@AndrzejWasowski
Andrzej Wąsowski

Advanced Programming

Program Correctness & Property-based Testing

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

- Assertions
- Pre-conditions
- Post-conditions
- Contracts
- Invariants
- Property-Based Testing
- Homework & ScalaTest

AGENDA

Assertions

```
1 // assertion: an empty list has size zero
2 assert (List().size == 0)
3
4 def f (l: List[Int]) :Unit = {
5   // A contract for Cons: If I have a list, and add an element to it
6   // it will certainly have at least one element in it
7   assert (Cons(42,l).size > 0)
8
9   // A contract for Cons: If length of a list is n, and you add (Cons)
10  // an element to the list , it will have size n+1
11  assert (l.size + 1 == Cons(42,l).size)
12
13  // A contract for Cons: If you add (Cons) an element to the list,
14  // then this element will be the head of the list.
15  assert (Cons(42,l).headOption == Some (42))
16 }
17
18 def head (l: List[Int]) :Int = {
19  // Precondition for head: l is not empty
20  assert ( ... ) // Exercise
21  ...
```

- Analyze **examples** + exercise
- **English** turned into **predicates** over program state (variables, values)
- Invariants, pre-conditions, post-conditions, contracts
- In Scala: `assert` is a **method** on the `Predef` object, automatically imported
- 2-argument version: produce an error msg

Assertions

Assertion: A Boolean predicate over the program state (variables and values), usually expressed as a logical proposition that evaluates to true in a correct execution.

- Help the programmer read the code
- Facilitate fail-fast programming
- Help compilers compile efficiently
- Support **testing** and **verification**
- Flavours of predicates (Scala terminology), differences mostly visible in verification
  - **Assert**: Must hold, the checker should prove it, a goal, bug if does not hold; A test
  - **Assume**: Consider satisfying executions; Verifier assumes this; An axiom
  - **Require**: Enforce before an action, bug if violated, **blame the caller**, a pre-condition
  - **Ensuring**: enforce the property after some action, bug if violated, **blame the callee**, a post-condition; Interestingly an expression: `(x) ensuring { x =>P(x) }`, returns `x`
  - During execution **all fail** with different exceptions.
- Assertions **not** specific to imperative programming, used in FP too
- Write **small assertions** if possible, test a single violation at a time. Gives better feedback.

Pre-conditions for Option

```
1 def isEmpty :Boolean = this match { case None => true; case _ => false }
2 // Get the value held by an option
3 // Precondition: !this.isEmpty
4 def get :A = ...
5
6 // Apply a binary operator to all elements of this iterable collection
7 // Like foldRight, but without a zero element. Examples:
8 // List(1,2,3) reduce[Int] (_+_) == 6
9 // List(1) reduce[Int] (_+_) == 1
10 // Some (1) reduce[Int] (_+_) == 1
11 // Nil reduce[Int] (_+_) == ???
12 // precondition: !this.isEmpty
13 def reduceRight[B >: A](op: (A, B) => B): B
14
15 // Count the number of elements in a collection that satisfy a predicate
16 // Task: what is the precondition?
17 def count(p: (A) => Boolean): Int
18
19 // Always fail with an exception
20 // Task: what is the precondition for f to avoid failure?
21 def f = throw new Exception
```

- **Pre-condition true**
  means
  always succeeds
  (no pre-condition)
- **Pre-condition false**
  means
  never succeeds
  (always fail)

Pre-conditions

Pre-condition: A Boolean **predicate** such that **if it holds before execution of some code** then the code behaves correctly.

- Typically considered at the **entry point to a function** or procedure
- Can constrain **arguments** + other **variables** and **values** in scope
- **Enforced using assertions** in most programming languages
- A function can have **several pre-conditions** (a single example)
  - Consider: `def factorial(n: Int): Int`
  - $n >= 0$ is a pre-condition
  - $n >= 42$ is a pre-condition, too. **Why?**
  - $n >= 0$ is weaker, $n >= 42$ is stronger. **Why**?
- The **weakest pre-condition**: the minimal assumption for a block of code (typically a function) to behave correctly, so to satisfy the expected post-condition
- We specify the weakest pre-conditions for functions to achieve **complete specifications**
- **Stronger pre-conditions** are used in testing (when testing only an aspect of logics)

Post-conditions

```
1  def size[A](t: Tree[A]): Int
2  // A post-condition: forall t. size(t) >= 1
3
4  def depth[A](t: Tree[A]): Int
5  // A post-condition: forall t. depth(t) >= 0
6
7  // A helper function
8  def get (t: Tree[Int]): Int = t match {
9    case Leaf (n) => n
10   case Branch (l,_) => get (l)
11 }
12 // Now we use it
13 def maximum(t: Tree[Int]): Int
14 // A post-condition: forall t. maximum (t) >= get (t)
15
16 // We previously defined
17 def nextInt: (Int, RNG) = ...
18 // The postcondition: true
19 // Not the strongest post-condition for random,
20 // but it is hard to give a stronger one.
21 // This would require formally defining the pseudo-
22 // random stream (likely not needed)
```

- Are these the **strongest post-conditions** we could think of? Could we guarantee more?
- Strongest post-conditions may be **difficult** to write
- The `maximum` post-condition **relates** more than one function (we shall return to this)
- A **true** post-condition means that we cannot say anything useful about the result
- (like a false pre-condition meant that there is no useful conditions in which the function could be called)

Post-conditions

Post-condition A Boolean **predicate** describing a correct result of an execution of a fragment of code.

- Typically **applied to functions**
- For a function it **constraints the return value**
- In imperative programs also **constraints variables and values in scope**
- The **strongest post-condition**: the maximum guarentee for a code fragment given by a correct execution, so an execution that satisfies a pre-condition
- Strongest post-conditions are used to define the **complete behavior** of code
- **Testing:** check post-conditions with assertions after a function call
- Giving strongest post-conditions is often difficult without replicating the code
- **Decompose the post-condition** and test for several weaker conditions instead

Contracts

```
1 // Returns this scala.Option if it is nonempty, otherwise return
2 // the result of evaluating alternative.
3 def orElse[B >: A](alternative: => Option[B]): Option[B]
4 // pre-condition: true
5 // post-condition:
6 //   (!this.isEmpty => orElse(alternative) == this       ) &&
7 //   ( this.isEmpty => orElse(alternative) == alternative)
8
9 def maximum (t: Tree[Int]): Int
10 // pre-condition: true
11 // post-condition:
12 //    (forAll (t) { _ <= maximum (t) }) &&
13 //    (exist  (t) { _ == maximum (t) })
14 // (assume that forAll/exist implemented for trees as static methods)
15
16 // For partial functions a pre-condition is not true.
17 def get[A] (oa: Option[A]): A = ...
18 // pre-condition: !oa.isEmpty
19 // post-condition: Some(get(oa)) == oa
```

- We are using **algebraic laws** again in all three examples
- A key aspect of correct behavior is that it **interacts well** with other parts of our API

Contracts

Contract. A contract for a function is a **pair of a pre-condition and a post-condition**. The caller of the function must ensure that the pre-condition is satisfied at the call time, and the callee has to ensure that the post-condition holds after termination, **if** the pre-condition was satisfied.

- Ideally a contract **minimizes assumptions** and **maximizes guarantees** (the pre-condition is the weakest, the post-condition is the strongest)
- Writing these ideal complete contracts may be difficult
- In testing, we can use **weaker contracts**:
  - If ϕ holds before the call then ψ holds after the call
  - Not necessarily the weakest/strongest
  - We decompose the ideal contract into tests and test-cases, or, in other words pairs of stronger preconditions and weaker post-conditions
- **NB**. The weakest contract is false => true
  - Any terminating function satisfies it vacously

Invariants

- An **invariant** is a related concept: A property that should always hold (for an object/concept at runtime)
- **Loop invariant** holds at every loop iteration (not useful in FP). Why?
  - Insertion sort: Elements left of the current index form a sorted sequence
- In FP loop invariants are replaced by contracts of recursive functions
- **Data structure invariant** holds always for instances f the data structure:
  - AVL trees: The difference of height between the left and right subtree is at most 1 for any AVL tree node

Property-based testing in a nutshell /1

Actual test code from exercises in the prior weeks

Three weak/partial contracts/laws for `map` on trees (trivial preconditions)

```
"identity is a unit with map" in {
  forAll ("t") { (t: Tree[Int]) => Tree.map (t) (x => x) should be (t) }
}

"map does not change size" in {
  forAll ("t","f") { (t: Tree[Int], f: Int => Int) =>
    val t1 = Tree.map (t) (f)
    Tree.size (t1) should equal (Tree.size(t))
  }
}

"map is 'associative'" in {
  forAll ("t","f","g")  { (t: Tree[Int], f: Int=>Int, g: Int=>Int) =>
    val t1 = Tree.map (Tree.map (t) (f)) (g)
    val t2 = Tree.map (t) (g compose f)
    (t1) should equal (t2)
  }

}
```

- `forAll` generates random `Tree[Int]`
- `"t"`: an optional parameter to improve error reporting
- `forAll` checks if the predicate (lambda) holds for all random data. Fail if false
- `should be`: infix assertion, fails if no equality
- This code uses of **scalatest** library
- We implement a similar lib next week
- Mentimeter on tests 2 and 3

This code can be found in the option code, under `src/test/scala/adpro/ExercisesSpec.scala`, line ca. 110

Property-based testing in a nutshell /2

Test-data generators

```
def genTree[A] (implicit arbA: Arbitrary[A]): Gen[Tree[A]] =
  Gen.frequency (7 -> true, 1 -> false)
  .flatMap { stop =>
    if (stop) arbA.arbitrary map { Leaf(_) }
    else for {
      l <- genTree[A]
      r <- genTree[A]
    } yield (Branch (l,r))
  }

implicit def arbitraryTree[A] (implicit arbA: Arbitrary[A])
  : Arbitrary[Tree[A]] = Arbitrary[Tree[A]] (genTree (arbA))
```

- This week we learn how to **use** the API
- Next we **build** such API, incl. implicit values + `Gen` monad
- When you test a new data type: **create a generator** for it
- Normally you write much **more tests** than generators

- `genTree` generates a random tree of `A`, if it **can find** an implicit generator of arbitrary `A`
- `Gen.frequency` is a `Gen[Boolean]`, returns true/false with relative frequencies 7/1 (polymorphic)
- `arbA.arbitrary` generates an arbitrary `A`, **mapped** into a leaf containing the A (`Gen[Leaf[A]]`)
- The last two lines provide an **implicit** generator of arbitrary trees of As (wrapping `genTree`)

This code can be found in the option code, under `src/test/scala/adpro/ExercisesSpec.scala`, line ca. 15

Scenario tests vs Property tests

- Most traditional unit-level and integration-level testing uses **scenarios**
- Examples, test cases, **one path through a system** per test
- Scenarios are **obtained from requirements**
- Good for **covering requirements**.
- Good for testing **special corner cases**
- Good for recording **regression tests**
- **Automation key** for success
- Test libraries are more important than debuggers, in the sense that if you automate tests, you **reuse the effort**
- BDD, TDD, JUnit (XUnit), Cucamber/RSpec, scalatest

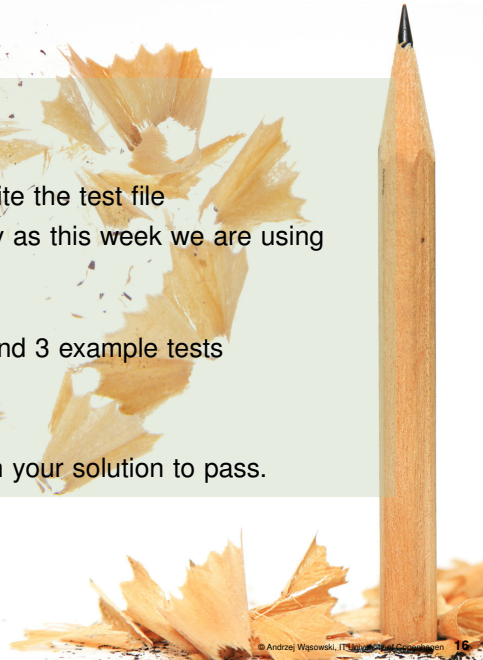
- **Property-based testing (PBT)** tests **algebraic laws** that should hold for an API
- The process is:
  1. Formulate laws (pre/post/invariant thinking helps)
  2. Generate random data
  3. Test the laws on this data
- Gives **better** (? rather alternative) **coverage** than scenario testing
- It may catch things that you **did not predict**, due to randomness
- Gets closer to **verification** but remains easy
- You use **formal specifications** to test

PBT and Contracts

```
1 // Recall the specification     // pre-condition: !oa.isEmpty
2 def get[A] (oa: Option[A]): A   // post-condition: Some(get(oa)) == oa
3
4 // When testing in the classical way, create a value 'oa', run get, inspect the result
5 "test get contract" in { // Example with scalatest ('JUnit' for Scala):
6     val oa = Some(1)
7     (get (oa)) should be (1)
8 }
9
10 // Two weaknesses:
11 // 1. For another integer, write another test; create repetive code (or iterate a collection)
12 // 2. The contract specification is not explicit in the test.
13 // A property-based test does the iteration implicitly and solves these problems:
14
15 "test get contract (PBT)" in {
16   forAll { (oa: Option[Int]) =>          // <-- quantifier
17     whenever (!oa.isEmpty)               // <-- pre-condition
18     { (Some(get(oa))) should be (oa) }  // <-- post-condition
19   }
20 } // An explicit contract; 100 different tests without test case tables and iterations
```

Homework

- The goal is to **test our streams library**
- We invert the situation from previous weeks:
  assume that the type (Stream) is implemented, you write the test file
- The **README file is key**. Follow the task meticulously as this week we are using automatic grading
- **Automatic grading**, so follow instructions carefully
- The build system is **set up with the testing library**, and 3 example tests
- There is even an **example generator** of streams
- **Add more tests**, and possibly more generators
- There must be **several property-based tests** (PBT) in your solution to pass.



Resources on ScalaTest

- Scalatest provides a multitude of interfaces
- Stick to **AnyFreeSpec** (like the example code)
- Switching to other flavor fine (won't break grader) but then read different docs chapter
- **Scalatest's guide to property testing**: `http://www.scalatest.org/user_guide/property_based_testing`.
- You can ignore the scalacheck part. Scalatest provides two syntaxes for property testing (scalatest and scalacheck) the homework project is configured for both. They are different and incompatible (should not be used in the same test).
- OK to use scalacheck API via scalatest if you know what you're doing; won't break the grader
- **Using matchers**: `http://www.scalatest.org/user_guide/using_matchers`
  Read to learn about other assertions than `should be`
- **Cheat-sheet** on FreeSpec: `http://www.scalatest.org/at_a_glance/FreeSpec`. Use for find-in-page (not AnyFreeSpec but API seems to be the same, docs not updated)
- Additionally, next week we study the design of `Prop` and `Gen`, to deepen understanding