Lecture 3

Generics and Type Inference

Figure 5.1 shows two functions that calculate (1) the length of a list of integers and (2) the length of a list of strings. These two functions are almost identical. The only difference between them in the type of element in the list. We can abstract away the differences by writing a *generic* length function:

```
def length[A](alist: List[A]): Int = alist match {
  case Nil => 0
  case _ :: tail => 1 + length(tail)
}
```

In the code above, `A` is a *type argument*. It truly is an argument to the function, just like `alist`. i.e., when `length` is invoked, we need to specify `A`:

```
length[Int](List(1, 2, 3))
length[String](List("A", "B", "C", "D"))
```

However, `A` holds a type instead of a value.

Scala lets you elide type arguments in most cases. So, we could also write:

```
length(List(1, 2, 3))
length(List("A", "B", "C", "D"))
```

But, it is important to note that under the hood, Scala figures out that we meant `A` to be `Int` on the first line and `String` on the second line and inserts the type arguments for us. In fact, we relied on this in the definition of `length`: Scala expands the recursive call `length(tail)` to `length[A](tail)`. Note that Scala isn't just "guessing" the right type. It uses the type of the value-arguments to infer the missing type argument.

However, Scala's type inference is not perfect. It saves you typing, but don't expect it to figure out exactly what you mean. It can make mistakes. For example, if we write `List(1, "2", 3)`, Scala infers that it has type `List[Any]`. But, this is almost certainly not what we meant! We probably meant to construct a list of integers or strings. If we instead wrote `List[Int](1, "2", 3)`, Scala would have signalled a type error.

Generic Data Structures

Figure 5.4 In the last lecture, we build a data structure that represents binary search trees with strings for values. We can write a similar data structure with integer-values, as shown in fig. 5.4. But, this will force us to redefine the functions we wrote earlier to proceed `SBinTree`s. Instead, we can write a generic type for binary trees where the values have some unknown type `A`:

```
sealed trait BinTree[A]
case class Leaf[A]() extends BinTree[A]
case class Node[A](lhs: BinTree[A], key: Int, value: A, rhs: BinTree[A]) extends BinTree[A]
```

```
def ilength(alist : List[Int]): Int = {
  alist match {
  case Nil => 0
  case _ :: tail => 1 + ilength(tail)
  }
}
```

(a) Length of a list of integers.

```
def slength(alist : List[String]): Int = {
  alist match {
    case Nil => 0
    case _ :: tail => 1 + slength(tail)
  }b
}
```

(b) Length of a list of integers.

Figure 5.1: Two very similar length functions.

```
sealed trait IBinTree
case class ILeaf() extends IBinTree
case class INode(
  lhs: IBinTree,
  key: Int,
  value: Int,
  rhs: IBinTree)
  extends IBinTree
```

Figure 5.2: Values are integers.

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

Figure 5.3: Values are strings.

Figure 5.4: Two non-generic binary trees.

```
def incrs(alist: List[Int]): List[Int] =
  alist match {
    case Nil => Nil
    case h :: t => h + 1 :: incrs(t)
  }
```

(a) A function that increments all numbers in a list.

```
def lengths(alist: List[String]): List[Int] =
  alist match {
  case Nil => Nil
  case h :: t => h.length :: lengths(t)
}
```

(b) A function that calculates the lengths of all strings in a list.

```
def doubles(alist: List[Int]): List[Int] =
  alist match {
    case Nil => Nil
    case h :: t => h * 2 :: doubles(t)
}
```

(c) A function that doubles all numbers in a list.

```
def negates(alist: List[Int]): List[Int] =
  alist match {
    case Nil => Nil
    case h :: t => -h :: negates(t)
}
```

(d) A function that negates all numbers in a list.

Figure 5.5: Four different functions that transform elements of a list.

Here are some examples of binary trees with values of different types:

- This binary tree has integer values:

  ```
  Node(Leaf(), 10, 500, Leaf())
  ```

- This binary tree has string values:

  ```
  Node(Leaf(), 10, "five hundred", Leaf())
  ```

- This binary tree has boolean values:

  ```
  Node(Leaf(), 10, true, Leaf())
  ```

We can easily update the `insert`, `find`, and `size` functions from the last lecture to work with generic binary trees.

Higher-Order Functions on Lists

The Map Function

Study the four functions in fig. 5.5. The function `incrs` adds one to every element in a list of integers, the function `lengths` calculates the length of every string in a list, the function `doubles` doubles every number in a list, and the function `negates` negates every number in a list. Hopefully, you've noticed that these four functions have a lot of in common. *The only difference between them is the operation that they perform on the head of the list.*

Here is a variant of `doubles` that makes the operation explicit, by moving it into a separate function:

```
def f(n: Int): Int = n * 2

def doubles(alist: List[Int]): List[Int] = alist match {
  case Nil => Nil
  case h :: t => f(h) :: doubles(t)
}
```

In this version, we're simply applying a function f to h, where f is the doubling function. *You should apply the same refactoring to the other functions.* E.g., instead of directly writing `head.length`, refactor the function so that this expression is in a helper function.

Once we've re-written the operation as `f(h)`, all three functions look identical: the only difference is that each refers to a different function. Instead of writing three functions that are almost identical, we can write one function that takes f as an argument:

```
def map(f: Int => Int, alist: List[Int]): List[Int] = alist match {
  case Nil => Nil
  case h :: t => f(h) :: map(f, t)
}
```

With this function, we can make our examples much more succinct:

```
def doubles(alist: List[Int]) = {
  def f(n: Int): Int = n * 2
  map(f, alist)
}

def incrs(alist: List[Int]) = {
  def f(n: Int): Int = n + 1
  map(f, alist)
}

def lengths(alist: List[String]) = {
  def f(str: String): Int = str.length
  map(f, alist)
}
```

Unfortunately, the definition of `lengths` above does not type-check. Scala reports two type errors and they are both very informative:

```
<console>:15: error: type mismatch;
 found   : String => Int
 required: Int => Int
         map(f, alist)
             ^
<console>:15: error: type mismatch;
 found   : List[String]
 required: List[Int]
         map(f, alist)
                ^
```

The `map` function only works of lists of integers. However, if you look at the definition of `map` closely, you'll see that all the `Int`-specific code has been factored out into `f`. We can make the function even more generic by introducing two type-parameters:

```
def map[A,B](f: A => B, alist: List[A]): List[B] = alist match {
  case Nil => Nil
  case h :: t => f(h) :: map[A, B](f, t)
}
```

The type-parameter A is the type of the elements in *alist* and the type-parameter B is the type elements in the produced list. In the code above, we've make all the type-arguments explicit. But, as we've discussed before, we can rely on type-inference to fill them in:

```
def map[A,B](f: A => B, alist: List[A]): List[B] = alist match {
  case Nil => Nil
  case h :: t => f(h) :: map(f, t)
}
```

The Filter Function

Another common pattern when programming with lists is to select certain elements that have some property. For example, here is a function that consumes a list, and produces a new list that only contains the even numbers:

```
def filterEven(alist: List[Int]): List[Int] = alist match {
  case Nil => Nil
  case head :: tail =>
    (head % 2 == 0) match {
      case true => head :: filterEven(tail)
      case false => filterEven(tail)
    }
}
```

This is a very common pattern too. For example, we could write a function to select the odd numbers or the prime numbers. If we had a list of strings, we could select all the strings with length 5 or all the strings that represent English-language words. All these functions have the same shape: they test the value of `head` in some way. If the test succeeds, `head` is added in the output list. But, if the test fails, it is excluded.

Following the same strategy we used to derive `map`, we first package the `head`-test into a function:

```
def f(n: Int): Boolean = head % 2

def filterEven(alist: List[Int]): List[Int] = alist match {
  case Nil => Nil
  case head :: tail =>
    f(head) match {
      case true => head :: filterEven(tail)
      case false => filterEven(tail)
    }
}
```

Now that the pattern is clearer, we generalize `filterEven` to take `f` as an argument:

```
def filter(f: Int => Boolean, alist: List[Int]): List[Int] = alist match {
  case Nil => Nil
  case head :: tail =>
    f(head) match {
      case true => head :: filter(f, tail)
      case false => filter(f, tail)
    }
}

def filterEven(alist: List[Int]): List[Int] = {
  def f(n: Int): Boolean = head % 2
  filter(f, alist)
}
```

Finally, just as we did for `map`, we can generalize the type of `filter` so that it can be applied to `List[A]`:

```
def filter[A](f: A => Boolean, alist: List[A]): List[A] = alist match {
  case Nil => Nil
  case head :: tail =>
    f(head) match {
      case true => head :: filter(f, tail)
      case false => filter(f, tail)
    }
}
```

Sorting

You've written sorting functions that sort lists is ascending or descending order. But, the following higher-order function can be used to sort any lists of any type, given an arbitrary "less than" operator.

```
def insert[A](lessThan: (A, A) => Boolean, x: A, alist: List[A]): List[A] = alist match {
  case Nil => List(x)
  case hd :: tl => {
    if (lessThan(x, hd)) {
      x :: hd :: tl
    }
    else {
      hd :: insert(lessThan, x tl)
    }
  }
}

def sort[A](lessThan: (A, A) => Boolean, alist: List[A]) = alist match {
  case Nil => Nil
  case hd :: tl => insert(lessThan, hd, sort(lessThan, tl))
}
```

```
def sortAscending(alist: List[Int]): Int = {
  def f(x: Int, y: Int): Boolean = x < y
  sort(f, alist)
}

def sortDescending(alist: List[Int]): Int = {
  def f(x: Int, y: Int): Boolean = x > y
  sort(f, alist)
}

def sortByLengthAscending(alist: List[String]): Int = {
  def f(x: String, y: String): Boolean = x.length < y.length
  sort(f, alist)
}
```

Builtin Methods on Lists

All the higher-order list-processing functions that we've defined are already built-in to Scala as methods on lists.

- We can map over a list using the `map` method:

  ```
  def f(n: Int): Int = n + 1

  assert(List(10, 20, 30).map(f) == List(11, 21, 22))
  ```

- We can filter a list using the `filter` method:

  ```
  def f(n: Int): Boolean = n % 2 == 0

  assert(List(0, 1, 2, 3, 4, 5).map(f) == List(0, 2, 4))
  ```

- We can sort a list using the `sortWith` method:

  ```
  def f(x: Int, y: Int): Boolean = x > Y

  assert(List(3, 7, 9, 11).sortWith(f) == List(11, 9, 7, 3))
  ```

  There are several other higher-order functions that you can discover from the documentation or the Scala console.