K 1

Monadic Effects

1.1 Outline

1. Definition and context for monads in category theory and computer science
2. Explanation of how monads can model effects in general
3. Demonstration of constructing and using the stateful monad, building it up from scratch in Haskell or Haskell-like psuedocode
   ( ) Explain the Functor, Applicative, and Monad typeclasses, and how they build up the mathematical definition of a monad
4. Outline some problems with monadic effects
   ( ) different effects are not composable
   ( ) paper: *The Awkward Squad*

1.2 Introduction to Monads

TODO: introduce in programmer-friendly way first, mention category only for like a sentence if at all. give examples of writing code in Java/C/etc. and how we want to be able to do something similar but in a functional way

The concept of *monad* originates in the branch of mathematics by the name of Category Theory. As highly abstract as monads are, they turn out to be a very convenient structure for formalizing implicit contexts within functional programming languages. Though a background in the Categorical approach to monads and monad-related structures probably cannot hurt one's understanding of computational monads, this section will follow a more programmer-friendly path.

One particularly general effect is the **stateful** effect, where an implicit, mutable state is accessible within a stateful computation. This effect was implemented by $\mathbb{B}$ by the introduction of a globally-accessible, mutable table of variables. This implementation required, however, several new syntactical structures and reduction rules. Is there a way to implement something similar in just 𝔸?

It turns out there is — with a certain tradeoff. Since 𝔸 is pure, such an implementation cannot provide true mutability. However, we can model mutability in 𝔸 as a function from the initial state to the modified state. Call a σ-computation of α to be a stateful computation where the state is of type σ and the result is of type α. To describe the type of such computations purely, consider the following:

```
type stateful σ a := σ -> σ × α.
```

Relating to the description of the stateful effect, `stateful` σ α is the type of functions from an initial σ-state to a pair of the affected σ-state and the α-result. So if given a term m : `stateful` σ α, one can purely compute the affected state and result by providing m with an initial state.

For `stateful` to truly be an effect, it needs to implicitly facilitate the stateful effect to some sort of internal context. So let us see how we can construct terms that work with `stateful`. In 𝔹's implementation of this effect, it posited two primitive terms: `read` and `write`. Using `stateful` we can define these terms in 𝔸 as:

```
term read : stateful σ σ := (s : σ) => (s, s).
term write (s' : σ) : stateful σ s := (s : σ) => (s', •).
```

Observe that `read` is a σ-computation that does not modify the state and returns the state, `write` is a σ-computation that replaces the state with a given $s' : \sigma$ and returns $\bullet$. In this way, using `read` and `write` in 𝔸 fills exactly the same role as a simple 𝔹 stateful effect of one mutable variable.

Since the stateful effect is in fact an effect, we should also be able to *sequence* stateful computations to produce one big stateful computation that does performs the computations in sequence. It is sufficient to define the sequencing of just two effects, since any number of effects can be sequenced one step at a time. So, given two σ-computations m : `stateful` σ α and m' : `stateful` σ β the sequenced σ-computation should first compute the m-affected state and then pass it to m'.

```
term sequence (m : stateful σ α) (m' : stateful σ β) : stateful σ β :=
  (s : σ) =>
    let (s', a) : σ × α := m s in
    m' s'.
```

In general, the term that sequences effects is called “>>” and infixed.

However, in this form it becomes clear that `sequence` throws away some information — the α-result of the first stateful computation. To avoid this amounts to allowing m' to reference m's result, and can be modeled by typing m' instead as $\alpha \to$ `stateful` σ β. A sequence that allows this is called a *binding-sequence*, since it sequences m, m' and also *binds* the first parameter of m' to the result of m.

```
term bind (m : stateful σ α) (fm : α -> stateful σ β) : stateful σ β :=
  (s : σ) =>
    let (s', a) : σ × α := m s in
    fm a s'.
```

In general, the term that binding-sequences effects is called ">>=" and infixed.

Additionally, one may notice that one of `sequence` and `bind` is superfluous i.e. can be defined in terms of the other. Consider the following re-definition of `sequence`:

```
term sequence (m : stateful σ α) (m' : stateful σ β) : stateful σ β :=
  bind m ((a : α) => m')
```

This construction demonstrates how `sequence` can be thought of as a trivial binding-sequence, where the bound result of m is ignored by m'.

So far, we have defined all of the *special* operations that 𝔹 provides for stateful computations. The key difference between 𝔹 and our new 𝔸 implementation of the stateful effect is that 𝔹 treats stateful computations just like any other kind of pure computation, whereas 𝔸 "wraps" σ-stateful computations of α using the `stateful` σ α type rather than just pure α. What is still missing in our 𝔸 implementation is the ability to use non-stateful computations within or on stateful computations. In other words, we should be able to *lift* non-stateful computations to stateful computations that internally don't actually end up having stateful effects. There are two such computations to consider:

- **return**: The non-stateful computation, with parameter $a : \alpha$, that results in the same $a : \alpha$. In other words, the computation that does nothing and results in a.

- **map**: The computation, with parameter non-stateful computation $f : \alpha \to \beta$ and stateful computation m : `state` σ α, that results in f applied to the result of m. In other words, the function the lifts $f : \alpha \to \beta$ to a function `stateful` σ $\beta \to$ `stateful` σ β.

These computations are implemented in 𝔸 as follows:

```
term return (a : α) : stateful σ a :=
  (s : σ) => (s, a)
```

```
term map (f : α -> β) (m : stateful σ α) : stateful σ β :=
  (s : σ) =>
    let (s', a) : σ × α := m s in
    (s', f a)
```

1.2.1 Definition of Monad

Now we have implemented a concrete approach to the stateful effect using pure terms in 𝔸. However, only the `read` and `write` terms were meant for exclusive use with `stateful`. The other necessary terms, as they correspond to use the impure implementation in 𝔹, are intended to interoperate with all effects and interchangeably. To summarize, the capabilities the terms implement are

- **map**
- **return**
- **binding-sequence**

With these in mind, the definition of monad is as follows:

Definition 1.2.1. A type $M : Type \to Type$ is a **monad** if there exist terms of the following types:

- `map : (α -> β) -> M α -> M β`
- `return : α -> M α`
- `bind : M α -> (α -> M β) -> M β`

However, within 𝔸 we currently have no type-oriented way to assert that a type M is associated with the expected constructions for qualifying as a monad. In order to formally and generally reference monads in 𝔸, we first need to introduce simple type-classes.

1.3 Typeclasses

Classes are conceived of in many different forms in many different programming languages. The core concept is that a collection of different structures can be considered as the range of a parameter by requiring that the variable only assumes the structure meets the requirements associated with the class, and each structure that is a member of the class is required to meet those requirements.

Classes can be considered in the realm of types as allowing the generalization of a type parameter over many different types as long as all of those types are a member of a specified **type-class**. Type-classes are useful for defining abstract properties of types that vary in implementation from type to type.

As a contrived example, consider typing problem of constructing an `add` term that works for both `integers` and `float`. Ideally, we want a function that looks something like `add` : *number* → *number*, where *number* is some type. However, since `integer` and `float` are not the same type, `number` cannot be just one of them. So it appears that we *must* have two terms for add: . This solution is ugly because it doesn't take formal advantage of the intention that `add` is the *same* operation in these two cases, even though it is applied on different types, and instead creates a few heavily-overlapping names. These two types share the feature of being *addable* — the requirement of both a `add-integer` and a `add-float` is ignorant of this.

A different solution that does in fact take advantage of the fact that `integer` and `float` are both *addable* is that of creating a type-class for *addable*.

```
primitive type Addable : Type -> Type.
primitive type Addable-instance : α( : Type) => α( -> α) -> Addable α.
```

Next we instantiate `integer` and `float` as instances of the `Addable` type-class.

```
term Addable-integer : Addable integer :=
  Addable-instance ((x, y) => x y)
```