CS2030S Recitation Problem Set 7

Brian Cheong

InfiniteList<T>

- What is head?
  - A Producer that produces our value
  - Think of it as the instructions to create the current value
- What is tail?
  - A Producer that produces the next InfiniteList
  - Think of it as the instructions to create the next InfiniteList

InfiniteList<T> (APIs)

- `iterate` : `init` is your initial value, `next` transforms the current value to the next one
- `head` : gets your current value.
- `tail` : gets the next InfiniteList.
- `get` : get the value `n` elements away from the current one.

Question 1a: background

Fibonacci Sequence

- First described by Indian mathematician Pingala
- Popularised by Fibonacci
- Basically Fibonacci noticed rabbits are loving and thus breed a lot
- So it is

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

Question 1a:

- Enough about mating rabbits...
- What are we supposed to do?
  - Create `InfiniteList<BigInteger> fib(BigInteger a, BigInteger b)`
  - return infinite list of fibonacci numbers
  - a and b is your current number, and your next number
  - If head is called, return a (current number)
  - Fib is an encoding of 2 fibonacci numbers at a particular point.
  - Tail should be b as the current number and $a + b$ as the next number.

Question 1a:

```
InfiniteList<BigInteger> fib(BigInteger a, BigInteger b) {
    return new InfiniteList<>(
        () -> a,
        () -> fib(b, a.add(b))
    );
}
```

Question 1b:

- `ZipWith`
  - You're given another list and a curried `mapper` function. Basically zip 2 list together to produce a result list.
  - Realise the type of `mapper` : $T \rightarrow S \rightarrow R$, R is the type of the resultant list after zipping
  - Simply, apply mapper on the current element (T) to produce S and apply on S to produce R

Question 1b:

```
public <S, R> InfiniteList<R> zipWith (InfiniteList<? extends S> list,
        Transformer<? super T,
        ? extends Transformer<? super S, extends R>> mapper) {

    return new InfiniteList<>(
        () -> mapper.transform(this.head()).transform(list.head()),
        () -> this.tail().zipWith(list.tail(), mapper)
    );
}
```

Question 2

- Write `fib` again, such that it returns the first n Fibonacci numbers as a `Stream<BigInteger>`
  - use `iterate`, `map` and `limit` can be found here
  - use `Pair<T>`
  - Previously `fib` represented 2 Fibonacci numbers, now we can use `Pair` to do that
  - we iterate to create Pairs of Fibonacci numbers
  - First will be a number, second will be the next number.
  - Limit at n
  - then we just map and get the first of each pair

Question 2

```
Stream<BigInteger> fib(int n) {
    return Stream.iterate(new Pair<>(BigInteger.ONE, BigInteger.ONE),
        pair -> new Pair<>(pair.second, pair.first.add(pair.second)))
        .limit(n).map(pair -> pair.first);
}
```

Question 3

- Write `product` that takes in 2 `List` and produce a `Stream` combining each element from `list1` with every element in `list2` using `BiFunction`
  - `BiFunction` takes in 2 things and combines it into 1
  - For each element in `list1`, iterate across entire `list2`
  - Intuition we convert `list1` and `list2` to `Stream` since we want a stream anyway
  - then use `flatMap` to map each element in `stream1` to it's own copy of `stream2` (we don't want nested stream)
  - Both the element from `stream1` and `stream2` are in scope, and we can apply the `BiFunction`.

Question 3

```
<T, U, R> Stream<R> product(List<? extends T> list1, List<? extends U> list2,
                            BiFunction<? super T, ? super U, ? extends R> func) {
    return list1.stream().flatMap(ele1 ->
        list2.stream().map(ele2 ->
            func.apply(ele1, ele2)));
}
```

Question 4

- Omega numbers
  - the i-th Omega number is the number of distinct prime factors in the number i
  - Example: Omega number of 1 is 0, Omega number of 2 is 1, Omega number of 6 is 2 (2, 3)

Question 4

- Implement `omega` for n numbers
  - Create a stream of n numbers from $1...n$
  - for each number, create a new stream from $2...i$ (now is a stream of stream)
    - use filter to keep numbers that divide i and is a prime number
    - count the numbers
    - reduces our stream of streams back to a stream of numbers

Question 4

- Given in lecture

```
boolean isPrime(int x) {
    return IntStream.range(2, x).noneMatch(n -> x % n == 0);
}
```

Question 4

```
Stream<Long> omega(int n) {
    return Stream
        .iterate(1, i -> i <= n, i -> i + 1)
        .map(i -> Stream
            .iterate(2, x -> x <= i, x -> x + 1)
            .filter(x -> (i % x == 0 && isPrime(x)))
            .count());
}
```