Exercise sheet 09— Solutions

1 Stack-Based Calculator

The methods in this task are very simple. All we need to know is that we have to return a value of type `Calc[X] = State[List[Int], X]`. But we never create a value of that type ourselfs directly. All we do is use methods which give us a value of the correct type and the we use those values in for comprehesions.

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

To implement `push` we just ask for the current state via `get` while specifying the type. Then we prepend the given argument to the state and save it via `set`.

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

The method `pop` is very similar. Instead of prepending we just drop, though. Since we might run into an empty stack we need to handle that so we just do `getOrElse 0` to yield 0 as a default.

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

The method `add` consists only of combinators. We don't use `get` and `set` directly at all. This happens quite often when you use some monadic abstraction to solve a problem. You build your own custom DSL and the original monad fades into the background and you work only with your DSL.

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

Same as before, only the operator changed.

2 Candy Machine

There a quite a few ways to solve this task. We started with building an update method which gets some input and a machine and returns a new machine.

```scala
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
```

This function implements the requirements from the task sheet very succinctly. The first `case` makes sure the machine is always `unlocked` when someone enters `coins` as long as there are `candies` left.

The second case is matched when the nob is turned. Note that we match on `false` to make sure to only go into this case when the machine is not locked.

In all other casees the machine stays as is. Using this function now makes it very easy to implement `simulateMachine`:

```scala
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)))
```

The hartest part to understand surely is `inputs.traverse(input => modify(update(input)))`. Here we use the list of inputs and turn every input into a `State`-transition. The input is passed to `update`, which turns `update` (formerly `(Input, Maschine) => Maschine`) int a function of type `Maschine => Maschine`. `modify` is defined in Cats and takes a `S => S` and turns it into a `State[S, Unit]`, which ask for the state, transforms it and then saves it again. Would we have used `map` instead of `traverse` we would have gotten a value of type `List[State[Maschine, Unit]]` as a result. But since we used `traverse` the types get swapped and we get a value of type `State[Maschine, List[Unit]]`.

Now we just need to call `get` and yield `coins` and `candies`.

3 Applicative Filtering

3.1 filterA

The implemenation is short but a lot is going on:

```scala
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))
```

First things first: The signature is rather simple. We work inside a `F[_]` which we know to be a Applicative (`using Applicative[F]`).

Within the method we start with `foldRight` with two parts:

1. The accumulator element used is just an empty list inside of `F`, i.e. of type `F[List[A]]`.
2. The fold function gets an element of the original list in `a` as well as the entries already created in `fas`. `a` of type `A` gets turned into a `F[Boolean]` via `p(a)`.

Then we use `map2` to unpack `p(a)` and `fa` and bind it to `incl` and `t`. Inside the function we check whether `incl` is true and if so prepend the `a` to the list. Otherwise we leave the list unchanged. Since `foldRight` works from back to front, the list in `F[_]` is already in the right order even though we prepend instead of append.

3.2 powerset

The implemention is straight forward:

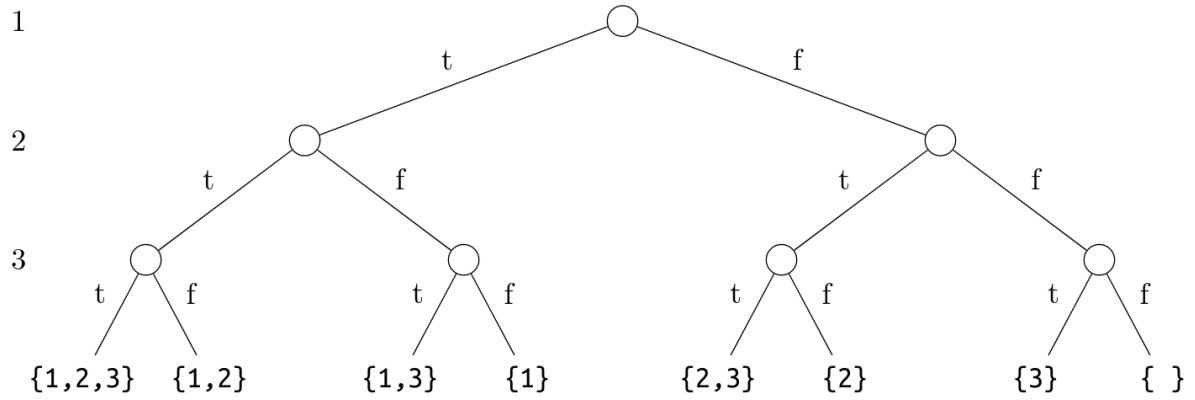
```
def powerset[A](l: List[A]): List[List[A]] =
  filterA(l)(_ => List(false, true))
```

Of course, the question is: Why does this work?

It works because `List` models the effect of multiple values. It splits the computation into multiple paths. One with `true` and one with `false`.

This means that for each element there are two paths to take. One in which the element is inside the resulting list and the other path in which it is not part of that list. This way, all possible lists are generated. The empty list is the result of the computation in which every element has been filtered out. The list containing all the elements is generated the opposite way. All the other lists are somewhere in between.

3.3 Trees




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

Just as a list a tree modles multiple values two. It has a richer inner structure though and keeps it. Each inner node reprensents a fork in the road. To the left is the path with the element in the list, to the right the path without. As is expected, the list with all elements is on the far left, the list without any elements on the far right.