IPRF – Structures de données en OCaml

Christophe Mouilleron

ensiie
école nationale supérieure d'informatique
pour l'industrie et l'entreprise

Plan

Plan

Plan

Les ensembles

Interface pour la structure de données ensemble :

- insertion dans un ensemble
- retrait d'un ensemble
- test d'appartenance
- test si ensemble vide
- cardinal
- union / intersection / différence

Les ensembles

Interface pour la structure de données ensemble :

- insertion dans un ensemble $O(n)$
- retrait d'un ensemble $O(n)$
- test d'appartenance $O(n)$
- test si ensemble vide $O(1)$
- cardinal $O(n)$
- union / intersection / différence $O(nm)$

Implantation possible via des listes

Les ensembles

Interface pour la structure de données ensemble :

- insertion dans un ensemble — $O(\log n)$
- retrait d'un ensemble — $O(\log n)$
- test d'appartenance — $O(\log n)$
- test si ensemble vide — $O(1)$
- cardinal — $O(n)$
- union / intersection / différence — $O(n+m)$

Implantation usuelle via des arbres équilibrés

Structure indépendante du type des éléments

Difficultés :

- arbres de recherche $\Rightarrow$ usage d'une fonction de comparaison
- ordre variable suivant l'application

Structure indépendante du type des éléments

Difficultés :

- arbres de recherche $\Rightarrow$ usage d'une fonction de comparaison
- ordre variable suivant l'application

Solution 1 : passer l'ordre en argument à toutes les fonctions

✗ pénible

✗ source d'erreur si plusieurs ordres

Structure indépendante du type des éléments

Difficultés :

- arbres de recherche $\Rightarrow$ usage d'une fonction de comparaison
- ordre variable suivant l'application

Solution 1 : passer l'ordre en argument à toutes les fonctions

✗ pénible

✗ source d'erreur si plusieurs ordres

Solution 2 : créer un nouveau type à partir d'un ordre

✓ ordre fixé une fois pour tout $\Rightarrow$ code plus court et plus fiable

✗ un peu moins générique un type par ordre

Le foncteur `Set.Make`

En OCaml : solution 2

module = regroupement de définitions de types et de fonctions

- `List`, `Printf`, etc.
- fichier `foo.ml` $\rightsquigarrow$ module `Foo`

foncteur = *fonction* qui crée un module à partir de modules existants

- version propre du mécanisme de template compil. séparée

Le foncteur `Set.Make`

En OCaml : solution 2

module = regroupement de définitions de types et de fonctions

- `List`, `Printf`, etc.
- fichier `foo.ml` $\rightsquigarrow$ module `Foo`

foncteur = *fonction* qui crée un module à partir de modules existants

- version propre du mécanisme de template compil. séparée

Pour les ensembles :

- un module pour lier un type et un ordre
- un foncteur `Set.Make`

Le foncteur `Set.Make` – Exemple

Pour définir les ensembles d'entiers en OCaml :

1. On définit le module `Int` pour les entiers ordonnées.

```
module Int =
  struct
    type t = int
    let compare = fun x y -> x - y
  end
;;
```

2. On crée le nouveau module `IntSet`.

```
module IntSet = Set.Make(Int) ;;
```

Fonctions disponibles sur les ensembles

Quelques fonctions disponibles après appel au foncteur `Set.Make` :

- `empty` = constante correspondant à l'ensemble vide — $O(1)$
- `is_empty` = test si ensemble vide — $O(1)$
- `cardinal` — $O(n)$
- `mem` = test d'appartenance — $O(\log n)$
- `add` / `remove` = ajout / retrait — $O(\log n)$
- `union` / `inter` / `diff` — $O(n \log n)$
- `fold` (mais pas `map`)

Plan

Les associations

Interface pour une association :

- ajout d'une entrée (clé + valeur) $O(\log n)$
- retrait d'une entrée via sa clé $O(\log n)$
- test d'appartenance d'une clé $O(\log n)$
- recherche de la valeur associée à une clé $O(\log n)$
- nombre d'associations clé/valeur $O(n)$

Implantation usuelle similaire à celle des ensembles

Association générique

Contraintes ?

- on doit pouvoir comparer des clés
- aucune contrainte sur le type des valeurs

En OCaml : foncteur `Map.Make`

- prend un type ordonné en argument — type des clés
- renvoie un module générique
  - → utilisable pour n'importe quel type de valeurs

Exemple :

```
module Annuaire = Map.Make(String) ;;
Annuaire.add "Bob" [4; 2] Annuaire.empty ;;
```

Fonctions disponibles sur les associations

Quelques fonctions disponibles après appel au foncteur `Map.Make` :

- `empty` = constante correspondant à l'association vide $O(1)$
- `is_empty` = test si association vide $O(1)$
- `cardinal` $O(n)$
- `mem` = test d'appartenance d'une clé $O(\log n)$
- `find` = trouve la valeur associée à une clé donnée $O(\log n)$
- `add` / `remove` = ajout / retrait $O(\log n)$
- `fold` (clé + valeur) / `map` (valeur)

Plan

1. Ensembles et associations
   - Module `Set`
   - Module `Map`

2. Applications
   - Résolution de SUBSET SUM avec backtracking
   - Modélisation des graphes orientés

Plan

Problème SUBSET SUM

Entrée : un ensemble s de valeurs dans $\mathbb{N}$ et une somme à atteindre t

Questions :

- existe-t-il $s' \subset s$ dont la somme des éléments fait t ?
- trouver une telle solution
- trouver toutes les solutions

Exemple avec $s = \{1, 2, 4, 9\}$:

- $t = 6$? Oui $\{2, 4\}$
- $t = 8$? Non

Solution en OCaml – version 1

Idée :

- $t = 0 \Rightarrow$ gagné
- $t < 0$ ou $s = \emptyset \Rightarrow$ perdu
- sinon, prendre un élément e de s et essayer avec et sans e

On reprend le module `IntSet` vu précédemment :

```
let rec subset_sum_v1 = fun s -> fun t ->
  if t = 0 then true
  else if t < 0 || IntSet.is_empty s then false
  else let e  = IntSet.choose s in
       let s' = IntSet.remove e s in
       subset_sum_v1 s' (t-e) || subset_sum_v1 s' t ;;
```

Solution en OCaml – version 2

Version où on retourne une solution :

- ⇝ ajout d'un paramètre `acc` de type `IntSet.t` initialement $\emptyset$
- ⇝ utilisation d'un type option : `None | Some sol`

```
let rec subset_sum_v2 = fun s -> fun t -> fun acc ->
  if t = 0 then Some acc
  else if t < 0 || IntSet.is_empty s then None
  else let e    = IntSet.choose s    in
       let s'   = IntSet.remove e s  in
       let acc' = IntSet.add e acc   in
       match subset_sum_v2 s' (t-e) acc' with
       | None     -> subset_sum_v2 s' t acc
       | Some sol -> Some sol ;;
```

Solution en OCaml – version 3

Version avec utilisation d'une exception :

```
exception Found_sol of IntSet.t;;

let rec aux = fun s t acc ->
  if t = 0 then raise (Found_sol acc)
  else if t < 0 || IntSet.is_empty s then None
  else let e    = IntSet.choose s    in
       let s'   = IntSet.remove e s  in
       let acc' = IntSet.add e acc   in
       let _    = aux s' (t-e) acc'  in
       aux s' t acc ;;

let subset_sum_v3 = fun s t ->
  try  aux s t IntSet.empty
  with Found_sol sol -> Some sol ;;
```

Plan

Définition d'un type `graph`

Un graphe est un ensemble de sommets . . .

- on numérote ces sommets avec des entiers — par simplicité
- on se donne un ordre sur les sommets

```
module Int =
  struct
    type t = int
    let compare = fun x y -> x - y
  end ;;
```

- on définit le type des ensembles de sommets

```
module IntSet = Set.Make(Int) ;;
```

Définition d'un type `graph` (suite)

. . . et un ensemble d'arêtes

- représentation via une association sommet $\rightarrow$ successeurs
- on définit le type des associations avec des sommets pour clés

```
module IntMap = Map.Make(Int) ;;
```

- on définit enfin le type `graph`

```
type graph = IntSet.t IntMap.t ;;
```

Opérations sur les graphes

- graphe vide — `IntMap.empty`
- ajout d'un sommet `v` — `IntMap.add v IntSet.empty`
- ajout d'une arête de `u` vers `v` =
  - ajouter `v` aux successeurs de `u` — `IntMap.find` + `IntSet.add`
  - mettre à jour les successeurs de `u` — `IntMap.add`
- tester si `v` successeur de `u` — `IntMap.find` + `IntSet.mem`
- traitement sur tous les sommets — `IntMap.fold`
- traitement sur toutes les arêtes — `IntMap.fold` + `IntSet.fold`