Pointeurs

Jean-Baptiste Rouquier

Pour profiter du canevas de code source, ouvrez le fichier pointeurs.ml.

Un pointeur est une adresse indiquant l'endroit où l'on peut trouver une valeur. L'exemple de base sont les références (le type `ref`) : une valeur de type `'a ref` n'est pas directement le type `'a` mais indique où l'on peut trouver la valeur de type `'a`.

Cela permet par exemple de donner à une fonction l'adresse de ses arguments plutôt que ses arguments eux-mêmes. Comme les adresses ont une taille fixe, on peut donner rapidement à une fonction un argument de grande taille.

Cela permet aussi à la fonction de modifier le contenu de cette adresse. Si l'on donne un entier n à la fonction `succ` (successeur), elle va *renvoyer* $n+1$ car elle ne peut pas (heureusement !) changer la valeur de n. En revanche la fonction `incr` prend une référence et *modifie* son contenu, elle ne renvoie que unit. En Caml il faut déclarer explicitement cette possibilité pour les enregistrements par le mot-clef `mutable`, ainsi les références peuvent être implémentées par

```
type 'a ref = {mutable contents : 'a}
```

Une troisième possibilité courante est la construction de structures de données. Un arbre peut être vu comme un ensemble de boîtes, chaque boîte contenant un élément et un pointeur vers son père (sauf pour la racine). Ou au contraire chaque chaque boîte contenant une liste de pointeurs, chacun vers un fils.

Il y a en Caml plusieurs types pour représenter les pointeurs. Contrairement à C il n'y a pas de pointeur `NULL`, on utilise pour cela le type option, plus propre. Seule les références correspondent précisément aux pointeurs de C. Mais d'autres types utilisent des pointeurs de façon confortable.

Les différents champs des enregistrements sont chacun un pointeur. Une liste non vide est également constituée d'un élément et d'un pointeur sur une liste de même type. Les listes (comme les références) bénéficient d'une syntaxe particulière pour faciliter leur utilisation.

Enfin on peut considérer un index dans un tableau comme un pointeur : cet entier n'est pas directement la valeur mais l'endroit où l'on peut la trouver. On peut donner cette adresse à une fonction qui pourra modifier la valeur stockée, et le TP sur les tas a montré que les tableaux pouvaient représenter des données structurées.

Remarque. C'est sur les pointeurs que se fait la distinction entre `==` (ou `!=`) qui teste si les adresses sont les mêmes (précisément, si les deux arguments sont stockés au même endroit dans la RAM), tandis que `=` (ou `<>`) teste si les contenus sont identiques (en les parcourant récursivement, ce qui ne termine pas si les structures de données sont cycliques).

1 Tas de Tarjan

Aussi appelés structure ***union-find***. On va implémenter une structure de données impérative codant des *relations d'équivalence*, permettant d'une part de fusionner deux classes (union) et d'autre part de trouver un représentant canonique d'une classe en en connaissant un élément (find). Disposer d'un représentant canonique permet en particulier de dire si deux éléments sont dans la même classe.

Un élément est dit ***réductible*** s'il n'est pas le représentant canonique de sa classe. Un ensemble de n éléments sera représenté par un tableau de taille n dont les cases contiennent le type

```
type element = Representant | Reductible of int
```

On a donc

```
type classes = element array
```

Un élément réductible pointe vers un autre élément de la classe.

Exemple. On considère l'ensemble $[\![0, 9]\!]$ et la relation d'équivalence $\sim$ définie par

$$x \sim y \iff x \equiv y \mod 3$$

Cette relation a trois classes d'équivalence, on choisit pour représentants canoniques les éléments 0, 1 et 2. On peut coder cette relation par exemple par la valeur

```
let modulo_3 =
  [|Representant; Representant; Representant;
    Reductible 0; Reductible 1; Reductible 2;
    Reductible 3; Reductible 4; Reductible 2;
    Reductible 0|]
```

Question 1.1. Écrire la fonction `isoles : int -> classes` qui prend la taille de l'ensemble et renvoie une relation d'équivalence où chaque élément est seul dans sa classe. Chaque élément est donc le représentant canonique de sa classe (remarque : c'est la relation d'équivalence «égalité»).

Pour trouver le représentant canonique de la classe de l'élément i, deux cas sont possibles : soit i est déjà le représentant canonique de sa classe (et c'est fini), soit il pointe vers un élément j, et l'on recommence avec j.

Question 1.2. Écrire la fonction qui renvoie le représentant canonique de la classe d'un élément :

```
val find : classes -> int -> int
```

Remarque. Pour s'assurer que cette fonction termine, il faudra donc imposer qu'il n'y a jamais de cycles dans la structure de données. Comme il n'y en a pas au début, il suffit de vérifier que `union` n'en crée pas. Si l'on dessine le graphe dont les sommets sont les éléments et ayant un arc partant de chaque élément réductible (arc qui va bien sûr sur l'élément pointé par ce réductible), on obtient une forêt dont chaque arbre est une classe.

Pour fusionner les classes des éléments i et j, il suffit de déclarer que le représentant canonique de la classe de i n'est plus représentant canonique mais réductible et qu'il pointe vers le représentant canonique de la classe de j.

Question 1.3. Écrire la fonction qui fusionne deux classes :

```
val union : classes -> int -> int -> unit
```

Question 1.4. Que se passe-t-il si i et j sont dans la même classe ? Modifier si besoin la fonction `union` pour que `find` termine toujours.

Question 1.5 (bonus). Cet algorithme est presque linéaire (en théorie, et linéaire dans la pratique) si on lui ajoute l'astuce suivante. Comme on effectue beaucoup de fusions, trouver un représentant canonique demande de passer par de plus en plus d'éléments. L'idée est donc, chaque fois que l'on appelle `find i` sur un élément i réductible, de faire pointer i directement sur le représentant canonique de sa classe au lieu de l'élément sur lequel il pointait avant. On court-circuite ainsi le chemin que l'on suivait à chaque fois dans la version précédente.

Intégrer cette modification à `find`.

Question 1.6. Tester ces fonctions à l'aide du code fourni et vérifier les résultats.

Remarque. Cette structure union-find est utilisée dans l'implémentation de l'algorithme de Kruskal.

2 Arbres aléatoires

Voici un algorithme pour tirer au sort un arbre binaire avec la loi uniforme parmi tous les arbres binaires ayant un nombre fixé de sommets[1].

[1] J.-L. Rémy, Un procédé itératif de dénombrement d'arbres binaires et son application a leur génération aléatoire. RAIRO Informatique Théorique 19 (1985), 179-195.

Soit C_n le nombre d'arbres binaires à n nœuds internes. $C_n = \frac{1}{n+1}\mathrm{C}_{2n}^n$ est le n$^{\mathrm{e}}$ nombre de Catalan. L'algorithme de Rémy consiste à interpréter comme une relation de récurrence constructive le fait que

$$(n+1)\,C_n \;=\; 2\,(2n-1)\,C_{n-1}$$

On peut interpréter $(n+1)\,C_n$ comme le nombre d'arbres à n nœuds internes (et donc $n+1$ feuilles) ayant un feuille marquée. $2\,(2n-1)\,C_{n-1}$ est le nombre d'arbres à $n-1$ nœuds internes dont on a marqué l'un des $2n-1$ nœuds (internes ou feuilles), et pour lesquels on a en plus choisi un booléen (gauche ou droite).

Il suffit alors de trouver une bijection entre ces deux ensembles. Pour ajouter un nœud interne et une feuille à un arbre ayant $n-1$ nœuds internes, on prend le sous-arbre du nœud marqué, on remplace le nœud marqué par un nouveau nœud dont le fils gauche (resp. droit) est une feuille si le booléen est gauche (resp. droit) et l'autre est le sous-arbre précédent. Cette nouvelle feuille est celle qui est marquée pour la bijection réciproque.

On ajoute ainsi un nœud interne à chaque étape. Il suffit d'itérer cette construction en tirant au sort le nœud à marquer et le booléen à chaque étape, et en oubliant la marque de la feuille.

On représente un arbre par un tableau, chaque case contenant `None` si c'est une feuille, l'index de ses deux fils i et j si c'est un nœud interne (i.e. `Some(i,j)`).

Question 2.1. Écrire la fonction `random_tree_vect : int -> (int * int) option array` qui prend le nombre n de nœuds internes et renvoie un arbre aléatoire à $2n+1$ sommets selon cette représentation.

Pour tirer au sort un booléen, on peut utiliser la fonction `Random.bool : unit -> bool`. Pour un entier, `Random.int n` renvoit un entier de $[\![0, n[\![$ selon la loi uniforme (n exclu, donc). De même `Random.float x` renvoie un flottant de $[0; x[$ selon la loi uniforme.

Question 2.2. On pose `type tree = Leaf | Node of tree * tree`, écrire la fonction de conversion `tree_of_vect : (int * int) option array -> tree` et la fonction `random_tree : int -> tree` qui prend aussi le nombre de nœuds internes.

Pour dessiner des arbres le module `Draw_tree` est fourni avec le TP. Pour le compiler, copiez le fichier `draw_tree.ml` et `draw_tree.mli` dans le même répertoire que votre code source puis tapez dans une console

```
ocamlc -c draw_tree.mli draw_tree.ml
```

Question 2.3. Tester avec le code fourni qui permet de dessiner les arbres (sélectionner la fenêtre graphique de Caml et appuyer sur une touche pour passer à l'arbre suivant).

3 Liste doublement chaînées

Les listes de Caml sont
- simplement chaînées : chaque cellule d'une liste de type `'a list` contient un élément de type `'a` et un pointeur vers la cellule suivante ;
- fonctionnelles : on ne peut pas modifier le troisième élément d'une liste. Il faut pour cela construire une nouvelle liste ayant ses deux premiers éléments identiques et une troisième cellule contenant l'élément modifié et pointant sur la quatrième cellule de la première liste. Cette nouvelle liste n'utilise donc que trois cases mémoire supplémentaires, à partir du rang 4 les éléments sont physiquement au même endroit que ceux de la première liste. On nomme cela le partage.
  En contrepartie de l'immutabilité, on gagne beaucoup de mémoire grâce au partage, et lorsque l'on passe une liste à une fonction on est assuré que cette fonction ne va pas modifier la liste.

Nous allons implémenter des listes
- doublement chaînées : chaque cellule d'une liste contient un élément, un pointeur vers la cellule précédente et un pointeur vers la cellule suivante ;
- impératives : on peut modifier les listes en place, comme les tableaux (mais le coût d'accès à un élément n'est pas constant).

Une cellule sera représentée par le type `cell` ; une liste (type `liste`) contient un pointeur vers la première cellule et un vers la dernière :

```
type 'a cell = {
  mutable a:'a;
  mutable prec:'a cell option;
  mutable suiv:'a cell option}
type 'a liste = {
  mutable deb:'a cell option;
  mutable fin:'a cell option}
```

On veillera à ce que le pointeur `prec` de la cellule «`suiv`» et le pointeur `suiv` de la cellule «`prec`» pointent sur la cellule courante. La liste vide sera représentée par les deux pointeurs `deb` et `fin` égaux à `None`, et ce sera le seul cas où l'un de ces pointeurs pourra être `None`.

Attention, cette structure de donnée est cyclique (une cellule contient un pointeur vers la cellule suivante qui contient un pointeur vers la cellule de départ). Il ne faut donc pas utiliser de comparaisons `=` ou `<>`, ni demander au toplevel d'afficher une valeur de ce type (il s'arrêterait quand même, mais l'affichage est de toute façon peu utilisable).

On définit l'exception `Empty` (voir le code fourni) et l'on écrira `raise Empty` (au lieu de `failwith "liste vide"`) pour signaler l'erreur dans les cas où une fonction tente d'accéder à un élément d'une liste vide.

Question 3.1. Écrire les fonctions de construction de listes à zéro ou un élément :

```
empty : unit -> 'a liste
singleton : 'a -> 'a liste
```

Question 3.2. Écrire les fonctions permettant d'ajouter un élément à une liste :

```
add_deb : 'a liste -> 'a -> unit
add_fin : 'a liste -> 'a -> unit
```

Question 3.3. Écrire les fonctions permettant de retirer un élément d'une liste (et qui renvoient cet élément) :

```
take_deb : 'a liste -> 'a
take_fin : 'a liste -> 'a
```

Question 3.4. Écrire les fonctions de conversion avec le type `list` original :

```
to_list : 'a liste -> 'a list
of_list : 'a list -> 'a liste
```

Question 3.5. Écrire enfin `rev_liste : 'a liste -> unit` qui renverse une liste en place (sans utiliser la conversion vers un autre type).

Question 3.6. Vérifier que tout marche bien grâce au code fourni.