Programmer avec les monades

- Revisiter la notion de Monade, esquissée avec la compréhension de listes
- Pousser les monades, et resoudre quelque puzzle

On se rappelle...

Il semble qu'on puisse écrire facilement des programmes par comprehension si on dispose d'un type de donnée 'a t équipé avec des fonctions

```
bind : 'a t -> ('a -> 'b t) -> 'b t
return : 'a -> 'a t
zero : 'a t
```


Définition

Une *monade* est constitué par

- un type polymorphe *'a t*
- une opération *bind: 'a t −> ('a −> 'b t) −> 'b t*
- une opération *return : 'a −> 'a t*

et satisfaisant les trois équations vues lors du dernier cours pour les listes :

bind m return	=	m
bind (return e) f	=	f e
bind e1 (fun x -> bind e2 (fun y -> e3))	=	
bind (bind e1 (fun x -> e2)) (fun y -> e3)		*x non libre dans e3*

Ces équations sont utiles pour prouver des propriétés, mais elles sont souvent oubliées par les programmeurs...

Rappel sur les équations

Les équations que nous avons vu au début

bind m **return**	=	m
bind (**return** e) f	=	f e
bind e1 (**fun** x −> **bind** e2 (**fun** y −> e3))	=	
bind (**bind** e1 (**fun** x −> e2)) (**fun** y −> e3)		*x non libre dans e3*

S'écrivent comme suit avec l'opérateur infixe

m >>= **return**	=	m
return e >>= f	=	f e
e1 >>= **fun** x −> e2 >>= **fun** y −> e3	=	
(e1 >>= **fun** x −> e2) >>= **fun** y −> e3		*x non libre dans e3*

L'intuition

Le type *'a t* contient des *calculs* sur des valeurs de type *'a*, et les deux opérations *bind: 'a t −> ('a −> 'b t) −> 'b t* et *return : 'a −> 'a t* fournissent les briques de base pour construire et composer des calculs.

Une monade particulière peut fournir d'autres briques pour construire des calculs :

- dans le cours sur la comprehension des listes, par exemple, on a vu qu'il nous fallait aussi une opération constante *zero : 'a t* avec l'équation *bind zero f = zero* pour coder les compréhensions.
- pour les exemple de ce cours, nous allons aussi ajouter des opérations de *somme* de monades, comme *++ : 'a t -> 'a t -> 'a t* qui permettent de cumuler le résultat de plusieurs calculs.

Quelques repères historiques

Les *monades* sont un concept mathématique apparu dans la théorie des catégories, et qui a été largement repris en informatique :

1989 Eugenio Moggi les utilise pour construire une sémantique dénotationnelle modulaire des langages de programmation

1992 Philip Wadler popularise l'utilisation des monades en programmation fonctionnelle ; aujourd'hui les monades sont un des concepts les plus utilisés dans la communauté Haskell

1995 Peter Buneman, Val Tannen et leurs étudiants construisent des langages de requête concis et optimisables basés sur les monades des collections (similaire à ce qu'on a vu avec la comprehension des listes)

La monade des listes...

Nous avons vu ça dans le cours précédent

```
module ListMonad =
  struct
    type 'a t = 'a list
    let return x = [x]
    let bind l f = List.fold_right (fun x acc -> (f x)@acc) l []
    let zero = []
    let ( >>= ) l f = bind l f
  end;;
|| module ListMonad :
||   sig
||     type 'a t = 'a list
||     val return : 'a -> 'a list
||     val bind : 'a list -> ('a -> 'b list) -> 'b list
||     val zero : 'a list
||     val ( >>= ) : 'a list -> ('a -> 'b list) -> 'b list
||   end
```


... et la compréhension

Traduction uniforme

La traduction du cours précedent peut s'écrire comme suit

$$T[\![\ e\ |x \leftarrow e']\!] \quad = \quad e' \ >>= (\textbf{fun}\ x -> \text{return}\ e)$$

$$T[\![\ e\ |x \leftarrow e'; y \leftarrow e'';\ Q\ ]\!] \quad = \quad e' \ >>= (\textbf{fun}\ x -> T[\![\ e\ |y \leftarrow e'';\ Q\ ]\!])\ e'$$

$$T[\![\ e\ |x \leftarrow e'; f;\ Q\ ]\!] \quad = \quad e' \ >>= (\textbf{fun}\ x -> \textbf{if}\ f\ \textbf{then}\ T[\![\ e\ |\ Q\ ]\!]\ \textbf{else}\ \text{zero})$$

Donc avec les opérations de la monade des listes on peut écrire *toutes* les fonctions qu'on peut décrire à l'aide de la compréhension.Cela montre aussi qu'avec toute monade possédant un *zero* on peut faire de la compréhension !

La compréhension et les équations

Notez comme les équations des monades font parfaitement sens avec la notation de la compréhension des listes... par exemple :

unité 1

$$
\begin{aligned}
T[\![[\ x\ | x \leftarrow e']]\!] &= \text{e' >>= (\textbf{fun} x-> return x)} \\
&= \text{e' >>= return} \\
&= \text{e'}
\end{aligned}
$$

unité 2

$$
\begin{aligned}
T[\![[\ e\ | x \leftarrow [e']]]\!] &= \text{(return e') >>= (\textbf{fun} x-> return e)} \\
&= \text{(\textbf{fun} x-> return e) e'} \\
&= \text{[e \{x := e'\}]}
\end{aligned}
$$

Extensions de la monade des listes

Traitons le choix et le non déterminisme.

Etant donné un calcul l de type `'a t` et un deuxième calcul l' de type `'a t`, on peut souhaiter les fusionner en un seul calcul $l++l'$ qui combine les deux.

Somme de calculs (listes)

Avec la monade des listes, le choix le plus naturel est simplement de définir l'opération ++ comme la concatenation des deux listes :

```
let (++) l l' = List.append l l';;
```

Cela nous donne

```
module ListMonadPlus = struct
    include ListMonad
    let (++) l l' = List.append l l'
  end
;;
|| module ListMonadPlus :
||   sig
||     type 'a t = 'a list
||     val return : 'a -> 'a list
||     val bind : 'a list -> ('a -> 'b list) -> 'b list
||     val zero : 'a list
||     val ( >>= ) : 'a list -> ('a -> 'b list) -> 'b list
||     val ( ++ ) : 'a list -> 'a list -> 'a list
||   end
```


Trois nouvelles équations

Avec cette nouvelle opération viennent trois nouvelles équations, qui sont faciles à vérifier sur la monade des listes :

Equation pour la somme

$$
\begin{aligned}
\text{m ++ zero} &= \text{m} \\
\text{zero ++ m} &= \text{m} \\
\text{m ++ (m' ++ m'')} &= \text{(m ++ m') ++ m''}
\end{aligned}
$$

En général, on parle d'une *monade avec zero et plus* quand on dispose d'une monade avec des opérations zero et ++ satisfaisant les équations ci-dessus.

Application : suppressions

Problème :

Étant donnée une liste l, construire toutes les listes qu'on peut obténir à partir d'elle en supprimant un et un seul élément.

Avec la compréhension, on serait temptés d'écrire :

```
let rec suppr l =
   match l with
      [] -> []
     [_] -> [ [] ]
   | a::r -> [ l' | l' <- ``soit r,
                              soit a devant un des (suppr r)''
              ]
```

Voyons de concrétiser cette idée

Application : suppressions

```
let rec suppr l =
   match l with
      [] -> [] | [_] -> [ [] ]
   | a::r ->
     [ l' | l' <- (List.append [r]
                    (List.map (fun x -> a::x) (suppr r))) ]
```

Et on peut même supprimer la compréhension, vu que
[l' | l' <− e] = e.

Application : suppressions

On peut écrire notre code très simplement avec notre monade avec somme :

```
open ListMonadPlus;;

let rec suppr l =
    match l with
       [] -> zero
     | [_] -> return []
     | a::r -> (return r) ++
                ((suppr r) >>= (fun l' -> return (a::l')));;
|| val suppr : 'a list -> 'a list list = <fun>

suppr [1;2;3];;
|| - : int list list = [[2; 3]; [1; 3]; [1; 2]]
```


Application : sousensembles

Problème :

Étant donnée une liste l représentant un esemble S, construire les listes représentant tous les sousensembles de S.

Maintenant qu'on a compris la puissance de +, on écrira :

```
let rec subsets =
  function
    [] -> return []
  | a::r -> let ss = (subsets r)  in
              ss ++ (ss >>= (fun s -> return (a::s)))
;;
|| val subsets : 'a list -> 'a list list = <fun>

subsets [1;2];;
|| - : int list list = [[]; [2]; [1]; [1; 2]]
```

Autre application de la somme

Avec l'ajout de la somme de monades, on est maintenant capables de calculer *toutes* les solutions d'un puzzle, sans recourir à *aucun constructeur de liste explicite*.

```
module type MonadPlusList = sig
    type 'a t
    val return : 'a -> 'a t
    val bind : 'a t -> ('a -> 'b t) -> 'b t
    val zero : 'a t
    val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
    val ( ++ ) : 'a t -> 'a t -> 'a t
    val reveal : 'a t -> 'a list
  end;;


module ListMonadPlus : MonadPlusList= struct
  include ListMonadPlus
  let reveal m = m
  end;;
```


Exemple (concours Kangourou 2013)

Problème :
Anita, Berenice, Carmen et Dikra sont nées la même année. Leurs anniversaires sont, dans le désordre : le 20/2, le 12/4, le 12/5 et le 25/5. Berenice et Anita sont nées le même mois. Anita et Carmen sont nées le même jour, mais de mois différents.
Laquelle des quatres est la plus agée ?

Solution

```
type anniv = {day:int;month:int};;

open ListMonadPlus;;

let annivs = (return {day=20;month=2}) ++
             (return {day=12;month=4}) ++
             (return {day=12;month=5}) ++
             (return {day=25;month=5});;
|| val annivs : anniv ListMonadPlus.t = <abstr>


let rec alldifferent = function
    [] -> true
  | a::r -> alldifferent r && (not (List.mem a r));;
```


Solution

```
let solutions =
annivs >>= (fun anita   ->
annivs >>= (fun berenice ->
annivs >>= (fun carmen  ->
annivs >>= (fun dikra   ->
  if (berenice.month = anita.month) &&
     (anita.day=carmen.day) &&
     alldifferent [anita;berenice;carmen;dikra]
  then return (anita,berenice,carmen,dikra) else zero))))
|| val solutions : (anniv * anniv * anniv * anniv) ListMo
|| >

reveal solutions;;
|| - : (anniv * anniv * anniv * anniv) list =
|| [({day = 12; month = 5}, {day = 25; month = 5}, {day =
||    {day = 20; month = 2})]
```

Interlude

Avec l'ajout de la somme de monades, on est maintenant capables de calculer *toutes* les solutions d'un puzzle, sans recourir à *aucun constructeur de liste explicite*...
Mais on calcule vraiment *toutes* les solutions...
Cela peut être très cher si on n'en veut qu'une !
Voyons comment réaliser une monade avec sommes qui permet de calculer *une seule* solution.

McCarty's amb operator

Un des créateurs de Algol, John McCarthy, propose dans un article l'opérateur binaire `amb`, avec l'idée que `amb e1 e2` échoue si les deux arguments échouent, et sinon retourne la valeur d'une des opérateurs qui n'échouent pas...

J. Mccarthy.
A basis for a mathematical theory of computation.
In Computer Programming and Formal Systems, pages 33–70. North-Holland, 1963.

Adaptons notre ++ et nos monades pour simuler à peu près `amb`

Somme avec retour en arrière

Commençons simple : supposons de travailler avec des valeurs booléennes.

```
module type NonDetMonad = sig
  include MonadPlusList
  val reveal : bool t -> bool option
end;;


module MonadNonDet: NonDetMonad = struct
type 'a t = ('a -> bool option) -> bool option
let return v = fun k -> k v
let (++) e1 e2 = fun k -> match e1 k with
                              Some v -> Some v
                            | None   -> e2 k
let zero = fun k -> None
let bind e f = fun k -> e (fun x -> f x k)
let reveal e = e (fun x -> Some x)
let (>>=) e f = bind e f
end;;
```


Quelques tests simples

```
open MonadNonDet
let e =
  (return false ++ return true) >>= (fun x -> Printf.eprintf "x=%b␣%!" x
   if x then return true else zero);;
|| val e : bool MonadNonDet.t = <abstr>

reveal e;;
|| x=false x=true - : bool option = Some true

let e' =
  (return false ++ return true) >>= (fun x -> Printf.eprintf "x=%b␣%!" x
  (return false ++ return true) >>= (fun y -> Printf.eprintf "y=%b␣%!" y
   if x&&y then return true else zero));;
|| val e' : bool MonadNonDet.t = <abstr>

reveal e';;
|| x=false y=false y=true x=true y=false y=true - : bool option = Some tru
||
```

Somme avec retour en arrière

Si on veut pouvoir traiter des valeurs quelconques, il faut utiliser des fonctions polymorphes de première classe (disponibles seulement comme champs d'enregistrements !).

```
module type NonDetMonad = sig
  include MonadPlusList
  val reveal : 'a t -> 'a option
end;;


module MonadNonDet : NonDetMonad = struct
type 'a t = {f: 'b.  ('a -> 'b option) -> 'b option}
let return v = { f = fun k -> k v}
let (++) e1 e2 = { f = fun k -> match e1.f k with
                            Some v -> Some v
                          | None   -> e2.f k}
let zero = {f=fun k -> None}
let bind e f = {f = fun k -> e.f (fun x -> (f x).f k)}
let reveal e = e.f (fun x -> Some x)
let (>>=) e f = bind e f
end;;
```

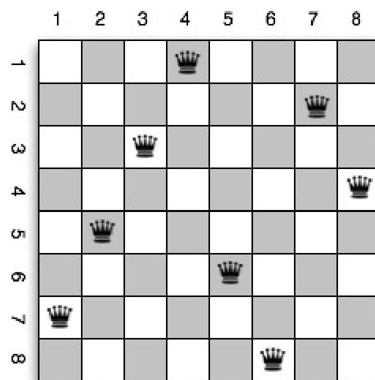

Solution révisitée

Maintenant on peut résoudre le puzzle sans engendrer toutes les solutions (l'ordre de visite reste très important).

```
open MonadNonDet

let annivs = (return {day=20;month=2}) ++ (return {day=12;month=4}) ++
             (return {day=12;month=5}) ++ (return {day=25;month=5});;
|| val annivs : anniv MonadNonDet.t = <abstr>

let solutions =
annivs >>= (fun anita    ->
annivs >>= (fun berenice ->
annivs >>= (fun carmen   ->
annivs >>= (fun dikra    ->
  if (berenice.month = anita.month) &&
     (anita.day=carmen.day) &&
     alldifferent [anita;berenice;carmen;dikra]
  then return (anita,berenice,carmen,dikra) else zero))));;
|| val solutions : (anniv * anniv * anniv * anniv) MonadNonDet.t = <abstr>

reveal solutions;;
|| - : (anniv * anniv * anniv * anniv) option =
|| Some
||  ({day = 12; month = 5}, {day = 25; month = 5}, {day = 12; month = 4},
||    {day = 20; month = 2})
```


Le problème `n-queens`

Revoyons l'ensemble de ces notions avec le problème bien connu des *n*-reines. Il s'agit de placer *n* reines sur un échiquier de taille $n \times n$ de telle sorte qu'aucune reine soit en prise.
Dans la figure, un exemple de configuration correcte pour $n = 8$.



Représentation des données

On encode une configuration de l'échiquier de taille $n \times n$ avec une liste de *n* entiers [l1 ; l2 ;...; ln] : l'entier lj donne la ligne sur laquelle se trouve la reine de la colonne *j*. Avec cette convention, la configuration de la figure est [7;5;3;1;6;8;2;4] .

```
type board = int list;;
|| type board = int list
```

Condition pour ne pas être en prise

Il est facile de vérifier les conditions pour que les reines ne soient pas en prise :

- on ne peut pas avoir deux reines dans la même colonne, par construction
- deux reines en li et lj sont sur la même ligne ssi li=lj
- deux reines en li et lj sont sur la même diagonale ssi *abs*(li −lj)= *abs*(i−j)

Condition pour ne pas être en prise : le code

```
let rec safeadd board row dist =
  match board with
    [] -> true
  | row'::board' -> row<>row' && abs(row-row')<> dist
                    && safeadd board' row (dist+1);;
|| val safeadd : int list -> int -> int -> bool = <fun>


let compatible board row = safeadd board row 1;;
|| val compatible : int list -> int -> bool = <fun>
```


Touver toutes les solutions

Il nous faut la fonction auxiliaire `range`.On la définit de telle sorte qu'elle fournisse un element de `int ListMonad.t` :

```
open ListMonadPlus

let range i j =
  let rec aux acc n =
        if n < i then acc
        else aux (return n ++ acc) (n-1)
  in aux zero j;;
|| val range : int -> int -> int ListMonadPlus.t = <fun>
```


Touver toutes les solutions

Maintenant on peut écrire

```
let rec fill_queens n board =
  function
      0 -> return board
    | i -> (range 1 n) >>= fun row ->
              if compatible board row
              then fill_queens n (row::board) (i-1)
              else zero;;
|| val fill_queens : int -> int list -> int -> int list Li
|| fun>

let nqueens_all n = fill_queens n [] n;;
|| val nqueens_all : int -> int list ListMonadPlus.t = <fu
```

Testons le code !

Touver une solution

Si nous voulons une seule solution, nous pouvons utiliser notre monade des listes avec somme non deterministe.
On a juste besoin de redefinir `range` pour qu'il fournisse un element de `int MonadNonDet.t` (notez que le code de range est inchangé !) :

```
open MonadNonDet
let range i j =
  let rec aux acc n =
          if n < i then acc
          else aux (return n ++ acc) (n-1)
  in aux zero j;;
|| val range : int -> int -> int MonadNonDet.t = <fun>
```

Programmation Fonctionnelle Avancée
└La monade des Listes et le choix

Trouver une solution

Maintenant, nous pouvons réutiliser *exactement* le même code (mais qui fait appel à une modade différente) aussi pour la fonction principale :

```
let rec fill_queens n board =
  function
      0 -> return board
    | i -> (range 1 n) >>= fun row ->
               if compatible board row
               then fill_queens n (row::board) (i-1)
               else zero;;
|| val fill_queens : int -> int list -> int -> int list MonadNonDet.t = <
|| n>


let nqueens_one n = fill_queens n [] n;;
|| val nqueens_one : int -> int list MonadNonDet.t = <fun>


reveal(nqueens_one 8);;
|| - : int list option = Some [4; 2; 7; 3; 6; 8; 5; 1]
```

Programmation Fonctionnelle Avancée
└La monade des Listes et le choix

On peut factoriser le code

Maintenant, utilisons la puissance du langage des modules pour factoriser le code et retarder le choix de la monade à l'execution.
Pour cela :

- il faut adapter les signatures, qui diffèrent pour `reveal` :

```
ListMonadPlus.reveal;;
|| - : 'a ListMonadPlus.t -> 'a list = <fun>

MonadNonDet.reveal;;
|| - : 'a MonadNonDet.t -> 'a option = <fun>
```

- ensuite on utilisera les modules de prémière classe

Programmation Fonctionnelle Avancée
└La monade des Listes et le choix

Unifier les signatures

```
module type MonadLP =
  sig
    include module type of ListMonadPlus
    val reveal : 'a t -> 'a list option
  end;;
|| module type MonadLP =
||   sig
||     type 'a t
||     val return : 'a -> 'a t
||     val bind : 'a t -> ('a -> 'b t) -> 'b t
||     val zero : 'a t
||     val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
||     val ( ++ ) : 'a t -> 'a t -> 'a t
||     val reveal : 'a t -> 'a list option
||   end
```

Unifier les signatures

```
module MonadDet : MonadLP =
  struct
    include ListMonadPlus
    let reveal l =
      match reveal l with
[] -> None
      | l' -> Some l'
  end;;
|| module MonadDet : MonadLP

module MonadND : MonadLP =
  struct
    include MonadNonDet
    let reveal l =
      match (reveal l) with
Some v -> Some [v]
      | None -> None
  end;;
|| module MonadND : MonadLP
```


Factorisation du code I

```
let nqueens n all =
  let module M =
    (val (if all then (module MonadDet : MonadLP)
                 else (module MonadND: MonadLP)) : MonadLP)
  in let open M in
     let range i j =
       let rec aux acc n =
               if n < i then acc
               else aux (return n ++ acc) (n-1)
       in aux zero j in
     let rec fill_queens n board =
       function
           0 -> return board
         | i -> (range 1 n) >>= fun row ->
                  if compatible board row
                  then fill_queens n (row::board) (i-1)
                  else zero
     in reveal (fill_queens n [] n);;
|| val nqueens : int -> bool -> int list list option = <fun>
```


Factorisation du code II

```
nqueens 6 true;;
|| - : int list list option =
|| Some
||  [[5; 3; 1; 6; 4; 2]; [4; 1; 5; 2; 6; 3]; [3; 6; 2; 5; 1; 4];
||    [2; 4; 6; 1; 3; 5]]

nqueens 6 false;;
|| - : int list list option = Some [[5; 3; 1; 6; 4; 2]]
```


Conclusions

- la programmation avec des monades est un méchanisme puissant et générique, qui permet d'écrire des programmes à un niveau d'abstraction très élévé ...
- ... cela se base sur un choix bien spécifique de combinateurs, dont bind, return, zero et ++
- on peut donc facilement transporter nos programmes sur *toute structure* qui dispose de ces opérations