

Programmation fonctionnelle avancée

Notes de cours

Cours 7

4 novembre 2015

Sylvain Conchon

sylvain.conchon@lri.fr



1/1

12/1

2

Notions introduites

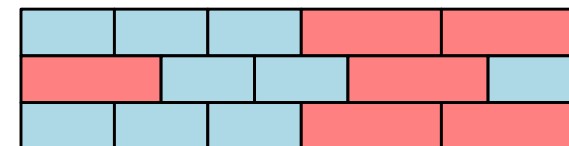
Construction d'un mur de briques

(1/2)

- Les entiers `int64`
- Les tables de hachage
- La technique de mémoïsation

On souhaite construire un mur avec des briques de longueur 2 (■) et de longueur 3 (■), dont on dispose en quantité illimitée.

Voici par exemple un mur de longueur 12 et de hauteur 3 :

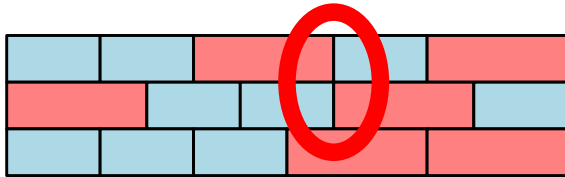


3/1

34/1

4

Mais pour être solide, le mur ne doit jamais superposer 2 jointures.



Combien y a-t-il de façons de construire un mur de longueur 32 et de hauteur 10 ?

5/1

56/1

6

Une solution récursive

On calcule **récurivement** le nombre de façons $W(r, h)$ de construire un mur de hauteur h , dont la rangée de brique la plus basse r est donnée.

► Cas de base :

$$W(r, 1) = 1$$

► Sinon :

$$W(r, h) = \sum_{r' \text{ compatible avec } r} W(r', h - 1)$$

On utilise la fonction `sum` suivante pour calculer la fonction récursive `W` :

```
let sum f l =
  List.fold_left (fun acc x -> Int64.add (f x) acc) 0L l
```

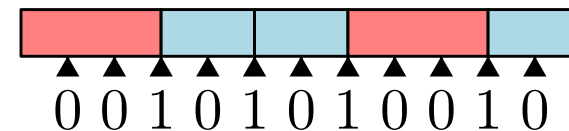
7/1

78/1

Réprésentation des rangées de briques

On représente les rangées de briques par des **entiers** en base 2 dont les chiffres 1 correspondent à la présence de jointures.

Par exemple, la rangée suivante



est représentée par l'entier $338 (= 00101010010_2)$

8

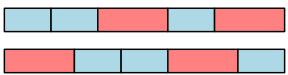
Avec cette représentation, il est facile de vérifier que deux rangées sont compatibles par une simple opération de ET logique (**land** en Ocaml).

► Rangées **compatibles**



$$\begin{array}{l} 00101010010_2 \\ \text{land } 01010100100_2 \\ = 00000000000_2 = 0 \end{array}$$

► Rangée **incompatibles**



$$\begin{array}{l} 01010010100_2 \\ \text{land } 00101010010_2 \\ = 00000010000_2 \neq 0 \end{array}$$

On construit par récurrence sur n une liste rows de toutes les rangées possibles de longueur n .

- Cas de base : les rangées de longueur 2 et 3 sont représentées par l'entier 0.
- Sinon, on calcule récursivement les rangées de longueur $n - 2$ et $n - 3$ et on utilise les deux fonctions suivantes :
 - add2 ajoute une brique de longueur 2 à droite d'une rangée par un double décalage logique à gauche (**lsl** en Ocaml) et en ajoutant 10_2 .
 - add3 ajoute une brique de longueur 3 d'une manière similaire.

Mais ça ne marche pas :-(

Dépassement de capacité

Le calcul prend beaucoup beaucoup trop de temps...

Le problème est qu'on retrouve souvent les mêmes couples (r, h) en argument de W , et donc qu'on calcule plusieurs fois la même chose.

- Pour remédier à ce problème, on va stocker dans une table $W(r, h)$ déjà calculés.
- C'est technique est la **mémoïsation**.

Il faut de grands entiers pour compter ce nombre de façons de construire ce mur.

- Il faut calculer avec des **entiers 64 bits**.

```
# 0L;;
- : int64 = 0L
# Int64.add 1L 4L;;
- : int64 = 5L
# Printf.printf "%Ld\n" 1L;;
1
- : unit = ()
```

- Le module `Int64` permet de manipuler des entiers 64 bits de type `int64`.
- On ajoute un `L` après les constantes entières pour créer des entiers 64 bits.
- L'addition est `Int64.add`, la multiplication `Int64.mul` etc.

Une table de hachage permet d'associer des clés à des valeurs à l'aide d'un tableau de taille `n` et d'une fonction de **hachage** :

`hash : clé → int`

- Pour une clé `k` associée à une valeur `v`, on range le couple (k, v) dans la case du tableau `hash(k) mod n`
- Comme plusieurs clés peuvent se retrouver dans la même case (on parle de **conflits**), on stocke une **liste** de couples dans chaque case.

13/1

114/1

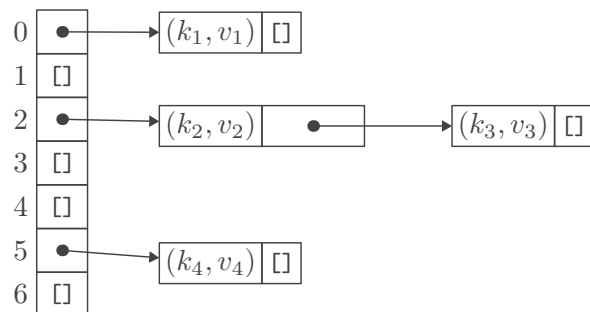
1

Les tables de hachage

Signature

Exemple :

- $n = 7$
- $\text{hash}(k_1) = 0 \bmod 7$
- $\text{hash}(k_2) = \text{hash}(k_3) = 2 \bmod 7$
- $\text{hash}(k_4) = 5 \bmod 7$



Une signature minimale pour des tables de hachage :

```
type ('a, 'b) t
```

```
val create : unit -> ('a, 'b) t
val cardinal : ('a, 'b) t -> int
val add : 'a -> 'b -> ('a, 'b) t -> unit
val find : 'a -> ('a, 'b) t -> 'b
val remove : 'a -> ('a, 'b) t -> unit
end
```

- Le type `('a, 'b) t` est celui des tables de hachage où `'a` est celui des clés et `'b` représente le type des valeurs associées aux clés.

15/1

116/1

1

Pour réaliser cette structure, il est nécessaire de disposer

- ▶ d'une fonction de hachage
- ▶ d'une fonction d'égalité sur le type 'a.

On utilisera la fonction générique de hachage fournit par OCaml :

```
Hashtbl.hash : 'a -> int
```

Pour l'égalité, on utilisera simplement l'opérateur =

Le type ('a, 'b) t des tables de hachage est défini de la manière suivante :

```
type ('a, 'b) t = {
  mutable size : int;
  buckets : ('a * 'b) list array;
}
```

17/1

118/1

1

Création d'une table de hachage

La création d'une table vide nécessite de choisir la taille du tableau des *buckets* :

```
let array_length = 5003
```

On crée une table de hachage vide avec un tableau de taille `array_length` ne contenant que des listes vides :

```
let create () =
  { size = 0;
    buckets = Array.create array_length []; }
```

19/1

120/1

Recherche d'un élément

La fonction `find` cherche la valeur associée à une clé dans la liste d'association qui lui correspond.

```
let find x h =
  let i = (Hashtbl.hash x) mod array_length in
  let rec lookup = function
    | [] -> raise Not_found
    | (k, v) :: _ when x = k -> v
    | _ :: b -> lookup b
  in
  lookup h.buckets.(i)
```

2

```
let add x v h =
  let i = (Hashtbl.hash x) mod array_length in
  let b = h.buckets.(i) in
  if not (mem_bucket x b) then begin
    h.size <- h.size + 1;
    h.buckets.(i) <- (x, v) :: b
  end
```

- On utilise la fonction suivante pour vérifier la présence d'une clé dans une liste d'associations :

```
let mem_bucket x =
  List.exists (fun (y, _) -> x = y)
```

La suppression d'un élément dans une table de hachage procède de manière similaire à l'insertion.

```
let remove x h =
  let i = (Hashtbl.hash x) mod array_length in
  let b = h.buckets.(i) in
  if mem_bucket x b then begin
    h.size <- h.size - 1;
    h.buckets.(i) <-
      List.filter (fun (y, _) -> y <> x) b
  end
```

On utilise une table de hachage pour **mémoïser** la fonction $W(r, h)$.

Pour cela, on écrit deux fonctions `w` et `memo_w` **mutuellement récursives** :

- `w` effectue le calcul, en appelant `memo_w` récursivement
- `memo_w` consulte la table, et si besoin appelle `w` pour la remplir

Le schéma suivant permet de mémoïser n'importe quelle fonction :

```
let table = create ()
let rec f x = ... memo_f x ...
and memo_f x =
  try find table x
  with Not_found ->
    let v = f x in add table x v; v
```

Mieux qu'un schéma de programmation

On peut capturer ce schéma de programmation directement dans une fonction grâce à l'ordre supérieur et au polymorphisme.

```
let memo ff =  
  let h = create 5003 in  
  let rec f x =  
    try find h x  
    with Not_found ->  
      let v = ff f x in add h x v; v  
  in f
```

On utilise la fonction `memo` de la manière suivante :

```
let f = memo (fun f x -> ... f x ...)
```

Système de modules

25/1

226/1

2

Unités de compilation

Le principe de base du génie logiciel est le découpage d'une application en plusieurs parties indépendantes appelées **unités de compilation**.

Cela permet notamment :

- ▶ de mieux **maîtriser** la complexité de logiciels de **grandes tailles**
- ▶ de réaliser un **développement en équipe**
- ▶ de **recompiler** rapidement un programme en ne recompilant que **ce qui est nécessaire** après une modification

Unités de compilation en Ocaml

En OCaml, chaque unité de compilation est un couple de deux fichiers : le fichier **interface** et le fichier **implémentation**

Ces fichiers portent le **même préfixe**, seules les extensions diffèrent :

- ▶ le fichier **interface** (`.mli`) définit les types (abstraits ou concrets) et les signatures des valeurs visibles à l'extérieur ;
- ▶ le fichier **implémentation** (`.ml`) contient les définitions (concrètes) de tous les types et de toutes les valeurs (visibles ou non) de l'unité de compilation.

27/1

228/1

2

- ▶ Si `module.mli` et `module.ml` sont les fichiers d'interface et d'implémentation d'une unité de compilation, on utilisera le nom `Module` pour désigner cette unité. On utilisera également la notation `Module.v` pour faire référence à la valeur `v` de `Module`.
- ▶ La directive `open Module` évite d'utiliser la notation pointée pour faire référence aux valeurs de `Module`.

Attention : si deux unités `M` et `N` contiennent la même valeur `v`, alors seule la déclaration de `N` est visible après les deux directives consécutives `open M` et `open N`.

On utilise un fichier d'interface pour spécifier quels types ou valeurs d'une implémentation sont accessibles de « l'extérieur »

Cela permet notamment :

- ▶ de **cacher** certains composants (type ou valeur) ;
- ▶ de **restreindre** le type de certains composants exportés ;
- ▶ de rendre **abstraits** certains types

La syntaxe utilisée dans les fichiers d'interface est la suivante :

- ▶ les valeurs sont déclarées en utilisant le mot-clé `val`.

```
val f : int -> 'a -> a list
```

- ▶ les types sont déclarés à l'aide du mot-clé `type`

```
type t = A | B
```

- ▶ les types abstraits sont des déclarations sans définitions

```
type t
```

- ▶ les fichiers d'interface doivent être compilés
`ocamlc -c fichier.mli`
- ▶ le fichier compilé porte l'extension `.cmi`
- ▶ seul le fichier `.cmi` est nécessaire pour la compilation séparée

Lors de la compilation d'un fichier d'implémentation `.ml`

- ▶ s'il n'y a pas d'interface, un fichier `.cmi` est généré automatiquement avec tous les types et valeurs exportés
- ▶ sinon, le compilateur vérifie que les types **inférés** sont « compatibles » avec les types **déclarés** dans l'interface

L'idée principale du découpage est que pour concevoir une unité de compilation il est seulement nécessaire de connaître les interfaces des autres unités.

- ▶ Lorsqu'une unité `M1` fait référence à une unité `M2`, on dit que `M1` **dépend** de `M2`.
- ▶ L'unité `M1` peut faire référence à `M2` soit dans son interface, soit dans son implémentation.
- ▶ Dans un programme avec plusieurs unités de compilation, la relation « **dépend de** » forme un **graphe de dépendances**.

Le graphe de dépendances définit une **ordre partiel** de compilation

- La phase de **compilation** effectue le **typage** et la production de codes **à trous** (on parle de fichiers **objets**)

L'option **-c** des compilateurs (**ocamlc** ou **ocamlopt**) permet de compiler sans faire d'édition de liens

Les fichiers objets portent l'extension **.cmo** (en *bytecode*) ou **.cmx** (en natif)

- La phase d'**édition de liens** construit un exécutable en « remplissant » les trous, selon l'ordre des fichiers donnés en arguments

33/1

Les notions de **modules** et **interfaces** sont en réalité plus fines que les fichiers et correspondent à des constructions du langage

On définit une interface I dans un programme comme ceci :

```
module type I = sig
  val a : int
  val f : int -> int
end
```

on définit un module M ayant cette interface comme ceci :

```
module M : I = struct
  let a = 42
  let b = 3
  let f x = a * x + b
end
```

le compilateur fait alors les mêmes opérations que si I était un fichier **.mli** et M un fichier **.ml**

334/1

3

Modules paramétrés

- Comme les fonctions, les modules peuvent avoir des **paramètres**
- Ces modules paramétrés sont des **foncteurs**
- Le langage impose que ces paramètres soient des **modules**

Voici par exemple la déclaration d'un module paramétré **M** ayant un module **S** de signature **T** en paramètre :

```
module M ( S : T ) = struct
  ...
end
```

Pour créer une instance de M, il suffit de l'appliquer à un module ayant la signature T. Par exemple, si on suppose que B est un module ayant la signature T, alors on crée une instance de M de la manière suivante :

```
module A = M(B)
```

35/1

Interface d'un dictionnaire

```
module type DICO = sig
  type key
  type 'a dico
  val empty : 'a dico
  val add : key -> 'a -> 'a dico -> 'a dico
  val mem : key -> 'a dico -> bool
  val find : key -> 'a dico -> 'a
  val remove : key -> 'a dico -> 'a dico
end
```

336/1

3

Un dictionnaire doit être **indépendant** du type des clés, mais il est important pour l'implémentation de pouvoir les **comparer**

On définit donc un dictionnaire comme un module paramétré par un module **Key** ayant la signature **ORDERED** suivante :

```
module type ORDERED = sig
  type t
  val compare : t -> t -> int
end

module MakeDico ( Key : ORDERED ) : DICO = struct
  ...
end
```