Some Words on Scala

- Scala is object-oriented.
  - every value is an object
  - classes and traits: types and behavior of objects
  - inheritance

- Scala is functional.
  - every function is a value
  - anonymous functions
  - higher-order functions
  - support of currying
  - pattern matching
Scala By Example I (from [4])

- class in Java:

```java
public class Person {
    public final String name;
    public final int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

- class in Scala:

```scala
class Person(val name: String, val age: Int) {}
```
filtering in Java:

```java
Person[] people; Person[] minors; Person[] adults;

ArrayList<Person> minorsList = new ArrayList<Person>();
ArrayList<Person> adultsList = new ArrayList<Person>();

for (int i = 0; i < people.length; i++)
    (people[i].age < 18 ? minorsList : adultsList).add(people[i]);

minors = minorsList.toArray();
adults = adultsList.toArray();
```

filtering in Scala:

```scala
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```
```scala
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)

    Array.concat(
      sort(xs filter (pivot >)),
      xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```
Testing (Scala) Programs

Question: Does a program obey its specification?

• Obtaining a definitive answer is often not feasible
  • techniques of formal verification
  • generate all test cases

• Pragmatic approach: Generate many test cases to gain confidence in the program for covering
  • standard cases
  • corner cases

⇒ ScalaCheck
The specification of properties ...

- ... helps to understand what the program shall do
- ... helps to understand what the program actually does
- ... helps to talk about the program
- ... can help to find an algorithm
- ... can be valuable for debugging
What does ScalaCheck do?

- **User:**
  - specification of properties which should always hold
  - definition of random data for testing properties
  - no worries about missed test cases

- **ScalaCheck:**
  - automatic generation of test cases
  - checking if properties hold
  - shrinking (minimization of failing test cases)

- **ScalaCheck is ...**
  - ... an automated, property based testing tool for Scala/Java
  - ... an extended port of Haskell QuickCheck
  - ... available at www.scalacheck.org
A First Example

An unsorted list $L$ has the same length as the list $L'$ obtained by sorting the elements of $L$.

Example

```java
object MyProperties extends Properties("MyProperties") {

  property("same length") =
  forAll { (a: [Int]) =>
    a.length == sort(a).length
  }
}
```
ScalaChecks Highlights

- automatic testing of properties
- automatic generation of test data (also for custom data types)
- precise control of test data generation
- automatic simplification of failing test cases
- support for stateful testing of command sequences
- simplification of failing command sequences
- direct testing of property object from the command

Example

```scala
scala> import org.scalacheck.Prop.forAll
import org.scalacheck.Prop.forAll

scala> val overflow = forAll { (n: Int) => n > n-1 }
overflow: org.scalacheck.Prop = Prop

scala> overflow.check
! Falsified after 6 passed tests.
> ARG_0: -2147483648
```
Basic Concepts

• properties
  org.scalacheck.Prop

• generators
  org.scalacheck.Gen

• test runner
  org.scalacheck.Test
• testable unit in ScalaCheck
• class: org.scalacheck.Prop
• generation:
  • specification of new property
  • combination of other properties
  • use specialized methods

Example

```scala
scala> object StringProps extends Properties("String") {
  |
  | property("startsWith") = forAll ( (a:String, b:String) => (a+b).startsWith(a))
  |
  | property("substring") = forAll ((a:String, b:String) => (a+b).substring(a.length) == b)
  |
} defined module StringProps

scala> StringProps.check
+ String.startsWith: OK, passed 100 tests.
+ String.substring: OK, passed 100 tests.
```
Universally Quantified Property (Forall Property)

• create property: `org.scalatest.Prop.forAll`
  • in: function which returns Boolean or a property
  • out: property

• check property: call of `check` method

Example

```scala
import org.scalatest.Prop.forAll

val propReverseList = forAll {
  l: List[String] =>
  l.reverse.reverse == l
}

val propConcatString = forAll {
  (s1: String, s2: String) =>
  (s1 + s2).endsWith(s2)
}
```
Data Generator

- generation of test data for
  - custom data types
  - subsets of standard data types
- representation: org.scalacheck.Gen

Example

```scala
val myGen = for {
  n ← Gen.choose(10, 20)
  m ← Gen.choose(2*n, 500)
} yield (n, m)

val vowel = Gen.oneOf('A', 'E', 'I', 'O', 'U')

val vowel1 = Gen.frequency((3, 'A'), (4, 'E'),
                           (2, 'I'), (3, 'O'), (1, 'U'))
```
sealed abstract class Tree

case class Node(left: Tree, right: Tree, v: Int) extends Tree

case object Leaf extends Tree

import org.scalacheck._
import Gen._
import Arbitrary.arbitrary

val genLeaf = value(Leaf)

val genNode = for {
  v <- arbitrary[Int]
  left <- genTree
  right <- genTree
} yield Node(left, right, v)

def genTree: Gen[Tree] = oneOf(genLeaf, genNode)
Statistics on Test Data

- collect info on created test data
- inspection of distribution
- only trivial test cases?

Example

```scala
def ordered(l: List[Int]) = l == l.sort(_ > _)

val myProp = forAll { l: List[Int] =>
  classify(ordered(l), "ordered") {
    classify(l.length > 5, "large", "small") {
      l.reverse.reverse == l
    }
  }
}
```

scala> myProp.check
+ OK, passed 100 tests.
> Collected test data:
78% large
16% small, ordered
6% small
Conditional Properties

- sometimes specifications are implications
- implication operator
- restricts number of test cases
- problem: condition is hard or impossible to fulfill
- property does not only pass or fail, but could be undecided if implication condition does not get fulfilled.

Example

```java
property("firstElement") = Prop forall { 
    (xs: List[Int]) => (xs.size > 0) =>
    (xs.head == xs(0))
}
```
Combining Properties

combine existing properties to new ones

val p1 = forAll(...)

val p2 = forAll(...)

val p3 = p1 && p2

val p4 = p1 || p2

val p5 = p1 == p2

val p6 = all(p1, p2) // same as p1 && p2

val p7 = atLeastOne(p1, p2) // same as p1 || p2
Test Case Execution

- module Test
  - execution of the tests
  - generation of the arguments
  - evaluation of the properties
  - increase of size of test parameters
  - reports success (passed) after certain number of tries

- testing parameters in Test.Params
  - number of times a property should be tested
  - size bounds of test data
  - number of tries in case of failure
  - callback

- statistics in Test.Result
- test properties with Test.check
Test Case Minimisation

- ScalaCheck tries to shrink failing test cases before they are reported
- default by Prop.forAll
- no shrinking: Prop.forAllNoShrink

Example

```scala
val p1 = forAllNoShrink(arbitrary[List[Int]])(
  l => l == l.removeDuplicates)

counter example:
List(8, 0, -1, -3, -8, 8, 2, -10, 9, 1, -8)

val p3 = forAll((l: List[Int]) =>
  l == l.removeDuplicates)

counter example: List(-5, -5)
```
Customized Shrinking (from [5])

- definition of custom shrinking methods is possible
- implicit method which returns \texttt{Shrink}[^T] instance
- important: instances get smaller (otherwise loops possible)

**Example**

```scala
def shrinkTuple2[T1, T2] (s1: Shrink[T1], s2: Shrink[T2]):
  Shrink[(T1, T2)] = Shrink { case (t1, t2) =>
    for (x1 <- shrink(t1)) yield (x1, t2)
    append
    for (x2 <- shrink(t2)) yield (t1, x2)
  }
```
• what about testing combinations of functions?
• solution: org.scalatest.Commands
• example: Test the behavior of a counter

Example

```scala
class Counter {
    private var n = 0
    def inc = n += 1
    def dec = n -= 1
    def get = n
    def reset = n = 0
}
```
object CounterSpecification extends Commands {

val counter = new Counter

case class State(n: Int)

def initialState() = { ... }

case object Dec extends Command { ... }
case object Inc extends Command { ... }
case object Get extends Command { ... }

def genCommand(s: State): Gen[Command] =
Gen.oneOf(Inc, Dec, Get)
}
References


[2] ScalaCheck Project Site: www.scalacheck.org

