Übungsblatt 02— Lösungen

1 dropWhile

Wir wollen Elemente so lange vom Beginn der Liste entfernen bis `f false` zurückgibt. Hierzu gehen wir ähnlich wie bei `drop` rekursiv durch die Liste. Jedoch müssen wir beim `match` auch den Wert in einer Variable speichern, nicht nur den Tail:

```
def dropWhile(f: A => Boolean): List[A] =
  this match
    case Cons(x, xs) =>
      if f(x) then xs.dropWhile(f)
      else this
    case Nil => this
```

Wenn unsere Liste noch ein Element hat, nehmen wir den Wert davon und geben ihn an die Funktion `f`. Trifft die Funktion zu, rufen wir `dropWhile` rekursiv auf dem Tail der Liste auf, ansonsten sind wir fertig und geben die aktuelle Liste unverändert zurück (genau so bei einer leeren Liste).

Scala erlaubt es auch, zu einem `case` eine Bedingung hinzuzufügen, indem man zwischen Pattern und Pfeil `if <boolean expression>` hinzufügt:

```
def dropWhile(f: A => Boolean): List[A] =
  this match
    case Cons(a, as) if f(a) => as.dropWhile(f)
    case _                   => this
```

Dies nennt man ein *pattern guard*, und bewirkt, dass ein case nur dann matcht, wenn die gegebene Bedingung erfüllt ist. Jedoch hat dies den Nachteil, dass der Compiler bei Verwendung von Pattern Guards nicht mehr prüfen kann, ob jede Möglichkeit vom `match` abgedeckt wird.

2 Folds — Schritt für Schritt

Zur Erinnerung:

```
@annotation.tailrec
final def foldLeft[B](z: B)(f: (B, A) => B): B = this match
  case Nil         => z
  case Cons(a, as) => as.foldLeft(f(z, a))(f)
```

a) `foldLeft` und Addition

```
Cons(1, Cons(2, Cons(3, Nil))).foldLeft(0)((x,y) => x + y)
Cons(2, Cons(3, Nil)).foldLeft(0 + 1)((x,y) => x + y)
Cons(3, Nil).foldLeft(1 + 2)((x,y) => x + y)
Nil.foldLeft(3 + 3)((x,y) => x + y)
6
```

b) Inkrementieren jedes Elements einer Liste um 1 mit `foldRight`

```
Cons(1, Cons(2, Nil)).foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t))
Cons(2, Cons(2, Nil).foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t)))
Cons(2, Cons(3, Nil.foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t))))
Cons(2, Cons(3, Nil: List[Int]))
```

3 reverse

Implementation von `reverse` via `foldLeft`

```
def reverse: List[A] =
  foldLeft(Nil: List[A])((b, a) => Cons(a, b))
```

Wir starten mit einer leeren Liste. Da `foldLeft` im Gegensatz zu `foldRight` das vorderste Element der Liste zuerst mit dem Startwert kombiniert, wird dieses auch als erstes vor das Ende der Liste gehängt $\implies$ die Reihenfolge kehrt sich um.

4 append

Implementation von `append` via foldRight:

```
def append[AA >: A](r: List[AA]): List[AA] =
  foldRight(r)((a, as) => Cons(a, as))
```

Wir haben in der Vorlesung gesehen, dass `foldRight` mit `Nil` als Startwert und dem Cons-Konstruktor als Operator wieder die Liste ergibt, da wir alle Werte an den Startwert vorne anhängen. Um hieraus eine Listen-Konkatenation zu bauen, reicht es, den Startwert durch die hinten anzuhängende Liste zu ersetzen: wir hängen nun alle Elemente vor diese, statt vor ein `Nil`.

5 map, filter und flatMap

map: Beinahe identisch zum foldRight-Aufruf, der unsere Liste beibehält, nur die Funktion `f` wird zusätzlich auf den Werten aufgerufen.

```
def map[B](f: A => B): List[B] =
  foldRight(Nil: List[B])((a, as) => Cons(f(a), as))
```

filter: Wir testen in der fold-Funktion, ob die Bedingung `p` auf das aktuelle Element zutrifft. Falls ja, hängen wir es wie gehabt an das aktuelle Zwischenergebnis an. Falls nicht, behalten wir das letzte Zwischenergebnis bei.

```
def filter(p: A => Boolean): List[A] =
  foldRight(Nil: List[A])((a, as) => if p(a) then Cons(a, as) else as)
```

flatMap: Die übergebene Funktion gibt für jedes Element der Liste eine neue Liste zurück. Anstatt also das Ergebnis mit einem Cons vorne anzuhängen (was eine `List[List[B]]` ergeben würde), benutzen wir `append`, um die Liste elementweise vorne anzuhängen.

```
def flatMap[B](f: A => List[B]): List[B] =
  foldRight(Nil: List[B])((a, as) => f(a).append(as))
```

Exkurs: map, filter und flatMap in stack safe

Problem: Unsere Implementierung von `foldRight` ist nicht tail-rekursiv, d.h. sie ist nicht *stack-safe* (eine lange Liste führt zu einer zu großen Rekursionstiefe, wir können einen StackOverflowError erhalten).

Allerdings haben wir `map`, `filter` und `flatMap` via `foldRight` implementiert. `foldLeft` ist zwar stack-safe, aber unterscheidet sich semantisch hinsichtlich der Reihenfolge, in der die Elemente kombiniert werden. Eine Lösung für dieses Problem ist es, `foldRight` mit `foldLeft` zu implementieren indem wir die Liste vorher mit `reverse` umkehren:

```
def foldRight[B](z: B)(f: (A, B) => B): B = reverse.foldLeft(z)((b, a) => f(a, b))
```

Die Implementation von `foldRight` auf `List` in der Standardbibliothek nutzt diesen Trick.

6 zip

a) zipAdd

Wir wollen die Elemente zweier Listen von Integern paarweise addieren, also jeweils die Elemente mit dem gleichen Index. Falls die Listen nicht gleich lang sind, hören wir auf, sobald eine der Listen endet (erhalten also immer ein Ergebnis mit der Länge der kürzeren Liste).

```
def zipAdd(a: List[Int], b: List[Int]): List[Int] =
  (a, b) match
    case (Nil, _) | (_, Nil)          => Nil
    case (Cons(ah, at), Cons(bh, bt)) => Cons(ah + bh, zipAdd(at, bt))
```

Da wir beide Listen gemeinsam durchlaufen wollen, fassen wir diese in ein Tupel zusammen und matchen dann auf beide gleichzeitig (ein verschachteltes Pattern matching wäre auch möglich, aber unübersichtlicher).

Wir sehen hier auch, dass mit `|` mehrere Pattern kombiniert werden können. Dies geht nur, wenn diese keine Variablen zuweisen (Unterstriche sind erlaubt). Dies bewirkt das selbe, wie wenn wir die beiden Pattern in separaten `case`-Zeilen mit der selben Expression geschrieben hätten. `(Nil, _) | (_, Nil)` matcht genau dann, wenn wir das Ende von mindestens einer Liste erreicht haben. Im zweiten `case` können wir daher in beiden Listen auf ein `Cons` matchen und uns die Heads und Tails der Listen holen. Die Heads addieren wir dann und rufen auf den Tails `zipAdd` rekursiv auf.

Da unsere Methode nur mit Listen funktioniert, deren Elementtyp `Int` ist, implementieren wir sie auf dem Companion-Object. Direkt auf dem Enum wäre dies nur mit Implicits möglich, die wir erst später im Semester genauer behandeln werden. Hier trotzdem kurz eine solche Implementierung (nicht klausurrelevant):

```
def zipAdd[AA >: A](l: List[AA])(using num: Numeric[AA]): List[AA] =
  import num.*
  (this, l) match
    case (Nil, _) | (_, Nil)          => Nil
    case (Cons(ah, at), Cons(bh, bt)) => Cons(ah + bh, at.zipAdd(bt))
```

Der Körper der Methode ändert sich kaum, die Signatur wird jedoch deutlich komplizierter. Ohne Givens und `using` genauer zu erklären, kann man sich dies so vorstellen, dass `using num: Numeric[AA]` dazu auffordert, nach einer Implementation von Zahlenoperationen für den Typ `AA` zu suchen und sie zur Verfügung zu stellen (damit wir den Plus-Operator nutzen können). Der Parameter `AA` muss ein Obertyp der Elementtypen unserer Liste und der übergebenen zweiten Liste sein (ähnlich wie z.B. bei `append`). `Numeric` ist eine sogenannte Typklasse, diese werden wir in einer späteren Vorlesung detailliert behandeln.

b) zipWith

Wir verallgemeinern `zipAdd` auf beliebige Elementtypen und Operationen. Diesmal können wir wieder auf dem `List`-Enum definieren, ohne dass wir Implicits benötigen, da wir über die Elementtypen nichts wissen müssen.

Zwei Typparameter `B` und `R` stellen die Elementtypen der übergebenen zweiten Liste sowie der zurückgegebenen Liste dar. Den Typ `A` haben wir bereits von der Liste, auf der wir `zipWith` aufrufen. Außerdem fügen wir unserer Signatur einen Parameter für die Funktion hinzu, die je ein Element aus jeder Liste nimmt und kombiniert.

Nun müssen wir nur noch den Plus-Operator durch `f` ersetzen.

```
def zipWith[AA >: A,B](l: List[AA])(f: (A, AA) => B): List[B] =
  (this, l) match
    case (Nil, _) | (_, Nil)          => Nil
    case (Cons(a, as), Cons(b, bs))   => Cons(f(a, b), as.zipWith(bs)(f))
```

Die folgende Lösung geht noch einen Schritt weiter und macht unsere Implementation tailrekursiv. Dazu benötigen wir eine Hilfsfunktion. Wir nutzen wie bereits in der Vorlesung den Trick, unser Ergebnis rückwärts zusammenzubauen und anschließend wieder umzudrehen:

```
def zipWithTailrec[B,R](l: List[B])(f: (A, B) => R): List[R] =
  def go(agg: List[R])(a: List[A], b: List[B]): List[R] =
    (a, b) match
      case (Nil, _) | (_, Nil)          => agg
      case (Cons(ah, at), Cons(bh, bt)) => go(Cons(f(ah, bh), agg))(at, bt)
  go(Nil)(this, l).reverse
```

7 Parametricity

`curry` zum Umwandeln von Funktionen mit zwei Parametern in solche mit einem Parameter und Funktionstyp als Rückgabe:

```
def curry[A,B,C](f: (A,B) => C): A => B => C =
  a => b => f(a, b)
```

Um auf die Implementation zu kommen, ist es am einfachsten, vom Rückgabetyp aus zu arbeiten:

- `A => B => C` ist äquivalent zu `A => (B => C)`.
- Also ist unser Rückgabetyp eine Funktion, die einen Parameter vom Typ `A` bekommt. Wir fangen mit diesem an:
  `(a: A) =>`
- Der Rückgabetyp dieser Funktion wiederum, ist eine Funktion mit einem Parameter vom Typ `B`:
  `(a: A) => (b: B) =>`
- Diese Funktion soll einen Wert vom Typ `C` zurück geben. Unsere einzige Möglichkeit, einen solchen Wert zu bekommen, ist die übergebene Funktion `f`:
  `(a: A) => (b: B) => f(a,b)`
- Da die Typen auch aus dem Rückgabetyp der Funktion eindeutig sind, können wir diese wie oben weglassen, der Compiler ergänzt sie für uns.

`uncurry` als Umkehrfunktion von `curry` funktioniert sehr ähnlich:

```
def uncurry[A,B,C](f: A => B => C): (A, B) => C =
  (a, b) => f(a)(b)
```

Da unser Rückgabetyp eine Funktion mit zwei Parametern ist, beginnen wir mit der entsprechenden Parameterliste für die anonyme Funktion, und schauen dann, wie wir aus der übergebenen Funktion `f` einen Wert passend zum erwarteten Rückgabetyp bekommen.