Functional Programming

Midterm Solution

Friday, November 6 2015

Exercise 1: Sliding Differences (5 points)

Part 1: Calculating the list of differences

There are many possible solutions to the problem. Here are some (arguably elegant) of them:

```
def differences(ls: List[Int]): List[Int] =
  ((0 :: ls) zip ls) map { case (l, r) => r - l }

def differences2(ls: List[Int]): List[Int] = {
  @tailrec
  def loop(xs: List[Int], acc: List[Int]): List[Int] = xs match {
    case x :: Nil => acc.reverse
    case x :: y :: ys => loop(y :: ys, (y - x) :: acc)
  }
  loop(0 :: ls, Nil)
}
```

Part 2: Rebuilding the original list

There are many possible solutions to the problem. Here are some (arguably elegant) of them:

```
def rebuildList(ls: List[Int]): List[Int] = ls match {
  case Nil => Nil
  case x :: Nil => ls
  case x :: y :: ys => x :: rebuildList((x + y) :: ys)
}

def rebuildList2(ls: List[Int]): List[Int] = ls match {
  case Nil => Nil
  case x :: xs =>
    (xs.foldLeft(x :: Nil) {
      case (y :: acc, elem) => (elem + y) :: y :: acc
    }).reverse
}

def rebuildList3(ls: List[Int]): List[Int] =
  xs.scan(x)((l, elem) => l + elem)
```

Exercise 2: Subtyping (5 points)

We first state the following subtyping relation for functions:

`A1 => B1 <: A2 => B2` iff `A1 >: A2` and `B1 <: B2`

1. `Map[V, Y]` **is not related to** `Map[W, X]`.
   - Non related as `Map` is invariant in first type argument and `V =!= W`.
2. `Iterable[Pair[V, Y]] => V <: Map[V, Y] => V`.
   - `Iterable[Pair[V, Y]] >: Map[V, Y]` from definition of `Map`.
   - `V <: V`.
3. `Map[Map[V, Y], Y]` **is not related to** `Map[Iterable[Pair[V, X]], X]`.
   - Similar to 3.1, `Map` is invariant in first type argument and `Map[V, Y] =!= Iterable[Pair[V, X]]`.
4. `(Y => Y) => V <: (X => Y) => W`.
   - As `Y <: X` and `Y >: X`, we have `(Y => Y) >: (X => Y)`.
   - As `V <: W` and `(Y => Y) >: (X => Y)`, we have `(Y => Y) => V <: (X => Y) => W`
5. `W => (Y => X)` **is not related to** `W => (V => Y)`.
   - `Y => X` and `V => Y` are non related as `Y` and `V` are non related.

Exercise 3: Binary Search Tree (5 points)

Question 1:

```
def computeMinMax(b: Node): (Int, Int) = b match {
  case Node(Leaf, k, Leaf) => (k, k)
  case Node(left: Node, k, Leaf)  => computeMinMax(left) match {
    case (min, max) => (Math.min(min, k), Math.max(k, max))
  }
  case Node(Leaf, k, right: Node) => computeMinMax(right) match {
    case (min, max) => (Math.min(min, k), Math.max(k, max))
  }
  case Node(left: Node, k, right: Node) =>
    (computeMinMax(left), computeMinMax(right)) match {
      case ((min1, max1),(min2, max2)) =>
        (Math.min(Math.min(min1, k), min2), Math.max(Math.max(max1, k), max2))
    }
}
```

Or

```
def computeMinMax(b: Node): (Int, Int) = {
  def rec(b: Node, accMin: Int, accMax: Int): (Int, Int) = b match {
    case Leaf = (accMin, accMax)
    case Node(left, k, right) =>
```

```
      val (leftMin, leftMax) = rec(left, accMin, accMax)
      val (rightMin, rightMax) = rec(right, leftMin, leftMax)
      (Math.min(rightMin, k), Math.min(rightMax, k))
  }
  rec(b, b.value, b.value)
}
```

Questions 2 and 3:

There are three main approaches to solving this problem:

1) Using an inner function returning the min/max

```
def isBinarySearchTree(b: Tree): Boolean = {
  def rec(b: Node): (Boolean, Int, Int) = b match {
    case Node(Leaf,       k, Leaf)        =>
      (true, k, k)
    case Node(left: Node, k, Leaf)        =>
      rec(left)  match { case (b,min,max) => (b && max < k, min, k) }
    case Node(Leaf,       k, right: Node) =>
      rec(right) match { case (b,min,max) => (b && min > k, k, max) }
    case Node(left: Node, k, right: Node) =>
      (rec(left), rec(right))  match {
        case ((b1, min1, max1), (b2, min2, max2)) =>
          (b1 && b2 && max1 < k && k < min2, min1, max2)
      }
  }

  b match {
    case Leaf => true
    case b: Node => rec(b)._1
  }
}
```

Or,

```
def isBinarySearchTree(tree: Tree): Boolean = {
  def rec(tree: Tree, minAllowed: Int, maxAllowed: Int): Boolean = tree match {
    case Leaf => true
    case Node(left, k, right) =>
      k >= minAllowed && k <= maxAllowed &&
      rec(left, minAllowed, k - 1) && rec(right, k + 1, maxAllowed)
  }
  rec(tree, Int.MinValue, Int.MaxValue)
}
```

2) Using direct functions (can be solved incrementally)

```
def isBinarySearchTree(b: Tree): Boolean = {
  def isBoundedStart(t: Tree): Boolean = t match {
```

```
      case Leaft => true
      case Node(left, k, right) =>
        isBoundedRight(left, k) && isBoundedLeft(k, right)
    }

    def isBoundedLeft(elem: Int, t: Tree): Boolean = t match {
      case Leaf => true
      case Node(left, k, right) =>
        k > elem && isBounded(elem, left, t) && isBoundedLeft(k, right)
    }

    def isBoundedRight(t: Tree, elem: Int): Boolean = t match {
      case Leaf => true
      case Node(left, k, right) =>
        k < elem && isBounded(k, right, elem) && isBoundedRight(left, k)
    }

    def isBounded(min: Int, t: Tree, max: Int): Boolean = t match {
      case Leaf => true
      case Node(left, k, right) =>
        k > min && k < max && isBounded(min, left, k) && isBounded(k, right, max)
    }

    isBoundedStart(b)
  }
```

3) Using min and max (cumbersome)

```
  def minTree(b: Node): Int = b match {
    case Node(Leaf, elem, _) => elem
    case Node(left, elem, _) => leftMostTree(left)
  }
  def maxTree(b: Node): Int = b match {
    case Node(_, elem, Leaf) => b
    case Node(_, elem, right) => rightMostTree(left)
  }

  def isBinarySearchTree(b: Tree): Boolean = b match {
    case Leaf => true
    case b: Node =>
      val min = minTree(b) - 1
      val max = maxTree(b) + 1

      def isWithinBounds(t: Tree, min: Elem, max: Elem): Boolean = t match {
        case Leaf => true
        case Node(left, elem, right) =>
          isWithinBounds(left, min, elem) &&
          min < elem && elem < max &&
          isWithinBounds(right, elem, max)
      }

      isWithinBounds(b, min, max)
```

```
}
```

Exercise 4: Structural Induction (10 points)

In addition to axioms 1 to 4, we can refer to the following formulas from the definitions of `S` and `Bst`:

```
(5) S(Leaf) = {}
(6) S(Node(l, e, r)) = S(l) U S(r) U {e}

(7) Bst(Leaf) = {}
(8) Bst(Node(l, e, r)) =
      Bst(l) &&
      Bst(r) &&
      (for all k in S(l) : k < e) &&
      (for all k in S(r) : k > e)
```

Part 1

We have to prove that:

```
For all t: Tree such that Bst(t) and all v: Int:
  S(add(t, v)) === S(t) U {v}
```

We do this by structural induction on the shape of `t`. It turns out that the part of the precondition `Bst(t)` is irrelevant for this proof.

Base case, i.e., Leaf

We have to prove that:

```
S(add(Leaf, v)) === S(Leaf) U {v}
```

From the left-hand-side:

```
S(add(Leaf, v))
  (1)
= S(Node(Leaf, v, Leaf))
  (6)
= S(Leaf) U S(Leaf) U {v}
  (5)
= {} U S(Leaf) U {v}
  (union with {})
= S(Leaf) U {v}
```

which is the right-hand side.

Inductive case, i.e., Node

Induction hypothesis: the property holds for two trees `l` and `r`, i.e., `S(add(l, v)) = S(l) U {v}` and `S(add(r, v)) = S(r) U {v}` for any `v: Int`.

Assuming the IH is true, we have to prove that the property holds for a bigger node `Node(l, e, r)`, i.e., that:

```
S(add(Node(l, e, r), v)) = S(Node(l, e, r)) U {v}
```

We decompose this in three cases, `v == e`, `v < e` and `v > e`.

`v == e` From the left-hand-side:

```
S(add(Node(l, e, r), v))
  (2)
= S(Node(l, v, r))
  (6)
= S(l) U S(r) U {v}
```

From the right-hand-side:

```
S(Node(l, e, r)) U {v}
  (6)
= S(l) U S(r) U {e} U {v}
  v == e
= S(l) U S(r) U {v} U {v}
  union of sets, {v} U {v} === {v}
= S(l) U S(r) U {v}
```

`v < e` From the left-hand-side:

```
S(add(Node(l, e, r), v))
  (3)
= S(Node(add(l, v), e, r))
  (6)
= S(add(l, v)) U S(r) U {e}
  by induction hypothesis (on l)
= S(l) U {v} U S(r) U {e}
```

From the right-hand-side:

```
S(Node(l, e, r)) U {v}
  (6)
= S(l) U S(r) U {e} U {v}
  associativity of union
= S(l) U {v} U S(r) U {e}
```

`v > e` Not asked.

Part 2

We have to prove that:

```
For all t: Tree such that Bst(t) and all v: Int:
  Bst(add(t, v))
```

We do this by structural induction on the shape of `t`.

Base case, i.e., Leaf

Note that we have to prove this case because `Bst(Leaf)` is true (by `(7)`). If `Bst(Leaf)` had been false, we should have skipped this part as trivially satisfied.

We have to prove that:

```
Bst(add(Leaf, v)) === true
```

From the left-hand-side:

```
Bst(add(Leaf, v))
  (1)
= Bst(Node(Leaf, v, Leaf))
  (8)
= Bst(Leaf) && Bst(Leaf) &&
       (for all k in S(l) : k < e) &&
       (for all k in S(r) : k > e)
  (7) applied twice
= true && true &&
       (for all k in S(l) : k < e) &&
       (for all k in S(r) : k > e)
  (5) applied twice
= true && true &&
       (for all k in {} : k < e) &&
       (for all k in {} : k > e)
  because (for all k in {} : P) is always satisfied regardless of P
= true && true && true && true
  conjunction
= true
```

Inductive case, i.e., Node

Induction hypothesis: the property holds for two trees `l` and `r` *such that* ***`Bst(l)`*** *and* ***`Bst(r)`*** *are true*, i.e., `Bst(add(l, v))` and `Bst(add(r, v))` for any `v: Int`.

Assuming the IH is true, we have to prove that the property holds for a bigger node `Node(l, e, r)`, i.e., that:

```
Bst(add(Node(l, e, r), v)) === true
```

From the precondition that `Bst(t)` is true, we know that

```
(9) Bst(Node(l, e, r))
```

and we derive the following lemmas from `(9)` and `(8)`:

```
(10) Bst(l)
(11) Bst(r)
(12) (for all k in S(l) : k < e)
(13) (for all k in S(r) : k > e)
```

We decompose this in three cases, `v == e`, `v < e` and `v > e`.

`v == e`

```
Bst(add(Node(l, e, r), v))
  (2)
= Bst(Node(l, v, r))
  v == e
= Bst(Node(l, e, r))
  (9)
= true
```

`v < e`

```
Bst(add(Node(l, e, r), v))
  (3)
= Bst(Node(add(l, v), e, r))
  (8)
= Bst(l, v) && Bst(r) &&
       (for all k in S(add(l, v)) : k < e) &&
       (for all k in S(r) : k > e)
  by (10), we know that Bst(l), and therefore by induction hypothesis
= true && Bst(r) &&
       (for all k in S(add(l, v)) : k < e) &&
       (for all k in S(r) : k > e)
  by (11)
= true && true &&
       (for all k in S(add(l, v)) : k < e) &&
       (for all k in S(r) : k > e)
  by the theorem proved in Part 1, S(add(l, v)) === S(l) U {v}
= true && true &&
       (for all k in S(l) U {v} : k < e) &&
       (for all k in S(r) : k > e)
  decomposing 'for all' for S(l) and {v}
= true && true &&
       (for all k in S(l) : k < e) &&
       (for all k in {v} : k < e) &&
       (for all k in S(r) : k > e)
```

```
  (12)
= true && true &&
       true &&
       (for all k in {v} : k < e) &&
       (for all k in S(r) : k > e)
  v < e
= true && true &&
       true &&
       true &&
       (for all k in S(r) : k > e)
  (13)
= true && true &&
       true &&
       true &&
       true
  conjunction
= true
```

`v > e` Not asked.