gen[B]): Gen[Either[A, B]] =  
  boolean.flatMap(  
    isLeft =>  
      if (isLeft) ga.map(Left(_))  
      else gb.map(Right(_))  
  )
```



# Vers des générateurs pour mes types

---

```
trait Wheel
trait Engine

case class Car(engine: Engine, wheels: List[Wheel])

def car(genWheel: Gen[Wheel], genEngine: Gen[Engine]): Gen[Car] =
  for {
    engine <- genEngine
    wheels <- fixedList(genWheel, 4)
  } yield Car(engine, wheels)
```

Essentiellement, on a tout le nécessaire pour générer des valeurs pour nos types métier

Ou presque...

# Générer... des fonctions ?

---

```
def genFunction[A, B]: Gen[A => B] = ???

def constFunction[A, B](gb: Gen[B]): Gen[A => B] =
  gb.map(b => (_ => b))

def wildFunction[A, B](gb: Gen[B]): Gen[A => B] =
  const(_ => gb.run(Rng(Random.nextLong))._1)
```

# Un tour de magie

---

```
// Gen[A]    ~ Rng => (A, Rng)
// CoGen[A] ~ (A, Rng) => Rng
```

```
case class CoGen[A](consume: (A, Rng) => Rng)

def function[A, B](cogen: CoGen[A], gen: Gen[B]): Gen[A => B] = Gen { r =>
  def f(a: A): B = gen.run(cogen.consume(a, r))._1
  (f, r.next)
}
```

# Récap

---

On a vu la machinerie utilisée pour faciliter la génération de valeurs

Certains auront reconnu la monade “State”

On a toutefois omis quelques détails

ScalaCheck est plus “futé” sur la distribution des générateurs numériques

Le fait que les générateurs de ScalaCheck puissent être partiels est une complication

- En particulier pour générer des fonctions

# Étude de cas

---




AVEC DÉDÉ


# Un tour sur Wikipedia

**421**  
jeu de société

Ce jeu appartient au **domaine public**

<b>Format</b>	jeu de dés
<b>Mécanismes</b>	combinaisons chance
<b>Joueur(s)</b>	2 ou plus
<b>Âge</b>	à partir de 7 ans
<b>Durée annoncée</b>	à volonté

habileté physique	réflexion décision	générateur de hasard	info. compl. et parfaite
 Non	 Oui	 Oui	 Oui

[modifier](#)

# Notre modèle

---

```
case class Dice(h: Int, m: Int, l: Int) {  
  require(1 <= l && l <= m && m <= h && h <= 6)  
}  
  
object Dice {  
  val D421 = Dice(4, 2, 1)  
  val D111 = Dice(1, 1, 1)  
  def aces(n: Int) = Dice(n, 1, 1)  
  def same(n: Int) = Dice(n, n, n)  
  def sequence(n: Int) = Dice(n, n - 1, n - 2)  
  // ...  
}  
  
def compareDice(left: Dice, right: Dice): Int = ???  
// like Comparable (-1, 0 ou 1)
```

# Propriété naïve du 421

```
val genDice: Gen[Dice] = for {  
  a <- Gen.choose(1, 6)  
  b <- Gen.choose(1, 6)  
  c <- Gen.choose(1, 6)  
  (h, l) = (a max b max c, a min b min c)  
  m = a + b + c - h - l  
} yield Dice(h, m, l)  
  
implicit val arbDice: Arbitrary[Dice] = Arbitrary(genDice)  
  
val prop1 = forAll { (dice1: Dice, dice2: Dice) =>  
  val (best, worst) = (??? : (Dice, Dice))  
  // TODO: déterminer la meilleure combinaison parmi dice1, dice2  
  
  compareDice(best, worst) >= 0  
}
```



# Deuxième tentative

---

```
val genAces: Gen[Dice] =  
  Gen.choose(2, 6).map(Dice.aces)  
  
val genSame: Gen[Dice] =  
  Gen.choose(2, 6).map(Dice.same)  
  
val prop2 = forAll(genAces, genDice) { (aces: Dice, same: Dice) =>  
  compareDice(aces, same) == 1  
}
```

# Relation d'ordre

---

```
val propRefl = forAll { (d: Dice) =>
  compareDice(d, d) ?= 0
}

val propSym = forAll { (d1: Dice, d2: Dice) =>
  compareDice(d1, d2) == -compareDice(d2, d1)
}

val propTrans = forAll { (d1: Dice, d2: Dice, d3: Dice) =>
  (compareDice(d1, d2) >= 0 && compareDice(d2, d3) >= 0) ==>
    compareDice(d1, d3) >= 0
}

// Alternative avec cats
OrderTests(new Order[Dice] {
  override def compare(x: Dice, y: Dice): Int = compareDice(x, y)
})
```

# Pour aller plus loin

---

## User Guide

<https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md>

## Intégrations :

- alexarchambault/scalacheck-shapeless : derivation générique de générateurs
- 47deg/scalacheck-toolbox (inclus : générateurs de java.time.\*)
- amrhassan/scalacheck-cats : instances de cats pour ScalaCheck
- typelevel/discipline : verification de lois avec ScalaCheck

## Inspirations :

- [Functions and Determinism in Property-based Testing](#) – Erik Osheim
- [Practical ScalaCheck](#) – Noel Markham