Programmation Fonctionnelle Avancée

Cours 7

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Résumé de l'épisode précédent

On a vu des primitives impératives :

- Structures : références, enregistrements avec champs mutables, tableaux
- Boucles : `while`, `for`

On peut construire des structures complexes à partir de ces briques de base :

- Piles et files impératives (**vus en TP** (ou pas))
- Tables de hachage (aujourd'hui)

Plan

1 PFA (1) : Rappels d'OCaml ✓
2 PFA (2) : Rappels d'OCaml (suite et fin) ✓
3 PFA (3) : Arbres binaires de recherche ✓
4 PFA (4) : Modules, Foncteurs et Compilation séparée ✓
5 PFA (5) : Modules, Foncteurs et Compilation séparée (2) ✓
6 PFA (6) : Traits impératifs ✓
7 PFA (7) : Tables de hachage, modèle mémoire
  7.1 Corrigé commenté du TP
  7.2 Tables de hachage
  7.3 module Hashtbl

Corrigé commenté du TP 6, EXO 2

On fait un corrigé commenté du TP (~15 à 20 minutes)

Plan

1 PFA (1) : Rappels d'OCaml ✓
2 PFA (2) : Rappels d'OCaml (suite et fin) ✓
3 PFA (3) : Arbres binaires de recherche ✓
4 PFA (4) : Modules, Foncteurs et Compilation séparée ✓
5 PFA (5) : Modules, Foncteurs et Compilation séparée (2) ✓
6 PFA (6) : Traits impératifs ✓
7 PFA (7) : Tables de hachage, modèle mémoire
  7.1 Corrigé commenté du TP ✓
  7.2 Tables de hachage
  7.3 module Hashtbl

Dictionnaire

Le type abstrait de dictionnaire est très utile (le plus utile ?)

Il propose d'associer des clés (« arbitraires », i.e pas forcéments des entiers de 0 à (n-1)) à des valeurs. Si on a un dictionnaire efficace, pour lequel la lecture/mise à jour est en temps constant, on peut implémenter toutes les autres types structurés :

- Couple : $\{ \texttt{"left"} \mapsto v_l; \texttt{"right"} \mapsto v_r \}$
- Tableau : $\{ 0 \mapsto v_0; \ldots; n \mapsto v_n \}$
- Référence : $\{ \texttt{"content"} \mapsto v \}$
- Liste (chaînée) : $\texttt{Cons} \equiv \{ \texttt{"value"} \mapsto v; \texttt{"next"} \mapsto v_{next} \}$ $\texttt{Nil} \equiv \{ \}$
- Arbre binaire : $\texttt{Node} \equiv \{ \texttt{"value"} \mapsto v; \texttt{"left"} \mapsto v_l; \texttt{"right"} \mapsto v_r \}$

$\texttt{Leaf} \equiv \{ \}$

Exemple de tels langages : JavaScript, AWK (outil unix), ...

La plupart des langages proposent toutes ces structures nativement plus des dictionnaires (impératifs)

Implémentation des dictionnaires

Nous savons déjà implémenter des dictionnaires persistants avec des ABR

```
module SMap = Map.Make(String)

let dico0 = SMap.(add "Hello" 5 empty) (*{"Hello" ↦ 5 }*)
let dico1 = SMap.(add "Foo" 3 dico0)  (*{"Hello" ↦ 5, "Foo" ↦ 3 }*)
let dico2 = SMap.(add "Bar" 3 dico1) (*{"Hello" ↦ 5, "Foo" ↦ 3, "Bar" ↦ 3 }*)
let dico3 = SMap.(add "Hello" 0 dico2)(*{"Hello" ↦ 0, "Foo" ↦ 3, "Bar" ↦ 3 }*)
```

Comment obtenir une version impérative ?

Première solution

On utilise la même approche que pour la pile impérative, une référence vers une structure persistante

```
module Dict(X : OrderedType) = struct
    module M = Map.Make(X)
    let create () = ref M.empty
    let find m k = M.find k !m
    let add m k v = m := M.add k v !m
    let remove m k v = m := M.remove k  !m
    let fold f m acc = M.fold f !m acc
    ...
end
module SDict = Dict(String)
let dico = SDict.create ()
let () =
  SDict.add dico "Hello" 5;
  SDict.add dico "Foo" 3;
  SDict.add dico "Bar" 3;
  SDict.add dico "Hello" 0
```

Remarque sur cette approche

C'est pas si mal !

- On garde les performances de la structure de base (i.e. ajout en O(log(n)))
- Le code n'est pas complexe : chaque fonction sort la map de la référence et la remet dedans.
- On peut ajouter des fonctionalités ou améliorer les performances

Exemple, dans les `Map` persistantes, la taille est calculée en comptant les éléments (linéaire)

Exemple

```
module Dict(X : OrderedType) = struct
  module M = Map.Make(X)
  type 'a t = { mutable map : 'a M.t;
                mutable size : int }
  let create () = { map = M.empty; size = 0 }
  let add m k v =
    let i = if M.mem k m.map then 0 else 1 in
    m.map <- M.add k v m.map;
    m.size <- m.size + i

  let remove m k v =
    if M.mem k m.map then begin
      m.map <- M.remove k m.map;
      m.size <- m.size - 1
    end

  let cardinal m = m.size (* temps constant *)
end
```

Peut-on faire mieux que O(log(N)) pour l'accès pour un dictionnaire mutable ?

Tables de hachage : recherche

On suppose deux fonctions :

```
val hash : 'a -> int
val equal : 'a -> int
```

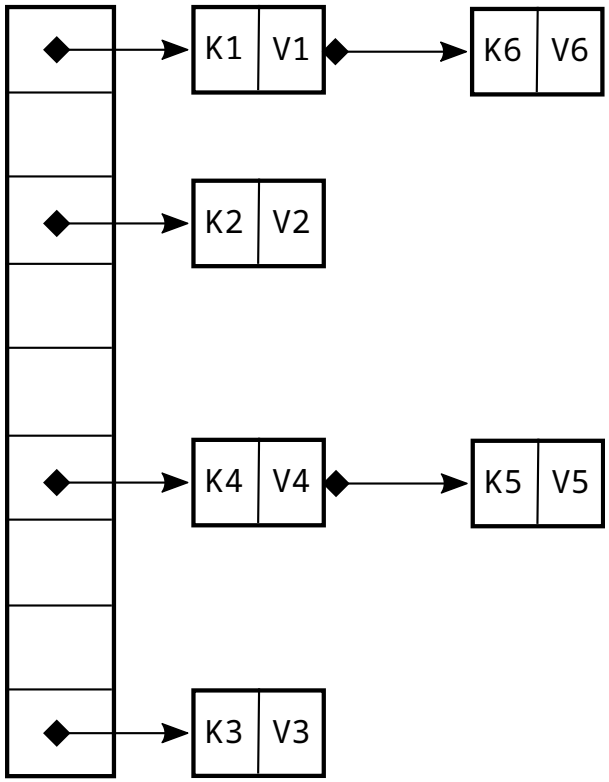
On peut implémenter ensuite la structure suivante

```
type ('k, 'v) t =
  { mutable data : ('k * 'v) list array; (* tableau de listes de paires *)
    mutable size : int; (* nombre d'éléments *)
  }

let find t k =
  let idx = (hash k) mod t.size in
  let bucket = t.data.(idx) in
  let rec loop l =
    match l with
      [] -> raise Not_found
    | (k0, v0) :: ll -> if equal k k0 then v0 else loop ll
  in loop bucket
```

Tables de hachage : ajout (v1)


```
let add t k v =
  let idx = (hash k) mod (Array.length t.data) in
  let bucket = t.data.(idx) in
  let rec loop l =
    match l with
      [] -> t.size <- t.size + 1; [ (k, v) ] (* mise à jour de t.size *)
    | (k0, v0) :: ll ->
           if equal k k0 then (k0,v)::ll
           else (k0,v0) :: loop ll
  in t.data.(idx) <- loop bucket

 (* Attention, add comporte un problème *)
```

Tables de hachage (suite)



Complexité en temps pour la recherche : proportionnelle à la longueur du *bucket* où se trouve(raît) la clé.
Deux clés peuvent se retrouver dans le même bucket pour deux raisons :

1. leurs *hashs* ont le même reste dans la division par la taille du tableau.
2. elles ont la même valeur de *hash*.

Tables de hachage (suite)

Quelle est la complexité moyenne attendue d'une recherche dans une table de hachage ?

O(1) (temps constant) amorti. Cette complexité en moyenne est atteinte si les *buckets* ne sont pas trop profonds, i.e. sont bornés par un facteur constant (en pratique un petit entier). Il faut pour cela deux conditions :

1. La table est agrandie et le contenu des *buckets* redistribué si un *bucket* déborde.
2. La fonction de hachage réalise une distribution uniforme des clés vers leur *hash*.

La fonction de hachage (`hash`) est toujours associée à une fonction d'égalité (`equal`) avec laquelle elle doit être cohérente :

$$\forall\ k_1\ \forall\ k_2\ ,\ \texttt{equal}\ k_1\ k_2\ \Rightarrow\ \texttt{hash}\ k_1 = \texttt{hash}\ k_2$$

Tables de hachage : ajout (v2)


```
let iter f t =
  for i = 0 to Array.length t.data - 1 do
    List.iter (fun (a,b) -> f a b) t.data.(i);
  done

let resize t =
  let nt = { data = Array.make (factor * Array.length t.data) []; size = 0 } in
  iter (fun k v -> add nt k v) t;
  t.size <- nt.size;
  t.data <- nt.data

let add t k v = add t k v; (* le add du slide 12 *)
  if factor * Array.length t.data > t.size then
  resize t
```

- Généralement, le facteur choisi est 2

Analyse de compléxité

- On suppose que le tableau est de taille initiale N, sans aucune valeur insérée
- On insère factor*N valeurs. Chaque insertion coûte la taille d'un « bucket »
- La factor*N + 1 ^{ème} insertion coûte factor*N + 1

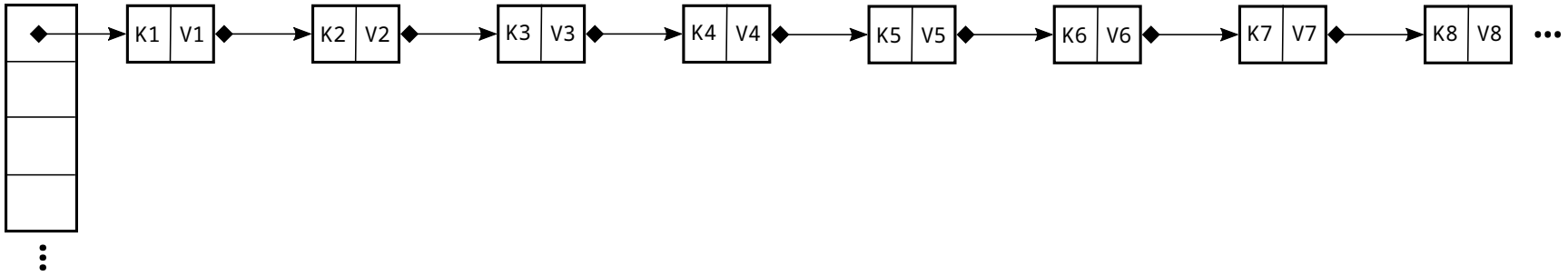
Si l'insertion dans un « bucket » est en temps constant, alors faire factor*N + 1 insertions coût factor*N + factor*N+1 = 2*factor*N+1

On met un temps linéaire pour faire un nombre linéaire d'insertions ⇒ Temps constant amorti.

Que se passe-t-il avec une mauvaise fonction de hachage ?

Cas dégénéré de la fonction constante

```
let hash _ = 0
```



malgré le redimensionnement des *buckets*, les entrées arrivent toujours dans le *bucket* 0, la complexité des opérations devient O(n). On appelle cela des collisions (deux valeurs ont des hash qui arrivent dans le même bucket).

Plan

1 PFA (1) : Rappels d'OCaml ✓
2 PFA (2) : Rappels d'OCaml (suite et fin) ✓
3 PFA (3) : Arbres binaires de recherche ✓
4 PFA (4) : Modules, Foncteurs et Compilation séparée ✓
5 PFA (5) : Modules, Foncteurs et Compilation séparée (2) ✓
6 PFA (6) : Traits impératifs ✓
7 PFA (7) : Tables de hachage, modèle mémoire
  7.1 Corrigé commenté du TP ✓
  7.2 Tables de hachage ✓
  7.3 module Hashtbl

module Hashtbl

Les deux fonctions

```
val Hashtbl.hash : 'a -> int (* dans le module Hashtbl *)
val equal : 'a -> 'a -> int (* visible de partout *)
```

Sont prédéfinies en OCaml. Elles sont exploitées par le module `Hashtbl` implémenter des tables de hachage génériques.

La fonction `hash` renvoie un entier positif pour toute valeur OCaml.

Pour éviter les collisions, la fonction utilisée est robuste (elle mélange bien tous les bits de la valeur). Elle peut éventuellement être initialisée avec un entier aléatoire pour ne pas pouvoir prévoir les valeurs de hash.

```
let h0 = Hashtbl.hash 0 (* 129913994 *)
let h1 = Hashtbl.hash 1 (* 883721435 *)
let h1000 = Hashtbl.hash (* 912777258 *)
let htoto = Hashtbl.hash "toto" (* 946608321 *)
let hpair = Hashtbl.hash ("foo", false) (* 523616810 *)
```

Fonction de hachage personalisée ?

Il est souvent souhaitable de redéfinir sa propre fonction de hachage et sa propre fonction d'égalité. On peut alors passer par le foncteur `Hashtbl.Make`

```
module type HashedType = sig
   type t
   val hash : t -> int
   val equal : t -> t -> bool
end
module HString = struct
  type t = string
  let hash s =
    let s = String.uppercase_ascci s in
    let acc = ref 0 in
    for i = 0 to String.length s - 1 do
      acc := 31* (Char.code s.[i]) + !acc;
    done;
    !acc
  let equal s1 s2 = String.((uppercase_ascci s1) (uppercase_ascci s2))
end
module Dico = Hashtbl.Make(HString)
```

Question subsidiaire

Peut on faire des dictionnaires persistants à partir des tables de hachage ?

Oui ...

```
module Map (K : HashedType) = struct
   module H = Hashtbl.Make(K)

   let create () = H.create 16
   let add k v h =
        let h2 = H.copy h in
        H.add h2 k v; h2

    ...
end
```

Combien coût l'insertion maintenant ? O(n) :(

Conclusion

On a vu toutes les briques de bases pour faire de la PF **A** :

- Structures persistantes (listes, ABR)
- Structures mutables (tableaux, tables de hachage)
- Les foncteurs et les modules (pour organiser le code)

On verra maintenant comment mélanger tout ça dans des concepts avancés pour faire de la programmation élégante et efficace