



Week 10 (Monday): Mutation and Functional Programming

CS-214 Software Construction

Outline

- ▶ Imperative Scala Constructs
- ▶ Side Effects May Include Confusion
- ▶ Making Arguments and Results Explicit
- ▶ Logic for Reasoning about State Change
- ▶ From Functions to Simple Imperative Code

Imperative Scala Constructs

Mutable Variables

Mutation = Change (of program variables or of the environment)

So far, we have considered **values**, which never change:

```
val a: Int = 42 // 42 forever.  
val b = a + 4   // remains 46 forever. The type is inferred (here: Int)
```

Now we consider **mutable variables**, e.g. local variables:

```
var a: Int = 42 // a has initial value 42  
a = a + 3      // a has current value 45  
var b = a + 4  // b has initial value 49. Type inferred: Int  
def c = a + 10 // when evaluated here, c gives 55  
a = a + 1     // a has a new current value: 46  
b             // 49  
c             // 56  
b = b + a     // now b has value 95
```

Local variables and while loop

We have already seen these in lectures on parallelism.

```
def mul(x: BigInt, y: BigInt): BigInt = {  
  require(y >= 0)  
  var bound = y  
  var res: BigInt = 0  
  while bound > 0 do  
    res = res + x  
    bound -= 1  
  res  
}.ensuring(_ == x * y)
```

Definition of whileDo

How could we define while using a function (call it whileDo)?

The function whileDo can be defined as follows:

```
def whileDo(condition: => Boolean)(command: => Unit): Unit =  
  if condition then  
    command  
    whileDo(condition)(command) // looks like infinite loop, but it isn't
```

Note: The condition and the command must be passed by name so that they're reevaluated in each iteration.

Note: whileDo is tail recursive

Using whileDo

```
def mul2(x: BigInt, y: BigInt): BigInt = {  
  require(y >= 0)  
  var bound = y  
  var res: BigInt = 0  
  whileDo(bound > 0):  
    res = res + x  
    bound -= 1  
  res  
} ensuring(_ == x * y)
```

Exercise

Write a function implementing a repeat loop that is used as follows:

```
repeatUntil {  
  command  
} ( condition )
```

It should execute `command` one or more times, until `condition` is true.

Use the following signature for `repeatUntil`:

```
def repeatUntil(command: => Unit)(condition: => Boolean) =
```

State Mutating First-Class Functions

```
scala> List(1,2,3).map(println(_))
```

```
1
```

```
2
```

```
3
```

```
val res0: List[Unit] = List((), (), ())
```

```
scala> List(1,2,3).foreach(println(_)) // result type is foreach is Unit
```

```
1
```

```
2
```

```
3
```

For Loops

```
for i <- 1 until 3 do
  System.out.print(f"$i ") // note that there is no `yield`
```

The above prints 1 2 on the terminal. Its meaning is given via foreach methods, defined on collections with elements of type T:

```
def foreach(f: T => Unit): Unit =
  // apply `f` to each element of the collection
```

Expanded by compiler to: (1 until 3).foreach(i => System.out.print(f"\$i "))

Example

```
for i <- 1 until 3; j <- "abc" do println(s"$i $j")
```

is expanded by the compiler to:

```
(1 until 3).foreach(i => "abc".foreach(j => println(s"$i $j")))
```

Other Forms of Mutable State

1) Classes with mutable fields:

```
case class Planet(var x: Double, var y: Double, var z: Double)
val p = Planet(0.0, 0.0, 0.0)
p.x = p.x + dx // change the x coordinate of p by dx
```

2) Arrays (immutable `.length`, mutable content):

```
// a : Array[Int]
a(k) = a(k) + 1 // increment a(k) by 1
```

3) (Other) mutable collections:

```
import scala.collection.mutable.*
val b: ListBuffer[Int] = ListBuffer(3) // b initially contains 3
b.addOne(42)                       // b now contains 3,42
```

Classes with State and Their Use

```
case class Accumulator(private var sum: BigInt):  
  def get = sum  
  def add(x: BigInt): Unit =  
    sum = sum + x  
  
val a = Accumulator(0)  
a.add(100)  
a.add(30)  
a.get    // 130
```

Example: Sum Using Accumulator Object

```
case class Accumulator(private var sum: BigInt):  
  def get = sum  
  def add(x: BigInt): Unit =  
    sum = sum + x
```

```
val a = Accumulator(0)  
scala> List(10,50,20).map(a.add(_))  
val res0: List[Unit] = List((), (), ())  
scala> a.get // 80
```

```
def sum(lst: List[BigInt]): BigInt =  
  val a = Accumulator(0)  
  lst.map(a.add(_))  
  a.get
```

Arrays

<https://www.scala-lang.org/api/2.13.3/scala/Array.html>

Important for efficiency, map to JVM arrays when you run on JVM

```
scala> val a = Array(10, 20, 30)
val a: Array[Int] = Array(10, 20, 30)
```

```
scala> val b = Array.fill(5)(42)
val b: Array[Int] = Array(42, 42, 42, 42, 42)
```

```
scala> val c = Array.tabulate(5)(i => 10*i)
val c: Array[Int] = Array(0, 10, 20, 30, 40)
```

```
scala> val d = new Array[Int](5)
val d: Array[Int] = Array(0, 0, 0, 0, 0)
```

update Method abbreviation

```
class FunArray[A](default: A):  
  private var f : Int => A = (x:Int) => default  
  def apply(i: Int): A = f(i)
```

```
  def update(ind: Int, newV: A): Unit =  
    val oldF = f  
    val newF = (i:Int) =>  
      if i == ind then newV  
      else oldF(i)  
    f = newF
```

```
val a = FunArray(42); a(100) // 42  
a(3) = 17  
a(100) // 42  
a(3) // 17
```

Mutable += versus Copying Concatenation

```
import scala.collection.mutable.*
val lst1 = ListBuffer(1, 2, 3)
val lst2 = ListBuffer(10, 20)
val lst3 = lst1 ++ lst2 // ListBuffer(1, 2, 3, 10, 20)
println(lst1)           // still ListBuffer(1, 2, 3)

val lst4 = lst1 += ListBuffer(100, 200) // ListBuffer(1, 2, 3, 100, 200)
println(lst1)           // ListBuffer(1, 2, 3, 100, 200)
```

Note that += implementation returns **this**, so lst1 and lst4 are the same object.

If we need both versions, we may prefer functional operations and immutable data structures.

State Machine Functionally

```
case class StateF(flipped: Vector[Boolean]):  
  def click(i: Int): StateF =  
    val st = flipped(i)  
    StateF(flipped.updated(i, !st))  
  
val s0 = StateF(Vector(false, false,  
                    false, false))  
val s1 = s0.click(1)  
val s2 = s1.click(2)  
val s3 = s2.click(1)
```

We retain access to old versions because objects do not change.

Need to pass along the right version: s0, s1, s2, s3

We copy $\log(n)$ amount of state. Access is $\log(n)$.

State Machine with Shallow Mutation

```
case class StateSh(var flipped: Vector[Boolean]):  
  def click(i: Int): Unit =  
    val st = flipped(i)  
    flipped = flipped.updated(i, !st)  
  
val s00 = StateSh(Vector(false, false,  
                        false, false))  
  
s00.click(1)  
s00.click(2)  
s00.click(1)
```

Object changes but only shallowly.

We use the same reference to a mutable object.

We copy $\log(n)$ amount of state. Access is $\log(n)$.

State Machine with Deep Mutation

```
case class StateD(flipped: Array[Boolean]):  
  def click(i: Int): Unit =  
    val st = flipped(i)  
    flipped(i) = !st  
  
val s01 = StateD(Array(false, false,  
                  false, false))  
  
s01.click(1)  
s01.click(2)  
s01.click(1)
```

We use the same reference to a mutable object.

No state copying. Access is $O(1)$.

Downside: if we want to keep the old version, we must copy the entire array.

Side Effects May Include: Confusion

Inputs, Outputs, and Side Effects

When we have mutation, to understand what function does we need to know its:

- ▶ inputs: parameter list (like before), **reads list: additional visible names**
- ▶ output: return value (like before)
- ▶ side effects (**modifies list: changes to state**), a kind of output

```
var logBuffer: String = ""
def log(msg: String): Unit =
  // reads: logBuffer;  modifies: logBuffer
  logBuffer = logBuffer + msg + "\n"

def sort(l: List[Int]): List[Int] =
  // reads: logBuffer, threshold;  modifies: logBuffer
  if l.length < threshold then insertionSort(l)
  else log("large list, using merge sort")
      mergeSort(l)
```

Convenience of State: Renaming Strings in a Tree

Given a binary tree storing strings, make a new version of it by adding distinct numbers to each of the strings in leaves.

```
sealed abstract class Tree
case class Leaf(s: String) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

val t = Node(Node(Leaf("x"), Leaf("y")), Node(Leaf("x"), Leaf("x")))
// Node(Node(Leaf(x),Leaf(y)),Node(Leaf(x),Leaf(x)))

val t1 = t.distVersion
// Node(Node(Leaf(x_1),Leaf(y_2)),Node(Leaf(x_3),Leaf(x_4)))
```

Such transformations can be used to rename variables in compilers and theorem provers

Implementation Using a Counter

```
case class Counter(var current: Long):  
  def next: Long = // not only returns Long, but also changes current  
    current += 1  
    current  
  
val c = Counter(0L)  
  
extension (t: Tree)  
  def distVersion: Tree =  
    t match  
      case Leaf(s) => Leaf(s + "_" + c.next.toString) // c.next is side effecting!  
      case Node(left, right) => Node(left.distVersion, right.distVersion)
```

Using distVersion

Repeating distVersion on same thing gives different results:

```
val t = Node(Node(Leaf("x"), Leaf("y")), Node(Leaf("x"), Leaf("x")))
```

```
val t1 = t.distVersion
```

```
// Node(Node(Leaf(x_1),Leaf(y_2)),Node(Leaf(x_3),Leaf(x_4)))
```

```
val t2 = t.distVersion
```

```
// Node(Node(Leaf(x_5),Leaf(y_6)),Node(Leaf(x_7),Leaf(x_8)))
```

Reasoning: Equality Relation is Reflexive

In mathematics we have

$$x == x$$

for every value x .

This also holds for (terminating) expressions in pure functional programs, e.g.:

$$(lst1 ++ lst2) == (lst1 ++ lst2)$$

A pure expression is one that uses only purely functional constructs that we have seen so far: `val`-s, recursion, `if` expressions, operations on values.

- ▶ no mutation
- ▶ no `println` or other I/O, no `scala.util.Random.nextInt`

Such expressions return the same value whenever they terminate.

If x_1 and x_2 are same then $f(x_1)$ and $f(x_2)$ are same.

Mutable State Ruins Such Mathematical Properties!

This assertion fails:

```
assert(t.distVersion == t.distVersion) // Print t.distVersion twice to see why
```

An even simpler example: for a counter `c = Counter(0L)`

```
assert(c.next == c.next) // fails: reduces to 1 == 2
```

and

```
def f(x: Long): Long =  
  if c.next > 10 then x else x + 100
```

Then `f(5)` in some cases evaluates to 5, in others to 105.

`x == x` does not always hold for expressions `x` that refer to mutable state.

How can we recover mathematical reasoning?

Diagnosis and Two Solutions

The Scala function:

```
def f(x: Long): Long =  
  if c.next > 10 then x else x + 100
```

is not a mathematical function $\text{Long} \Rightarrow \text{Long}$:

- ▶ its result depends on the value of `c.current`
- ▶ its outcome is not only the return value, but also a new value of `c.current`

Two solutions:

1. Make Arguments and Results Explicit
2. Develop Rules to Reason about Imperative Programs (Hoare logic, Dynamic logic)

Making Arguments and Results Explicit

Approach 1: Making Arguments and Results Explicit

The Scala function:

```
def f(x: Long): Long =  
  if c.next > 10 then x else x + 100
```

is not a mathematical function $\text{Long} \Rightarrow \text{Long}$:

- ▶ its result depends on the value of `c.current`
- ▶ its outcome is not only the return value, but also a new value of `c.current`

First, let us make the counter parameter explicit:

```
def f1(x: Long, c: Counter): Long =  
  if c.next > 10 then x else x + 100
```

But it is still not the case that $f1(x, c) = f1(x, c)$ because a call changes the value of `c.current`.

Making Arguments and Effects Explicit

A mathematical function that models a program function with side effects needs:

- ▶ additional *parameter* for each part of state that it reads
- ▶ additional *result* for each part of state that it changes

```
def f(x: Long): Long = ...           // Scala
def f_pure(x: Long, c: Long): (Long, Long) = ... // Pure Scala (math)
```

```
// Scala, method of Counter(var current):
```

```
  def next: Long = // not only returns Long, but also changes current
    current += 1
    current
```

```
// math: extra argument and an extra result, 2nd component of returned pair
```

```
def next_pure(current: Long): (Long, Long) =
  (current + 1, current + 1 /* new value of the counter */)
```

Removing mystery from `assert(c.next=c.next)`

```
assert(c.next == c.next)
```

really means

```
val v1 = c.next  
val v2 = c.next  
assert(v1 == v2)
```

which means

```
val (v1,c1) = next_pure(c)  
val (v2,c2) = next_pure(c1)  
assert(v1 == v2)
```

Which does not hold: not in math, not in Scala.

Meaning of function f

The Scala function:

```
def f(x: Long): Long =  
  if c.next > 10 then x else x + 100
```

becomes

```
def f_pure(x: Long, c: Long): (Long, Long) =  
  val (res, c1) = next_pure(c)  
  if res > 10 then (x, c1) else (x + 10, c1)
```

Let us Translate our String Renaming Function to Be Pure

```
sealed abstract class Tree
case class Leaf(s: String) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

extension (t: Tree)
  def distVersion: Tree =
    t match
      case Leaf(s) => Leaf(s + "_" + c.next.toString)
      case Node(left, right) => Node(left.distVersion, right.distVersion)

// we will translate it into a new function:
def distVersion_pure(c: Long): (Tree, Long) = ...
//           ^^^^           ^^^^
//           input counter   output counter
```

Translating String Renaming Function on Trees

```
case Leaf(s) => Leaf(s + "_" + c.next.toString)
```

~>

```
case Leaf(s) =>  
  val (res, c1) = next_pure(c)  
  (Leaf(s + "_" + res.toString), c1)
```

```
case Node(left, right) => Node(left.distVersion, right.distVersion)
```

~>

```
case Node(left, right) =>  
  val (left1, c1) = left.distVersion(c)  
  val (right1, c2) = right.distVersion(c1)  
  (Node(left1, right1), c2)
```

Result of Translation

```
extension (t: Tree)
  def distVersion_pure(c: Long): (Tree, Long) =
    t match
      case Leaf(s) =>
        val (res, c1) = next_pure(c)
        (Leaf(s + "_" + res.toString), c1)
      case Node(left, right) =>
        val (left1, c1) = left.distVersion_pure(c)
        val (right1, c2) = right.distVersion_pure(c1)
        (Node(left1, right1), c2)
```

Logic for Reasoning about State Change

Approach 2: Logic for Reasoning about State Change

Dynamic logic combines programs that change state and assertions.

Special case: Hoare logic (named after C.A.R Hoare)

We have seen function specifications. For functions returning Unit we have

```
def f: Unit = {  
  require(pre)  
  body  
}.ensuring(_ => post)
```

Statements of Hoare logic are triples: (pre, body, post)

- ▶ often denoted $\{pre\}body\{post\}$, or (in Rosen), $pre\{body\}post$

To say that a Hoare triple **holds** means that, for all states:

if pre holds in some state, then post holds after body executes from that state.

Example of Sequence of Hoare Logic Triples

Often we use post of one triple as pre of next one.

We can write such proofs using assertions: `assert(pre); body; assert(post)`

```
assert(res == (y - bound)*x && bound > 0)
res = res + x
assert(res == (y - bound + 1)*x && bound > 0)
bound -= 1
assert(res == (y - bound)*x && bound >= 0)
```

Two Hoare triples for individual statements imply triple for combined statement:

```
assert(res == (y - bound)*x && bound > 0)
res = res + x; bound -= 1
assert(res == (y - bound)*x && bound >= 0)
```

Proof of Multiplication Using Hoare Triples

```
def mulHoare(x: BigInt, y: BigInt): BigInt = {  
  require(y >= 0)  
  var bound = y  
  var res: BigInt = 0  
  assert(bound == y && res == 0)  
  while bound > 0 do  
    assert(res == (y - bound)*x && bound > 0)  
    res = res + x; bound -= 1  
    assert(res == (y - bound)*x && bound >= 0) // loop invariant  
  assert(res == (y - bound)*x && bound == 0)  
  assert(res == y*x && bound == 0)  
  res  
}.ensuring(_ == x * y)
```

Each assertion is preserved in next step, so they hold in all executions.

Hoare Logic Rules: Sequence of Statements

assert(p)

s1

assert(q)

assert(q)

s2

assert(r)

assert(p)

s1

s2

assert(r)

Horizontal bar ----- means: if things above hold, so do things below.

Hoare Logic Rules: While Loop (with Loop Invariant p)

```
assert(p && cond)
body
assert(p)
```

```
assert(p)
while cond do
  body
assert(p && !cond)
```

Hoare Logic Rules: If Statement

```
assert(p && cond)
s1
assert(q)
```

```
assert(p && !cond)
s2
assert(q)
```

```
assert(p)
if cond then
  s1
else
  s2
assert(q)
```

Hoare Logic Rules: Weakening

<code>assert(p)</code>	<code>p1 implies p</code>
<code>S</code>	
<code>assert(q)</code>	<code>q implies q1</code>

<code>assert(p1)</code>
<code>S</code>
<code>assert(q1)</code>

Hoare Logic Rules: Assignment Statement

```
assert(p(e))  
x = e  
assert(p(x))
```

Example:

```
assert(y + 1 > 5)  
x = y + 1  
assert(x > 5)
```

Note: these rules are for local vars storing values like Int (not sufficient for mutable data structures).

From Functions to Simple Imperative Code

We Can Translate Functional Programs to Simple Imperative Ones

Machine code (instructions for CPU) is a simple imperative language

Compilers translate languages like Scala to machine code

This can be done automatically and should be mostly left to compilers

We can do it by hand

- ▶ to learn how things work
- ▶ to improve stack and memory use

Example: Tail Recursion \Leftrightarrow Loops

```
def whileDo =  
  if condition then  
    command  
    whileDo
```



```
while condition do  
  command
```

We can transform one into another.

While loop does not consume stack on JVM, so it reduces chances of StackOverflow program crash.

Difficulty with General Recursion

Remembering where to return!

```
def eval(expr: Expr): Int =  
  expr match  
    case Const(i) => i  
    case Minus(e1, e2) =>  
      val v1 = eval(e1)  
      val v2 = eval(e2)  
      v1 - v2
```

When returning from a deep expression, we need to know if we have just evaluated v_1 or just evaluated v_2 .

Compiled program uses **stack** for this purpose.

We can use our own stack, e.g. a `var` of type `List`. More next week.

How to Represent Lists, Trees and Other Structures

```
abstract class IntSet
case class Empty() extends IntSet
case class Node(left: IntSet, elem: Int, right: IntSet) extends IntSet
                // 1           2           3
```

Program gets from the Operating System a chunk of memory, like `mem: Array[Addr]`

```
type Addr = Int
```

Program decides how to organize this array to store objects

References to objects are indices into `mem` array

```
def getLeft(n: Addr) = mem(n + 1)
def getElem(n: Addr) = mem(n + 2)
def getRight(n: Addr) = mem(n + 3)
```

Sketch of Creating a new Node

Let var nextFree: Int denote next available space in the array

```
def Node(left: Addr, elem: Int, right: Addr): Addr =  
  if nextFree + 3 >= mem.size then error("Out of memory")  
  else  
    val res = nextFree  
    mem(nextFree) = NodeTAG // so we know if it is Node, not Empty  
    mem(nextFree + 1) = left  
    mem(nextFree + 2) = elem  
    mem(nextFree + 3) = right  
    nextFree += 4  
    res
```

In reality, runtime also does garbage collection of unreachable nodes

Visibility of Object Identity

The Addr is not an integer on JVM or in Scala, but can be observed.

```
scala> class A  
// defined class A
```

```
scala> val a1 = new A  
val a1: A = A@6d31f106
```

```
scala> val a2 = new A  
val a2: A = A@32da97fd
```

```
scala> a1 == a2  
val res0: Boolean = false
```

Observing Object Identity Through Mutation

```
case class Counter(var current: Long):  
  def next: Long = // not only returns Long, but also changes current  
    current += 1  
    current
```

The following function will return true if c1 and c2 are same object:

```
def sameCounter(c1: Counter, c2: Counter): Boolean =  
  val x = c1.current  
  val n = c2.next  
  c1.current != x
```

Aliasing of mutable objects affects program behavior.

- ▶ makes informal and formal reasoning difficult
- ▶ options: 1) use memory array model 2) restrict aliasing (\approx Rust)

Another Side Effect Example: Vector Addition

```
extension (a1: Array[Int])
  def +(a2: Array[Int]): Array[Int] =
    val res = a1
    var i = 0
    (0 until res.length).foreach: i =>
      res(i) = a1(i) + a2(i)
    res

val a1 = Array(100, 100, 100)
val a2 = Array(5, 5, 5)
val example = a1 + a2 + a1 // Array(210, 210, 210) - why not 205 ?
```

Bonus Example: FunArray2 variant of FunArray

```
class FunArray2[A](default: A):  
  private var f : Int => A = (x:Int) => default  
  def apply(i: Int): A = f(i)  
  
  def update(ind: Int, newV: A): Unit =  
    val newF = (i:Int) =>  
      val oldF = f  
      if i == ind then newV  
      else oldF(i)  
    f = newF  
  
val a = FunArray2(42)  
a(100) // 42  
a(3) = 17  
a(100) // infinite loop
```

Bonus Example: FunArray2 After Replacing vals

```
class FunArray2[A](default: A):  
  private var f : Int => A = (x:Int) => default  
  def apply(i: Int): A = f(i)  
  
  def update(ind: Int, newV: A): Unit =  
    f = (i:Int) =>  
      if i == ind then newV  
      else this.f(i)  
  
val a = FunArray2(42)  
a(100) // 42  
a(3) = 17  
a(100) // infinite loop
```

Mutation and function values allow us to dynamically introduce recursion