1 Monads: What are they and what are they for?

Monads are models of computations that attempt to mimic the so-called **side effects** in functional programming. Side effects are any parts of the program that do not fit into the functional input-output paradigm. For example, algorithms that halt or are undefined at certain inputs, multi-valued functions, commands for data storage, writing, reading, and executing "physical" activities are side effects.

1.1 Working examples of monads

Example 1.1 (Option, exception or maybe monad)**.** Problem: find the `number : nat` associated to a `name : Name` in a list of "objects" `list (nat * Name)`. Use the type former "option" which is defined by

```
Inductive option (A : Type) : Type :=
    | Some : A -> option A
    | None : option A.
```

Here it is (monad_1.v):

```
Fixpoint find_name_return_value (name : Name) (data : list (Name * nat)) : option nat :=
  match data with
    | [] => None
    | dh :: dt => match (Name_eq_dec (let (n,k) := dh in n) name) with
                    | left _ => Some (let (n,k) := dh in k)
                    | right _ => find_name_return_value name dt
                  end
  end.

(*Tests*)
Definition data1 := [ (Anne, 23) ; (Brian,  32) ].

Definition data2 := [ (Anne, 23) ; (Brian, 18) ;  (Anne, 27) ; (Brian, 32) ].

Compute (find_name_return_value Brian data2).
(*       = Some 18
     : option nat*)
```

The program is almost correct, but there is at least one counter-intuitive solution in it. If there are two names with different numbers, it returns only the first numbers and it does not indicate that the result is not unique. To fix this problem, we use type former "list" as a side effect handling tool. List is defined as

```
Inductive list (A : Type) : Type :=
    nil : list A
  | cons : A -> list A -> list A.
```

which is also a monad.

Example 1.2 (List monad)**.** Problem: find the number : nat associated to a name : Name in a list of "objects" list (nat * Name). Do it so that the return value is Some _ : option nat, if the name is unique, None : option nat otherwise.

```
Fixpoint find_name_return_list (name : Name) (data : list (Name * nat)) (stack : list nat) {struct data} : list nat :=
  match data with
    | [] => stack
    | dh :: dt => match (Name_eq_dec (let (n,k) := dh in n) name) with
                    | left _ => find_name_return_list name dt ((let (n,k) := dh in k) :: stack)
                    | right _ => find_name_return_list name dt stack
                  end
  end.

Definition find_name_return_list' (name : Name) (data : list (Name * nat)) := find_name_return_list name data nil.


Compute (find_name_return_list' Brian data2).
(*      = [32; 18]
     : list nat*)

Definition find_name_return_value' (name : Name) (data : list (Name * nat)) : option nat :=
match (find_name_return_list' name data) with
    | [] => None
    | [x] => Some x
    | _ => None
end.

Compute (find_name_return_value' Brian data2).
(*      = None
     : option nat*)
```

1.2 SOGAT definition of monad in CS

In applied computer science, the **type theoretic definition of monads** (rather say Kleisli triad or extension system) are used instead of the categorical theoretic one. However, we can define easily the categorical version despite the fact that it is discovered only in 2010.

So, first, let us see the type theoretic definition. A SOGAT (Second Order Generalized Algebraic Theory) is an algebra that includes 1) multiple sorts, say Set, Type, or Type → Prop... (G in SOGAT), 2) equations or computational rules (AT in SOGAT), and 3) rules about first-order functions or functionals like $A \to B$ (S in SOGAT). The word "second" means we are not just on the level of types but we refer to functions of that type.

Remark 1.1 (Boxing-Unboxing)**.** Intuitively, a monad is a "boxing" strategy to handle side effect: we model and put the side effects into the input or the output of an algorithm as a gadget in a box. For example in case of option monad, if we have any type A and an inhabitant a of A, then if we want to, we put this element into the argument of Some:

$$\text{Some } a\text{, or graphically } \boxed{a} \qquad :)$$

which is a member of the new type "option A". If we don't want to refer to any inhabitants of A, then the output will be

$$\text{None, or graphically } \boxed{} \qquad :)$$

By this solution we avoid the non-well-defined cases, and the possible partially defined functions become totally defined ones. For this, see the examples above.

Definition 1.1 (Monad in CS, Extension System)**.** Let M : Type → Type be a map sending any type A to type MA. Let

$$\text{unit}_A : A \to MA$$

(or return_A) and

$$\text{bind}_{A,B} : (A \to MB) \to (MA \to MB)$$

be functions. M, unit_, and bind_, _ grouped together is called a monad, if the following hold

$$\begin{aligned}
\text{bind}_{A,A}\text{unit}_A &= \text{id}_A \\
(\text{bind}_{A,B}g) \circ \text{unit}_A &= g \qquad (g : A \to MB) \\
(\text{bind}_{B,C}f) \circ (\text{bind}_{A,B}g) &= \text{bind}_{A,C}((\text{bind}_{B,C}f) \circ g) \qquad (g : A \to MB, f : B \to MC)
\end{aligned}$$

♣

Remark 1.2 (Traditional bind and unboxing)**.** Mind that here $\text{bind}_{A,B} : (A \to MB) \to (MA \to MB)$ is rather the *extension operator*. Traditionally, bind is set up in the following form:

$$\text{bind}^{\text{trad}}_{A,B} : MA \to (A \to MB) \to MB$$

In this setup, bind first unboxes an MA, then performs the function $A \to MB$, then returns an MB:

$$\text{bind}^{\text{trad}}_{A,B} \sim (\boxed{a} : MA) \overset{\text{unbox}}{\mapsto} (a : A) \overset{f:A\to MA}{\mapsto} f(a) = \boxed{b} : MB$$

The traditional $\text{bind}^{\text{trad}}$ operator is denoted as

$$(a >>= f)$$

when $a : A$ and $f : A \to MB$.

Remark 1.3 (Commutative properties)**.** The equations above can be interpreted better, if one knows that they correspond to some property of a commutative diagram. In category Type, the following diagram commutes.

$$\begin{tikzcd}
A \arrow[r, "\text{unit}"] \arrow[dr, "g"'] & MA \arrow[loop above, "\text{id}"] \arrow[d, "\text{bind } g"] \\
B \arrow[r, "\text{unit}"] \arrow[dr, "f"'] & MB \arrow[d, "\text{bind } f"] \\
& MC
\end{tikzcd}$$

Right identity rule: $\text{bind}\,\text{unit} = \text{id}$

Left identity rule: $(\text{bind}\, g) \circ \text{unit} = g$

Associative rule: $(\text{bind}\, f) \circ (\text{bind}\, g) = \text{bind}\,((\text{bind}\, f) \circ g)$

Remark 1.4 (The SOGAT as typing rules)**.** Every SOGAT definition, as well as the latter one, can be rewrite as a deduction system, the following is a setup of monad as a proof system.

Type formation rule:

$$\frac{A : \text{Type}}{MA : \text{Type}}$$

Construction rules:

$$\frac{\vdash A : \text{Type} \qquad \vdash a : A}{\vdash \text{unit}\, a : MA}$$

$$\frac{\vdash A : \text{Type} \qquad a : A \vdash f : MB \qquad \vdash ma : MA}{\vdash \text{bind}(a.f; ma) : MB}$$

Computational rules:

$$\text{bind}(a.\text{unit}\, a; ma) = ma$$

$$\text{bind}(a'.g; \text{unit}\, a) = g[a \to a']$$

$$\text{bind}(b.f; \text{bind}(a.g; ma)) = \text{bind}(a.\text{bind}(b.f; g); ma)$$

Remark 1.5 (Coq implementation)**.** Also, one can give the Coq implementation:

```
(*Monad in CS, Extension system*)
Structure Monad : Type := mk_monad
{
  (*sort*)
  M : Type -> Type;

  (*operators*)
  bind : forall {A B : Type}, (A -> M B) -> M A -> M B;
  unit : forall {A : Type}, A -> M A;

  (*equations*)
  left_id_law : forall (A B : Type) (a : A) (f : A -> M B),
                  bind f (unit a) = f a;

  right_id_law : forall (A : Type) (ma : M A),
                  bind (unit) ma = ma;

  assoc_law : forall (A B C : Type) (ma : M A) (g : A -> M B) (f : B -> M C),
                  bind f (bind g ma) = bind (fun x => bind f (g x)) ma
}.
```

Example 1.3 (Option is a monad)**.** By the definition of extension system prove that the polymorph type former "option" is indeed a monad.

```
Definition unit_opt (A : Type) := fun (a : A) => Some a.

Definition bind_opt (A B: Type) := fun (f : A -> option B) (ma : option A) =>
                    match ma with
                      | Some a => f a
                      | None => None
                    end.


Theorem Option_is_a_Monad : Monad.
Proof.
apply mk_monad with (M := option) (bind := bind_opt) (unit := unit_opt).
  - intros; simpl; auto.
  - intros; induction ma; compute; auto.
  - intros; induction ma; compute; auto.
Qed.
```

2 Categories

Via the celebrated Curry–Howard–Lambek correspondence, any pure typed functional programming language has an algebraic representations among special kinds of categories, hence we need to know what a category is. A category as an algebraic theory is a generalized one, since it has two sorts: the *objects* and the *morphisms.* A category can be represented as a graph, the objects are the nodes, the morphisms are the edges or arrows. The operations are defined in the sort of morphisms, and they depend on objects. The first one is the nullary operation id_A for every node A. The second is the binary operation $\circ$, between any arrows f from B to C and g from A to B. id_A is a kind of neutral element, and $\circ$ is associative.

$$\begin{tikzcd}
A \arrow[loop left, "id_A"] \arrow[r, "h"] \arrow[rd, "g\circ h"] \arrow[rdd] & B \arrow[d, "g"] \arrow[dd, bend left, "f\circ g"] & (f\circ g)\circ h = f\circ(g\circ h) \\
id_A\circ h = h & C \arrow[d, "f"] & \\
 & D &
\end{tikzcd}$$

Mind that object are not necessarily sets, they might be anything. In fact the main thought about objects is the you shouldn't look into an object. As Milewski puts it:

> Category theory is extreme in the sense that it actively discourages us from looking inside the objects. An object in category theory is an abstract nebulous entity. All you can ever know about it is how it relates to other objects — how it connects with them using arrows. [...] The moment you have to dig into the implementation of the object in order to understand how to compose it with other objects, you've lost the advantages of your programming paradigm. [Milewski, 2019]

Instead of Set Theory, below we use Type Theory as a frame theory. In this world the sort "Type" is the largest universe containing all the data types, and $x : A$ means "x is of the type A". The dependent product type $\prod_{x:A} B$ is used to functions mapping A into B, more precisely, let $B : A \to \mathrm{Type}$, then $B\,x$ is a type and if

$$f : \prod_{x:A} B\,x$$

then for all $x : A$:

$$f\,x : B\,x.$$

A special sort is Prop which is the type of propositions. When $B\,x$: Prop, then

$$\mathrm{cons} : \prod_{x:A} B\,x$$

means cons is the proof for the proposition "for all x in A, proposition $B\,x$ holds". For example

$$\mathrm{comm} : \prod_{n,m:\mathrm{nat}} n+m=m+n$$

means naturals commute.

Definition 2.1. A category C consists of the following things:

- Ob_C : Class   (class of all objects)
- UHom_C : Class   (class of all morphism or arrows)
- $\mathrm{Hom} : \mathrm{Ob}_C \to \mathrm{Ob}_C \to \mathrm{UHom}_C$ ($\mathrm{Hom}(a,b)$ is the class of arrows from a to b; $f : \mathrm{Hom}(a,b)$ might be written as $f : a \to b$)
- $\mathrm{id} : \prod_{a\,:\,\mathrm{Ob}_C} \mathrm{Hom}(a,a)$ (identity morphism)
- $\mathrm{comp} : \prod_{a,b,c\,:\,\mathrm{Ob}_C} \mathrm{Hom}(a,b) \to \mathrm{Hom}(b,c) \to \mathrm{Hom}(a,c)$ (composition $f \circ g :=$ comp f g)

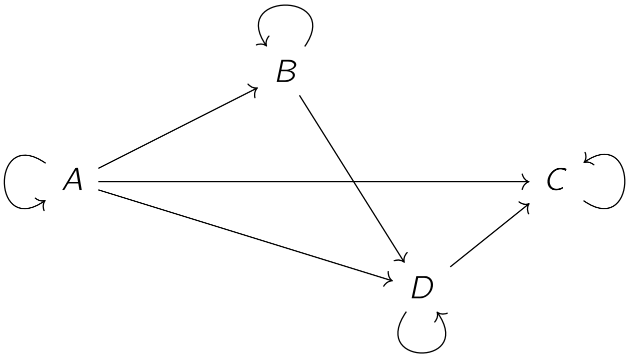
with the rules

$$\prod_{a,b\,:\,\mathrm{Ob}_C, f\,:\,\mathrm{Hom}(a,b)} f \circ \mathrm{id}_a = f;$$

$$\prod_{a,b\,:\,\mathrm{Ob}_C, g\,:\,\mathrm{Hom}(a,b)} \mathrm{id}_b \circ g = g;$$

$$\prod_{a,b,c,d\,:\,\mathrm{Ob}_C, h\,:\,\mathrm{Hom}(a,b), g\,:\,\mathrm{Hom}(b,c), f\,:\,\mathrm{Hom}(c,d)} f \circ (g \circ h) = (f \circ g) \circ h.$$

Example 2.1. a) *Directed graphs* provided that they possess all the loops and the edges from $e : A \to C$ when there are the edges $e_1 : A \to B$ and $e_2 : B \to C$. Here $\mathrm{Ob}(C) =$ Vertices, $\mathrm{UHom}(C) =$ Edges.



b) What is equivalent: *pre-ordered sets* (posets), in their directed graph representations:

$$a \leq b \qquad \iff \qquad \exists f : a \to b$$

(Binary relations over a set, where the relation is reflexive and transitive.)
c) Associative algebraic structure $(M, +)$ with identity element,

$$\forall a, b, c \in M \qquad (a+b)+c = a+(b+c), \qquad a+0 = 0+a = a$$

that is *monoids*. Here $\mathrm{Ob}(C) = \{*\}$ (a singleton) and $M = \mathrm{UHom} = \mathrm{Hom}(*, *)$.

For us one of the most frequently applied notion is the initial object in a category.

Definition 2.2. The object I is *initial* in the category C, if for every object A in C, there is a unique morphism $u : I \to A$.

$$\begin{tikzcd} I \arrow[r, dashed, "u"] & A \end{tikzcd}$$

(The *dual* is *terminal* $T : C$, that is, if for every $A : C$, there is a unique morphism $v : A \to T$.

$$\begin{tikzcd} A \arrow[r, dashed, "v"] & T \end{tikzcd}$$

)

Example 2.2. a) In Set (objects: sets, morphisms: functions), the empty set have the prperty that

$$\emptyset = u : \emptyset \to S$$

is unique, and for every set S, the function

$$v : S \to \{*\}$$

is unique. The singletons are terminal objects, the empty set is the initial object. Both are based on fortiori arguments.
b) Initial object in a poset $(P, \leq)$ is the least element m

$$\forall a \in P \qquad m \leq a$$

c) In Grp the trivial group

$$\{0\}$$

is both initial and terminal object (it is called a *zero object*). The latter is trivial, the former is due to the fact that the only group-morphism which maps $\{0\}$ to a group G is the $0 \mapsto 0$.

Definition 2.3. The morphism $f : X \to Y$ is isomorphism from X to Y, if there is a morphism $g : Y \to X$ such that $f \circ g = id_Y$ and $g \circ f = id_X$. In this case we write $X \overset{f}{\cong} Y$ or if the morphism is known, $X \cong Y$.

Remark 2.1. The initial object in a category is *unique up to isomorphism.*

$$\begin{tikzcd} I \arrow[loop left, "id_I"] \arrow[r, bend left, dashed, "u"] & J \arrow[l, bend left, dashed, "v"] \arrow[loop right, "id_J"] \end{tikzcd}$$

If I and J are initial objects as well, then id_I and id_J are unique, hence $u \circ v = id_J$ and $v \circ u = id_I$. What means that u is an isomorphism.

Now, we turn to a “hybrid” definition of syntax, still incorporated the universal algebraic notion of algebraic structure.

Definition 2.4. The *syntax* generated by the set X over the signature τ is the initial object of the category of all algebras generated by the set X (also called variables) over the signature τ. (Also called anarchic algebra or term algebra.)

The pure categorical syntax definition is based on the notion of F-algebras. But first, let us define functors. Functors are homomorphisms between categories.

Definition 2.5. Let C, D be categories. A functor $F : C \to D$ is a pair (F_O, F_H) of mappings $F_O : \mathrm{Ob}(C) \to \mathrm{Ob}(D)$ and $F_M : \mathrm{UHom}(C) \to \mathrm{UHom}(D)$ (indexes are not used further) such that

$$F(id_X) = id_{F(X)}$$

$$F(f \circ g) = F(f) \circ F(g)$$

$X, Y, Z : \mathrm{Ob}(C), g : X \to Y, f : Y \to Z$.
Hence, the following diagram commutes.

$$\begin{tikzcd} C \arrow[rr, "F"] & & D \end{tikzcd}$$

$$\begin{tikzcd} X \arrow[d, "g"'] \arrow[dr, "f \circ g"] & \\ Y \arrow[r, "f"'] & Z \end{tikzcd} \qquad \begin{tikzcd} F(X) \arrow[d, "F(g)"'] \arrow[dr, "F(f \circ g)"] & \\ F(Y) \arrow[r, "F(f)"'] & F(Z) \end{tikzcd}$$

There are incredibly many functors, from power set functor to dual functor of linear spaces.

Definition 2.6. 1. If C is a category, and $F : C \to C$ is an endofunctor of C, then an F *F-algebra* is a pair (A, α), where $A : \mathrm{Ob}(C)$ and

$$\alpha : F(A) \to A$$

(A is called the *carrier* of the F-algebra).

2. The *category of F-algebras* in C consists of the the F-algebras as objects and morphisms $f : A \to B$ such that the following diagram commutes.

$$\begin{tikzcd} F(A) \arrow[r, "\alpha"] \arrow[d, "F(f)"'] & A \arrow[d, "f"] \\ F(B) \arrow[r, "\beta"'] & B \end{tikzcd}$$

Example 2.3. a) If C has a terminal object 1, then by the functor

$$F(X) = 1$$

with the morphism

$$\alpha : 1 \to X$$

$(1, \alpha : 1 \to X)$ is an F-algebra (1 is the date type True in typed programming language). Especially, in Set, $1 = \{*\}$, and $\alpha = \{(*, x)\}$.
b) *Sum object* (Motivation: $F(X) = 1 \coprod 1$ the bi-pointed object, which is the data type Boole in the typed programming language.) In Set, for all set A, B, the set

$$A \coprod B = \{(n, x) \mid (n = 1 \wedge x \in A) \vee (n = 2 \wedge x \in B)\}$$

and the functions $l : A \to A \coprod B, a \mapsto (1, a)$ and $r : B \to A \coprod B, b \mapsto (2, b)$ possess the property that for any set C and functions $l' : A \to C$, $r' : B \to C$ there exists a unique function $u : A \coprod B \to C$ such that the following diagram commutes

$$\begin{tikzcd}
A \arrow[r, "l"] \arrow[dr, "l'"'] & A\coprod B \arrow[d, dashed, "u"] & B \arrow[l, "r"'] \arrow[dl, "r'"] \\
 & C &
\end{tikzcd}$$

$$u \circ l = l' \qquad u \circ r = r'$$

Since, $u(x) = \begin{cases} l'(a) & x = (1, a) \\ r'(b) & x = (2, b) \end{cases}$ uniquely defined.

c) *Product object* (Motivation: $F(X) = 1\coprod(X\prod X)$ is the Tree data type in typed programming languages.) In Set, for all set A, B, the set

$$A\prod B = \{(a, b) \mid a \in A \wedge b \in B\}$$

and the functions $pr_1 : A\prod B \to A, (a, b) \mapsto a$ and $pr_2 : A\prod B \to B, (a, b) \mapsto b$ possess the property that for any set C and functions $f : C \to A$, $g : C \to B$ there exists a unique function $u : C \to A\prod B$ such that the following diagram commutes

$$\begin{tikzcd}
A & A\prod B \arrow[l, "pr_1"'] \arrow[r, "pr_2"] & B \\
 & C \arrow[ul, "f"] \arrow[u, dashed, "u"'] \arrow[ur, "g"'] &
\end{tikzcd}$$

$u(x) = (f(x), g(x))$

Definition 2.7. Let F be an endofunctor in the category C. The *syntax* of the signature F is the initial algebra (object)

$$(X, \alpha : F(X) \to X)$$

in the category of F-algebras of F in C. (Here α is called the system of constructors of X.)

Theorem 2.1 (Lambek's Theorem)**.** If $(X, \alpha : F(X) \to X)$ is the initial F-algebra of the endofunctor F, then α is an isomorphism from $F(X)$ to X:

$$F(X) \cong X.$$

Hence, X is the fixpoint of F.

Proof. Let $I = (X, \alpha : F(X) \to X)$ be the initial F-algebra. Let us consider the F-algebra $(F(X), F(\alpha) : F(F(X)) \to F(X))$. I is initial, so there is a unique $i : X \to F(X)$ such the following diagram commutes:

$$\begin{tikzcd}
F(X) \arrow[r, "\alpha"] \arrow[d, "F(i)"'] & X \arrow[d, dashed, "i"] \\
F(F(X)) \arrow[r, "F(\alpha)"'] & F(X)
\end{tikzcd}$$

Now,

$$\alpha \circ i = id_X, \quad (*)$$

since id_X and $\alpha \circ i$ are both $X \to X$ morphisms and F-algebras, and X is initial. Hence, by (*) functority and F-algebra morphism properties:

$$id_{F(X)} = F(id_X) = F(\alpha \circ i) = F(\alpha) \circ F(i) = i \circ \alpha$$

□

3 Simple Types

One of the most important properties of typed, functional programming languages is that they form a category. The simplest model of the typed, functional programming language is the so-called Simple Type Theory (STT). Here the terms of the language are functions ($\lambda x.p$) and applications of functions ($p(q)$). The compound types are the function types of the form $A \to B$. In this context, functions become morphisms and types become objects of the category of programs. In the following, we will define both the STT programming language and the notion of category, and prove that STT forms a category in the above sense.

Definition 3.1 (STT-)**.** The following inference system is a fragment of Simple Type Theory (STT-). Types are defined by recursively as

$$\frac{}{\iota : \text{Type}} \qquad \frac{A : \text{Type} \quad B : \text{Type}}{A \to B : \text{Type}}$$

The contexts or the variable declarations are lists of types, this claim is denoted by Γ : Cnxt. The formation rules for contexts:

$$\frac{}{: \text{Cnxt}} \qquad \frac{\Gamma : \text{Cnxt} \qquad A : \text{Type}}{A, \Gamma : \text{Cnxt}}$$

The constructors of the types are:

$$\frac{\Gamma : \text{Cnxt} \qquad A\; B : \text{Type} \qquad A, \Gamma \vdash p : B}{\Gamma \vdash \lambda\, A\, p : A \to B}$$

$$\frac{\Gamma : \text{Cnxt} \qquad A\; B : \text{Type} \qquad \Gamma \vdash p : A \to B \qquad \Gamma \vdash q : A}{\Gamma \vdash p\$q : B}$$

$$\frac{\Gamma : \text{Cnxt} \qquad A : \text{Type}}{A, \Gamma \vdash \text{hyp}\,0 : A}$$

$$\frac{n : \text{nat} \qquad \Gamma : \text{Cnxt} \qquad A\ B : \text{Type} \qquad \Gamma \vdash \text{hyp}\,n : B}{A, \Gamma \vdash \text{hyp}\,\text{S}\,n : B}$$

The ternary relation $\Gamma \vdash p \equiv q : A$ models the computations. The equality rules for computation:

$$\frac{}{\Gamma \vdash p \equiv p : A} \quad \frac{\Gamma \vdash p \equiv q : A}{\Gamma \vdash q \equiv p : A} \quad \frac{\Gamma \vdash p \equiv q : A \qquad \Gamma \vdash q \equiv r : A}{\Gamma \vdash p \equiv r : A}$$

And the computational rule

$$\frac{}{\Gamma \vdash (\lambda\ A\ p)\$q \equiv p[q] : B}$$

The above setup is enough to prove that STT is a category.

Definition 3.2 (Category with morphism equality)**.**

References

[Milewski, 2019] Milewski, B. (2019). *Category theory for programmers*. Bartosz Milewski.