Functional Programming

Monads

Exercise 1. Puzzle via Oleg Kiselyov.

> "U2" has a concert that starts in 17 minutes and they must all cross a bridge to get there. All four men begin on the same side of the bridge. It is night. There is one flashlight. A maximum of two people can cross at one time. Any party who crosses, either 1 or 2 people, must have the flashlight with them. The flashlight must be walked back and forth, it cannot be thrown, etc.. Each band member walks at a different speed. A pair must walk together at the rate of the slower man's pace:
>
> - Bono: 1 minute to cross
> - Edge: 2 minutes to cross
> - Adam: 5 minutes to cross
> - Larry: 10 minutes to cross
>
> For example: if Bono and Larry walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If Larry then returns with the flashlight, a total of 20 minutes have passed and you have failed the mission.

Find all answers to the puzzle using a list comprehension. The comprehension will be a bit long but recursion is not needed.

Exercise 2. Assume `concat_map` as defined in lecture 6. What will the following expresions return? Why?

1.
   ```
   perform with (|->) in
     return 5;
     return 7
   ```

2.
   ```
   let guard p = if p then [()] else [];;
   perform with (|->) in
     guard false;
     return 7;;
   ```

3.
   ```
   perform with (|->) in
     return 5;
     guard false;
     return 7;;
   ```

Exercise 3. Define `bind` in terms of `lift` and `join`.

Exercise 4. Define a monad-plus implementation based on binary trees, with constant-time `mzero` and `mplus`. Starter code:

```
type 'a tree = Empty | Leaf of 'a | T of 'a t * 'a t
module TreeM = MonadPlus (struct
  type 'a t = 'a tree
  let bind a b = TODO
  let return a = TODO
  let mzero = TODO
  let mplus a b = TODO
end)
```

Exercise 5. Show the monad-plus laws for one of:

1. `TreeM` from your solution of exercise 4;
2. `ListM` from lecture.

Exercise 6. Why the following monad-plus is not lazy enough?

-
  ```
  let rec badappend l1 l2 =
    match l1 with lazy LazNil -> l2
    | lazy (LazCons (hd, tl)) ->
      lazy (LazCons (hd, badappend tl l2))
  let rec badconcat_map f = function
    | lazy LazNil -> lazy LazNil
    | lazy (LazCons (a, l)) ->
      badappend (f a) (badconcat_map f l)
  ```

-
  ```
  module BadyListM = MonadPlus (struct
    type 'a t = 'a lazy_list
    let bind a b = badconcat_map b a
    let return a = lazy (LazCons (a, lazy LazNil))
    let mzero = lazy LazNil
    let mplus = badappend
  end)
  ```

-
  ```
  module BadyCountdown = Countdown (BadyListM)
  let test5 () = BadyListM.run (BadyCountdown.solutions [1;3;7;10;25;50] 765)
  ```

-
  ```
  # let t5a, sol5 = time test5;;
  val t5a : float = 3.3954310417175293
  val sol5 : string lazy_list = <lazy>
  # let t5b, sol5_1 = time (fun () -> laztake 1 sol5);;
  val t5b : float = 3.0994415283203125e-06
  val sol5_1 : string list = ["((25-(3+7))*(1+50))"]
  # let t5c, sol5_9 = time (fun () -> laztake 10 sol5);;
  val t5c : float = 7.8678131103515625e-06
  val sol5_9 : string list =
    ["((25-(3+7))*(1+50))"; "(((25-3)-7)*(1+50))"; ...
  # let t5d, sol5_39 = time (fun () -> laztake 49 sol5);;
  val t5d : float = 2.59876251220703125e-05
  val sol5_39 : string list =
    ["((25-(3+7))*(1+50))"; "(((25-3)-7)*(1+50))"; ...
  ```

Exercise 7. Convert a "rectangular" list of lists of strings, representing a matrix with inner lists being rows, into a string, where elements are column-aligned. (Exercise not related to recent material.)

Exercise 8. Recall the overly rich way to introduce monads – providing the freedom of additional parameter

```
module type MONAD = sig
  type ('s, 'a) t
  val return : 'a -> ('s, 'a) t
  val bind :
    ('s, 'a) t -> ('a -> ('s, 'b) t) -> ('s, 'b) t
end
```

Recall the operations for the exception monad:

```
val throw : excn -> 'a monad
val catch : 'a monad -> (excn -> 'a monad) -> 'a monad
```

1. Design the signatures for the exception monad operations to use the enriched monads with `('s, 'a) monad` type, so that they provide more flexibility than our exception monad.

2. Does the implementation of the exception monad need to change? The same implementation can work with both sets of signatures, but the implementation given in lecture needs a very slight change. Can you find it without implementing? If not, the lecture script provides `RMONAD`, `RMONAD_OPS`, `RMonadOps` and `RMonad`, so you can implement and see for yourself – copy `ExceptionM` and modify:
```
module ExceptionRM : sig
  type ('e, 'a) t = KEEP/TODO
  include RMONAD_OPS
  val run : ('e, 'a) monad -> ('e, 'a) t
  val throw : TODO
  val catch : TODO
end = struct
  module M = struct
    type ('e, 'a) t = KEEP/TODO
    let return a = OK a
    let bind m b = KEEP/TODO
  end
  include M
  include RMonadOps(M)
  let throw e = KEEP/TODO
  let catch m handler = KEEP/TODO
end
```

Exercise 9. Implement the following constructs for *all* monads:

1. `for`...`to`...
2. `for`...`downto`...
3. `while`...`do`...
4. `do`...`while`...
5. `repeat`...`until`...

Explain how, when your implementation is instantiated with the `StateM` monad, we get the solution to exercise 2 from lecture 4.

Exercise 10. A canonical example of a probabilistic model is that of a lawn whose grass may be wet because it rained, because the sprinkler was on, or for some other reason. Oleg Kiselyov builds on this example with variables `rain`, `sprinkler`, and `wet_grass`, by adding variables `cloudy` and `wet_roof`. The probability tables are:

$$\begin{aligned}
P(\text{cloudy}) &= 0.5\\
P(\text{rain}|\text{cloudy}) &= 0.8\\
P(\text{rain}|\text{not cloudy}) &= 0.2\\
P(\text{sprinkler}|\text{cloudy}) &= 0.1\\
P(\text{sprinkler}|\text{not cloudy}) &= 0.5\\
P(\text{wet roof}|\text{not rain}) &= 0\\
P(\text{wet roof}|\text{rain}) &= 0.7\\
P(\text{wet grass}|\text{rain} \wedge \text{not sprinkler}) &= 0.9\\
P(\text{wet grass}|\text{sprinkler} \wedge \text{not rain}) &= 0.9
\end{aligned}$$

We observe whether the grass is wet and whether the roof is wet. What is the probability that it rained?

Exercise 11. Implement the coarse-grained concurrency model.

- Modify `bind` to compute the resulting monad straight away if the input monad has returned.

- Introduce `suspend` to do what in the fine-grained model was the effect of `bind (return a) b`, i.e. suspend the work although it could already be started.
- One possibility is to introduce `suspend` of type `unit monad`, introduce a “dummy” monadic value `Suspend` (besides `Return` and `Sleep`), and define `bind suspend b` to do what `bind (return ()) b` would formerly do.