Programming Principles

Midterm Solution

Friday, November 7 2014

Exercise 1: Merging sorted lists (5 points)

Part 1: Starting recursive

```
def merge[T](as: List[T], bs: List[T])(cmp: (T, T) => Boolean): List[T] = (as, bs) match {
  case (Nil, _) => bs
  case (_, Nil) => as
  case (x :: xs, y :: ys) =>
    if (cmp(x, y)) x :: merge(xs, bs)(cmp)
    else y :: merge(as, ys)(cmp)
}
```

Part 2: Going tail-recursive

```
def merge2[T](as: List[T], bs: List[T])(cmp: (T, T) => Boolean): List[T] = {

  @tailrec
  def loop(tmpAs: List[T], tmpBs: List[T], tmpRes: List[T]): List[T] = (tmpAs, tmpBs) match {
    case (Nil, _) => tmpRes.reverse ++ tmpBs
    case (_, Nil) => tmpRes.reverse ++ tmpAs
    case (x :: xs, y :: ys) =>
      if (cmp(x, y)) loop(xs, tmpBs, x :: tmpRes)
      else loop(tmpAs, ys, y :: tmpRes)
  }
  loop(as, bs, Nil)
}
```

Exercise 2: Streams (5 points)

Part 1

There are a few possible solutions. Here are 2. One of them uses the function from part 2.

```
def iterate[T](x: T)(f: T => T): Stream[T] =
   x #:: iterate(f(x))(f)

def iterate[T](x: T)(f: T => T): Stream[T] =
   iterated(f) map (g => g(x))
```

Part 2

There are a few possible solutions. Here are 2. One of them uses the function from part 1.

```
def iterated[T](f: T => T): Stream[T => T] =
  ((x: T) => x) #:: (iterated(f) map (_ andThen f))

def iterated[T](f: T => T): Stream[T => T] =
  iterate((x: T) => x)(_ andThen f)
```

Exercise 3: Equational Proof (5 points)

Part 1: axioms for `indexWhereAcc`

This follows straightforwardly from the implementation:

1. `indexWhereAcc(Nil, f, n) === n`
2. `indexWhereAcc(x :: xr, f, n) === n` if `f(x)` is true
3. `indexWhereAcc(x :: xr, f, n) === indexWhereAcc(xr, f, n+1)` if `f(x)` is false

Part 2: Proof of the lemma

We want to prove:

```
indexWhereAcc(xs, f, n) === indexWhereAcc(xs, f, 0) + n
```

We do it by structural induction on `xs`.

`Nil` case:

```
indexWhereAcc(Nil, f, n) =?= indexWhereAcc(Nil, f, 0) + n
     || (1)                          || (1)
     n                  =?=       0 + n
```

`x :: xr` case with `f(x)` is true:

```
indexWhereAcc(x :: xr, f, n) =?= indexWhereAcc(x :: xr, f, 0) + n
     || (2)                          || (2)
     n                      =?=    0 + n
```

`x :: xr` case with `f(x)` is false:

```
indexWhereAcc(x :: xr, f, n)    =?= indexWhereAcc(x :: xr, f, 0) + n
     || (3)                              || (3)
indexWhereAcc(xr, f, n+1)       =?= indexWhereAcc(xr, f, 1) + n
     || (inductive hypot.)               || (inductive hypot.)
indexWhereAcc(xr, f, 0) + (n+1) =?= (indexWhereAcc(xr, f, 0) + 1) + n
     || (arithmetics)                    || (arithmetics)
indexWhereAcc(xr, f, 0) + n + 1 =?= indexWhereAcc(xr, f, 0) + 1 + n
```

Part 3: Proof that the implementation satisfies the spec

`Nil` case:

```
indexWhere(Nil, f)       =?= 0
    || (def)
indexWhereAcc(Nil, f, 0) =?= 0
    || (1)
    0                    =?= 0
```

`x :: xr` case with `f(x)` is true:

```
indexWhere(x :: xr, f)       =?= 0
    || (def)
indexWhereAcc(x :: xr, f, 0) =?= 0
    || (2)
    0                        =?= 0
```

`x :: xr` case with `f(x)` is false:

```
indexWhere(x :: xr, f)       =?= 1 + indexWhere(xr, f)
    || (def)                           || (def)
indexWhereAcc(x :: xr, f, 0) =?= 1 + indexWhereAcc(xr, f, 0)
    || (3)
indexWhereAcc(xr, f, 1)      =?= 1 + indexWhereAcc(xr, f, 0)
    || (lemma)
indexWhereAcc(xr, f, 0) + 1  =?= 1 + indexWhereAcc(xr, f, 0)
```

Exercise 4: Subtyping (5 points)

Union of Sets

```
def union[A1 >: A](other: Set[A1]): Set[A1]
```

Explanation: For a detailed explanation, see lecture on covariance.

- If we set `other` to `Set[A]`, the expression val fruits = Set(new Apple).union(Set(new Peach)) will not typecheck.
- We therefore need a new type `A1` to authorize any type for the elements of `other`.
- However, we also want to ensure that the elements of `this` can be in a `Set[A1]`. Hence the constraint that `A1 >: A`.

Function Conformance

Explanation: for an assignment `val x: T = bla` to be valid, the type of `bla` must be a subtype of `T`. In all the exercises below, we need to verify this.

We also have the subtyping relation for functions:

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

We also know that the function type notation is right associative, i.e. `A => B => C` is the same as `A => (B => C)`

1. Is `A => D <: B => D`? yes, because of the above
2. Is `A => (D => C) <: A => (C => D)`?
   - `A >: A`.
   - So is `D => C <: C => D`? No, Because `D <: C`
3. Is `(D => A) => B <: (D => B) => A`?
   - Is `(D => A) >: (D => B)`? Yes, because `D <: D` and `A >: B`.
   - Is `B <: A`? Yes.

Exercise 5: Flattening (5 points)

The important part in this exercise was to pattern match an element of `ls` correctly. The naive solution is given below. It is quadratic in the number of "leaves" in `ls`.

```
def flatten(ls: List[Any]): List[Int] = ls match {
  case Nil => Nil
  case (x: Int) :: xs => x :: flatten(xs)
  case (x: List[Any]) :: xs => flatten(x) ++ flatten(xs)
}
```

There is a solution that is linear in the number of "leaves":

```
def flatten(ls: List[Any]): List[Int] = {

  def flattenConcat(tmpList: List[Any], tail: List[Int]): List[Int] = tmpList match {
    case Nil => tail
    case (x: Int) :: xs => x :: flattenConcat(xs, tail)
    case (x: List[Any]) :: xs => flattenConcat(x, flattenConcat(xs, tail))
  }

  flattenConcat(ls, Nil)
}
```

A `MatchError` is thrown if the pattern match fails on a certain pattern, there is no need to explicitly add a default case.