Übungsblatt 09— Lösungen

1 Stack-Based Calculator

Die Funktionen sind in dieser Aufgabe sehr einfach. Interessant ist lediglich, dass wir jeweils einen Wert vom Typ `Calc[X] = State[List[Int], X]` zurück geben müssen, den Typ selbst aber nie erstellen. Wir benutzen lediglich Funktionen, die uns schon den richtigen Typ geben und kombinieren diese in for-Comprehensions.

```
def push(nr: Int): Calc =
  for
    stack <- get
    _     <- set(nr :: stack)
  yield nr
```

Für die Funktion `push` holen wir uns einfach den momentanen Zustand über `get`, wobei wir den Typ angeben. Dann prependen wir den übergebenen Parameter an den State und speichern ihn über `set` wieder ab.

```
def pop: Calc =
  for
    stack <- get
    _     <- set(stack.drop(1))
  yield stack.headOption.getOrElse(0)
```

Die Funktion `pop` ist sehr ähnlich. Statt zu prependen droppen wir hier allerdings. Außerdem müssen wir den Fall behandeln, dass der Stack leer ist. In diesem Fall gibt uns `getOrElse 0` die 0 als Default zurück.

```
def add: Calc =
  for
    a <- pop
    b <- pop
    c <- push(a + b)
  yield c
```

Die Funktion `add` besteht nun lediglich aus Kombinatoren. Wir müssen die ursprünglichen `get` und `set` Methoden gar nicht mehr benutzen. Dieser Effekt trifft häufig auf, wenn man einen Monad benutzt, um ein Problem zu lösen. Man baut sich seine eigene DSL und der Monad tritt irgendwann in den Hintergrund.

```
def mul: Calc =
  pop.flatMap(a =>     // a <- pop
    pop.flatMap(b =>   // b <- pop
      push(a * b)))
```

Die Funktion `mul` ist genauso wie die Funktion davor. Lediglich der Operator wurde getauscht.

2 Candy Machine

Es gibt viele Möglichkeiten diese Aufgabe zu lösen. Wir haben uns zuerst eine Update-Funktion gebaut, welche einen Input und eine Maschine bekommt und dann eine neue Maschine zurück liefert:

```
private def update(input: Input)(machine: Machine): Machine =
  (input, machine) match
    case (Coin, Machine(_, candies, coins)) if candies > 0 =>
      Machine(locked = false, candies, coins + 1)
    case (Turn, Machine(false, candies, coins)) =>
      Machine(locked = true, candies - 1, coins)
    case (_, m) => m
```

Die Funktion bildet die Anforderungen aus der Aufgabenstellung etwas verkürzt ab. Der erste Fall stellt sicher, dass die Maschine `unlocked` ist, wenn eine Münze eingeworfen wird. Einzig ausgenommen ist Fall, in denen keine Süßigkeiten mehr in der Maschine sind.

Der zweite Fall kommt zum tragen, wenn der Griff an der Maschine gedreht wird. Aufgrund des `false` in der Bedingung jedoch nur, wenn die Maschine nicht gelockt ist.

In allen anderen Fällen ändert sich der Zustand der Maschine nicht. Gerüstet mit dieser Funktion lässt sich jetzt `simulateMachine` einfach lösen:

```
def simulateMachine(inputs: List[Input]): State[Machine, (Int, Int)] =
  inputs.traverse(input => modify(update(input))) // State[Machine, List[Unit]]
    .flatMap(_ => get.map(m => (m.coins, m.candies)))
```

Der schwierigste Teil ist hier sicherlich der mit `inputs.traverse(input => modify(update(input)))`. In diesem Teil wird die Liste von Inputs genommen und für jeden Input eine State-Transition gebaut. Der Input wird an `update` gegeben, was aus `update` (bis dahin `(Input, Maschine) => Maschine`) eine Funktion von `Maschine => Maschine` macht. `modify` ist in Cats für State definiert und nimmt eine Funktion `S => S` und macht daraus ein `State[S, Unit]`, welcher sich den State holt, mithilfe der Funktion transformiert und dann wieder setzt. Hätten wir `map` statt `traverse` benutzt, wären wir bei einer `List[State[Maschine, Unit]]` gelandet. `traverse` tauscht aber noch die Typen und macht daraus eine `State[Maschine, List[Unit]]`.

Hier müssen wir dann nur noch das Resultat mit `get` holen und die zwei Teile `coins` und `candies` zurückgeben.

3 Applicative Filtering

3.1 filterA

Die Implementation ist kurz, aber hat es in sich:

```
def filterA[F[_],A](l: List[A])(p: A => F[Boolean])(using AF: Applicative[F]): F[List[A]]
↪  =
  l.foldRight[F[List[A]]](AF.pure(Nil))((a, fas) =>
    p(a).map2(fas)((incl, b) => if incl then a :: b else b))
```

Der Reihe nach: Die Signatur ist recht simpel. Wir arbeiten in einem `F[_]`, von dem wir wissen, dass es ein Applicative ist (`using Applicative[F]`).

Innerhalb der Methode starten wir einen `foldRight` mit folgenden Teilen:

1. Das Start-Element (Akkumulator) für den Fold ist eine leere Liste in einem `F`, also vom Typ `F[List[A]]`.
2. Die Kominbationsfunktion bekommt einmal ein Element aus der ursprünglichen Liste in a und die bisher erzeugte Liste im Parameter `fas`. `a` hat den Typ `A` und wird mithilfe von `p(a)` zu einem `F[Boolean]` gemacht.

   Wir benutzen `map2` um `p(a)` und `fa` auszupacken und binden sie an die Variablen `incl` und `t`. Innerhalb der Funktion überprüfen wir dann ob `incl` wahr ist und wenn ja, dann prependen wir das a an die Liste, ansonsten lassen wir die Liste unverändert. Da `foldRight` "von Hinten nach Vorne" geht, ist die Liste in `F[_]` in der korrekten Reihenfolge, obwohl wir pre- und nicht appenden.

1. Das Start-Element (Akkumulator) für den Fold ist eine leere Liste in einem `F`, also vom Typ `F[List[A]]`.
2. Die Kominbationsfunktion bekommt einmal ein Element aus der ursprünglichen Liste in a und die bisher erzeugte Liste im Parameter `fas`. `a` hat den Typ `A` und wird mithilfe von `p(a)` zu einem `F[Boolean]` gemacht.

   Wir benutzen `map2` um `p(a)` und `fa` auszupacken und binden sie an die Variablen `incl` und `t`. Innerhalb der Funktion überprüfen wir dann ob `incl` wahr ist und wenn ja, dann prependen wir das a an die Liste, ansonsten lassen wir die Liste unverändert. Da `foldRight` "von Hinten nach Vorne" geht, ist die Liste in `F[_]` in der korrekten Reihenfolge, obwohl wir pre- und nicht appenden.

3.2 powerset

Die Implementation ist denkbar einfach:

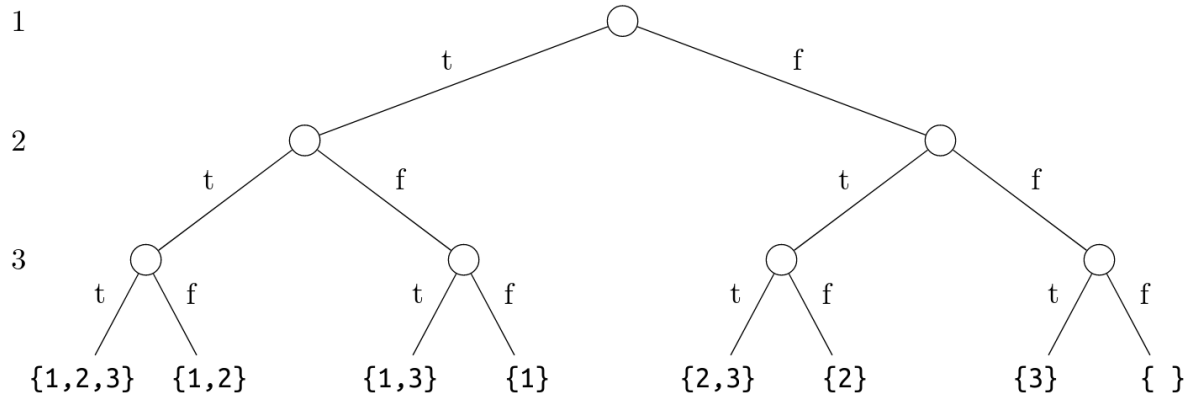
```
def powerset[A](l: List[A]): List[List[A]] =
  filterA(l)(_ => List(false, true))
```

Jetzt stellt sich nur die Frage: Warum funktioniert das?

Das funktioniert, weil `List` Indeterminismus modelliert. Sie spaltet also die Computation auf mehrere Pfade auf. Ein Pfad mit `true` und ein Pfad mit `false`.

Das bedeutet, es gibt für jedes Element zwei Pfade. Einen Pfad, in dem das Element in der Liste ist und einen, in dem es fehlt. Damit werden alle möglichen Listen generiert. Die Liste, die leer ist, ist der Pfad, in dem alle gefiltert wurden. Die Liste, die vollständig ist, ist der Pfad, in dem keine gefiltert wurden. Die anderen Listen liegen irgendwo in der Mitte.

3.3 Trees




```scala
List(1, 2, 3).filterA[Tree](_ => Branch(Leaf(false), Leaf(true)))
```

Ähnlich wie die Liste modelliert der Baum Indeterminismus. Allerdings behält er seine Struktur und liefert am Ende einen Baum zurück, in dem ich an jeder Kreuzung entscheiden kann, ob ich das jetzige Element in der Liste haben will oder nicht. Links bedeutet ja, rechts bedeutet nein. Folgerichtig ist das vollständige Resultat ganz links, das leere Resultat ganz rechts zu sehen.