1 The Mathematics of Monads

Monads are a general mechanism for extending functional languages with new features. They were introduced in the context of functional programming by Eugenio Moggi [1] and are by now regarded as a central tool of the functional language Haskell. There are literally dozens of tutorials available on the use of monads in Haskell. A reasonably good one is the introduction by Philip Wadler [2].

The concept of monad goes much further back, however. They were introduced by Roger Godement in the context of category theory as far back as 1958. The name *monad* was coined by Saunders MacLane [**?**]. Monads are strongly related to *adjunctions*, one of the most ubiquitous and useful concepts of category theory.

In this lecture we will not duplicate the content of these tutorials (the Wadler and Moggi papers are provided as handouts). Instead we will focus on the categorical foundations of monads and give several examples in the hope that this will aid in understanding the concept as it is used in functional programming.

2 Categories

A *category* consists of

- A collection of *objects* $A, B, \ldots$. These are often sets with some extra structure, such as groups or topological spaces, but abstractly they are just nodes of a directed graph. Think of them as types.
- A collection of typed *morphisms* $f : A \to B$ for each pair of objects A, B. The objects A and B are called the *domain* and *codomain* of f, respectively. Morphisms are typically structure-preserving functions from A to B, but abstractly they are just a directed edges in a graph.
- A typed *composition operator* $\circ$ such that if $f : A \to B$ and $g : B \to C$, then $g \circ f : A \to C$. Composition is associative.
- An identity morphism $\mathsf{id}_A : A \to A$ for each object A. The identity morphisms are both left and right identities for composition.

Examples of categories are: Sets and set functions (Set), groups and group homomorphisms, topological spaces and continuous maps, partially ordered sets and monotone maps, CPOs and continuous functions (CPO), vector spaces over $\mathbb{R}$ and linear maps. A partially ordered set $(X, \leq)$ forms a category whose objects are the elements of X with a unique morphism $x \to y$ iff $x \leq y$. Abstractly, a category is just a typed monoid.

3 Functors

A structure-preserving map between categories is called a *functor*. If $\mathcal{C}$ and $\mathcal{D}$ are categories, a functor $F : \mathcal{C} \to \mathcal{D}$ maps objects $A \in \mathcal{C}$ to objects $FA \in \mathcal{D}$ and morphisms $f : A \to B$ of $\mathcal{C}$ to morphisms $Ff : FA \to FB$ of $\mathcal{D}$ such that $F(g \circ f) = Fg \circ Ff$ and $F(\mathsf{id}_A) = \mathsf{id}_{FA}$. Functors are morphisms of categories, and there is a category of small categories, the "small" being there to avoid set-theoretic paradoxes.

Monads are based on *endofunctors*, which are functors from a category to itself. In most of our examples, the category is Set. Some examples are:

- The *powerset functor* $P : \mathsf{Set} \to \mathsf{Set}$. Here $PA = \{\text{subsets of } A\}$ and if $f : A \to B$, then $Pf : PA \to PB$ takes $A' \subseteq A$ to $\{f(a) \mid a \in A'\} \subseteq B$.
- The *term functor* $T_\Sigma : \mathsf{Set} \to \mathsf{Set}$ over a given signature Σ. Here Σ consists of various function symbols of various finite arities (for example, $\Sigma = \{f, g, c\}$ with f binary, g unary, and c a constant). The set of *terms* over variables X, denoted $T_\Sigma(X)$, is defined inductively; in our example, c is a term and any $x \in X$ is a term; if t is a term, then $g(t)$ is a term; and if s, t are terms, then $f(s, t)$ is a term. The functor T_Σ takes X to $T_\Sigma(X)$ and $f : X \to Y$ to $T_\Sigma(f) = \lambda t.\, t\{f(x)/x,\, x \in X\} : T_\Sigma(X) \to T_\Sigma(Y)$.
- The *option functor* $Opt : \mathsf{Set} \to \mathsf{Set}$. Here $Opt\, X = \{\mathsf{Some}\ x \mid x \in X\} \cup \{\mathsf{None}\}$, and if $f : X \to Y$, then $Opt\, f$ maps $\mathsf{Some}\ x$ to $\mathsf{Some}\ f(x)$ and None to None.
- The *lifting functor* $L : \mathsf{CPO} \to \mathsf{CPO}$, where L maps $\mathcal{D}$ to $\mathcal{D}_\perp$, and if $f : \mathcal{D} \to \mathcal{E}$ is a continuous map, then $Lf : \mathcal{D}_\perp \to \mathcal{E}_\perp$, where $Lf(\lfloor x \rfloor) = \lfloor f(x) \rfloor$ for $x \in \mathcal{D}$ and $Lf(\perp) = \perp$ (see Lecture 22, 7 March).
- If $(X, \leq)$ and $(Y, \leq)$ are partially ordered sets regarded as categories, then a functor $X \to Y$ is just a monotone function.

4 Monads

Each of these endofunctors gives rise to a monad in a natural way. A *monad* is a triple (F, η, μ) where

- $F : \mathcal{C} \to \mathcal{C}$ is an endofunctor on $\mathcal{C}$ (in most examples, $\mathcal{C} = \mathsf{Set}$);
- η is a collection of morphisms $\eta_A : A \to FA$, one for each object A; and
- μ is a collection of morphisms $\mu_A : F^2A \to FA$, one for each object A.

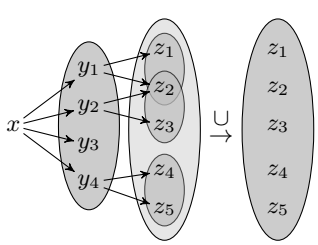
We can think of a monad as a mechanism for introducing new structure FA on an existing object A, as described by the endofunctor. The morphisms η_A and μ_A describe how morphisms are extended to handle the new structure.

There are a few axioms that η and μ must satisfy, but we will get to these later. First some examples.

4.1 The Powerset Monad

The *powerset monad* consists of the powerset functor P and morphisms $\eta_A(a) = \{a\}$ for $a \in A$ and $\mu_A(C) = \bigcup C$ for $C \in P^2A$. This is useful for modeling nondeterministic computation. A *nondeterministic function* from A to B is a function $f : A \to PB$, where $f(x) \subseteq B$ is the set of possible output values on input x.

Suppose we have two nondeterministic functions $f : A \to PB$ and $g : B \to PC$ and we wish to compose them. Intuitively, we would start from some $x \in A$, then compute $f(x)$. The output state could be any state in the set $f(x) \subseteq B$. Now from any one of those states $y \in f(x)$, we compute $g(y)$ to get a set of possible output states in C. The set of states we could be in after both steps is any state in some $g(y)$ for some $y \in f(x)$. This is the set $\bigcup\{g(y) \mid y \in f(x)\}$.



Breaking this down, we see that we first computed $f(x) \in PB$, then computed $Pg(f(x)) = \{g(y) \mid y \in f(x)\} \in P^2C$, then took the union to get the final set of output states $\bigcup\{g(y) \mid y \in f(x)\}$.

$$x \xrightarrow{f} f(x) \xrightarrow{Pg} \{g(y) \mid y \in f(x)\} \xrightarrow{\mu_C} \bigcup\{g(y) \mid y \in f(x)\}.$$

So the composition is given by $\mu_C \circ Pg \circ f : A \to PC$. This is known as the *Kleisli composition* of f and g, denoted $g \bullet f$. Do f first to get a set; then do Pg to the resulting set, which is the same as doing g to each element of the set, to get a set of sets; then knock it back down to just a set by taking the union.

We have not stated the axioms for η and μ, but they will imply that Kleisli composition is associative and that η is a left and right identity for Kleisli composition. Thus the collection of objects of $\mathcal{C}$ with morphisms $f : A \to PB$ under Kleisli composition and identities η_A form a category, called the *Kleisli category* of the monad. The type of $f : A \to PB$ is PB in Set, but B in the Kleisli category.

The action of a nondeterministic finite automaton is easily described in terms of Kleisli composition. Say we have an NFA with state set S. The transitions are typically specified by a ternary relation $\Delta \subseteq S \times \Sigma \times S$, or equivalently, a function $\Delta : S \times \Sigma \to PS$. This function can be extended by induction to $\widehat{\Delta} : S \times \Sigma^* \to PS$ as follows:

$$\widehat{\Delta}(s, \varepsilon) \triangleq \{s\} \qquad\qquad \widehat{\Delta}(s, xa) \triangleq \bigcup\{\Delta(t, a) \mid t \in \widehat{\Delta}(s, x)\}.$$

However, if we curry Δ and $\widehat{\Delta}$ to the form $\Delta : \Sigma \to S \to PS$ and $\widehat{\Delta} : \Sigma^* \to S \to PS$ respectively, we can just say that $\widehat{\Delta}$ is the unique extension of Δ to a monoid homomorphism from the free monoid Σ^* to the monoid of functions $S \to PS$ under Kleisli composition.

The Kleisli category of the powerset monad is isomorphic to the category of Rel of sets and binary relations under relational composition.

4.2 The Term Monad

The *term monad* consists of the term functor $T_\Sigma : \mathsf{Set} \to \mathsf{Set}$ with $\eta_X : X \to T_\Sigma(X)$ that takes the variable x to the term x and $\mu_X : T_\Sigma(T_\Sigma(X)) \to T_\Sigma(X)$ that takes a term over terms over X and just considers it a term over X. Neither η_X nor μ_X actually does anything to the term, they just change its type.

4.3 The Option Monad

The *option monad* consists of the option functor $Opt : \mathsf{Set} \to \mathsf{Set}$ with $\eta_X(x) = \mathsf{Some}\ x$, $\mu_X(\mathsf{Some\ Some}\ x) = \mathsf{Some}\ x$, and $\mu_X(\mathsf{Some\ None}) = \mu_X(\mathsf{None}) = \mathsf{None}$. Kleisli composition in this monad can be implemented in OCaml as follows:

```
let compose (f : 'a -> 'b option) (g : 'b -> 'c option) : 'a -> 'c option =
  fun (z : 'a) -> match f z with
  | Some x -> g x
  | None -> None
```

4.4 The List Monad

In OCaml, `'a list` is the type of lists of objects of type `'a`. This gives a monad with η the polymorphic function `fun x -> [x]` of type `'a -> 'a list` and μ the polymorphic function `List.flatten` with type `'a list list -> 'a list`. Kleisli composition is

```
let compose (f : 'a -> 'b list) (g : 'b -> 'c list) : 'a -> 'c list =
  fun (x : 'a) -> List.flatten (List.map g (f x))
```

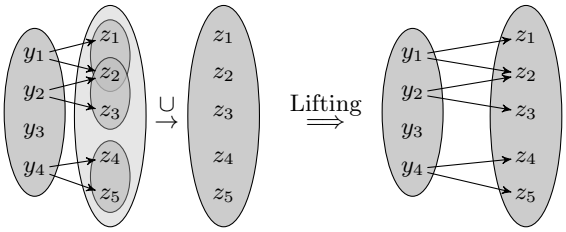
4.5 Closure Operators

Let $(X, \leq)$ be a partially ordered set regarded as a category. The monads on X are just the *closure operators*: functions $\mathrm{cl} : X \to X$ such that $x \leq \mathrm{cl}\, x$, $x \leq y \Rightarrow \mathrm{cl}\, x \leq \mathrm{cl}\, y$, and $\mathrm{cl}\,\mathrm{cl}\, x = \mathrm{cl}\, x$.

5 Lifting

If (F, η, μ) is a monad, then a morphism $f : A \to FB$ of the Kleisli category can be lifted to $f^\dagger : FA \to FB$ by: $f^\dagger \triangleq \mu_B \circ Ff$.

$$FA \xrightarrow{Ff} F^2B \xrightarrow{\mu_B} FB.$$



The lifting operation $(\cdot)^\dagger$ is invertible, as f can be recovered from $f^\dagger$ by composing it on the right with η_A. Thus the Kleisli category with Kleisli composition is isomorphic to the subcategory with objects FA and lifted morphisms under ordinary composition.

In the example involving nondeterministic finite automata above, lifting is just the subset construction that constructs an equivalent deterministic automaton. For the term monad, it constructs a substitution operator on terms from a given substitution. It is strange to think that two fundamental operations in vastly different areas turn out to be the same thing at this level of abstraction.

6 Monads in Functional Programming

In functional programs, monads are represented by

- a parameterized type constructor `'a m`, playing the role of the endofunctor;
- a polymorphic function `return : 'a -> 'a m`, playing the role of η; and
- a polymorphic function `bind : 'a m -> ('a -> 'b m) -> 'b m`, serving the same purpose as μ. The function `bind` is denoted `>>=` in Haskell and is used as an infix operator.

Recurrying `bind` to switch the order of its arguments, its type becomes `('a -> 'b m) -> 'a m -> 'b m`. We see then that it just maps a function of type `'a -> 'b m` to its lifted form. Then μ can be obtained from `bind` by setting `'a = 'b m` and applying it to the identity function on `'b m`:

```
let mu (x : 'b m m) : 'b m = bind x (fun y -> y)
```

Similarly, `bind` can be recovered from `mu`, but this computation depends on the definition of `m`. In particular, we need a `map` function for `m` of type `('a -> 'b) -> 'a m -> 'b m`. Category theoretically, this corresponds to the action of the functor `m` on morphisms. Then if we had `mu`, we could define

```
let bind (x : 'a m) (g : 'a -> 'b m) : 'b m = mu (map g x)
```

6.1 Example: Encoding Substitution

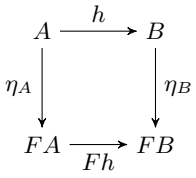
Let us use the term monad over the signature $\Sigma = \{f, g, c\}$ to encode substitution.

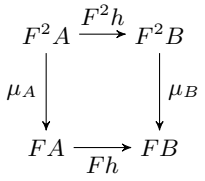
```
(* the type of terms over signature f, g, c with variables 'a *)
type 'a term = Var of 'a | F of ('a term) * ('a term) | G of ('a term) | C

(* the term monad *)
let rec bind (t : 'a term) (g : 'a -> 'b term) : 'b term =
  match t with
   | Var x -> g x
   | F (t1, t2) -> F (bind t1 g, bind t2 g)
   | G t1 -> G (bind t1 g)
   | C -> C

let return (x : 'a) : 'a term = Var x

(* lifting is just applying a substitution *)
let lift (g : 'a -> 'b term) (x : 'a term) : 'b term = bind x g

(* let's try it -- here is a substitution... *)
let s = function
  | 1 -> F (G (Var "y"), C)
  | 2 -> G (F (Var "y", Var "x"))
  | _ -> failwith "Unknown variable"

(* ...and a term to substitute into *)
let t = F (F (Var 1, C), G (Var 2))

# #load "monad.cmo";;
# open Monad;;
# lift s t;;
- : string Monad.term =
F (F (F (G (Var "y"), C), C), G (G (F (Var "y", Var "x"))))
```

Monads allow the programmer to encapsulate a strategy for composing computations involving some extended functionality. For example, the option monad allows the programmer to say uniformly how to compose functions that may not return values.

7 Axioms for Monads

Monads must satisfy the following axioms. First, η and μ must be *natural transformations*, which means they must behave the same way on all objects. Formally, this means that they must commute with all morphisms h of the category. So the following diagrams must commute:

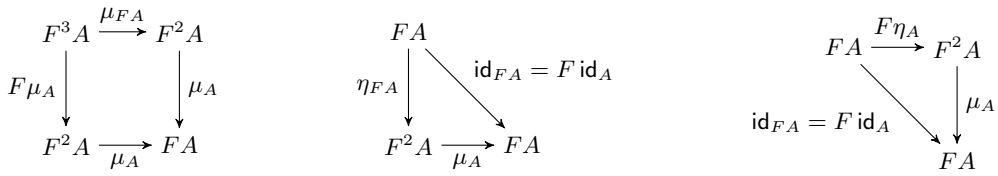




This is just a pictorial representation of the equations

$$\eta_B \circ h = Fh \circ \eta_A \qquad\qquad \mu_B \circ F^2 h = Fh \circ \mu_A.$$

In addition, η and μ must satisfy the following commutative diagrams.



or in other words

$$\mu_A \circ \mu_{FA} = \mu_A \circ F\mu_A \qquad \mu_A \circ \eta_{FA} = \mathsf{id}_{FA} \qquad \mu_A \circ F\eta_A = \mathsf{id}_{FA}.$$

These properties imply that Kleisli composition is associative and the η_A are left and right identities for Kleisli composition.

References

[1] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[2] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer Verlag, August 1992.