Programmation fonctionnelle avancée

Notes de cours

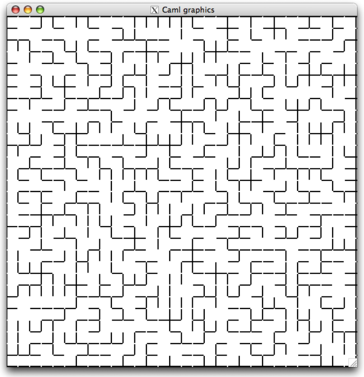
Cours 6

14 octobre 2015

Sylvain Conchon

sylvain.conchon@lri.fr

Programme : Construction d'un labyrinthe



Notions introduites :

- Le problème des classes disjointes
- Implémentation impérative de la structure *union-find*
- Le *Knuth Shuffle*

Construction d'un labyrinthe

- Le programme construit incrémentalement une relation d'équivalence sur l'ensemble E des cases du quadrillage.
- Chaque case (i, j) est « liée » avec sa voisine de coordonnées $(i, j+1)$ ou $(i+1, j)$.
- Le programme se termine quand la partition de E est réduite à une seule classe d'équivalence.

Relation d'équivalence

Une relation d'équivalence sur un ensemble rassemble des éléments « équivalents » dans une même classe.

Plus formellement, une relation d'équivalence $\mathcal{R}$ sur un ensemble $\mathcal{S}$ a les propriétés suivantes :

- réflexive : $\forall x \in \mathcal{S}.\ x\mathcal{R}x$
- symétrique : $\forall x, y \in \mathcal{S}.\ x\mathcal{R}y \Leftrightarrow y\mathcal{R}x$
- transitive : $\forall x, y, z \in \mathcal{S}.\ x\mathcal{R}y \wedge y\mathcal{R}z \Rightarrow x\mathcal{R}z$

Exemple :

- La relation $\mathcal{R}$ sur $\mathbb{N}$ définie par $x\mathcal{R}y$ ssi $x - y$ est pair.
- La relation $\mathcal{R}$ sur l'ensemble E des élèves d'un lycée définie par $x\mathcal{R}y$ ssi x et y sont dans la même classe.

Classe d'équivalence

Si un ensemble E est muni d'une relation d'ordre $\mathcal{R}$, on appelle classe d'équivalence de x suivant $\mathcal{R}$, l'ensemble

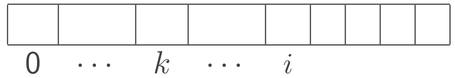
$$\{y \in E \mid y\mathcal{R}x\}$$

- Un élément appartenant à la classe d'équivalence de x est appelé représentant de la classe de x.
- L'ensemble des classes d'équivalence d'un ensemble E forme une partition de E.

Remarque : Une classe n'est jamais vide (tout élément est équivalent à lui-même).

L'algorithme de mélange de Knuth

Cet algorithme permet de mélanger un tableau de valeurs par génération d'une permutation aléatoire des éléments.



- On parcourt le tableau de la case `n-1` à `1`.
- Pour chaque case i, on tire au sort un nombre k entre 0 et i.
- On échange le contenu des cases i et k

Propriétés du *Knuth shuffle* :

- équitable (toutes les permutations ont la même chance d'être générées si la génération de nombres aléatoires est équitable).
- compléxité linéaire.
- aucun espace alloué supplémentaire.

Implémentation du `Knuth Shuffle`

```
let knuth_shuffle t =
 for j = Array.length t - 1 downto 1 do
  let k = Random.int (j+1) in
  let v = t.(j) in t.(j) <- t.(k); t.(k) <- v
 done
```


Union-Find

Ce programme utilise une structure de données impérative pour le problème des classes disjointes, connue sous le nom de union-find.

- Ce problème consiste à maintenir dans une structure de données une partition d'un ensemble fini.
- Sans perte de généralité, on peut supposer qu'il s'agit de l'ensemble des n entiers $\{0, 1, \ldots, n-1\}$.

Signature pour *union-find*

```
type t

val create : int -> t
val find : t -> int -> int
val union : t -> int -> int -> unit
```

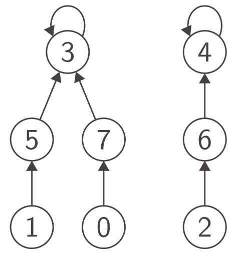
- L'opération `create n` construit une nouvelle partition de $\{0, 1, \ldots, n-1\}$ où chaque élément forme une classe à lui tout seul.
- L'opération `find` détermine la classe d'un élément, sous la forme d'un entier considéré comme le représentant de cette classe.
- Enfin l'opération `union` réunit deux classes de la partition, la structure de données étant modifiée en place.

Réprésentation des partitions

L'idée principale est de lier entre eux les éléments d'une même classe

- Dans chaque classe, ces liaisons forment un graphe acyclique où tous les chemins mènent au représentant, qui est le seul élément lié à lui-même.

La schéma suivant illustre une situation où l'ensemble $\{0, 1, \ldots, 7\}$ est partitionné en deux classes dont les représentants sont respectivement 3 et 4.

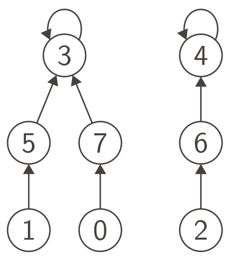


La structure est donc une forêt de graphes acycliques

Implémentation avec des tableaux

Une manière simple de réaliser cette forêt consiste à utiliser un tableau qui lie chaque entier i à un autre entier de la même classe.

Ainsi la partition du schéma de gauche est représentée par le tableau de droite :

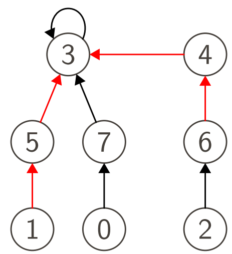


0	1			...			7
7	5	6	3	4	3	4	3

Réalisation de `find` et `union`

- L'opération `find` se contente de suivre les liaisons jusqu'à trouver le représentant d'un élément.
- L'opération `union` commence par trouver les représentants des deux éléments, puis lie l'un des deux représentants à l'autre.

Exemple : `find 1` suit les liaisons rouges à partir du nœud 1 pour trouver son représentant (ici 3). De même, `union 1 6` trouve tout d'abord les représentants de 1 et 6 (ici 3 et 4) puis lie le nœud 4 au nœud 3.



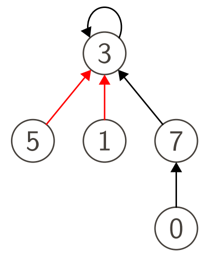
Amélioration

Afin d'atteindre de bonnes performances, on apporte deux améliorations.

- La compression de chemins
- L'équilibrage des classes

La compression de chemins

On *compresse les chemins* pendant la recherche effectuée par `find` : cela consiste à lier directement au représentant tous les éléments trouvés sur le chemin parcouru pour l'atteindre.



L'équilibrage des classes

- On maintient pour chaque représentant une approximation de la taille de sa classe, comme un un majorant de la longueur du plus long chemin dans sa classe.
- Cette information est stockée dans un second tableau et utilisée par la fonction `union` pour choisir le représentant d'une union.

Type de la structure *union-find*

Le type `t` est un enregistrement contenant deux tableaux :

- `rank` qui contient l'information sur la taille de chaque classe ;
- `parent` qui contient les liaisons.

```
type t = {
 rank : int array;
 parent : int array;
}
```

Classes initiales

- Chaque élément forme une classe à lui tout seul, c'est-à-dire que chaque élément est son propre représentant.
- La taille de chaque classe vaut 0.

```
let create n =
  { rank = Array.create n 0;
    parent = Array.init n (fun i -> i) }
```


La fonction find

```
let rec find t i =
  let p = t.parent.(i) in
  if p = i then
    i
  else begin
    let r = find t p in
    t.parent.(i) <- r;
    r
  end
```

Un appel `find t i` commence par calculer le parent `p` de `i`.

- Si c'est `i` lui-même, on a terminé et `i` est le représentant de la classe.
- Sinon, il suffit de calculer récursivement le représentant `r` comme `find t p`.

Cependant, pour réaliser la compression de chemins, on modifie le parent de `i`, pour lui donner la valeur `r`, avant de renvoyer `r`.

La fonction union (1/3)

Pour réaliser l'union des classes de deux éléments `i` et `j`, on commence par calculer leurs représentants respectifs `ri` et `rj`.

S'ils sont égaux, il n'y a rien à faire.

```
let union t i j =
  let ri = find t i in
  let rj = find t j in
  if ri <> rj then begin
  ...
```

Sinon, on compare la taille des deux classes...

La fonction union (2/3)

Si la classe de `ri` est strictement plus petite que celle de `rj`, on fait de `rj` le représentant de l'union, *i.e.* le parent de `ri`.

```
if t.rank.(ri) < t.rank.(rj) then
  t.parent.(ri) <- rj
```

Si en revanche la classe de `rj` est la plus petite, on procède symétriquement.

```
else begin
  t.parent.(rj) <- ri;
```

La fonction `union` (3/3)

L'information de taille n'a besoin d'être mise à jour que dans le cas où les deux classes ont la même taille, car c'est dans ce cas que la longueur du plus long chemin est susceptible d'augmenter.

```
    if t.rank.(ri) = t.rank.(rj) then
      t.rank.(ri) <- t.rank.(ri) + 1
    end
  end
```

Il est important de noter que la fonction `union` utilise la fonction `find` et donc réalise deux compressions de chemin, même dans le cas où il s'avère que `i` et `j` sont dans la même classe.

Tableaux persistants

Une structure de tableaux

On souhaite concevoir une structure de tableaux fonctionnels, c'est-à-dire où chaque écriture crée un nouveau tableau.

Mais on souhaite également avoir une structure efficace où les opérations `set` et `get` sont de même efficacité que celles des tableaux standard, i.e. $O(1)$, tant qu'on n'utilise pas le caractère persistant.

En revanche, on accepte de payer un certain coût lorsqu'on accède à des versions antérieures du tableau.

Signature des tableaux persistants

La signature d'une structure de tableaux persistants est la suivante :

```
type 'a t
val init : int -> (int -> 'a) -> 'a t
val length : 'a t -> int
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> 'a t
val iteri : (int -> 'a -> unit) -> 'a t -> unit
```

C'est exactement la signature des tableaux, à ceci près que la fonction `set` renvoie un nouveau tableau persistant, sans altérer son argument.

Principe

L'idée de base consiste à utiliser un tableau standard pour la version la plus récente du tableau persistant et, pour les versions antérieures, de l'information supplémentaire qui nous permettra de revenir en arrière.

```
type 'a t = 'a data ref
and 'a data =
  | Arr of 'a array
  | Diff of int * 'a * 'a t
```

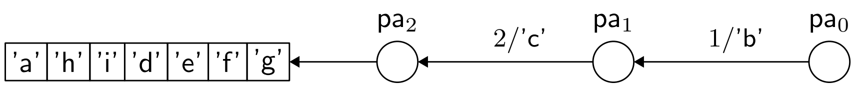

Exemple d'utilisation (1/4)

Illustrons l'utilisation de cette structure de données sur un exemple. On considère la série de déclarations suivantes définissant un tableau persistant pa0 puis deux autres, pa1 et pa2, obtenus par deux modifications successives :

```
let pa0 = init 7 (fun i -> Char.chr (Char.code 'a' + i))
let pa1 = set pa0 1 'h'
let pa2 = set pa1 2 'i'
```


Exemple d'utilisation (2/4)

La situation à l'issue de ces trois déclarations est la suivante :



pa2 est une référence vers `Arr [|'a';'h';'i';'d';'e';'f';'g'|]`

pa1 est une référence vers `Diff (2, 'c', pa2)`

pa0 une référence vers `Diff (1, 'b', pa1)`

Exemple d'utilisation (3/4)

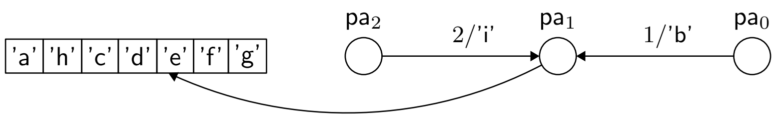
On effectue maintenant l'affectation suivante :

```
let pa3 = set pa1 2 'j'
```

Pour cela, il faut s'assurer que pa1 soit de la forme `Arr a` en effectuant trois opérations.

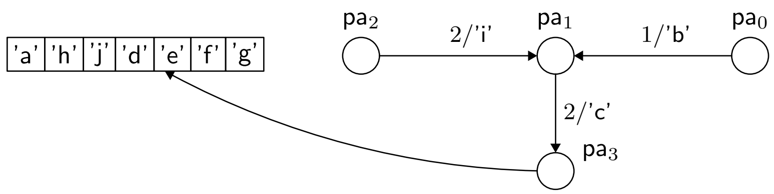
Exemple d'utilisation (4/4)

1. On inverse la chaîne de `Diff` menant de `pa1` au tableau `a` :



2. On crée une nouvelle référence pour `pa3`, contenant `Arr a`, où le contenu de `a` a été modifié pour contenir `'j'` en case 2.

3. On modifie la référence `pa1` pour qu'elle contienne maintenant `Diff (2, 'c', pa3)`.



Initialisation

La création d'un nouveau tableau persistant est immédiate :

```
let init n f = ref (Arr (Array.init n f))
```


Reroot

La fonction `reroot : 'a t -> 'a array` inverse une chaîne de `Diff` et renvoie le tableau standard obtenu par ce renversement :

```
let rec reroot pa = match !pa with
  | Arr a ->
      a
  | Diff (i, v, pa') ->
      let a = reroot pa' in
      let old = a.(i) in
      a.(i) <- v;
      pa := Arr a;
      pa' := Diff (i, old, pa);
      a
```


Longueur d'un tableau persistant et accès aux éléments

Grâce à `reroot`, les fonctions `length` et `get` sont immédiates en utilisant leurs équivalents du module `Array`

```
let length pa = Array.length (reroot pa)

let get pa i = (reroot pa).(i)
```

Itérateur `iteri`

L'opération `iteri` est simplement réalisée à l'aide de la fonction `Array.iteri`, qui applique une fonction à tous les élément d'un tableau en lui passant également l'indice de la case.

```
let iteri f pa = Array.iteri f (reroot pa)
```


Modification

La fonction `set` construit un nouveau tableau persistant à partir d'un tableau persistant `pa`, d'un index `i` et d'une valeur `v`.

```
let set pa i v =
  let a = reroot pa in
  let old = a.(i) in
  a.(i) <- v;
  let res = ref (Arr a) in
  pa := Diff (i, old, res);
  res
```