# Programmation Fonctionnelle Avancée

## Séance 6 : Writer Monad

Alexandros Singh

Université Paris 8

17 novembre 2023

## Logging

Supposons que nous voulions exécuter des fonctions pures tout en conservant un journal pendant l'exécution :

## Logging

Supposons que nous voulions exécuter des fonctions pures tout en conservant un journal pendant l'exécution :

- Par exemple, nous voulons conserver un journal pour déboguer notre code.

## Logging

Supposons que nous voulions exécuter des fonctions pures tout en conservant un journal pendant l'exécution :

- Par exemple, nous voulons conserver un journal pour déboguer notre code.
- Ou plus généralement, collecter des données sur notre code qui ne sont pas directement liées aux valeurs qu'il calcule.

Supposons que nous voulions exécuter des fonctions pures tout en conservant un journal pendant l'exécution :

- Par exemple, nous voulons conserver un journal pour déboguer notre code.
- Ou plus généralement, collecter des données sur notre code qui ne sont pas directement liées aux valeurs qu'il calcule.

Le problème suivant, qui concerne la collecte de données sur les performances de notre code, en est un exemple :

**Problème**

Effectuer une analyse empirique de la complexité d'un morceau de code en suivant le nombre d'opérations "primitives" effectuées pour différentes entrées.

Supposons par exemple que nous voulions analyser le tri à bulles et que nous ayons accès à une fonction `bubble_iter` qui effectue une itération de cet algorithme tout en enregistrant le nombre de swaps qu'elle a effectué :

```
# bubble_iter [1;2;3;4];;
- : int list * int = ([1; 2; 3; 4], 0)
# bubble_iter [1;2;4;3];;
- : int list * int = ([1; 2; 3; 4], 1)
# bubble_iter [4;3;1;2];;
- : int list * int = ([3; 1; 2; 4], 3)
```

Nous pouvons alors “composer” des itérations tout en gardant la trace du nombre total de swaps comme suit :

```
let two_iters l =
   let l1,c1 = bubble_iter l in
   let l2,c2 = bubble_iter l1 in
   (l2,c1+c2)
```

```
# two_iters [4;3;1;2];;
- : int list * int = ([1; 2; 3; 4], 5)
# two_iters [1;3;4;2];;
- : int list * int = ([1; 2; 3; 4], 2)
```

Pour implémenter l'algorithme complet, nous devons continuer à composer ces itérations jusqu'à ce que nous trouvions que la liste est triée. Supposons donc que nous ayons accès à à un prédicat `is_sorted` qui nous indique si une liste est ordonnée :

```
# is_sorted [1;2;3;4];;
- : bool = true
# is_sorted [4;1;3;2];;
- : bool = false
```

## Analyse du tri à bulles

Nous pouvons implémenter la fonction souhaitée qui exécute l'algorithme de tri à bulles tout en enregistrant le nombre de swaps effectués :

```
let rec iter_until_sorted l = match (is_sorted l) with
    | true -> (l,0)
    | false ->
            let (l_after_iter, s) = bubble_iter l in
            let (l_after_rec, r) = iter_until_sorted_n l_after_iter
            in (l_after_rec, s+r)
```

```
# iter_until_sorted_n [1;2;3;4];;
- : int list * int = ([1; 2; 3; 4], 0)
# iter_until_sorted_n [4;3;2;1];;
- : int list * int = ([1; 2; 3; 4], 6)
```

## Monade IntLogger

La “composition” d'itérations pourrait être être plus propre... en utilisant une monade !
La monade suivante devrait faire l'affaire :

```
module IntLogger : Monad with type 'a t = 'a * int = struct
    type 'a t = 'a * int

    let return x = (x, 0)
    let ( >>= ) m f =
        let (old_v, old_w) = m in
        let (new_v, new_w) = f old_v in
        (new_v, old_w + new_w)
end
```

Vérifiez les lois des monades !

Essayons d'abord de réimplémenter `bubble_iter` de façon monadique :

```
let rec bubble_iter l = match l with
    | [] -> return []
    | x :: [] -> return [x]
    | x :: y :: rest -> if (x >= y) then
                            let* a = bubble_iter (x :: rest) in
                            (y :: a, 1)
                        else
                            let* a = bubble_iter (y :: rest) in
                            return (x :: a)
```

Et maintenant, pour `two_iters` de façon monadique :

```
let two_iters_m l =
    bubble_iter l >>= bubble_iter
```

```
# two_iters_m [4;3;1;2];;
- : int list IntLogger.t = ([1; 2; 3; 4], 5)
# two_iters_m [1;3;4;2];;
- : int list IntLogger.t = ([1; 2; 3; 4], 2)
```

Et finalement, la fonction complète :

```
let rec iter_until_sorted l = match (is_sorted l) with
    | true -> return l
    | false ->
            bubble_iter l >>= iter_until_sorted
```

```
# iter_until_sorted [1;2;3;4];;
- : int list IntLogger.t = ([1; 2; 3; 4], 0)
# iter_until_sorted [4;3;2;1];;
- : int list IntLogger.t = ([1; 2; 3; 4], 6)
```

## Re-analyse du tri à bulles

Et si, au lieu d'enregistrer le nombre de swaps, nous voulions un log des swaps effectués ? Définissons d'abord une monade appropriée :

```
module StringLoggerLeft : Monad with type 'a t = 'a * string = struct
    type 'a t = 'a * string

    let return x = (x, "")
    let ( >>= ) m f =
        let (old_v, old_w) = m in
        let (new_v, new_w) = f old_v in
        (new_v, old_w ^ new_w)
end
```

Vérifiez les lois des monades !

Nous pourrions également permuter les arguments de la concaténation :

```
module StringLoggerRight : Monad with type 'a t = 'a * string = struct
    type 'a t = 'a * string

    let return x = (x, "")
    let ( >>= ) m f =
        let (old_v, old_w) = m in
        let (new_v, new_w) = f old_v in
        (new_v, new_w ^ old_w)
end
```

Vérifiez les lois des monades !

Puisque nous utiliserons les deux monades, définissons ce qui suit afin d'éviter toute ambiguïté :

```
let (>>=) = Monads.StringLoggerLeft.(>>=)
let (>>-) = Monads.StringLoggerRight.(>>=)
let (let>) x f =  x >>- f
```

La fonction `bubble_iter` devient :

```
let rec bubble_iter l = match l with
    | [] -> return []
    | x :: [] -> return [x]
    | x :: y :: rest ->
            if (x >= y) then
                let> a = bubble_iter (x :: rest) in
                (y :: a, (string_of_int x) ^ "<->" ^
                         (string_of_int y) ^ "\n")
            else
                let> a = bubble_iter (y :: rest) in
                return (x :: a)
```

## Re-analyse du tri à bulles

Le code pour `two_iters` ne change pas :

```
let two_iters l =
    bubble_iter l >>= bubble_iter
```

```
# two_iters [4;3;1;2];;
- : int list Code.Monads.StringLoggerLeft.t =
([1; 2; 3; 4], "4<->3\n4<->1\n4<->2\n3<->1\n3<->2\n")
# two_iters [4;3;1;2] |> snd |> print_string;;
4<->3
4<->1
4<->2
3<->1
3<->2
- : unit = ()
```

Que se passerait-il si nous utilisions >>- au lieu de >>= ?

## Re-analyse du tri à bulles

Il n'y a pas non plus de changement dans `iter_until_sorted`.

```
let rec iter_until_sorted l = match (is_sorted l) with
    | true -> return l
    | false ->
            bubble_iter l >>= iter_until_sorted
```

```
# iter_until_sorted [4;3;2;1];;
- : int list StringLogger.t =
([1; 2; 3; 4], "4<->1\n4<->2\n4<->3\n3<->1\n3<->2\n2<->1\n")
# iter_until_sorted [4;3;2;1] |> snd |> print_string;;
4<->1
4<->2
4<->3
3<->1
3<->2
2<->1
- : unit = ()
```

## Monade Writer

Ce type de monade est appelé monade d'écriture. Plus généralement, on remarque que pour tout monoïde `M` avec element d'identité `M.i` et opération binaire associative `M.(@)`, on peut définir une monade correspondante :

```
module MakeWriter (M : Monoid) : with type 'a t = 'a * M.m = struct
    type 'a t = 'a * M.m

    let return x = (x, M.i)
    let ( >>= ) m f =
        let (old_v, old_w) = m in
        let (new_v, new_w) = f old_v in
        (new_v, M.(@) old_w new_w)
end
```

Est-ce que cela respecte les lois monadiques pour n'importe quel monoïde ?

Il se peut aussi que vous rencontriez une version plus abstraite et plus riche en fonctionnalités :

```
module type WriterMonad2  = sig
    include Monad
    type m
    val tell: m -> unit t
    val writer: 'a * m -> 'a t
    val runWriter: 'a t -> 'a * m
end
```

## Monade Writer

Voici un foncteur qui, à partir d'un monoïde, produit de telles monades :

```
module MakeWriter2 (M : Monoid) : (WriterMonad2 with type m = M.m) =
struct
    type m = M.m
    let ( @ ) = M.(@)
    type 'a t = Writer of 'a * m
    let writer (r, o) = Writer (r, o)
    let tell (x : m) : unit t = Writer ((),x)
    let runWriter m = match m with
        | Writer (a,b) -> (a,b)
    let return x = Writer (x, M.i)
    let ( >>= ) m f =
        let (old_v, old_w) = runWriter m in
        let (new_v, new_w) = runWriter (f old_v) in
        writer (new_v, old_w @ new_w)
end
```