Programmation Fonctionnelle Avancée

Cours 9

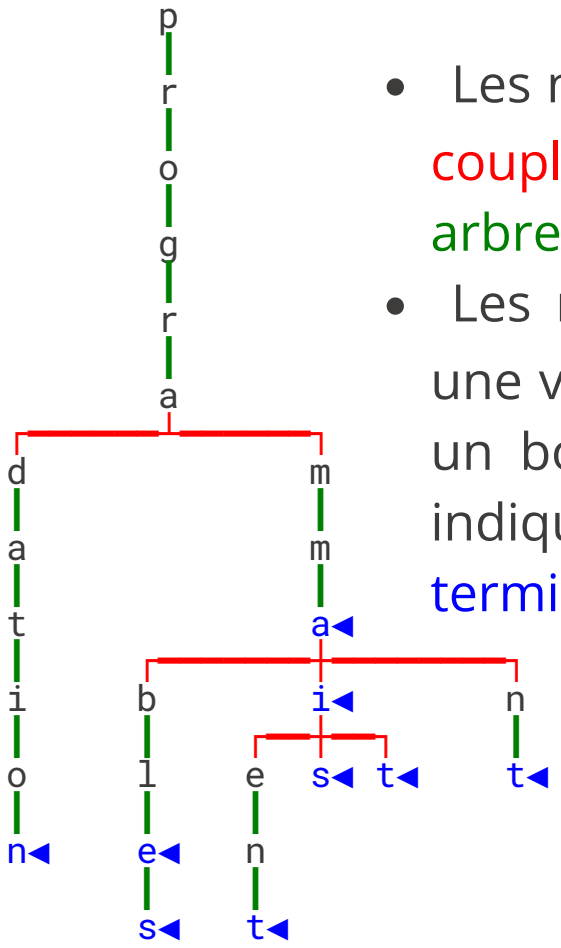
kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Résumé de l'épisode précédent

On a vu le *trie*, un structure de donnée permettant d'implémenter des dictionnaires :

	Accès	Données triées	Persistante
`Trie`	O(k)	✔(?)	✔



- Les nœuds sont des listes de couples (caractère, sous-arbre)
- Les nœuds contiennent soit une valeur (dictionnaires) soit un booléen (ensemble) pour indiquer qu'ils sont terminaux.

Plan

Corrigé commenté du TP 8, EXO 1

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

On continuera la feuille dans le TP 9 (quelques questions ajoutées)

Plan

Trier les mots d'un texte

On considère un texte $T = \{ s_1, ..., s_n \}$ constitué de n mots. Le nombre de caractères dans le texte est

$$|T| = \sum_{i=1..n} |s_i|$$

On considère l'algorithme suivant :

- Créer un *trie* t vide
- Pour i entre 1 et n, insérer s_i dans t
- Parcourir t avec un parcours préfixe pour afficher les clés dans l'ordre

On a donc trié l'ensemble des mots du texte. Quelle est la complexité ?

$$O(|T|)$$ 🤔

On a pu trier un texte en moins que $O(|T|\log(|T|))$, où est l'arnaque ?

Complexité des tris

Il faut être très précis dans les énoncés.

Un tri utilisant une *fonction de comparaison binaire* (qui compare 2 éléments entre eux) doit effectuer $O(N \log(N))$ comparaisons pour une collection de taille N.

Intuition: étant donné une collection de N éléments, il y a $N!$ permutations possibles. En faisant les comparaisons 2 à 2, on peut « trouver » la bonne permutation au mieux en $O(\log(N!)) = O(N \log(N))$

Complexité des tris

Mais ici on n'utilise pas une comparaison 2 à 2. On a une structure de données auxiliaire qui sait donner rapidement un ensemble de clés plus grandes.

On peut donc trier en temps linéaire, mais pour un ordre bien particulier, l'ordre lexicographique. On ne peut pas utiliser un *trie* pour un ordre arbitraire.

Exemple, si on a une liste de vecteurs (x,y) que l'on veut trier par taille croissante ($\sqrt{(x^2+y^2)}$), on ne peut pas utiliser un *trie*.

Ensembles

Dans toute la suite, on suppose que les *tries* qu'on manipule représentent des ensembles et non des dictionnaires. On stocke juste dans les nœuds un booléen qui dit si le nœud est terminal ou pas:

```
type trie =  Node of bool * (char * trie list)
(* pas de 'a, car on ne stocke pas de valeur *)
```

Unicité de la représentation

Pour un ensemble de clés données, le *trie* représentant cet ensemble est unique.

- `{ }` ⇒ `Node(false, [])`
- `{"A"}` ⇒ `Node(false, ['A', Node(true, [])])`
- `{"A", "AA", "BC", "C" }` ⇒

```
Node(false,
[ 'A', Node(true,  ['A', Node(true, [])]);
  'B', Node(false, ['C', Node(true, [])]);
  'C', Node(true,  [])
])
```

Ce n'est pas le cas d'autres structures. Exemple avec les tables de hachage:

- Ajouter "A" dans la table vide
- Ajouter "A" dans la table vide, puis ajouter 10000 autres clés, puis les retirer

Le tableau interne peut avoir été agrandi, mais c'est le même ensemble.

Unicité de la représentation : utilité

Si deux objets (immuables) égaux sont *structurellement égaux*, alors on peut partager leur représentation en mémoire. Cela permet d'accélérer des opérations et d'économiser de la mémoire. Avant de voir cette technique, on doit s'intéresser à la représentation en mémoire des valeurs OCaml.

Plan

Modèle mémoire ?

Le modèle mémoire d'un langage de programmation est la façon dont ce dernier représente ses valeurs en mémoire, i.e. comme des suites d'octets.

De ce point de vue, l'impact le plus grand pour langage est la présence d'un GC ou *garbage collector*.

GC

Le *garbage collector* (ou GC) est un "sous-programme" dont le rôle est de désallouer automatiquement la mémoire allouée. Exemple :

- sans GC : C, C++, Rust : on alloue et désalloue explicitement (mallloc/free)
- avec GC : OCaml, Java, Python, JavaScript, ...

Pourquoi le GC a-t'il un impact ?

Comment le GC détecte-t'il qu'un objet peut être désalloué ?

Il doit stocker avec chaque objet de l'information supplémentaire. La nature de cette information à un impact sur le modèle mémoire.

Représentation des valeurs en OCaml

OCaml distingue 2 sortes de valeurs à l'exécution

- Les immédiats : `int`, `char`, `bool`, `()` (unit), `[]` et les constructeurs constants comme `type color = Red | Blue | Green`
- Les blocs : tout le reste. Ils sont représentés par un *pointeur* vers une zone mémoire organisée d'une certaine façon (que l'on va détailler).

Ainsi, toutes les valeurs OCaml sont représentées de façon uniforme, un mot mémoire de 64 bits qui est soit un entier soit un pointeur.

Le GC

Comment désallouer ?

- Les immédiats n'ont pas besoin d'être désalloués
- Les pointeurs pointent vers un bloc mémoire contenant un nombre entier de mots de 64 bits. Le premier mot est un en-tête qui code :
  - le nombre de blocs suivants sur 54 bits
  - 2 bits d'espaces réservés pour le GC
  - Un tag sur 8 bits qui représente le constructeur pour les types sommes

Exemple, `let x = (41, 42)`. La variable `x` contient un pointeur vers un bloc de taille 3 * 64 bits = 3* 8 octets = 24 octets.

- Le premier bloc contient `0b|00000000|00|0...10|` pour dire qu'il y a 2 blocs ensuite
- Le second bloc contient 41
- Le troisième bloc contient 42

Le GC

Le GC parcours l'ensemble des valeurs pendant l'exécution du programme (en « parallèle »).

Il commence son parcours depuis les variables globales et la pile d'appel courante (variable locale visible)

Si le GC rencontre un pointeur, il va voir le bloc correspondant, et met un 1 dans les 2 bits réservés. Puis il parcours les blocs suivants et suis les pointeurs ≡ parcours du graphe des pointeurs.

Après le parcours, on re-parcourt une nouvelle fois. Toutes les valeurs marquées sont gardées (et la marque remise à 0), les autres sont désallouées.

Cette technique est appelée Mark and Sweep.

Pointeur vs. Entier ?

Problème, quand le GC voit la valeur « 42 » dans un bloc, est-ce qu'il s'agit de l'entier 42 (on ne fait rien) ou est-ce qu'il s'agit du pointeur vers l'adresse 42 (et il faut suivre ce pointeur).

Solution: Les entiers sont marqués en décalant à gauche de 1 bit mettant leur bit de poids faible à 1.

Ça marche car les adresses mémoires vers des blocs de 8 octets sont toujours des multiples de 8 donc pairs.

Donc: si on voit un nombre pair, c'est un pointeur on le suit
Si on voit un nombre impair, on le divise par 2 et on obtient la valeur avec laquelle on peut calculer.

Démos

- On utilise Compiler Explorer pour regarder du code assembleur OCaml
- On utilise une version instrumentée du toplevel OCaml pour voir les pointeurs.

Conclusion

On voit donc la différence qu'il y a entre « être structurellement égal » et être identique (pointer vers le même bloc)

Les *trie* on la propriété que si deux sous-arbres sont structurellement égaux, on peut les remplacer par le même pointeur.

On verra la prochaine fois comment utiliser cela pour rendre du code plus efficace