Reaktive Programmierung
Vorlesung 5 vom 20.05.14: ScalaCheck (and ScalaTest)

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Einführung in Scala
  - Die Scala Collections
  - Monaden
  - ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Was ist eigentlich Testen?

Myers, 1979

Testing is the process of executing a program or system with the intent of finding errors.

- Hier: testen is selektive, kontrollierte Programmausführung.
- Ziel des Testens ist es immer, Fehler zu finden wie:
  - Diskrepanz zwischen Spezifikation und Implementation
  - strukturelle Fehler, die zu einem fehlerhaften Verhalten führen (Programmabbruch, Ausnahmen, etc)

E. W. Dijkstra, 1972

Program testing can be used to show the presence of bugs, but never to show their absence.

Testmethoden

- Statisch vs. dynamisch:
  - Statische Tests analysieren den Quellcode ohne ihn auszuführen (statische Programmanalyse)
  - Dynamische Tests führen das Programm unter kontrollierten Bedingungen aus, und prüfen das Ergebnis gegen eine gegebene Spezifikation.
- Zentrale Frage: wo kommen die Testfälle her?
  - Black-box: Struktur des s.u.t. (hier: Quellcode) unbekannt, Testfälle werden aus der Spezifikation generiert;
  - Grey-box: Teile der Struktur des s.u.t. ist bekannt (z.B. Modulstruktur)
  - White-box: Struktur des s.u.t. ist offen, Testfälle werden aus dem Quellcode abgeleitet

Spezialfall des Black-Box-Tests: Monte-Carlo Tests

- Bei Monte-Carlo oder Zufallstests werden zufällige Eingabewerte generiert, und das Ergebnis gegen eine Spezifikation geprüft.
- Dies erfordert ausführbare Spezifikationen.
- Wichtig ist die Verteilung der Eingabewerte.
  - Gleichverteilt über erwartete Eingaben, Grenzfälle beachten.
- Funktioniert gut mit high-level-Spachen (Java, Scala, Haskell)
  - Datentypen repräsentieren Informationen auf abstrakter Ebene
  - Eigenschaft gut spezifizierbar
  - Beispiel: Listen, Listenumkehr in C, Java, Scala
- Zentrale Fragen:
  - Wie können wir ausführbare Eigenschaften formulieren?
  - Wie Verteilung der Zufallswerte steuern?

ScalaTest

- Test Framework für Scala

```
import org.scalatest.FlatSpec

class StringSpec extends FlatSpec {
  "A String" should "reverse" in {
    "Hello".reverse should be ("olleH")
  }

  it should "return the correct length" in {
    "Hello".length should be (5)
  }
}
```

ScalaTest Assertions 1

- ScalaTest Assertions sind Makros:

```
import org.scalatest.Assertions._
val left = 2
val right = 1
assert(left == right)
```

- Schlägt fehl mit `"2 did not equal 1"`
- Alternativ:

```
val a = 5
val b = 2
assertResult(2) {
  a - b
}
```

- Schlägt fehl mit `"Expected 2, but got 3"`

ScalaTest Assertions 2

- Fehler manuell werfen:

```
fail("I've got a bad feeling about this")
```

- Erwartete Exeptions:

```
val s = "hi"
val e = intercept[IndexOutOfBoundsException] {
  s.charAt(-1)
}
```

- Assumptions

```
assume(database.isAvailable)
```

ScalaTest Matchers

- Gleichheit überprüfen:

```
result should equal (3)
result should be (3)
result shouldBe 3
result shouldEqual 3
```

- Länge prüfen:

```
result should have length 3
result should have size 3
```

- Und so weiter...

```
text should startWith ("Hello")
result should be a [List[Int]]
list should contain noneOf (3,4,5)
```

- Siehe `http://www.scalatest.org/user_guide/using_matchers`

ScalaTest Styles

- ScalaTest hat viele verschiedene Styles, die über Traits eingemischt werden können
- Beispiel: `FunSpec` (Ähnlich wie RSpec)

```
class SetSpec extends FunSpec {
  describe("A Set") {
    describe("when empty") {
      it("should have size 0") {
        assert(Set.empty.size == 0)
      }

      it("should produce NoSuchElementException when head is
          invoked") {
        intercept[NoSuchElementException] {
          Set.empty.head
} } } } }
```

- Übersicht unter
  `http://www.scalatest.org/user_guide/selecting_a_style`

Blackbox Test

- Überprüfen eines Programms oder einer Funktion ohne deren Implementierung zu nutzen:

```
def primeFactors(n: Int): List[Int] = ???
```

- z.B.

```
"primeFactors" should "work for 360" in {
  primeFactors(360) should contain theSameElementsAs
      List(2,2,2,3,3,5)
}
```

- Was ist mit allen anderen Eingaben?

Property based Testing

- Überprüfen von Eigenschaften (Properties) eines Programms / einer Funktion:

```
def primeFactors(n: Int): List[Int] = ???
```

- Wir würden gerne so was schreiben:

```
forall x >= 1 -> primeFactors(x).product = x
                 && primeFactors(x).forall(isPrime)
```

Property based Testing

- Überprüfen von Eigenschaften (Properties) eines Programms / einer Funktion:

```
def primeFactors(n: Int): List[Int] = ???
```

- Wir würden gerne so was schreiben:

```
forall x >= 1 -> primeFactors(x).product = x
                 && primeFactors(x).forall(isPrime)
```

- Aber wo kommen die Eingaben her?

Testen mit Zufallswerten

- `def primeFactors(n: Int): List[Int] = ???`

- Zufallszahlen sind doch einfach!

```
"primeFactors" should "work for 360" in {
  (1 to 1000) foreach { _ ⇒
    val x = Math.max(1, Random.nextInt.abs)
    assert(primeFactors(x).product === (x))
    assert(primeFactors(x).forall(isPrime))
  }
}
```

Testen mit Zufallswerten

- `def primeFactors(n: Int): List[Int] = ???`

- Zufallszahlen sind doch einfach!

```
"primeFactors" should "work for 360" in {
  (1 to 1000) foreach { _ ⇒
    val x = Math.max(1, Random.nextInt.abs)
    assert(primeFactors(x).product === (x))
    assert(primeFactors(x).forall(isPrime))
  }
}
```

- Was ist mit dieser Funktion?

```
def sum(list: List[Int]): Int = ???
```

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

object Generator {
  def apply[T](f: ⇒ T) = new Generator[T] {
    def generate = f }
}
```

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

object Generator {
  def apply[T](f: ⇒ T) = new Generator[T] {
    def generate = f }
}
```

- `val integers = Generator(Random.nextInt)`

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

object Generator {
  def apply[T](f: ⇒ T) = new Generator[T] {
    def generate = f }
}
```

- `val integers = Generator(Random.nextInt)`

- `val booleans = Generator(integers.generate > 0)`

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

object Generator {
  def apply[T](f: ⇒ T) = new Generator[T] {
    def generate = f }
}
```

- `val integers = Generator(Random.nextInt)`

- `val booleans = Generator(integers.generate > 0)`

- `val pairs = Generator((integers.generate,integers.generate))`

Zufallsgeneratoren Kombinieren

- Ein generischer, kombinierbarer Zufallsgenerator:

```
trait Generator[+T] { self ⇒
  def generate: T
  def map[U](f: T ⇒ U) = new Generator[U] {
    def generate = f(self.generate)
  }
  def flatMap[U](f: T ⇒ Generator[U]) = new Generator[U] {
    def generate = f(self.generate).generate
  }
}
```

Einfache Zufallsgeneratoren

- Einelementige Wertemenge:

```
def single[T](value: T) = Generator(value)
```

- Eingeschränkter Wertebereich:

```
def choose(lo: Int, hi: Int) =
  integers.map(x ⇒ lo + x % (hi - lo))
```

- Aufzählbare Wertemenge:

```
def oneOf[T](xs: T*): Generator[T] =
  choose(0,xs.length).map(xs)
```

Beispiel: Listen Generieren

- Listen haben zwei Konstruktoren: `Nil` und `::`:

```
def lists: Generator[List[Int]] = for {
  isEmpty ← booleans
  list ← if (isEmpty) emptyLists else nonEmptyLists
}
```

Beispiel: Listen Generieren

- Listen haben zwei Konstruktoren: `Nil` und `::`:

```
def lists: Generator[List[Int]] = for {
  isEmpty ← booleans
  list ← if (isEmpty) emptyLists else nonEmptyLists
}
```

- Die Menge der leeren Listen enthält genau ein Element:

```
def emptyLists = single(Nil)
```

Beispiel: Listen Generieren

- Listen haben zwei Konstruktoren: `Nil` und `::`:

```
def lists: Generator[List[Int]] = for {
  isEmpty ← booleans
  list ← if (isEmpty) emptyLists else nonEmptyLists
}
```

- Die Menge der leeren Listen enthält genau ein Element:

```
def emptyLists = single(Nil)
```

- Nicht-leere Listen bestehen aus einem Element und einer Liste:

```
def nonEmptyLists = for {
  head ← integers
  tail ← lists
} yield head :: tail
```

ScalaCheck

- ScalaCheck nutzt Generatoren um Testwerte für Properties zu generieren

```
forAll { (list: List[Int]) ⇒
  sum(list) == list.foldLeft(0)(_ + _)
}
```

- Generatoren werden über implicits aufgelöst
- Typklasse `Arbitrary` für viele Typen vordefiniert:

```
abstract class Arbitrary[T] {
  val arbitrary: Gen[T]
}
```

Kombinatoren in ScalaCheck

```
object Gen {
  def choose[T](min: T, max: T)(implicit c: Choose[T]): Gen[T]
  def oneOf[T](xs: Seq[T]): Gen[T]
  def sized[T](f: Int ⇒ Gen[T]): Gen[T]
  def someOf[T](gs: Gen[T]*); Gen[Seq[T]]
  def option[T](g: Gen[T]): Gen[Option[T]]
  ...
}

trait Gen[+T] {
  def map[U](f: T ⇒ U): Gen[U]
  def flatMap[U](f: T ⇒ Gen[U]): Gen[U]
  def filter(f: T ⇒ Boolean): Gen[T]
  def suchThat(f: T ⇒ Boolean): Gen[T]
  def label(l: String): Gen[T]
  def |(that: Gen[T]): Gen[T]
  ...
}
```

Wertemenge einschränken

- Problem: Vorbedingungen können dazu führen, dass nur wenige Werte verwendet werden können:

```
val prop = forAll { (l1: List[Int], l2: List[Int]) ⇒
  l1.length == l2.length =⇒  l1.zip(l2).unzip() == (l1,l2)
}

scala> prop.check
Gave up after only 4 passed tests. 500 tests were discarded.
```

- Besser:

```
forAll(myListPairGenerator) { (l1, l2) ⇒
  l1.zip(l2).unzip() == (l1,l2)
}
```

Kombinatoren für Properties

- Properties können miteinander kombiniert werden:

```
val p1 = forAll(...)
val p2 = forAll(...)
val p3 = p1 && p2
val p4 = p1 || p2
val p5 = p1 == p2
val p6 = all(p1, p2)
val p7 = atLeastOne(p1, p2)
```

ScalaCheck in ScalaTest

- Der Trait `Checkers` erlaubt es, ScalaCheck in beliebigen ScalaTest Suiten zu verwenden:

```
class IntListSpec extends FlatSpec with Checkers {
  "Any list of integers" should "return its correct sum" in {
    check ( forall { (x: List[Int]) ⇒ x.sum ==
        x.foldLeft(0)(_ + _) })
  }
}
```