OCaml et la programmation fonctionnelle

Un fabuleux voyage au cœur de l'informatique (et des monades)

Antonin Décimo Carine Morel Pierre Nigron

18 mars 2019

Université Paris Diderot

Table des matières

OCaml : un langage multi-paradigmes

OCaml : un langage fonctionnel

- langage à expressions (et pas à instructions)

OCaml : un langage fonctionnel

- langage à expressions (et pas à instructions)
- fonctions d'ordre supérieur

```
let rec map f l =
match l with
| [] -> []
| h :: t -> f h :: map f t

let l = map (fun x -> x + 1) [1; 2; 3]
(* [2 ; 3 ; 4] *)
```

OCaml : un langage fonctionnel

- langage à expressions (et pas à instructions)
- fonctions d'ordre supérieur

```
let rec map f l =
match l with
| [] -> []
| h :: t -> f h :: map f t

let l = map (fun x -> x + 1) [1; 2; 3]
(* [2 ; 3 ; 4] *)
```

- applications partielles

```
let f x y = x * x + y * y
let g = f 2          (* fun y -> 4 + y * y *)
let b = g 3          (* 13 *)
```

OCaml : un langage fonctionnel

- langage à expressions (et pas à instructions)
- fonctions d'ordre supérieur

```
let rec map f l =
match l with
| [] -> []
| h :: t -> f h :: map f t

let l = map (fun x -> x + 1) [1; 2; 3]
(* [2 ; 3 ; 4] *)
```

- applications partielles

```
let f x y = x * x + y * y
let g = f 2          (* fun y -> 4 + y * y *)
let b = g 3          (* 13 *)
```

- filtrage par motifs (pattern-matching) et types algébriques

```
type 'a list = Nil | Cons of 'a * list
type 'a tree = Leaf of 'a
             | Node of 'a tree * 'a tree
```

OCaml : un langage multi-paradigmes

Un langage impératif !

```
let fresh =
  let c = ref (-1) in
  fun () ->
    c := !c + 1;
    !c

fresh ()  (* - : int = 0 *)
fresh ()  (* - : int = 1 *)
```

Un langage à objets !

OCaml : du typage

- typage statique (effectué à la compilation) ;
- typage fort (pas de transtypage implicite) ;

```
let x = 40 + 2.0
(* Error : This expression has type float but an
   expression was expected of type int *)
```

- inférence de type ;

```
let l = [1; 2]
(* val l : int list = [1; 2] *)
```

- du polymorphisme.

```
let rec map f l =
  match l with
  | [] -> []
  | h :: t -> f h :: map f t;;
(* val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

OCaml : et plus encore !

- compilé ou interprété ;
- ramasse-miettes ;
- un gestionnaire de paquets ;
- possibilité de liaison avec du C.

Le fonctionnel pur

Le fonctionnel pur

Un programme fonctionnel pur *n'a pas d'effets de bord*

```
let x = ref 0
let f y = !x * y
  f 2 (* 0 *)
  x := 21
  f 2 (* 42 *)
```

Quelques effets de bords :

- entrées et sorties (réseau, fichiers, interactions utilisateurs, ...) ;
- les exceptions ;
- affectations sur des données mutables.

Plus généralement : interaction observable avec le monde extérieur

Le fonctionnel pur, pourquoi ?

Comme une fonction mathématique ! La sortie ne dépend que de l'entrée.

Intérêts principaux :

- facilité à raisonner sur le programme ;
- preuves ! (Coq)
- possibilité d'optimisations.

Effets de bord et pureté

Mais, Carine, comment faire des effets de bord dans un langage fonctionnel pur ?

Les Monades

Les Monades

Qu'est-ce ? À quoi ça sert ?

- une façon d'ajouter des effets de bord dans des langages purs (monade IO, monade de non-déterminisme) ;
- un style de programmation avancé, efficace, et modulaire
  - monade d'état, monade liste, monade d'erreur
  - interpréteur monadique, parseur monadique
  - monade de continuation, monade d'exception

Des objets mathématiques (monoïdes dans la 2-catégorie des endo-foncteurs)

Les Monades en théorie

Soient deux types A et B, une monade est :

un constructeur de type M qui construit un type monadique MA ;

un convertisseur de type *return* qui encapsule un objet x dans la monade

$$return : A \to MA$$

un combinateur *bind* qui transporte un objet d'une monade à une autre en le transformant

$$bind : MA \to (A \to MB) \to MB$$

Les Monades en théorie

Soient deux types A et B, une monade est :

un constructeur de type M qui construit un type monadique MA ;

un convertisseur de type *return* qui encapsule un objet x dans la monade

$$return : A \to MA$$

un combinateur *bind* qui transporte un objet d'une monade à une autre en le transformant

$$bind : MA \to (A \to MB) \to MB$$

... satisfaisant les lois :

- (return x) »= f $\equiv$ f x
- m »= return $\equiv$ m
- (m »= f) »= g $\equiv$ m »= (fun x -> f x »= g)

Monade : une signature

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

Exemple 1 : Monade Maybe

La Monade Maybe

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

La Monade Maybe

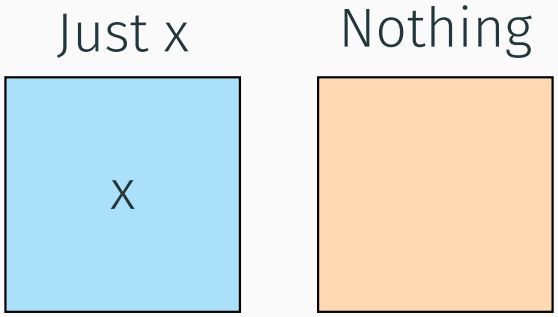
```
module type Monad = sig
 type 'a t
 val return : 'a -> 'a t
 val bind :
 'a t -> ('a -> 'b t) -> 'b t
 val (>>=) :
 'a t -> ('a -> 'b t) -> 'b t
 val _raise : 'a t (* !*)
end
```

```
module Maybe : Monad = struct
  type 'a t = Nothing
            | Just of 'a

  let return x = Just x
  let bind x f = match x with
    | Nothing -> Nothing
    | Just x -> f x
  let (>>=) = bind
  let _raise = Nothing (* !*)
end
```

La Monade Maybe

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

```
module Maybe : Monad = struct
  type 'a t = Nothing
            | Just of 'a

  let return x = Just x
  let bind x f = match x with
    | Nothing -> Nothing
    | Just x -> f x
  let (>>=) = bind
  let _raise = Nothing (* !*)
end
```



La Monade Maybe : utilisation

```
let div x y =
  if y = 0 then Nothing
  else Just (x/y)

let divM x y =
  if y = 0 then _raise
  else return (x/y)
```

La Monade Maybe : utilisation

```
let div x y =
  if y = 0 then Nothing
  else Just (x/y)

let divM x y =
  if y = 0 then _raise
  else return (x/y)

(* (((x / y1) / y2) / y3)) *)
let divM_1 x y1 y2 y3 =
  match divM x y1 with
  | Nothing -> Nothing
  | Just r1 ->
     match divM r1 y2 with
     | Nothing -> Nothing
     | Just r2 -> divM r2 y3
```

La Monade Maybe : utilisation

```
let div x y =
  if y = 0 then Nothing
  else Just (x/y)

let divM x y =
  if y = 0 then _raise
  else return (x/y)

(* (((x / y1) / y2) / y3)) *)
let divM_1 x y1 y2 y3 =
  match divM x y1 with
  | Nothing -> Nothing
  | Just r1 ->
     match divM r1 y2 with
     | Nothing -> Nothing
     | Just r2 -> divM r2 y3
```

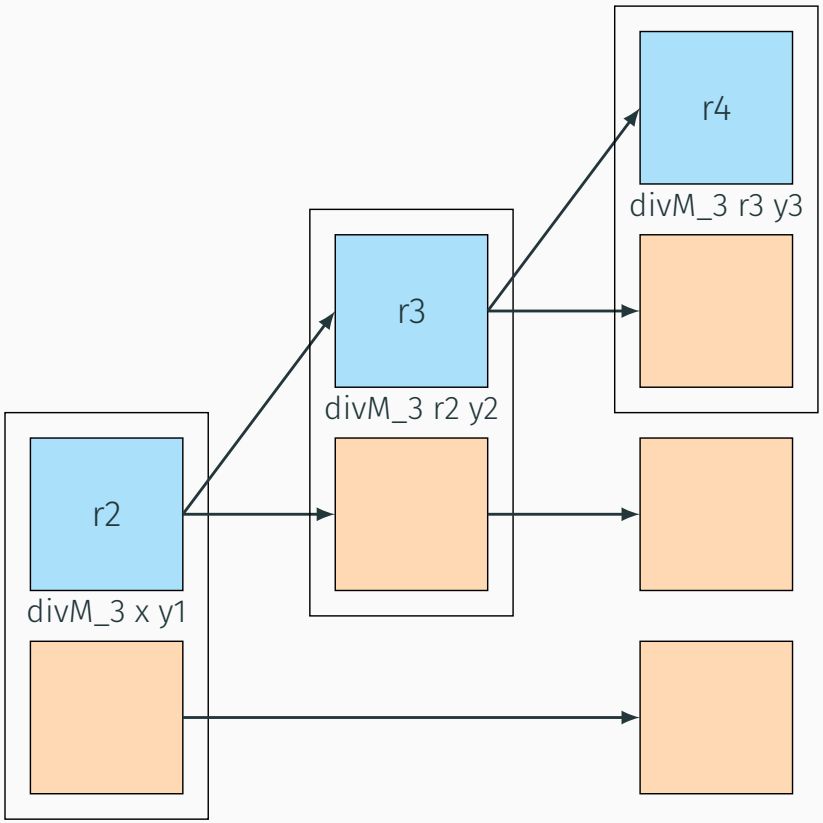
```
let divM_2 x y1 y2 y3 =
  bind (divM x y1)  (fun r2 ->
  bind (divM r2 y2) (fun r3 ->
  divM r3 y3))
```

La Monade Maybe : utilisation

```
let div x y =
  if y = 0 then Nothing
  else Just (x/y)

let divM x y =
  if y = 0 then _raise
  else return (x/y)

(* (((x / y1) / y2) / y3)) *)
let divM_1 x y1 y2 y3 =
  match divM x y1 with
  | Nothing -> Nothing
  | Just r1 ->
     match divM r1 y2 with
     | Nothing -> Nothing
     | Just r2 -> divM r2 y3
```

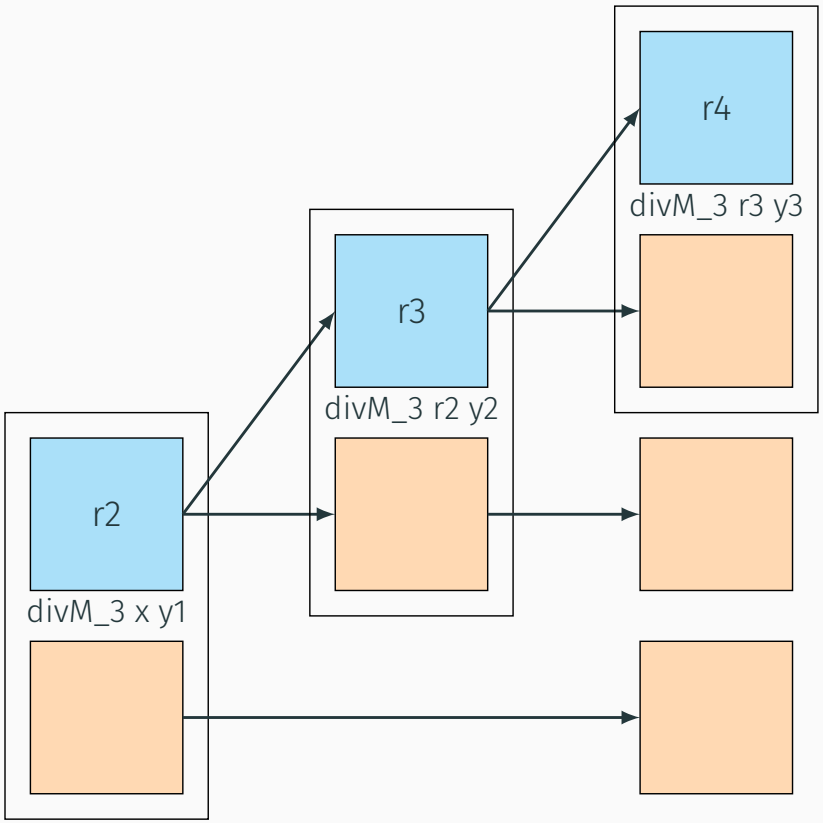
```
let divM_2 x y1 y2 y3 =
  bind (divM x y1)  (fun r2 ->
  bind (divM r2 y2) (fun r3 ->
  divM r3 y3))

let divM_3 x y1 y2 y3 =
  divM x y1
  >>= fun r2 -> divM r2 y2
  >>= fun r3 -> divM r3 y3
```

La Monade Maybe : explications




```
let divM_3 x y1 y2 y3 =
  divM x y1
  >>= (fun r2 -> divM r2 y2
  >>= fun r3 -> divM r3 y3)
```

La Monade Maybe : explications




```
let divM_3 x y1 y2 y3 =
  divM x y1
  >>= (fun r2 -> divM r2 y2
  >>= fun r3 -> divM r3 y3)
```

```
let divM_4 x y1 y2 y3 =
  return x
  >>= fun r1 -> divM r1 y1
  >>= fun r2 -> divM r2 y2
  >>= fun r3 -> divM r3 y3
```

La Monade Maybe : le bilan

```
(* style usuel *)
let divM_1 x y1 y2 y3 =
  match divM x y1 with
  | Nothing -> Nothing
  | Just r1 ->
     match divM r1 y2 with
     | Nothing -> Nothing
     | Just r2 -> divM r2 y3
```

```
(* style monadique *)
let divM_3 x y1 y2 y3 =
  divM x y1
  >>= fun r2 -> divM r2 y2
  >>= fun r3 -> divM r3 y3
```

Exemple 2 : Monade Random

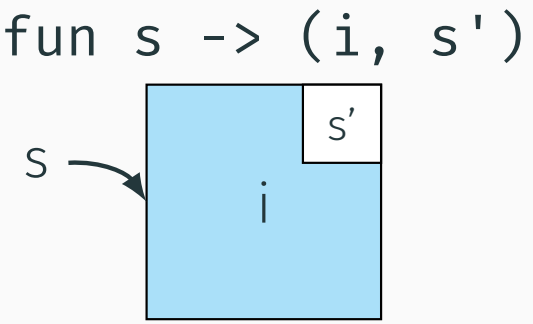
La Monade Random

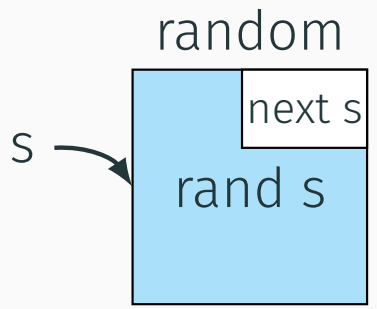
```
type seed = Seed of int
type 'a monad =
  seed -> 'a * seed

let return a =
  fun (Seed s)->(a,Seed s)

let bind (step:'a monad)
      (f:'a -> 'b monad)
    :'b monad=
  (fun (Seed s) ->
    let i, s' = step (Seed s)
    in f i s')

let (>>=) = bind
```

La Monade Random

```
type seed = Seed of int
type 'a monad =
  seed -> 'a * seed

let return a =
  fun (Seed s)->(a,Seed s)

let bind (step:'a monad)
      (f:'a -> 'b monad)
    :'b monad=
  (fun (Seed s) ->
    let i, s' = step (Seed s)
    in f i s')

let (>>=) = bind
```

```
let next (Seed s) =
  Seed (s*23 mod 17+1)
let rand (Seed s) = s - 1

let random:int monad =
  (fun (s:seed) ->
    (rand s, next s))

let run f =
  let seed = Random.int 17 + 1 in
  fst (f (Seed seed))
```





La Monade Random : les paires

```
type seed = Seed of int
type 'a monad =
  seed -> 'a * seed

let random:int monad =
  (fun (s:seed) ->
    (rand s, next s))

let return a =
  fun (Seed s)->(a,Seed s)

let bind (step:'a monad)
      (f:'a -> 'b monad)
    :'b monad=
  (fun (Seed s) ->
    let i, s' = step (Seed s)
    in f i s')

let (>>=) = bind

let run f =
  let seed = Random.int 17 + 1
  in fst (f (Seed seed))
```

La Monade Random : les paires

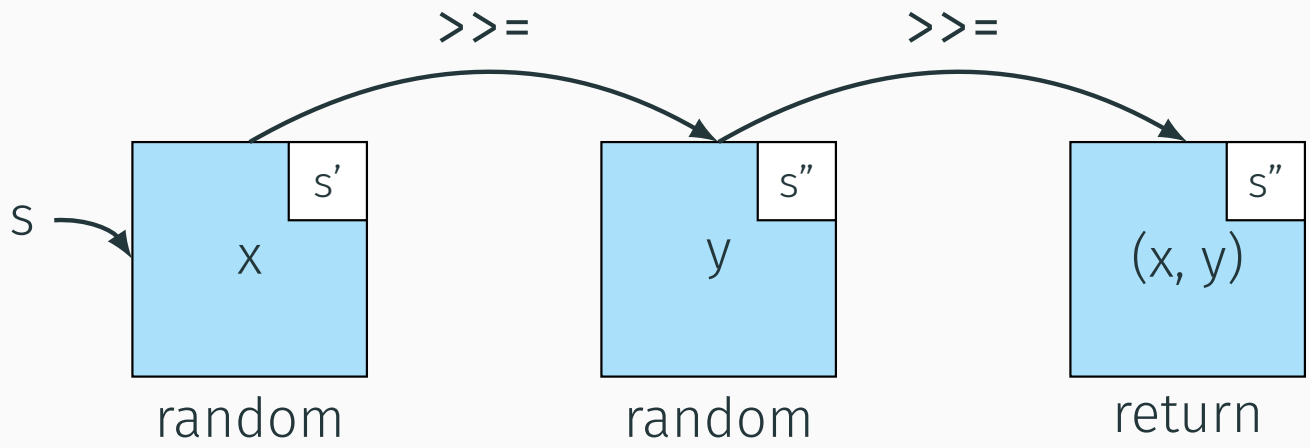
```
type seed = Seed of int
type 'a monad =
  seed -> 'a * seed

let random:int monad =
  (fun (s:seed) ->
    (rand s, next s))

let return a =
  fun (Seed s)->(a,Seed s)

let bind (step:'a monad)
      (f:'a -> 'b monad)
    :'b monad=
  (fun (Seed s) ->
    let i, s' = step (Seed s)
    in f i s')

let (>>=) = bind

let run f =
  let seed = Random.int 17 + 1
  in fst (f (Seed seed))
```

```
let rand_int () = run random

let mk_pair' =
  fun seed ->
  bind
    random
    (fun x seed' ->
      bind
        random
        (fun y -> return (x, y))
        seed')
    seed

let mk_pair =
  random >>= fun x ->
  random >>= fun y ->
  return (x, y)

let rand_pair () = run mk_pair
```

La Monade Random : les paires

```
let rand_int () =
  run random

let run f =
  let seed =
  Random.int 17 + 1
  in fst (f (Seed seed))
```

```
let mk_pair =
  random >>= fun x ->
  random >>= fun y ->
  return (x, y)

let rand_pair () = run mk_pair
```



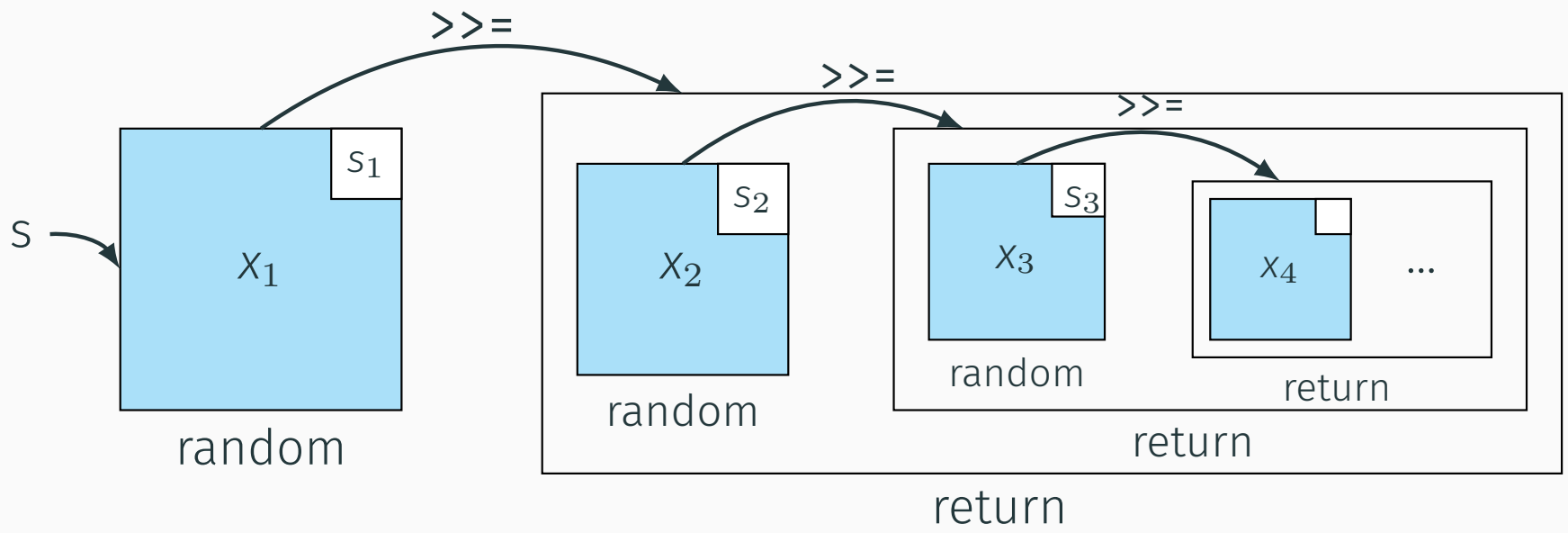
La Monade Random : les listes

```
type seed = Seed of int
type 'a monad =
  seed -> 'a * seed

let random:int monad =
  (fun (s:seed) ->
    (rand s, next s))

let return a =
  fun (Seed s)->(a,Seed s)

let bind (step:'a monad)
      (f:'a -> 'b monad)
    :'b monad=
  (fun (Seed s) ->
    let i, s' = step (Seed s)
    in f i s')

let (>>=) = bind

let run f =
  let seed = Random.int 17 + 1
  in fst (f (Seed seed))
```

```
let rec mk_list n =
  if n = 0 then return [] else
    random >>= fun x ->
    mk_list (n-1) >>=  fun l ->
    return (x::l)

let rand_list () =
  run (random >>= fun l->mk_list
```

La Monade Random : les listes

```
let rec mk_list n =
  if n = 0 then return [] else
    random >>= fun x ->
    mk_list (n-1) >>=  fun l ->
    return (x::l)

let rand_list () =
  run (random >>= fun l->mk_list l)
```



Conclusion

La programmation fonctionnelle

Un style de programmation qui se concentre sur l'algorithme.

De l'expressivité, de la concision.

De la puissance pour tous les types de programmeurs, allant du L1 à Yann Régis-Gianas.

Les Monades

Un outil extrêmement puissant, mais difficile à saisir.

- permet d'avoir des effets de bord en gardant les bonnes propriétés de la programmation fonctionnelle pure ;
- mais aussi une sorte de patron de conception aux stéroïdes.

Les Monades

Un outil extrêmement puissant, mais difficile à saisir.

- permet d'avoir des effets de bord en gardant les bonnes propriétés de la programmation fonctionnelle pure ;
- mais aussi une sorte de patron de conception aux stéroïdes.

Mais surtout
Pas indispensable pour faire de la programmation fonctionnelle (il y a des librairies !)

La programmation fonctionnelle à l'horizon

Est-elle réservée aux langages fonctionnels ?

- C++ `std::functional`, Java lambda, Golang, Rust, ...
- C

```c
int square (int) __attribute__ ((const));
int hash (char *) __attribute__ ((pure));
```

ExCamera [3] : un encodeur vidéo en C++, fonctionnel pur

Références


Dominus Carnufex. *Comprendre les monades*. 2016. URL : https://dominuscarnufex.github.io/cours/monades/ (visité le 17/03/2019).


Guiseppe Castagna. *Monads*. 2016. URL : https://www.irif.fr/~gc/slides/monads.pdf (visité le 17/03/2019).


Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter et Keith Winstein. «Encoding, fast and slow : Low-latency video processing using thousands of tiny threads». In : *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, p. 363-376.


Xavier Leroy. *Functional programming languages*. 2017. URL : `https://xavierleroy.org/mpri/2-4/monads.2up.pdf` (visité le 17/03/2019).


Wikibooks. *Haskell/Understanding monads/State — Wikibooks, The Free Textbook Project*. 2019. URL : `https://en.wikibooks.org/w/index.php?title=Haskell/Understanding_monads/State&oldid=3520151` (visité le 17/03/2019).