CSC 347 - Concepts of Programming Languages

Unit Tests and Contracts

Instructor: James Riely

Compute the Factorial of a Number

```
1 Implement an iterative Scala 3 program
2 that computes the factorial of a number
```

```scala
def factorial(n: Int) : Int =
  var result = 1
  for i <- 1 to n do
    result *= i
  result
```

- How do we use `factorial`?
- Is the code correct?
- How can we convince ourselves whether code is correct?

Learning Objectives

❓ How can we build trust in (auto-generated) code?

- Read and express unit tests
- Read and express contracts

Unit Tests

💡 Unit tests provide sample inputs and test expected outputs of code in isolation

- Unit tests specify **expected behavior**
- Unit tests are **documentation**
  - How to initialize and use some code
  - What to expect as a result on legal inputs
  - What to expect as an exception on illegal inputs
- Unit tests **prevent quality regression**

Unit Tests with ScalaTest

```scala
def factorial(n: Int) : Int =
  var result = 1
  for i <- 1 to n do
    result *= i
  result
```

```scala
class FactorialTests
  extends AnyFlatSpec with should.Matchers:

  "Factorial" should "compute 0! correctly" in {
    factorial(0) should be (1)
  }

  it should "compute 1! correctly" in {
    factorial(1) should be (1)
  }

  it should "compute 3! correctly" in {
    factorial(3) should be (6)
  }
```

- Matchers `should be` etc. express expected results
- ⚠ Can only test a finite number of examples

Contracts

💡 Contracts are formal and verifiable interface specifications

- **Preconditions** are expectations that must be met by a client
- **Postconditions** are guarantees made by the component
- **Invariants** are maintained (e.g., every loop iteration)

Formal Specification and Verification

Specification

- Hoare triples: `{P} C {Q}`
- Modal logic: `P -> [C]Q`
- Code:

```
def C() = {
  require (P)
  // ...
} ensuring (Q)
```

Verification

- **Theorem proving** shows correctness mathematically for all possible inputs and all possible executions
- **Model checking** shows correctness for finite-length executions
- **Runtime verification** checks contracts while executing the software and throws exceptions when violated

Contracts in Scala

```scala
def factorial(n: Int) : Int = {
  require (n>=0)
  var result = 1
  for i <- 1 to n do
    result *= i
  result
} ensuring {
  (result: Int) =>
    result == factorialContract(n)
}
```

⚠ Useful?

```scala
def factorialContract(n: Int) : Int =
  var result = 1
  for i <- 1 to n do result *= i
  result
```

💡 Alternative solution!

```scala
def factorialContract(n: Int) : Int =
  if n<=1 then 1
  else n * factorialContract(n-1)
```

- Contract specifies input assumptions (`require`) and output guarantees (`ensuring`)
- Contract can be an alternative implementation
- ⚠ Runtime checking adds computational overhead

Summary

Unit Tests

- Execute code with some inputs and test for expected outputs
  - ⊕ documents known and expected use and behavior
  - ⊖ finite number of tests
  - Best for: regression testing

Contracts

- Express input assumptions and output guarantees
  - ⊕ show correctness for all possible inputs and executions
  - ⊖ computational overhead, extra effort
  - Best for: correctness-critical applications