@AndrzejWasowski

Andrzej Wąsowski

Advanced Programming

Program Correctness & Property-based Testing

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

Property-Based Testing

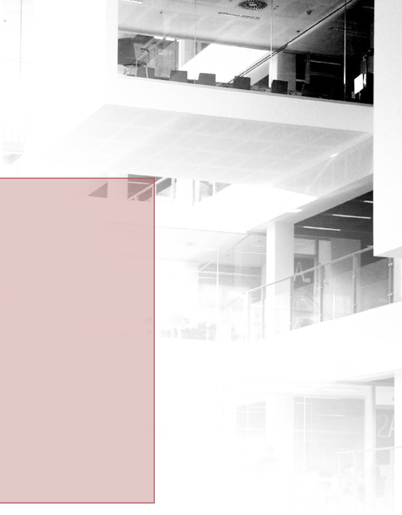
What problem we are solving today?

The Problem

- Scenario testing (unit testing) tries only the situations for which you created test-cases, and which you anticipated
- This **increases the cost** of testing (you need to create many test-cases)
- This **lowers the coverage** of tests (only things you anticipated are tried)

The Solution

- **Automate** generation of test data
- Use **randomized inputs**, generated in controlled manner (which we know how to do from the State chapter)
- **Generalize** tests to properties that should hold on many possible inputs



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

AGENDA

Assertions

```scala
// assertion: an empty list has size zero
assert(List().size == 0)

def f(l: List[Int]): Unit =
  // A contract for Cons: If I have a list, and add an element to it
  // it will certainly have at least one element in it
  assert(Cons(42, l).size > 0)

  // A contract for Cons: If length of a list is n, and you add (Cons)
  // an element to the list, it will have size n+1
  assert(l.size + 1 == Cons (42, l).size)

  // A contract for Cons: If you add (Cons) an element to the list,
  // then this element will be the head of the list.
  assert(Cons(42, l).headOption == Some (42))

def head(l: List[Int]): Int =
  // Precondition for head: l is not empty
  assert( ... ) // <-----
```

- Analyze **examples**
- **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

```scala
def isEmpty: Boolean = this match { case None => true; case _ => false }
// Get the value held by an option
// Precondition for 'this': !this.isEmpty
def get: A = ...

// Apply a binary operator to all elements of this iterable collection
// Like foldRight, but with an implicit initial element. Examples:
// List(1,2,3).reduce[Int] { _+_ } == 6
// List(1).reduce[Int] { _+_ } == 1
// Some(1).reduce[Int] { _+_ } == 1
// Nil.reduce[Int] { _+_ } == 0
// precondition for 'this': true
def reduceRight[B >: A](op: (A, B) => B): B

// Count the number of elements in a collection that satisfy a predicate
// Task: what is the precondition for 'this'?
def count(p: A => Boolean): Int

// Always fail with an exception
// Task: what is the precondition for this.f to avoid failure?
def f = throw 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
  - $n >= 2$ is a pre-condition, too. **Why?**
  - $n >= 0$ is weaker, $n >= 2$ 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 one aspect of logics)

Post-conditions

```scala
def size[A](t: Tree[A]): Int
// A post-condition: forall t. size(t) >= 1

def depth[A](t: Tree[A]): Int
// A post-condition: forall t. depth(t) >= 0

// A helper function
def get(t: Tree[Int]): Int = t match
  case Leaf(n) => n
  case Branch(l, _) => get(l)

// Now we use it
def maximum(t: Tree[Int]): Int
// A post-condition: forall t: maximum(t) >= get(t)

// We previously defined
def nextInt: (Int, RNG) = ...
// The postcondition: true
// Not the strongest post-condition for random,
// but it is hard to give a stronger one.
// This would require formally defining the pseudo-
// 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

```
// Returns this scala.Option if it is nonempty, otherwise return
// the result of evaluating alternative.
def orElse[B >: A](alternative: => Option[B]): Option[B]
// pre-condition: true
// post-condition:
//   (!this.isEmpty => orElse(alternative) == this        ) &&
//   ( this.isEmpty => orElse(alternative) == alternative)

def maximum (t: Tree[Int]): Int
// pre-condition: true
// post-condition:
//    (forall a in t: { a <= maximum(t) }) &&
//    (exist  a in t: { a == maximum(t) })
// (forAll/exist could be implemented for trees as static methods)

// For partial functions a pre-condition is not true.
def get[A](oa: Option[A]): A = ...
// pre-condition: !oa.isEmpty
// 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
(with trivial/true preconditions)

```
property("Ex04.02: identity is a unit with map") =
  forAll { (t: Tree[Int]) => Tree.map(t)(identity[Int]) == t }

property("Ex04.03: map does not change size") =
  forAll { (t: Tree[Int], f: Int => Int) =>
    Tree.size(Tree.map(t)(f)) == Tree.size(t) }

property("Ex04.04: map is ’associative’") =
  forAll { (t: Tree[Int], f: Int => Int, g: Int => Int) =>
    Tree.map (Tree.map(t)(f))(g) == Tree.map(t)(g compose f) }
```

- `forAll` generates random `Tree[Int]`
- `forAll` checks if the predicate (lambda) holds for all random data. Fail if false
- This code uses the **ScalaCheck** library
- We implement a similar lib next week

This code can be found in `030-option/Exercises.test.scala`

Property-based testing in a nutshell /2

Test-data generators

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

given arbitraryTree[A: Arbitrary]: Arbitrary[Tree[A]] =
  Arbitrary[Tree[A]](genTree)
```

- This week we learn how to **use** the API
- Next we **build** such API, incl. givens + 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** a given 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 a **given** generator of arbitrary trees of `As` (wrapping `genTree`)

This code can be found in `030-option/Exercises.test.scala`

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

```scala
// Recall the specification     // pre-condition: !oa.isEmpty
def get[A](oa: Option[A]): A    // post-condition: Some(get(oa)) == oa

// When testing in the classical way, create a value 'oa', run get, inspect the result
property ("test get contract") = // Example with ScalaCheck simple Boolean property
  val oa = Some(1)               // (like an assertion)
  get (oa) == 1                  // Any Boolean function (predicate) is treated like an assert

// Two weaknesses:
// 1. For another integer, write another test; create repetive code (or iterate a collection)
// 2. The contract specification is not explicit in the test.
// A property-based test does the iteration implicitly and solves these problems:

property ("test get contract (PBT)") =
  forAll { (oa: Option[Int]) =>         // <-- quantifier
    !oa.isEmpty ==>                     // <-- pre-condition (implication)
      Some(get(oa)) == oa }             // <-- post-condition

// An explicit contract; 100 different tests without test case tables and iterations
```

Homework

- The goal is to **test our lazy list library**
- We invert the situation from previous weeks:
  assume that the type (LazyList) is implemented, you write the test file
- **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 lazy lists
- **Add more tests**, and possibly more generators
- There must be **several property-based tests** (PBT) in your solution to pass.

Resources on ScalaCheck

- ScalaCheck: `https://github.com/typelevel/scalacheck/blob/main/doc/UserGuide.md`
- `Gen`, a key type: `https://javadoc.io/doc/org.scalacheck/scalacheck_3/1.16.0/org/scalacheck/Gen.html`
- `Arbitrary`, a key type: `https://javadoc.io/doc/org.scalacheck/scalacheck_3/1.16.0/org/scalacheck/Arbitrary.html`.
- I also find source code at `https://github.com/typelevel/scalacheck/blob/v1.16.0/core/shared/src/main/scala/org/scalacheck/Gen.scala` useful to read (actually I read it more often than the docs).
- Additionally, next week we study the design of `Prop` and `Gen`, to deepen understanding (you can start reading Ch. 8)

Further reading: Blogs! (Read one for context)

- **C++:** couldn't find a nice blog post; read
  the user guide of rapidcheck if C++ is your fare: `https://github.com/emil-e/rapidcheck/blob/master/doc/user_guide.md`
- **C#**: `https://www.codit.eu/blog/property-based-testing-with-c/` with `https://fscheck.github.io/FsCheck/`
- **F#**: `https://www.codit.eu/blog/practically-property-based-testing-your-strict-domain-model-in-f-c/`
  using `https://fscheck.github.io/FsCheck/`
- **Go:** `https://earthly.dev/blog/property-based-testing/` uses a small CSV file manipulation tool as an example
- **Haskell:** `https://www.fpcomplete.com/blog/2017/01/quickcheck/` with Quickcheck, of course!
- **Java:** `https://yoan-thirion.medium.com/improve-your-software-quality-with-property-based-testing-70bd5ad9a09a`
  with junit-quickcheck. There is also `https://jqwik.net` which seems to be a rich library used also in Kotlin and Groovy
- **JavaScript/TypeScript:**
  `https://mokkapps.de/blog/property-based-testing-with-type-script` using `https://github.com/dubzzz/fast-check`
- **Kotlin:** `https://instil.co/blog/from-tdd-to-pbt-via-kotest/`
  Uses `https://kotest.io` which apparently also runs on non-jvm tool chains of Kotlin.
- **Python:** `https://medium.com/clarityai-engineering/property-based-testing-a-practical-approach-in-python-with-hypothesis-and-pandas-6082d737c3ee`
  and `https://datascience.blog.wzb.eu/2019/11/08/property-based-testing-for-scientific-code-in-python/`,
  the latter with more data-science angle, both using Hypothesis.
- **Rust:** with `https://blog.auxon.io/2021/02/01/effective-property-based-testing/`
  with proptest, and `https://dev.to/itminds/introduction-to-property-based-testing-via-rust-3h6f` with quickcheck
- **Scala:** `medium.com/analytics-vidhya/property-based-testing-scalatest-scalacheck-52261a2b5c2c`,
  shows how to PBT within ScalaTest (the unit testing framework for Scala)