INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

Exercises 03: Algebraic Data Types and Functional Data Structures

Exercise 1:

Given the ADT for lists that we have written in class, implement the following functions:

- `concat` takes a list of lists containing objects of type A as input and returns one list of objects of type A.
- `map` takes as input a list of objects of type A and a function f that takes as input objects of type A and returns objects of type B. This function should return a list containing objects of type B.
- `fmap` takes as input a list of objects of type A and a function f that takes as input objects of type A and **returns a list of objects of type B**. This function should return a list containing objects of type B.
- **dropWhile** takes as input a list of As and a function taking an A and returning a Boolean value. This function returns a list that "removes" elements from the beginning from left to right as long as the elements satisfy the condition. You can implement this function using **pattern guards**. We have seen an example of pattern guards in Chapter 3. You can also look up Scala's documentation to learn more about pattern guards.

Try implementing these functions with pattern matching. You may also avail of `foldRight` to implement (some of) these functions.

```
enum List[+A]:
  case Nil
  case Cons(head: A, tail: List[A])

object List:
    def foldRight[A,B](l: List[A], z: B, f: (A,B) => B): B = l match
        case Nil => z
        case Cons(h, t) => f(h, foldRight(t, z, f))

    def append[A](l1: List[A], l2: List[A]): List[A] = l1 match
        case Nil => l2
        case Cons(h, t) => Cons(h, append(t, l2))

    def apply[A](l: A*): List[A] =
        if(l.isEmpty) Nil
        else Cons(l.head, apply(l.tail: _*))

    def concat[A](l: List[List[A]]): List[A] = ???
    def map[A,B](l: List[A], f: A => B): List[B] = ???
    def fmap[A,B](l: List[A], f: A => List[B]): List[B] = ???
    def dropWhile[A](l: List[A], f: A => Boolean): List[A] = ???
```

Attention!

In the book, the authors ask you to write your version of `flatMap`. The result computed by `flatMap` in the book and the one described in the Scala standard library are different. This is why we will define a `fmap` in the exercises. Please consult that standard library's documentation to learn more about Scala's `flatMap` as it is a useful function. Scala's `flatMap` basically applies `map` and then `flatten`.

Solution 1:

```
    def concat[A](l: List[List[A]]): List[A] = l match
        case Nil => Nil
        case Cons(h,t) => append(h, concat(t))

    def map[A,B](l: List[A], f: A => B): List[B] = l match
        case Nil => Nil
        case Cons(h,t) => Cons(f(h), map(t, f))

    def fmap[A,B](l: List[A], f: A => List[B]): List[B] = l match
        case Nil => Nil
        case Cons(h,t) => append(f(h), fmap(t, f))

    def dropWhile[A](l: List[A], f: A => Boolean): List[A] = l match
        case Cons(h, t) if f(h) => dropWhile(t, f)
        case _ => l
```

Exercise 2:

Given `val x = List(1,2,3,4)`, demonstrate the use of these functions. Use `map` and `fmap` to add three to each element of that list. Use `dropwhile` to remove all elements that are lower or equal to 3. Use `concat` to concatenate the list with itself, resulting in the list `List(1,2,3,4,1,2,3,4)`.

Solution 2:

```
scala> val x = List(1,2,3,4)
val x: List[Int] = Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
scala> import List._
scala> map(x, _ + 3)
val res0: List[Int] = Cons(4,Cons(5,Cons(6,Cons(7,Nil))))
scala> fmap(x, x => List(x + 3))
val res1: List[Int] = Cons(4,Cons(5,Cons(6,Cons(7,Nil))))
scala> dropWhile(x, _ <= 3)
val res2: List[Int] = Cons(4,Nil)
scala> concat(List(x, x))
val res3: List[Int] = Cons(1,Cons(2,Cons(3,Cons(4,Cons(1,Cons(2,Cons(3,Cons(4,Nil)))))))
```

Exercise 3:

The functions `concat`, `map`, `append`, and `fmap` can be easily defined in terms of `foldRight`. If you haven't used `foldRight` to write these functions, then now is the time. Create the functions `concat2`, `map2`, `append2`, and `fmap2` that rely on `foldRight`. Demonstrate that they yield the same results.

Solution 3:

```
    def concat2[A](l: List[List[A]]): List[A] =
        foldRight(l, Nil: List[A], append)

    def map2[A,B](l: List[A], f: A => B): List[B] =
        foldRight(l, Nil: List[B], (a, acc) => Cons(f(a), acc))

    def fmap2[A,B](l: List[A], f: A => List[B]): List[B] =
        foldRight(l, Nil: List[B], (a, acc) => append(f(a), acc))

    def append2[A](l1: List[A], l2: List[A]): List[A] =
        foldRight(l1, l2, (l, acc) => Cons(l, acc))
```

```
scala> map2(x, _ + 3) == map(x, _ + 3)
val res0: Boolean = true
scala> fmap2(x, x => List(x + 3)) == fmap2(x, x => List(x + 3))
val res1: Boolean = true
scala> concat2(List(x,x)) == concat(List(x,x))
val res2: Boolean = true
scala> append2(x,x) == append(x,x)
val res3: Boolean = true
```

Exercise 4:

Given the following ADT for binary trees:

```
enum Tree[+A]:
    case Leaf(value: A)
    case Branch(left: Tree[A], right: Tree[A])

    def size: Int = ???
    def depth: Int = ???
    def map[B](f: A => B): Tree[B] = ???
```

Create the following functions:

1. `size` returns the size of a tree. Both branches and leaves count as one. We have seen this function in class, but try not to look at the slides unless you are stuck.
2. `depth` returns the depth of a tree. Objects of the type Int (and Doubles, ...) have a method `max` that returns the max of two numbers. For instance, `1.max(5)` yields `5`.
3. `map` takes as input a function taking as input As and returning Bs. The result is a tree with the same structure in which all As are transformed into Bs.

Solution 4:

```
    def size: Int = this match
        case Leaf(_) => 1
        case Branch(l, r) => 1 + l.size + r.size

    def depth: Int = this match
        case Leaf(_) => 0
        case Branch(l, r) => 1 + (l.depth.max(r.depth))

    def map[B](f: A => B): Tree[B] = this match
        case Leaf(a) => Leaf(f(a))
        case Branch(l, r) => Branch(l.map(f), r.map(f))
```

Exercise 5:

Extend the ADT for binaries you have created with the function `fold`. `fold` is a function that takes, for binary trees, two functions. The first function is applied to leaves and returns objects of type B. The second is a function that is applied to the left and right of a branch. Redefine `map`, `depth`, and `size` using `fold`.

```
    // I Have not provided the full signature of this function.
    // Try to figure out what the signatures of these functions will be.
    // Can you also explain why?
    ///def fold[B](f: ??? => ???, g: ??? => ???): B = ???
    def size2: Int = ???
    def depth2: Int = ???
    def map2[B](f: A => B): Tree[B] = ???
```

Solution 5:

```
    def fold[B](f: A => B, g: (B,B) => B): B = this match
        case Leaf(a) => f(a)
        case Branch(l, r) => g(l.fold(f, g), r.fold(f, g))

    def size2: Int =
        fold(a => 1, 1 + _ + _)

    def depth2: Int =
        fold(a => 0, (d1, d2) => 1 + (d1.max(d2)))

    def map2[B](f: A => B): Tree[B] =
        fold(a => Leaf(f(a)), Branch(_, _))
```

Exercise 6:

> Attention!
>
> This exercise is hard-ish and good to learn more about Scala!

Rewrite your list ADT so that you can use OOP-style message passing. In other words, you can apply map by using `x.map(_ + 3)` instead of `map(x, _ + 3)`. Keep `apply` and `append` as they are defined in your companion object. You will also keep `concat` in your companion object, but you will have to rewrite it. Demonstrate its use. Be careful: you will have to import functions declared in your companion object in your ADT's scope.

Solution 6:

The key here is to:

- Ensure we import the necessary functions in our ADT.
- Remove the type parameter that has been declared at the ADT level (`A` in this case).
- Remove one of the parameters and replace it with the `this` keyword.

```
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

    import List._

    def foldRight[B](z: B, f: (A,B) => B): B = this match
        case Nil => z
        case Cons(h, t) => f(h, t.foldRight(z, f))

    def map[B](f: A => B): List[B] = this match
```

```
        case Nil => Nil
        case Cons(h,t) => Cons(f(h), t.map(f))

    def fmap[B](f: A => List[B]): List[B] = this match
        case Nil => Nil
        case Cons(h,t) => append(f(h), t.fmap(f))

    def dropWhile(f: A => Boolean): List[A] = this match
        case Cons(h, t) if f(h) => t.dropWhile(f)
        case _ => this

object List:
    def apply[A](l: A*): List[A] =
        if(l.isEmpty) Nil
        else Cons(l.head, apply(l.tail: _*))

    def append[A](l1: List[A], l2: List[A]): List[A] = l1 match
        case Nil => l2
        case Cons(h, t) => Cons(h, append(t, l2))

    def concat[A](l: List[List[A]]): List[A] =
        l.foldRight(Nil: List[A], append)
```

Since our ADT relies on functions declared in our companion object, we do not have to import the functions of List. However, we still have to import functions declared inside that companion object if we want to use `append` and `concat` in our program.

```
scala> val x = List(1,2,3,4)
val x: List[Int] = Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
scala> val a = x.map(_ + 3)
val a: List[Int] = Cons(4,Cons(5,Cons(6,Cons(7,Nil))))
scala> val b = x.fmap(x => List(x + 3))
val b: List[Int] = Cons(4,Cons(5,Cons(6,Cons(7,Nil))))
scala> val c = x.dropWhile(_ <= 3)
val c: List[Int] = Cons(4,Nil)
```

Exercise 7:

> Attention!
>
> This exercise is really hard! You are warned. :-)

We have rewritten our ADT list so that is uses OOP-style message passing. You may have noticed that we have not "promoted" the function `append`. In this exercise, you will try to move that function in the ADT. In other words, the following should be possible:

```
val x = List(1,2,3,4)
val d = x.append(x)
```

With a "naive" implementation of A, you will likely face an error like "covariant type A occurs in contravariant position in type A." The problem is that our List is covariant in A. That means that if all C ARE A, then all List[C] are List[A]. The following lines of code are valid:

```
scala>val x: List[Int] = List(1,2,3,4)
val x: List[Int] = Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
```

```
scala>val y: List[Any] = x
val y: List[Any] = Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
```

But, Scala cannot compile your function as it can anticipate the following problem:

```
scala> y.append(List("Foo", "Bar"))
```

Why? Not only are the values of x and y equal, but so are their references:

```
scala> x == y
val res0: Boolean = true

scala> x eq y
val res1: Boolean = true
```

So from this, it should be clear that we cannot add String objects to a list of Int objects. There is a solution, however. We need to inform Scala that our append object may take as input a List of objects of type B, but (!) with a restriction: `[B >: A]`. This is a **lower type bound**[1][2] and means that B is constrained to be a supertype of A. If we append a List of String objects to a List of Int objects, **Scala is informed to look for a common supertype**.

```
scala>val d = x.append(x)
val d: List[Int] = Cons(1,Cons(2,Cons(3,Cons(4,Cons(1,Cons(2,Cons(3,Cons(4,Nil))))))))

scala>val e = x.append(List("foo", "bar"))
val e: List[Matchable] = Cons(1,Cons(2,Cons(3,Cons(4,Cons(foo,Cons(bar,Nil))))))
```

You will also need to rewrite the other functions that rely on append, obviously. Good luck!

Solution 7:

```
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

    // We don't need this anymore
    // import List._

    def foldRight[B](z: B, f: (A,B) => B): B = this match
        case Nil => z
        case Cons(h, t) => f(h, t.foldRight(z, f))

    def map[B](f: A => B): List[B] = this match
        case Nil => Nil
        case Cons(h,t) => Cons(f(h), t.map(f))

    // This function had to be redefined
    def fmap[B](f: A => List[B]): List[B] = this match
        case Nil => Nil
        case Cons(h,t) => f(h).append(t.fmap(f))

    def dropWhile(f: A => Boolean): List[A] = this match
        case Cons(h, t) if f(h) => t.dropWhile(f)
        case _ => this
```

[1] https://docs.scala-lang.org/tour/lower-type-bounds.html
[2] https://www.geeksforgeeks.org/scala-lower-bound/

```
    // Our append!
    def append[B >: A](l2: List[B]): List[B] = this match
        case Nil => l2
        case Cons(h, t) => Cons(h, t.append(l2))

object List:
    def apply[A](l: A*): List[A] =
        if(l.isEmpty) Nil
        else Cons(l.head, apply(l.tail: _*))

    // def append[A](l1: List[A], l2: List[A]): List[A] = l1 match
    //     case Nil => l2
    //     case Cons(h, t) => Cons(h, append(t, l2))

    // This function had to be redefined
    def concat[A](l: List[List[A]]): List[A] =
        // We could have also written
        // l.foldRight(Nil: List[A], _.append(_))
        l.foldRight(Nil: List[A], (l, acc) => l.append(acc))
```

References

[1] Paul Chiusano and Rnar Bjarnason. 2015. Functional Programming in Scala (2nd. ed.). Manning Publications Co., USA.