Lecture 2

1 Immutability

Consider the function `incrList` in fig. 3.1, which increments all numbers in a list. We want to emphasize that `incrList` *produces a new list* and leaves the original list unchanged.

You can observe this behavior in the Scala console:

```
scala> val original = List(10, 20, 30)
original: List[Int] = List(10, 20, 30)

scala> val result = incrList(original)
result: List[Int] = List(11, 21, 31)

scala> original
res0: List[Int] = List(10, 20, 30)

scala> result
res1: List[Int] = List(11, 21, 31)
```

At no point did we change or *mutate* the original list. In fact, lists in Scala are *immutable* data structures. There is no way to update their head or tail. Programming with immutable data structures is a key part of *functional programming*, which we will emphasize in the first half of this course.

2 Case Classes

Scala classes tend to be shorter than their Java counterparts. For example, fig. 3.2a shows a `Point` class in Java and fig. 3.2b shows the equivalent class in Scala, which is much shorter! If we start adding methods, you'll have to write more code in Scala too. But, simple classes tend to be very short. But, we are not going to use classes yet. Instead, we are going to use *case classes*, which are unique to Scala.

If you write `case class`—

```
case class Point(x: Double, y: Double)
```

```
def incrList(lst: List[Int]): List[Int] = lst match {
  case Nil => Nil
  case head :: tail => (head + 1) :: incrList(tail)
}
```

Figure 3.1: A function that increments all numbers in a list.

```java
public class Point {

  double x;
  double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

}

Point pt = new Point(1, 2)
```

(a) A Java class that represents a point.

```
class Point(x : Double, y : Double)
val pt = new Point(1, 2)
```

(b) A Scala class that represents a point.

Figure 3.2: Scala code is usually much shorter and simpler than Java code.

—instead of just `class`, you get several conveniences. First, you can create values without writing `new`:

```
val pt = Point(1, 2)
```

Second, case classes have an automatically generated `toString` method that prints the fields:

```
scala> pt
pt: Point = Point(1.0,2.0)
```

Finally, all fields are public by default, so you can easily write simple functions, such as this one, without writing getters:

```
def magnitude(pt: Point) : Double = {
  math.sqrt(pt.x * pt.x + pt.y * pt.y)
}

test("3-4-5 triangles") {
  assert(magnitude(Point(3, 4)) == 5)
}
```

3 Sealed Case Classes

Imagine you're a new age librarian, tasked with cataloging information on the Internet. There are many types of information. Here are some significant ones:

```
case class Tweet(user: String, number: Long)
case class Xkcd(number: Int)
case class HackerNews(item: Int, points: Int)
```

Here are some examples:

```
// https://twitter.com/PLT_Borat/status/248038616654299136
val tweet = Tweet("PLT_Borat", 248038616654299136L)
// http://xkcd.com/1316/
val comic = Xkcd(1316)
// https://news.ycombinator.com/item?id=8169367
val news = HackerNews(8169367, 305)
```

Let's write a function called `getURL` that maps these items to their URLs, which is easy to do with pattern matching.

```
def getURL(item: Any): String = item match {
  case Tweet(user, number) => "https://twitter.com/" + user + "/status/" + number
  case Xkcd(number) => "http://xkcd.com/" + number
  case HackerNews(item, _) => "http://news.ycombinator.com/item?id=" + item
  case _ => sys.error("not a library item")
}
```

This definition is unsatisfactory. `getURL` takes values of `Any` type. So, it is really easy to get a runtime error:

```
scala> getURL("hello")
java.lang.RuntimeException: not a library item
 at scala.sys.package$.error(package.scala:27)
  at cmpsci220.package$$anonfun$1.apply(package.scala:13)
  at cmpsci220.package$$anonfun$1.apply(package.scala:13)
  at .getURL(<console>:38)
  ... 33 elided
```

To eliminate this kind of error, we need an `Item` type:

```
sealed trait Item
case class Tweet(user: String, number: Long) extends Item
case class Xkcd(number: Int) extends Item
case class HackerNews(item: Int, points: Int) extends Item
```

A *trait* in Scala is like an *interface*. They're also much more versatile than interfaces, but we'll get into that later.

Now, we can rewrite `getURL`, restricting the argument type:

```
def getURL(item: Item): String = item match {
  case Tweet(user, number) => "https://twitter.com/" + user + "/status/" + number
  case Xkcd(number) => "http://xkcd.com/" + number
  case HackerNews(item, _) => "http://news.ycombinator.com/item?id=" + item
}
```

This time, applying `getURL` to a non-item produces a type error as expected:

```
scala> item("hello")
<console>:43: error: type mismatch;
 found   : String("hello")
 required: Item
              getURL("hello")
```

A really nice feature of `match` is that it checks to ensure you've handled all cases. For example, suppose we forgot to write the HackerNews case. Scala prints the following error:

```
<console>:18: warning: match may not be exhaustive.
It would fail on the following input: HackerNews(_, _)
       def getURL(item: Item): String = item match {
                                        ^
error: No warnings can be incurred under -Xfatal-warnings.
```

More Pattern Matching You can use `match` to also match concrete values. For example, here is variant of getURL that censors a particular tweet from PLT Borat:

```
def getURL(item: Item): String = item match {
  case Tweet("PLT_Borat", 301113983723794432L) => "http://disney.com"
  case Tweet(user, number) => "https://twitter.com/" + user + "/status/" + number
  case Xkcd(number) => "http://xkcd.com/" + number
  case HackerNews(item, _) => "http://news.ycombinator.com/item?id=" + item
}
```

Imagine you'd added the censoring line, but accidentally removed the line that handles all other Tweets. Again, Scala will catch the error:

```
sealed trait SBinTree
case class SNode(lhs: BinTree, key: Int, value: String, rhs: BinTree) extends SBinTree
case class SLeaf() extends SBinTree

def find(k: Int, t: SBinTree): String = t match {
  case SLeaf() => sys.error("not found")
  case SNode(_, key, value, _) if (k == key) => value
  case SNode(lhs, key, _, _) if (k < key) => find(k, lhs)
  case SNode(_, key, _, rhs) => find(k, rhs)
}

def insert(k: Int, v: String, t: SBinTree): SBinTree = t match {
  case SLeaf() => SNode(SLeaf(), k, v, SLeaf())
  case SNode(lhs, key, _, rhs) if (k == key) => SNode(lhs, key, v, rhs)
  case SNode(lhs, key, value, rhs) if (k <= key) =>  SNode(insert(k, v, lhs), key, value, rhs)
  case SNode(lhs, key, value, rhs) => SNode(lhs, key, value, insert(k, v, rhs))
}

def size(t: BinTree): SBinTree = t match {
  case SLeaf() => 1
  case SNode(lhs, k, v, rhs) => size(lhs) + size(rhs) + 1
}
```

Figure 3.3: Binary search trees.

```
<console>:62: warning: match may not be exhaustive.
It would fail on the following inputs:
  Tweet("PLT_Borat", (x: Long forSome x not in 301113983723794432L)),
  Tweet((x: String forSome x not in "PLT_Borat"), 301113983723794432L),
  Tweet((x: String forSome x not in "PLT_Borat"), _),
  Tweet(_, (x: Long forSome x not in 301113983723794432L))
       def getURL(item: Item): String = item match {
                                        ^
error: No warnings can be incurred under -Xfatal-warnings.
```

It is a long error message. But, if you read it carefully, you'll see that it is very precisely describing exactly the cases that are missing.

4 Binary Search Trees

Figure 3.3 shows a type definition for binary search trees, where keys are integers and values are strings (which is why we call them `SBinTrees`). There are two kinds of binary trees: (1) empty trees, or `SLeaf()`s, and (2) non-empty trees, or `SNode(..)`s, which have a left-subtree, a numeric key, a string value, and a right-subtree. Given this definition, we can write canonical functions such as `size`, `find`, and `insert`, as shown in the same figure. We emphasize that the `insert` function does not modify the original tree, and instead produces a new binary tree with the element added.

4.1 Testing

There is a lot of code in fig. 3.3 that needs to be tested and there are two categories of tests that we can write. First, we should *test all the cases* of a function. For example, the `insert` function has four cases and the four tests below are chosen to exercise each case:

```
test("insert into empty tree") {
  assert(insert(200, "A", SLeaf()) == SNode(SLeaf(), 200, "A", SLeaf()))
```

```
}

test("insert into left-subtree") {
  assert(insert(200, "A", SNode(SLeaf(), 500, "B", SLeaf()))
    == SNode(SNode(SLeaf(), 200, "A", SLeaf()), 500, "B", SLeaf()))
}

test("insert into right-subtree") {
  assert(insert(700, "A", SNode(SLeaf(), 500, "B", SLeaf()))
    == SNode(Leaf(), 500, "B", SNode(SLeaf(), 700, "A", SLeaf())))
}

test("insert and replace") {
  assert(insert(200, "A", SNode(SLeaf(), 200, "B", SLeaf()))
    == SNode(SLeaf(), 200, "A", SLeaf()))
}
```

A robust test suite should exercise `insert` on larger examples, but it is important to understand exactly what each case is doing, which is what the descriptive string states.

The second kind if test is to *test properties that relate functions to each other.* There isn't a neat recipe for these kinds of tests. You have to think hard about what your code is actually doing. But, here are some properties for binary trees.

Property 1 *After we insert a key-value into a tree, we should be able to find it again.*

The following tests check that this property holds on examples where a key-value is inserted on the left, inserted on the right, or replaces the root:

```
val t1 = SNode(SLeaf(), 200, "A", SLeaf())

test("find after insert on right") {
  assert(find(500, insert(500, "B", t1)) == "B")
}

test("find after insert on left") {
  assert(find(100, insert(100, "B", t1)) == "B")
}

test("find after insert replaces") {
  assert(find(200, insert(200, "B", t1)) == "B")
}
```

Property 2 *Inserting a key-value into a binary tree produces a tree with size one greater than the original tree.*

The following tests check that this property holds for the three cases of `insert`:

```
val t1 = SNode(SLeaf(), 200, "A", SLeaf())

test("insert increases size (rhs)")
  assert(size(insert(500, "B", t1) == size(t1) + 1)
}

test("insert increases size (lhs)") {
  assert(size(insert(100, "B", t1) == size(t1) + 1)
}

test("insert increases size (replacement)") {
  assert(size(insert(200, "B", t1) == size(t1) + 1)
}
```

Unfortunately, the third test case fails because the property is wrong. When `insert` replaces a value in the tree, the size of the tree does not change. Here is a revised version of this property that accounts for this corner-case:

Property 3 *Inserting a key-value into a binary tree either (1) produces a tree with size one greater than the original tree or (2) the size of the produced tree is the same and the key was present in the original tree.*

We can write a function to test this property for any given k, v, and t and use it to write our tests compactly:

```
def check(k: Int, v: String, t: SBinTree): Boolean = {
  if (size(insert(k, v, t)) == size(t) + 1) {
    true
  }
  else {
    try {
      val r = find(k, t)
      true
    }
    catch {
      case exn:Exception => false
    }
  }
}

val t1 = SNode(SLeaf(), 200, "A", SLeaf())

test("insert increases size (rhs)")
  assert(check(500, "B", t1))
}

test("insert increases size (lhs)") {
  assert(check(100, "B", t1))
}

test("insert increases size (replacement)") {
  assert(check(200, "B", t1))
}
```

There are other properties that we can test. For example:

```
insert(k, v, insert(k, v, t)) == insert(k, v, t)
```

Moreover, if we write more functions to process binary trees, the number of interesting properties will keep growing.

5 Lists

Although we've used Scala's builtin lists, we can use case classes to write very similar types. For example, we can write a type for lists of integers as follows:

```
sealed trait IList
case class INil() extends IList
case class ICons(head: Int, tail: IList) extends IList
```

The following function adds n to every number in a list:

```
def add(n: Int, lst: IList): IList = lst match {
  case INil() => INil()
  case ICons(head, tail) => ICons(head + n, add(n, tail))
}
```

A test suite should test the two cases in the function. Moreover, we can also test simple properties of this function alone.

For example, for all `m`, `n`, and `lst`:

```
add(n, add(m, lst)) == add(m + n, lst)
```

The following function appends two lists together:

```
def append(lst1: IList, lst2: IList): IList = lst1 match {
  case INil() => lst2
  case ICons(head, tail) => ICons(head, append(tail, lst2))
}
```

Again, we should test the two cases of this function. But, there are several other properties that we can test. For example, if we append three lists together, the order in which we apply the function does not matter:

```
append(lst1, append(lst2, lst3)) == append(append(lst1, lst2), lst3)
```

i.e., `append` is *associative*.

Here is another property of `append`:

```
append(lst, INil()) == lst
```

Notice that this is *not* a case of `append`: the function pattern-matches on the first argument. This property shows that both cases of `append` behave in a similar way when the second argument is `INil()`.

Finally, we can write properties that relate the `add` and `append` functions:

```
append(add(n, lst1), add(n, lst2)) == add(n, append(lst1, lst2))
```