école normale supérieure paris—saclay

Monads

Félix Yvonnet

December 5, 2023

Introduction

> "A monad is just a monoid in the category of endofunctors, what's the problem?"
>
> — Saunders Mac Lane — James Iry

Outline

Functional Programming

Functional Programming

A **side effect** is when a function **relies on**, or **modifies**, something **outside its parameters** to do something

Pure functions

Definition

A function is **pure** if, given the same inputs,

a) always returns the same output

b) does not have any side effects

Pureness

```ocaml
let var = ref 1
let impure (var: int ref): int =
    let var := !var * 2 in
    !var

(* 2 *)
let _ = print_int (impure (var))
(* 4 *)
let _ = print_int (impure (var))
```

```ocaml
let var = ref 1
let pure (var: int ref): int =
    let m = 2 in
    m

(* 2 *)
let _ = print_int (pure (var))
(* 2 *)
let _ = print_int (pure (var))
```

Advantages

- Easier to test

Advantages

- Easier to test
- Easier to run in parallel

Advantages

- Easier to test
- Easier to run in parallel
- Predictable

Advantages

- Easier to test
- Easier to run in parallel
- Predictable
- ...

Problem

But it's necessary: I/O operations, databases, probabilistic aspects of computer science, exceptions…

All Hail Functional Programming

Define a structure to handle these cases

A Solution ?

```
let impure (): int =
    let v = 3+f() in
    v
```

A Solution ?

```
let purer (f: ()->int): int =
    let v = 3+f() in
    v
```

A Solution ?

```
let purerer ((+): int->int)
            (f: ()->int): int =
    let v = 3+f() in
    v
```

The Solution

Monad

Overview

A bit of history

- 1958 by Roger Godement for category theory

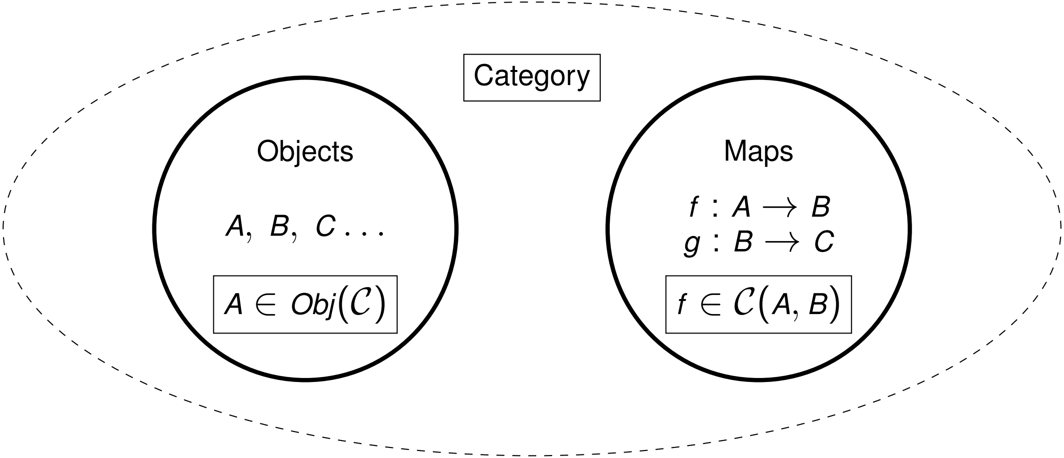
A bit of history

- 1958 by Roger Godement for category theory
- 1980s – 1990s in programming

A bit of history

- 1958 by Roger Godement for category theory
- 1980s – 1990s in programming
- Wait, seriously ? Theoretical computer science is actually usefull ?!

Category


Category
Objects
$A, \; B, \; C \ldots$
$A \in Obj(\mathcal{C})$
Maps
$f : A \to B$
$g : B \to C$
$f \in \mathcal{C}(A, B)$

Rules

- Composition: $g \circ f$ is a map
- Don't worry about order: $\left(g \circ f\right) \circ h = g \circ \left(f \circ h\right)$
- There is id_A for all A (with id properties: $id_B \circ f = f = f \circ id_A$)

Examples

1 — category

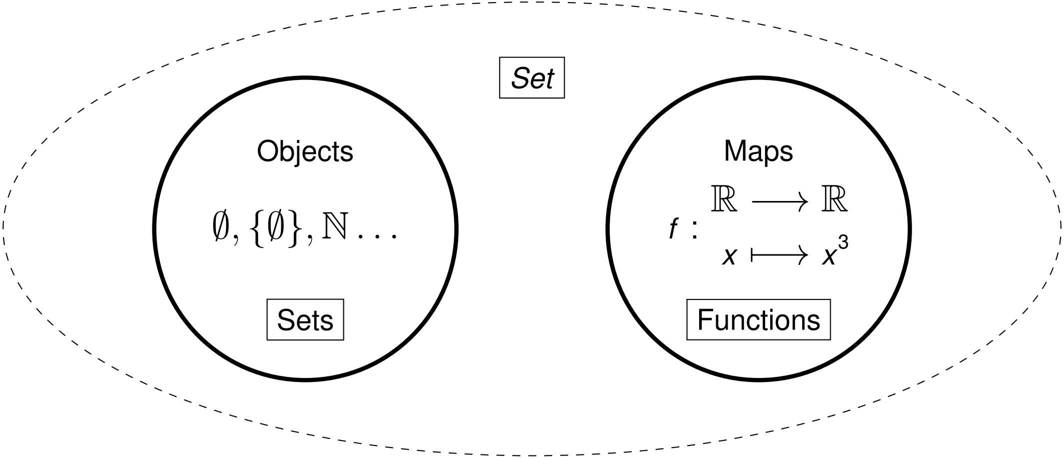
$$\begin{tikzcd} 1 \arrow[loop above, "id_1"] \end{tikzcd}$$

Examples

2—category

$$\begin{tikzcd} 1 \arrow[loop above, "id_1"] \arrow[r, bend right, "g"'] & 2 \arrow[loop above, "id_2"] \arrow[l, bend right, "f"'] \end{tikzcd}$$

Examples


Set
Objects
$\emptyset, \{\emptyset\}, \mathbb{N} \dots$
Sets
Maps
$f : \mathbb{R} \longrightarrow \mathbb{R}$
$x \longmapsto x^3$
Functions

Examples

In programming:

```
module ListCategory = struct
    type A = int
    type B = int list

    let to_list (a: A): B = [a]
    let len (b: B): A = List.lenght b
    let id_A (a: A) = a
    let id_B (b: B) = b
end
```

Functors

Transforms elements from a category to another preserving the structure

Functors

$F : \mathcal{C} \to \mathcal{D}$ is a functor iff

$$A \in Obj(\mathcal{C}) \xrightarrow{\quad F \quad} FA \in Obj(\mathcal{D})$$

$$f \in \mathcal{C}(A, B) \xrightarrow{\quad F \quad} Ff \in \mathcal{D}(FA, FB)$$

école normale supérieure paris-saclay

Rules

$F : \mathcal{C} \to \mathcal{D}$ is a functor iff

$$F(id_A) = id_{FA}$$

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

Examples

For $\mathcal{C}$ a category:

Examples

For $\mathcal{C}$ a category:

$$\begin{aligned} && \mathcal{C} &\longrightarrow \mathcal{C} \\ F = id_{\mathcal{C}} : && A \in Obj(\mathcal{C}) &\longmapsto A \\ && f \in \mathcal{C}(A, B) &\longmapsto f \end{aligned}$$

Examples

$$F' : \begin{array}{c} Set \longrightarrow \textbf{Set} \\ A \in Obj(\textbf{Set}) \longmapsto \mathcal{P}(A) = \{x \mid x \subset A\} \\ f \in \textbf{Set}(A, B) \longmapsto \begin{pmatrix} \mathcal{P}(A) \longrightarrow \mathcal{P}(B) \\ V \longmapsto f(V) \end{pmatrix} \end{array}$$

Examples

In programming:

```
module type Ordered = sig
  type a
  val compare : a -> a -> a
  val id_a: a -> a
end
```

Examples

In programming:

```
 module OrderedInt = struct
  type a = int
  let compare x y =
    if x > y then 1
    else (
        if x=y then 0
        else -1
    )
  let id_a x = x
end
```

Examples

In programming:

```
module Inverse (M : Ordered) : Ordered =
  struct
    type a = M.a
    let compare x y = M.compare y x
    let id_a x = x
end

module InvOrderInt = Inverse(OrderedInt)
```

école normale supérieure paris-saclay

Endofunctors

$F : \mathcal{C} \to \mathcal{C}$ is an endofunctor

Monoid

- Take your favorite set $(\mathbb{N}, \{0\} \ldots)$ called *M*

Monoid

- Take your favorite set $(\mathbb{N}, \{0\} \ldots)$ called M
- Choose a particular element called e

Monoid

- Take your favorite set $(\mathbb{N}, \{0\} \ldots)$ called M
- Choose a particular element called e
- Add a bit of structure

Monoid

- Take your favorite set $(\mathbb{N}, \{0\} \ldots)$ called M

- Choose a particular element called e

- Add a bit of structure
  - $* : M \times M \to M$

Monoid

- Take your favorite set $(\mathbb{N}, \{0\} \ldots)$ called M
- Choose a particular element called e
- Add a bit of structure
  - $* : M \times M \to M$
  - $\forall m \in M,\ m * e = m = e * m$

Monoid

- Take your favorite set $(\mathbb{N}, \{0\} \ldots)$ called M
- Choose a particular element called e
- Add a bit of structure
  - $* : M \times M \to M$
  - $\forall m \in M,\ m * e = m = e * m$
- $(M, e, *)$ is called a monoid

Examples

- $(\mathbb{N}, 0, +)$
- $(\mathbb{R}, 1, \times)$
- $(\mathcal{P}(X), \emptyset, \cup)$

Examples

- $(\mathbb{N}, 0, +)$
- $(\mathbb{R}, 1, \times)$
- $(\mathcal{P}(X), \emptyset, \cup)$
- $(\{F : \mathcal{C} \to \mathcal{C}\}, id_{\mathcal{C}}, \circ)$

First Monad

First Monad

$(\{id_{\mathcal{C}}\}, id_{\mathcal{C}}, \circ)$ is a monad

Power Set Monad

$F_{\textbf{Set}\to\textbf{Set}}$ the functor that maps $A \mapsto \mathcal{P}(A)$ and $f \mapsto (V \mapsto f(V))$

Power Set Monad

$F_{\mathbf{Set}\to\mathbf{Set}}$ the functor that maps $A \mapsto \mathcal{P}(A)$ and $f \mapsto (V \mapsto f(V))$

$(\{F^k \mid k \in \mathbb{N}\}, id_{\mathcal{C}}, \circ)$ is a monad

école normale supérieure paris-saclay

Maybe Monad

$$F^*_{\mathbf{Set}\to\mathbf{Set}} \begin{cases} A \longmapsto A \cup \{*\} \\ f_{A\to B} \longmapsto \begin{pmatrix} x \in A \mapsto x \\ * \mapsto * \end{pmatrix} \end{cases}$$

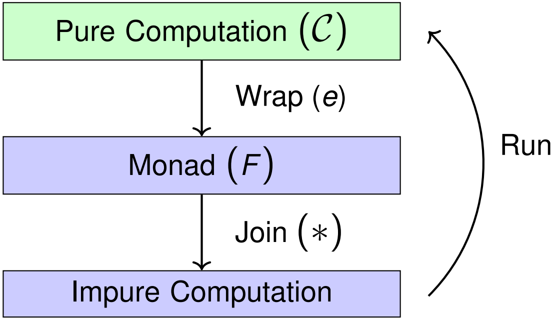
$$\left(\left\{F^*, id_{\mathbf{Set}}\right\}, id_{\mathbf{Set}}, \circ\right)$$

Overview

Use cases

- when the access to a value is not possible (I/O, Errors. . .)
- when the access to a value should be limited (List, pointers. . .)
- the monad represent a context one can access

Abstraction



In OCaml

```
module type Monad = sig
    type 'a t
    val return : 'a -> 'a t
    (* bind *)
    val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
end
```

Bind

No join ?

$$\textit{bind} \equiv \textit{join} \text{ and } \textit{map}$$

école normale supérieure paris-saclay

Monad laws

Kleisli triple

```
(return x) >>= f = f x

 f >>= return = f

(h >>= f) >>= g
      = h >>= ( fun x -> (f x >>= g) )
```

école normale supérieure paris–saclay

Log

```
(* Encryption functions *)
let enc x = x + 1
let dec x = x - 1
```

Log

```
let enc_log x = (
    x + 1,
    Printf.sprintf "enc (%i) = %i" x (x+1)
)
let dec_log x = (
    x - 1,
    Printf.sprintf "dec(%i) = %i" x (x+1)
)
```

Log

```
let id x = dec (enc x)

let id_fail x = dec_log (enc_log x)
```

Log

```
module EncWithLog: Monad = struct
  type 'a t = 'a * string
  let return = fun x -> (x, "")










end
```

Log

```
module EncWithLog: Monad = struct
  type 'a t = 'a * string
  let return = fun x -> (x, "")

  let ( >>= ): 'a t -> ('a -> 'b t) -> 'b t =
    fun (x, s1) f ->
        let (y, s2) = f x in
        (y, s1 ^ "\n" ^ s2)




end
```

Log

```
module EncWithLog: Monad = struct
  type 'a t = 'a * string
  let return = fun x -> (x, "")

  let ( >>= ): 'a t -> ('a -> 'b t) -> 'b t =
    fun (x, s1) f ->
        let (y, s2) = f x in
        (y, s1 ^ "\n" ^ s2)

  let log (name: string) (f: 'a->'b): 'a -> 'b t =
    fun x ->
        (f x, Printf.sprintf "Called %s on %i" name x)
end
```

Log

```
open EncWithLog
let x = 3
let enc_log = log "enc" enc
let dec_log = log "dec" dec
let (id, log) =
    (return x) >>=
    enc_log >>=
    dec_log
```

List

```
module List: Monad = struct
  type 'a t = 'a list
  let return = fun (x: 'a) -> [x]





end
```

List

```
module List: Monad = struct
  type 'a t = 'a list
  let return = fun (x: 'a) -> [x]

  let ( >>= ) =
    fun l f = List.fold_left (
                fun acc x -> f x @ acc
          ) [] l
end
```

Conclusion

Monad:

- complex formal definition

Conclusion

Monad:

- complex formal definition
- global concept:

Conclusion

Monad:

- complex formal definition
- global concept:
  - pure code

Conclusion

Monad:

- complex formal definition
- global concept:
  - pure code
  - encapsulation

Conclusion

Monad:

- complex formal definition
- global concept:
  - pure code
  - encapsulation
  - good file organisation I guess?

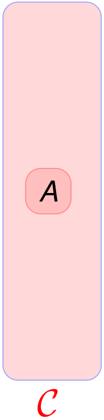
Conclusion

Monad:

- complex formal definition
- global concept:
  - pure code
  - encapsulation
  - good file organisation I guess?
- can be a little verbose sometime

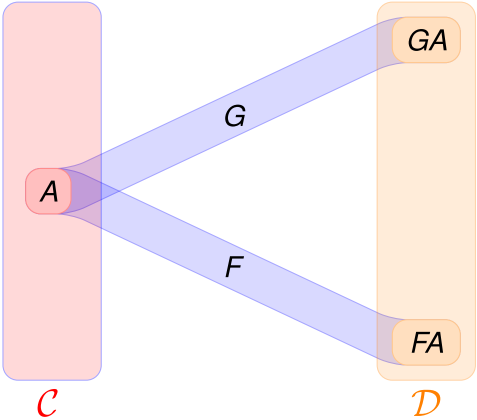
Overview

Natural Transformations



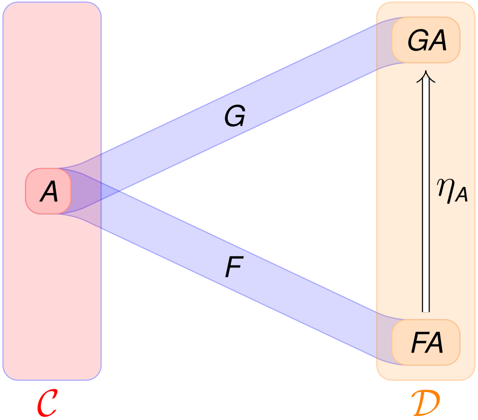
Natural Transformations

For $F, G : \mathcal{C} \to \mathcal{D}$,



Natural Transformations

For $F, G : \mathcal{C} \to \mathcal{D}$, $\eta = (\eta_A)_{A \in Obj(\mathcal{C})} : F \Rightarrow B$ is a natural transformation iff:



Rules

For $F, G : \mathcal{C} \to \mathcal{D}$, $\eta : F \Rightarrow B$ is a natural transformation iff:

$$\begin{tikzcd}
FA \arrow[r, "Ff"] \arrow[d, "\eta_A"'] & FB \arrow[d, "\eta_B"] \\
GA \arrow[r, "Gf"'] & GB
\end{tikzcd}$$

For every map $f \in \mathcal{C}(A, B)$

école normale supérieure paris-saclay

Examples

For $F : \mathcal{C} \to \mathcal{D}$ a functor. $\eta : F \Rightarrow F$?

école normale supérieure paris-saclay

Examples

For $F : \mathcal{C} \to \mathcal{D}$ a functor. $\eta : F \Rightarrow F$?

$$\eta_A = id_{FA} : FA \to FA$$

Examples

- **CRing** category of commutative rings with ring homeomorphisms ie $f(a+b) = f(a) + f(b)$ and $f(a \times b) = f(a) \times f(b)$
- **Grp** category of groups with group homeomorphisms ie $f(a+b) = f(a) + f(b)$

Examples

$$\begin{aligned} * : \quad \mathbf{CRing} &\longrightarrow \mathbf{Grp} \\ A &\longmapsto A^* \text{ the unit group} \\ f_{A \to B} &\longmapsto f_{|A^* \to B^*} \end{aligned}$$

is a functor from **CRing** to **Grp**

Examples

$$\mathbf{CRing} \longrightarrow \mathbf{Grp}$$

$$GL_n : \quad \begin{array}{rcl} A & \longmapsto & GL_n(A) \\ f_{A \to B} & \longmapsto & \begin{pmatrix} GL_n(A) & \longrightarrow & GL_n(B) \\ M = (m_{i,j})_{1 \leq i,j \leq n} & \longmapsto & GL_n(f)(M) = (f(m_{i,j}))_{1 \leq i,j \leq n} \end{pmatrix} \end{array}$$

is a functor from **CRing** to **Grp**

Examples

A natural transformation $\eta : GL_n \Rightarrow (*)$?

Examples

A natural transformation $\eta : GL_n \Rightarrow (*)$?

$(\det_A : GL_n(A) \to A^*)_{A \in \mathbf{CRing}}$ as $f^* \circ \det_A = \det_B \circ GL_n(f)$

Monad

- $\mathcal{C}$ a category
- $T : \mathcal{C} \longrightarrow \mathcal{C}$ an endofunctor
- $\eta : id_{\mathcal{C}} \Rightarrow T$ a natural transformation
- $\mu : T^2 \Rightarrow T$ a natural transformation

Monad

$$
\begin{tikzcd}
T^3X \arrow[r, "T\mu_X"] \arrow[d, "\mu_{TX}"'] & T^2X \arrow[d, "\mu_X"] \\
T^2X \arrow[r, "\mu_X"'] & TX
\end{tikzcd}
\qquad
\begin{tikzcd}
TX \arrow[r, "\eta_{TX}"] \arrow[d, "T\eta_X"'] \arrow[dr, equal] & T^2X \arrow[d, "\mu_X"] \\
T^2X \arrow[r, "\mu_X"'] & TX
\end{tikzcd}
$$

$\mu \circ T\mu = \mu \circ \mu T$ **and** $\mu T\eta = \mu\eta T = id_{\mathcal{C}}$

Link to Monoids

TX are the types, $\mu \equiv e$ and $\eta \equiv *$

Link to programming

T is the type creator, $\mu \equiv$ *return* and $\eta \equiv$ *join*

Comonads

- Dual of monads
- ”whenever you see large datastructures pieced together from lots of small but similar computations there’s a good chance that we’re dealing with a comonad”
- Extract data from a context: Streams
- In category of vector space with tensor product defines coalgebra

Coalgebra

- F : **Set** $\to$ **Set** st $FX = \mathbb{N} \times X$

Coalgebra

- $F : \mathbf{Set} \to \mathbf{Set}$ st $FX = \mathbb{N} \times X$
- $(Stream\mathbb{N}, \alpha)$

Coalgebra

- $F : \mathbf{Set} \to \mathbf{Set}$ st $FX = \mathbb{N} \times X$
- $(Stream\mathbb{N}, \alpha)$
- $\alpha(s) = (head(s), rest(s))$

Coalgebra

- $F : \textbf{Set} \to \textbf{Set}$ st $FX = \mathbb{N} \times X$
- $(\mathit{Stream}\mathbb{N}, \alpha)$
- $\alpha(s) = (\mathit{head}(s), \mathit{rest}(s))$
- defines a stream (stderr, stdin. . .)

More

- Strong monads: no functions but type of functions (val in OCaml)
- do notation for efficient programming language from Kleisli composition
  - “do this, do that, and return the result”
  - do prog $\equiv$ prog
  - do $prog_1$ $prog_2$ $\equiv$ $prog_1$ bind $(\backslash\_ \to prog_2)$
  - do (x $\leftarrow$ $prog_1$) $prog_2$ $\equiv$ $prog_1$ bind $(\backslash x \to prog_2)$