Functional Programming

Midterm Solution

Friday, November 6 2015

Exercise 1: List functions (10 points)

You are asked to implement the following List functions using only the specified List API methods.

Please refer to the appendix on the last page as a reminder for the behavior of the given List API methods.

(a) Implement `scanLeft` using only `foldLeft` and `::` (cons).

```scala
def reverse[A](xs: List[A]) = xs.foldLeft(List[A]())((r,c) => c :: r)

def scanLeft[A, B >: A](xs: List[A])(z: B)(op: (B, B) => B): List[B] =
    reverse(xs.foldLeft((z, List(z))){
        case ((acc, accList), curr) =>
            val res = op(acc,curr)
            (res, res :: accList)
    }._2)

//Here is a sample usage:
def add(acc: String, elem: String) = {
    println("Called \"add\" with acc=" + acc + " and elem=" + elem + ".")
    acc + elem
}

val xs = List("A", "B", "C")

scanLeft(xs)("z")(add)

//It produces the following output:
//Called "add" with acc=z and elem=A.
//Called "add" with acc=zA and elem=B.
//Called "add" with acc=zAB and elem=C.
//res0: List[String] = List(z, zA, zAB, zABC)
```

(b) Implement `flatMap` using only `foldRight` and `::` (cons).

```scala
def flatMap[A,B](xs: List[A])(f: A => List[B]): List[B] =
    xs.foldRight(List[B]())( (outCurr, outAcc) =>
        f(outCurr).foldRight(outAcc)( (inCurr, inAcc) => inCurr :: inAcc ) )

//Here is a sample usage:
val fruits = List("apple", "banana", "orange")

flatMap(fruits)(_.toUpperCase.toList)

//It produces the following output:
//res0: List[Char] = List(A, P, P, L, E, B, A, N, A, N, A, O, R, A, N, G, E)
```

Exercise 2: Subtyping (10 points)

Given the following hierarchy of classes:

```
trait Producer[+A]
trait Consumer[-A]
trait Factory[+A, -B] extends Producer[A] with Consumer[B]
```

Recall that `+` means covariance and `-` means contravariance.

Consider also the following typing relationships for `W`, `V`, `X` and `Y`:

- `V <: W`
- `Y <: X`

Fill in the subtyping relation between the types below using symbols:

- `<:` in case `T1` is a subtype of `T2`;
- `>:` in case `T1` is a supertype of `T2`;
- $\times$ in case `T1` is neither a supertype nor a supertype of `T2`.

Solution

```
Producer[X]                    >:  Producer[Y]
Producer[Consumer[X]]          <:  Producer[Consumer[Y]]
Factory[Producer[X], X]        ><  Factory[Factory[Y, Y], Y]
Factory[Y, Y] => Producer[V]   <:  Factory[Y, X] => Producer[W]
List[Factory[Y, Y]]            ><  List[Consumer[X]]
```

Exercise 3: Structural Induction (10 points)

Question recap

We want to prove that:

```
list.foldLeft(z)(add) === z + sum(list)
```

Using the following axioms:

```
(1)  sum(Nil) === 0
(2)  sum(x :: xs) === x + sum(xs)
(3)  Nil.foldLeft(z)(f) === z
(4)  (x :: xs).foldLeft(z)(f) === xs.foldLeft(f(z, x))(f)
(5)  add(a, b) === a + b
(6)  a + b === b + a
(7)  (a + b) + c === a + (b + c)
(8)  a + 0 === a
```

Proof

We prove the above lemma by structural induction over `list`.

Case `list` is `Nil`

We want to prove that

```
Nil.foldLeft(z)(add) === z + sum(Nil)
```

This case is a base case. There is no induction hypothesis. The proof is:

```
 Nil.foldLeft(z)(add) =?= z + sum(Nil)
--------------------
     || (3)
     z                =?= z + sum(Nil)
                              --------
                                || (1)
     z                =?= z + 0
                          -----
                           || (8)
     z                =?=  z
```

which concludes the proof.

Case `list` is `x :: xs`

This case is not a base case. Our induction hypothesis is that the lemma holds for `list === xs` (which is a constituent of `x :: xs`, and is therefore smaller, making this induction well-founded).

Assuming that (induction hypothesis):

```
xs.foldLeft(z)(add) === z + sum(xs)
```

we want to prove that:

```
(x :: xs).foldLeft(z)(add) === z + sum(x :: xs)
```

The proof is:

```
 (x :: xs).foldLeft(z)(add)  =?= z + sum(x :: xs)
--------------------------
    || (4)
xs.foldLeft(add(z, x))(add) =?= z + sum(x :: xs)
            ---------
              || (5)
xs.foldLeft(z + x)(add)     =?= z + sum(x :: xs)
-----------------------
     || (induction hypothesis)
  (z + x) + sum(xs)         =?= z + sum(x :: xs)
                                    ------------
                                      || (2)
  (z + x) + sum(xs)         =?= z + (x + sum(xs))
                                ----------------
                                      || (7) (right-to-left)
  (z + x) + sum(xs)         =?= (z + x) + sum(xs)
```

which concludes the proof.

Exercise 4: Graph Reachability (10 points)

1. You are asked to compute the set of all nodes reachable in **exactly** `n` steps from a set of initial nodes.

```
def reachable(n: Int, init: Set[Node], edges: List[Edge]): Set[Node] = n match {
  case 0 => init
  case _ =>
    val next = init.flatMap(node => edges.filter(_.from == node).map(_.to))
    reachable(n - 1, next, edges)
}
```

2. Compute all cycles of size 3 using the above function.

```
def cycles3(nodes: Set[Node], edges: List[Edge]): Set[Node] =
  nodes.filter(node => reachable(3, Set(node), edges).contains(node))
```