

## 1 Combinateurs

On retrouve souvent les mêmes schémas de parcours pour des structures de données telles que listes, arbres binaires, etc. Il est possible, en utilisant l'ordre supérieur (le passage de fonctions en argument) de programmer de tels schémas une fois pour toutes. Cela devient même nécessaire lorsque l'on manipule une structure de données dont la représentation interne est abstraite (masquée) dans l'interface (voir la section 2).

Les schémas les plus fréquents sont :

- *map*, le morphisme structurel, où l'on recopie une structure de donnée en une structure de même forme, mais où les éléments  $x$  sont remplacés par des éléments  $f x$  ;
- *fold*, le pliage, consistant à appliquer une opération binaire aux éléments de la structure, tour à tour ;
- *iter*, l'itération, consistant à appliquer une fonction à effet de bord à tous les éléments de la structure.

### Exercice 1 : combinateurs de liste

Donner le type puis la définition de *map*, *fold* et *iter* pour les listes.

#### Corrigé

$map : (\alpha \rightarrow \beta) \rightarrow \alpha list \rightarrow \beta list$

let rec *map*  $f =$  fonction

| []  $\rightarrow$  []

|  $a :: l \rightarrow$  let  $r = f a$  in  $r :: map f l$

$fold\_left : (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta list \rightarrow \alpha$

let rec *fold\_left*  $f accu =$  fonction

| []  $\rightarrow accu$

|  $a :: l \rightarrow fold\_left f (f accu a) l$

$fold\_right : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha list \rightarrow \beta \rightarrow \beta$

```

let rec fold_right f l accu =
  match l with
  | [] → accu
  | a :: l → f a (fold_right f l accu)

iter : (α → unit) → α list → unit

let rec iter f = function
  | [] → ()
  | a :: l → f a; iter f l

```

## Exercice 2 : utilisation

Choisir et utiliser convenablement les combinateurs précédents pour réaliser les programmes suivants :

- impression de tous les éléments d'une liste d'entiers ;
- somme des éléments d'une liste d'entiers ;
- liste des représentations sous forme de chaîne d'une liste d'entiers ;
- longueur d'une liste.

### Corrigé

```

let print_int_list = iter (fun n → Printf.printf "%i\n" n)

let somme_elts = fold_left (+) 0

let chn = map string_of_int

let long l = fold_right (fun x r → r + 1) l 0

```

## 2 Manipulation d'ensembles finis

La bibliothèque standard de Ocaml comprend deux modules utiles pour les manipulations d'ensembles finis de données : *Set* et *Map*. Tous deux sont de nature fonctionnelle et utilisent une représentation efficace à base d'AVL. Ils sont donc paramétrés par un type d'éléments munis d'une relation d'ordre selon le type suivant.

```

module type OrderedType =
  sig
    type t
    val compare : t → t → int
  end

```

Noter que la fonction *compare* rend un entier. Par convention son signe indique le sens de l'inégalité entre les deux arguments. Ocaml offre en standard une fonction *compare* convenable, de type  $\alpha \rightarrow \alpha \rightarrow \text{int}$  (il s'agit de polymorphisme ad-hoc, vu en cours). Mais il est loisible d'en prendre une autre.

### Exercice 3 : types ordonnés

Définir :

- un module ordonnant le type des chaînes ;
- un module ordonnant le type des entiers sans utiliser la fonction *compare* standard ;
- un module ordonnant des couples (entier, chaîne) de sorte que deux couples  $(n, c)$  et  $(m, d)$  soient confondus si  $n = m$ .

#### Corrigé

```

module OrdString = struct
  type t = string
  let compare = compare
end

module OrdInt = struct
  type t = int
  let compare = (-)
end

module OrdCpl = struct
  type t = int × string
  let compare (n, c) (m, d) = n - m
end

```

## 2.1 Ensembles finis

Voici l'interface du module *Set*.

```

module type S =
  sig
    type elt
    type t
    val empty : t

```

```

val is_empty : t → bool
val mem : elt → t → bool
val add : elt → t → t
val singleton : elt → t
val remove : elt → t → t
val union : t → t → t
val inter : t → t → t
val diff : t → t → t
val compare : t → t → int
val equal : t → t → bool
val subset : t → t → bool
val iter : (elt → unit) → t → unit
val fold : (elt → α → α) → t → α → α
(* fold f s a calcule (f xN ... (f x2 (f x1 a))...) *)
val for_all : (elt → bool) → t → bool
val exists : (elt → bool) → t → bool
val filter : (elt → bool) → t → t
val partition : (elt → bool) → t → t × t
val cardinal : t → int
val elements : t → elt list
val min_elt : t → elt
val max_elt : t → elt
val choose : t → elt
val split : elt → t → t × bool × t
end
module Make (Ord : OrderedType) : S with type elt = Ord.t

```

#### Exercice 4 : utilisation

Calculer le produit des éléments d'un ensemble d'entiers. On pourra faire en sorte que le calcul s'interrompe dès qu'un 0 est trouvé.

#### Corrigé

```

module Sint = Set.Make (OrdInt)

Version de base
let prod ens = Sint.fold ( × ) ens 1

Version avec interruption sur 0
exception Zero
let prod ens =
  try Sint.fold (fun x y → if x = 0 then raise Zero else x × y) ens 1
  with Zero → 0

```

## 2.2 Applications à domaine fini

Le module *Map* permet d'associer une information à une clé. Ses principales opérations sont les suivantes.

```
module type S =
  sig
    type key
    type  $\alpha$  t
    val empty :  $\alpha$  t
    val is_empty :  $\alpha$  t  $\rightarrow$  bool
    val add : key  $\rightarrow$   $\alpha$   $\rightarrow$   $\alpha$  t  $\rightarrow$   $\alpha$  t
    val find : key  $\rightarrow$   $\alpha$  t  $\rightarrow$   $\alpha$ 
    val remove : key  $\rightarrow$   $\alpha$  t  $\rightarrow$   $\alpha$  t
    val mem : key  $\rightarrow$   $\alpha$  t  $\rightarrow$  bool
    val iter : (key  $\rightarrow$   $\alpha$   $\rightarrow$  unit)  $\rightarrow$   $\alpha$  t  $\rightarrow$  unit
    val map : ( $\alpha$   $\rightarrow$   $\beta$ )  $\rightarrow$   $\alpha$  t  $\rightarrow$   $\beta$  t
  end
module Make (Ord : OrderedType) : S with type key = Ord.t
```

### Exercice 5 : utilisation

Fabriquer un module pour une table des symboles, où chaque symbole est une chaîne. Étant donné une table associant une adresse entière aux symboles entrés, donner :

- un programme pour traduire les adresses d'une valeur entière ;
- un programme pour imprimer la table.

#### Corrigé

```
module Tbl = Map.Make (OrdString)

Version de base
let translate n tbl = Tbl.map (fun x  $\rightarrow$  n + x) tbl

Version avec  $\eta$ -reduction
let translate n = Tbl.map ((+) n)

Attention : printf ne doit pas être  $\eta$ -réduit
let printtbl = Tbl.iter (fun s n  $\rightarrow$  Printf.printf "%s : %i\n" s n)
```

### Exercice 6 : discussion sur les interfaces

Le module *Map* comporte une fonction *map*, (ainsi que *List*) mais pas *Set*. Pourquoi? Réaliser une fonction *map* pour un ensemble fini et l'utiliser

pour obtenir l'ensemble des longueurs d'un ensemble de chaînes. Indication : commencer par écrire le type des fonctions désirées.

### Corrigé

D'une manière générale, un appel `map f s` reproduit la structure de `s` en remplaçant chaque élément `x` par `f x`. Le faire sur les clés d'un AVL (ou d'une structure ordonnée) ne garantit le respect de l'ordre que si `f` est croissante. En outre, on change d'ensemble ordonné dès que `f` est de type  $\alpha \rightarrow \beta$  avec  $\alpha \neq \beta$ . Par exemple que se passe-t-il si `f` n'est pas injective ?

Dans le cas de `Map`, `f` ne s'applique pas aux clés, donc pas de problème. Dans le cas de `Set`, une réalisation est possible en recréant un AVL au moyen d'appels à `add`. Ce n'est pas tout à fait dans le même esprit et l'utilisateur peut facilement le programmer.

```
module Sstring = Set.Make (OrdString)

let map_string_int f e =
  Sstring.fold (fun x → Sint.add (f x)) e Sint.empty

let longueurs = map_string_int String.length
```

## 3 Représentation d'un graphe ou d'un automate

### 3.1 Expressions régulières

#### Exercice 7 : type des expressions régulières

Donner un type de donnée pour des expressions régulières pouvant comporter le mot vide  $\varepsilon$ , des terminaux, des séquences, des choix, l'opérateur  $*$  de Kleene.

### Corrigé

```
type regexp =
  | Epsilon
  | Terminal of char
  | Seq of regexp × regexp
  | Alt of regexp × regexp
  | Iter of regexp
```

### 3.2 Distributeur de valeurs fraîches

Lorsque l'on construit un automate qui reconnaît le langage d'une expression régulière, on a besoin « d'inventer » de nouveaux états et de nouvelles tran-

sitions. C'est une situation fréquente, on a souvent besoin d'un programme qui distribue à la demande une nouvelle valeur. On appelle parfois un tel programme *gensym* (générateur de symbole). En pratique on se contente d'un compteur d'entiers : un symbole est simplement représenté par un numéro.

### Exercice 8 : gensym simple

Réaliser une fonction *gensym* (donner d'abord son type) en faisant en sorte que le compteur interne ne soit pas visible en dehors de la définition de *gensym*.

#### Corrigé

```
let gensym =  
  let r = ref 0 in  
  let gs () =  
    let x = !r in  
    r := x + 1;  
    x  
  in gs
```

### Exercice 9 : gensym multiple

Modifier *gensym* en une fonction *mk\_gensym* qui retourne à la demande une fonction comme *gensym*, de sorte à avoir des distributeurs de valeurs fraîches qui soient indépendants.

Donner deux exemples d'applications de *mk\_gensym*, l'une pour distribuer de nouveaux état, l'autre pour distribuer des nouvelles transitions.

#### Corrigé

```
let mk_gensym () =  
  let r = ref 0 in  
  let gs () =  
    let x = !r in  
    r := x + 1;  
    x  
  in gs  
  
let genstate = mk_gensym ()  
let gentrans = mk_gensym ()
```

### 3.3 Automate non déterministe d'une expression régulière

Plusieurs stratégies sont possibles pour construire un automate non déterministe reconnaissant le langage exprimé par une expression régulière. On suit ici l'algorithme de Thompson, assez simple à programmer. Chaque expression régulière est reconnue par un automate ayant exactement un état de départ et un état accepteur différents l'un de l'autre. Le seul cas délicat est celui de l'étoile.

#### Exercice 10 : algorithme de Thompson

Envisager différentes représentations de l'automate et programmer l'algorithme de Thompson.

#### Corrigé

Une représentation assez simple à manipuler consiste à numéroter chaque transition et à associer (par *Map*) à chacune sa source, sa destination et son étiquette. À cause des  $\varepsilon$ -transitions, on étiquettera par un type *option*.

La fonction *thompson* prend en entrée deux états  $e_0$  et  $e_1$ , un ensemble (*Map*) de transitions et une expression régulière  $r$  et elle insère des transitions de sorte que  $r$  soit reconnue entre  $e_0$  et  $e_1$ .

```
module Trans = Map.Make (Ordint)
type  $\alpha$  trans = {src : Ordint.t; dest : Ordint.t; etiq :  $\alpha$  }
let new_trans e0 e1 c et =
  let numt = gentrans () in
  let t = {src = e0; dest = e1; etiq = c} in
  Trans.add numt t et
```

Algorithme de Thompson

```
let rec thompson e0 e1 tr = fonction
  | Epsilon  $\rightarrow$  new_trans e0 e1 None tr
  | Terminal (c)  $\rightarrow$  new_trans e0 e1 (Some c) tr
  | Seq (r1, r2)  $\rightarrow$ 
    let e = genstate () in
    thompson e e1 (thompson e0 e tr r1) r2
  | Alt (r1, r2)  $\rightarrow$  thompson e0 e1 (thompson e0 e1 tr r1) r2
  | Iter (r)  $\rightarrow$ 
    let e2 = genstate () in
    let e3 = genstate () in
    let tr = new_trans e0 e2 None tr in
    let tr = new_trans e3 e1 None tr in
    let tr = new_trans e3 e2 None tr in
    thompson e2 e3 tr r
```

```

Type de nfa_of_regep :
regep → int × int × char option trans Trans.t

let nfa_of_regep r =
  let e0 = genstate () and e1 = genstate () in
    e0, e1, thompson e0 e1 Trans.empty r

```

### 3.4 Transformations d'automates

On transforme l'automate non-déterministe obtenu en un automate déterministe. Il est alors utile de disposer rapidement, pour un état donné  $e$ , des transitions partant de  $e$ . Plus exactement, pour chaque  $e$ , on veut une fonction qui, pour chaque étiquette  $c$ , donne l'ensemble des destinations atteignables en une transition partant de  $e$  et étiquetée par  $c$ . Cela donnera en particulier l'ensemble des états que l'on peut atteindre à partir de  $e$  par une  $\varepsilon$ -transition.

#### Exercice 11 : changement de représentation

Concevoir une nouvelle structure de données en termes de *Map* et *Set* pour représenter un automate sous la forme indiquée précédemment. Écrire le programme qui effectue le changement de représentation.

#### Corrigé

```

module Ordeti = struct
  type t = char option
  let compare x y = match x, y with
    | None, None → 0
    | None, _ → -1
    | _, None → 1
    | Some (c), Some (d) → compare c d
end

module Chtrans = Map.Make (Ordeti)
module Ftrans = Map.Make (Ordint)

let addtrans _ t ftr =
  let m = Ftrans.find t.src ftr in
  let cibles = Chtrans.find t.etiq m in
  let cibles = States.add t.dest cibles in
  Ftrans.add t.src (Chtrans.add t.etiq cibles m) ftr

Famille des transitions indexée par l'état de départ
let f_transition nfa = Trans.fold addtrans nfa Ftrans.empty

```

### Exercice 12 : clôture par $\varepsilon$

Écrire la fonction qui calcule la clôture par  $\varepsilon$  d'un ensemble d'états.

#### Corrigé

```
let epsilon_from_states ftr sources =  
  let rec loop vues affaire =  
    if States.is_empty affaire then vues  
    else  
      let e = States.choose affaire in  
      let affaire = States.remove e affaire in  
      let eps_e = Chtrans.find None (Ftrans.find e ftr) in  
      loop (States.union vues eps_e) (States.diff affaire eps_e) in  
  from_states States.empty sources
```

### Exercice 13 : déterminisation

Écrire la fonction qui détermine un automate.

#### Corrigé