Category Theory: Monads

98-317: Hype for Types

Introduction

This week we learned about practical uses of category theory. In this homework, you will use monads inside Standard ML.

Turning in the Homework: Submit your `handin.zip` file to Gradescope.

Implementing a Monad

First, let's implement a monad!

In `MONAD.sig`, you'll find the `MONAD` signature (which includes the `MAPPABLE` signature, found in `MAPPABLE.sig`).

```
signature MAPPABLE =
  sig
    type 'a t
    val map : ('a -> 'b) -> 'a t -> 'b t
  end
```

```
signature MONAD =
  sig
    include MAPPABLE

    val return : 'a -> 'a t

    val >>= : 'a t * ('a -> 'b t) -> 'b t
    val >=> : ('a -> 'b t) * ('b -> 'c t) -> 'a -> 'c t
    val join : 'a t t -> 'a t
  end
```

As mentioned in lecture, there are a few invariants left implicit by the signatures. However, for the purpose of this homework, we won't go into detail on what they are; anything well-typed and "sensible" is likely correct.

In `OptionMonad.sml` and `ListMonad.sml`, we've included the implementation of the option and list monads from lecture.

Task 1 Implement `structure LogMonad` in `LogMonad.sml`.

We've included tests in `log-test.sml` that you can run using:

```
smlnj sources.cm log-test.sml
```

The expected results are given as `(* EXPECT: expected-result-here *)` in the `log-test.sml` file.

Monad Automation

You may have noticed that implementing some of the monad functions can get quite redundant; in particular, `map`, `>>=`, `>=>`, and `join` all have a similar flavor. If so, you wouldn't be surprised to learn that given only `return` and `>>=`, you can implement all of the others![1]

Task 2 In `MkMonad.sml`, implement `functor MkMonad` which takes in `return` and `>>=` and produces a structure ascribing to `MONAD`. (Hint: use your intuition, and follow the types!)

Huzzah! Now, to define a monad, we'll just use your functor. (No need to define the other functions manually, unless there's an efficient implementation we'd prefer to use.)

[1] This works for `>=>` and `join`, too! Bonus: try implementing `>>=` in terms of them.

Probabilistic Programming

In `ProbabilityMonad.sml`, we've implemented the probability monad for you![2] Now, to use it! Direct your attention to `probability-test.sml`.

Wakeup

Many people start the semester waking up early, but this doesn't always last. What are the chances you still wake up early by the end of the semester? We'll use the probability monad as a tool to figure it out, working under the following assumptions:

> On the 0th day, we fix whether you woke up early or late.
>
> On the $n + 1$th day:
>
> - if you wake up early on day n, there's a 70% chance you wake up early the next day (and 30% chance you wake up late)
> - if you wake up late on day n, there's a 90% chance you wake up late the next day (and 10% chance you wake up early)

At a high level, we assume you have a decent shot to wake up early if you're already waking up early, but a very high chance to wake up late again if you're already waking up late.

We model one day of this algorithm as follows:

```
datatype wakeup = Early | Late

val transition : wakeup -> wakeup t =
 fn Early => [(Early, 0.7), (Late, 0.3)]
  | Late  => [(Late, 0.9), (Early, 0.1)]
```

Given what happened yesterday, we give a probability distribution over what might happen today.

To run the simulation for n days, we simply use monadic bind to sequence together n calls to `transition`:

```
(* simulate : int -> wakeup -> wakeup t *)
fun simulate 0 w = return w
  | simulate n w = simulate (n - 1) w >>= transition
```

Note that in the implementation, we locally shadow `>>=` so that it automatically "compresses" the resulting distribution (so there's at most one entry in the distribution per possibility).

[2] Using your `MkMonad`, of course.

Now, we can use the `simulate` function to observe the distribution after many iterations. Based on the described model, we'll assume we start waking up early.

```
> smlnj sources.cm probability-test.sml
(* ... *)
- simulate 0 Early;
val it = [(Early,1.0)] : wakeup t
- simulate 1 Early;
val it = [(Early,0.7),(Late,0.3)] : wakeup t
- simulate 2 Early;
val it = [(Early,0.52),(Late,0.48)] : wakeup t
```

Task 3 Open the REPL and experiment with some other values of `n` and observe how the distribution changes. As `n` gets large, the probability of waking up early converges on a particular probability: what is it? Set `lateAtEndOfSemester` to this value.

Bonus Task Alter the probabilities in the original `transition` function and observe how the distribution changes in the limit.

Pro Tip Next time you have a homework involving probability, check your work by modeling the scenario using the probability monad!

Monopoly Jail

Now, it's your turn to model a scenario!

> In the game Monopoly, if you land in "jail", you roll dice to attempt to get out.
>
> - Each try, you roll two six-sided dice. If the results are equal, you're out of jail! Otherwise, you're still in jail.
> - After three failed attempts, you have to pay to get out of jail. Oh no!

Let's model it using the probability monad! First, we have some starter code:

```
(* 6-sided die *)
val d6 = List.tabulate (6, fn i => (i + 1, 1.0 / 6.0))

datatype state = InJail | OutOfJail
```

The value `d6` is a distribution over numbers between 1 and 6, each with equal probability.

Task 4 Implement `jailRoll : state -> state t` which, given a state, computes the distribution according to the given scenario. In particular, if your state is `InJail`, you should sample `d6` twice and switch to state `OutOfJail` only when the two samples are equal.

Using `getOutOfJail`:

```
val getOutOfJail = jailRoll >=> jailRoll >=> jailRoll
```

we sequence three attempts to get out of jail.

Load the file and see what distributions `ifOutOfJail` and `ifInJail` are.

- `ifOutOfJail` should be `[(OutOfJail,1.0)]`, since we start out of jail in the first place, and
- `ifInJail` should be `[(InJail,1-P),(OutOfJail,P)]`, for some to-be-found probability of escaping `P`.

Functional Imperative Programming

It's time for some *real* imperative programming!

We implemented the Imperative Monad in `ImperativeMonad.sml`[3], which we can now make use of.

You don't have to worry about the implementation of `ImperativeMonad`; instead, you can simply use the given helper functions, which look imperative (aside from the monadic piping required to use them).

We'll try to write some C-like imperative code to solve the "Fizz" problem (a simplification of "FizzBuzz"):

```
int counter = 0;
string output = "";

string fizz() {
  output = output ^ (counter % 3 == 0) ? "Fizz" : Int.toString
   x;

  if (counter < 10) {
    counter = counter + 1;
    return fizz();
  } else
    return output;
}
```

Task 5 Finish the implementation of `fizz` in `sandbox.sml`, updating the output (as in the C code).[4][5]

When you load `sandbox.sml`, you should get the following output if you uncomment the line `(* val result = ... *)`:

```
val result =
  {state={counter=10,output="Fizz12Fizz45Fizz78Fizz10"},
  value="Fizz12Fizz45Fizz78Fizz10"} : {state:?.state, value:
   string}
```

Notice, in addition to the state ("global variables" `counter` and `output`), we get our answer `"Fizz12Fizz45Fizz78Fizz10"`.

[3] Often, this is called the *state monad*.

[4] Hint: consider using the helper function `setOutput`.

[5] Another hint: remember your "CPS"! Notice that variables `counter` and `output` are in scope.