Übungsblatt 04— Lösungen

1 Alte Bekannte

Die Funktionen `map`, `filter`, `flatMap` und `append` für `LazyList` via `foldRight` unterscheiden sich nicht weiter von den entsprechenden Implementationen für `List`:

```
def map[B](f: A => B): LazyList[B] =
  foldRight(empty[B])((h, t) => cons(f(h), t))
```

```
def filter(p: A => Boolean): LazyList[A] =
  foldRight(empty[A])((h, t) => if p(h) then cons(h, t) else t)
```

```
def flatMap[B](f: A => LazyList[B]): LazyList[B] =
  foldRight(empty[B])((h, t) => f(h).append(t))
```

```
def append[B >: A](b: => LazyList[B]): LazyList[B] =
  foldRight(b)(cons(_, _))
```

Das geänderte Verhalten durch Laziness steckt hier jeweils komplett in der Verwendung des Smart Constructors `cons` und unserer nicht-strikten Implementation von `foldRight` aus der Vorlesung.

2 takeWhile

2.1 via Pattern Matching

```
def takeWhile(p: A => Boolean): LazyList[A] = this match
  case Cons(h, t) if p(h()) => cons(h(), t().takeWhile(p))
  case _                    => Empty
```

Prinzipiell funktioniert `takeWhile` wie `take`, nur statt einer Zahl, die wir herunterzählen, nutzen wir jetzt die übergebene Bedingung um zu prüfen, ob wir weiter Elemente nehmen müssen.

2.2 via `foldRight`

```
def takeWhileViaFoldRight(p: A => Boolean): LazyList[A] =
  foldRight(empty[A])((h, t) => {
    if p(h) then cons(h, t)
    else empty
  })
```

Unser z-Parameter für `foldRight` ist das Listenende. In der übergebenen Funktion erhalten wir den jeweiligen Head der `LazyList`, sowie die Ergebnisliste, an die wir vorne das Element anhängen, falls es die Bedingung erfüllt. Wird die Bedingung nicht erfüllt, geben wir direkt `empty` zurück. Da in diesem Zweig `t` nicht benutzt wird, beendet dies die Rekursion.

2.3 via unfold

Zur Erinnerung hier unsere Implementation von `unfold` aus der Vorlesung:

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): LazyList[A] =
  f(z) match
    case Some((a, s)) => cons(a, unfold(s)(f))
    case None => empty
```

Mit `unfold` können wir aus einem Startwert eine `LazyList` generieren. Da wir eine `LazyList` mit Elementen aus der, die wir schon haben generieren wollen, ist die Idee folgende: wir nehmen die aktuelle Liste als Startwert, von dieser nehmen wir das erste Element und prüfen auf die Bedingung. Wird sie erfüllt, geben wir ein `Some` zurück und generieren so ein Element für die Ausgabe. Den neuen Startwert setzen wir auf den Tail, so dass wir weiter durch die Liste laufen. Wenn die Bedingung nicht erfüllt wird oder das Ende der Liste erreicht ist, geben wir `None` zurück, um die Rekursion zu beenden.

```
def takeWhileViaUnfold(p: A => Boolean): LazyList[A] = unfold(this){
  case Cons(h, t) if p(h()) => Some((h(), t()))
  case _                    => None
}
```

3 tails

Wie gerade eben wollen wir mit `unfold` eine Liste generieren, die aus Teilen unserer bisherigen Liste besteht. Jedoch wollen wir diesmal alle Suffixe, d.h. alle Teillisten, die man durch Entfernen von 0 oder mehr Elementen vom Anfang der Liste erhalten kann. Hier gibt es diverse Möglichkeiten, dies umzusetzen. Zwei davon stellen wir vor.

```
def tails: LazyList[LazyList[A]] = unfold(this) {
    case Empty => None
    case s => Some((s, s.drop(1)))
  }.append(LazyList(empty))
```

In der ersten Variante beginnen wir unser `unfold` mit der Liste als Startwert. Da das erste Element unserer Liste direkt diese selbst ist (0 Elemente entfernt), geben wir sie in der unfold-Funktion zurück und nehmen als nächsten Startwert den Tail. Da wir keine `tail`-Funktion in der Vorlesung implementiert haben, nutzen wir hier das äquivalente `drop(1)`. Erreichen wir das Ende der Liste, geben wir `None` zurück, um die Rekursion zu beenden. Aufgrund der Art, wie wir `unfold` definiert haben, können wir in diesem letzten Schritt kein Element zurückgeben, weshalb wir das letzte Element, die leere Liste, mittels `append` anhängen. Wichtig: `append` erwartet eine Liste vom Typ `LazyList[LazyList[A]]`. Würden wir statt `LazyList(empty)` einfach nur `empty` schreiben, wäre das immer noch valide, aber würde nichts anhängen.

```scala
def tailsAlt: LazyList[LazyList[A]] = LazyList(this).append(unfold(this){
    case Empty => None
    case Cons(_, t) => Some((t(), t()))
  })
```

In der zweiten Variante geben wir statt der aktuellen Teilliste immer deren Tail zurück, indem wir auf das `Cons` matchen. Dadurch fehlt die leere Liste am Ende nicht, dafür jedoch die vollständige Liste am Anfang. Also hängen wir das Ergebnis des `unfold`-Aufrufs an eine Liste an, in die wir unsere Startliste gepackt haben.