Programmation avancée

TD1 : Valeurs et persistance

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse Paul Robert

1 Préliminaires : représentations de valeurs et partage

On propose de commencer en réfléchissant à la représentation des valeurs OCaml et aux questions de partage sur des petits exemples. Pour chaque exemple, on pourra vérifier sa réponse en s'aidant de la bibliothèque `memgraph_kitty` (mais réfléchissez d'abord, il faut être capable de répondre à toutes ces questions sans l'aide de la bibliothèque !).

On utilise `memgraph_kitty` de la façon suivante :

- L'installer : `opam install memgraph_kitty`
- Lancer le terminal `kitty` (On peut simplement utiliser la commande `kitty` depuis un terminal classique) ;
- Lancer un top-level OCaml : `ocaml`
- Utilisation de la librairie :

```
# #use "topfind";;
# #require "memgraph_kitty";; (* importe memgraph_kitty *)
# open Memgraph_kitty;;
# let l = [1; 2; 3];;
# show ["l", l];; (* dessine la liste `l` *)
```

En cas de souci, on pourra alternativement utiliser la version en ligne à l'adresse : `https://armael.github.io/jsoo-memgraph-toplevel`

Dans la version en ligne il suffit de définir une valeur pour afficher sa représentation:

```
# let l = [1; 2; 3];;  (* affiche la représentation de `l` *)
```

Exercice 1 (Tableaux de listes).

Rappel *: la fonction* `Array.init` *de type* `int -> (int -> 'a) -> 'a array` *crée un tableau de la taille indiquée en premier argument, dont le contenu est déterminé par la fonction passée en deuxième argument. Autrement dit,* `Array.init n f` *renvoie le tableau* `[| f 0; f 1; ...; f (n-1) |]`*.*

1. *Considérons les tableaux renvoyés par respectivement l'expression OCaml :*

```
(* cas 1 *)
let l = [1; 2; 3] in
Array.init 4 (fun _ -> l)
```

et

```
(* cas 2 *)
let l = [2; 3] in
Array.init 4 (fun _ -> 1 :: l)
```

1. *Combien de* ***blocs mémoire*** *sont alloués sur le tas dans chaque cas (on pourra s'aider d'un dessin, sur le même principe que vu en cours) ?*
2. *Peut-on distinguer, dans un programme OCaml, entre le tableau produit dans le premier cas et celui produit dans le deuxième cas ? Autrement dit, peut-on écrire un programme OCaml qui, étant donné un tableau produit par l'un des deux cas, identifie le cas dont il s'agit ?*

Exercice 2 (Tableaux de références). *Même question, en considérant maintenant les deux programmes suivants :*

```
(* cas 1 *)
let r = ref 0 in
Array.init 4 (fun _ -> r)
```

et

```
(* cas 2 *)
Array.init 4 (fun _ -> ref 0)
```

1. *Combien de blocs mémoire sont alloués sur le tas dans chaque cas ?*
2. *Peut-on distinguer les tableaux produits dans les cas 1 et 2 ? Si oui, écrire un programme qui le fait : une fonction du type* `int ref array -> bool` *renvoyant* `true` *si le tableau d'entrée a été produit par le cas 1, et* `false` *sinon.*

Considérons maintenant la définition d'arbres binaires suivante, et la fonction `complet42 n` qui produit un arbre complet de hauteur `n` contenant 42 à chaque nœud.

```
type t =
  | Node of t * int * t
  | Leaf

let rec complet42 n =
```

```
  if n = 0 then Leaf
  else Node (complet42 (n-1), 42, complet42 (n-1))
```

Exercice 3 (Arbres).

1. *Combien de blocs mémoire sont alloués par un appel à* `complet42 3` *?*
2. *Dans le cas général, combien de blocs mémoire sont alloués par un appel à* `complet42 n` *?*
3. *Implémenter une fonction* `complet42_compact n` *qui renvoie un arbre équivalent à* `complet42 n`*, mais en allouant un nombre linéaire de blocs mémoire (astuce : utiliser le partage).*

Considérons la fonction `map` sur les arbres définie ci-dessous, qui transforme un arbre en appliquant son argument `f` sur l'entier contenu dans chaque nœud.

```
let rec map_tree (f : int -> int) : t -> t = function
  | Node (t1, x, t2) -> Node (map_tree f t1, f x, map_tree f t2)
  | Leaf -> Leaf
```

Exercice 4 (Arbres, suite).

1. *Combien de blocs mémoire y a t-il (en fonction de* `n`*) dans le résultat de* `map_tree (fun x -> x + 1) (complet42 n)` *?*
2. *Combien de blocs mémoire y a t-il (en fonction de* `n`*) dans le résultat de* `map_tree (fun x -> x + 1) (complet42_compact n)` *?*

Pour finir, considérons la fonction prenant un arbre en argument, et renvoyant un arbre dont la valeur du nœud le plus à droite a été incrémentée :

```
let rec incr_tree : t -> t = function
  | Node (t1, x, Leaf) -> Node (t1, x + 1, Leaf)
  | Node (t1, x, t2) -> Node (t1, x, incr_tree t2)
  | Leaf -> Leaf
```

Et on considère le scénario suivant :

```
let t1 = complet42 n in
let t2 = incr_tree t1 in
...
```

Exercice 5 (Arbres, fin).

1. *Combien de blocs mémoire sont alloués par* `incr_tree t1` *(en fonction de* `n`*) ?*
2. *Dessiner la représentation en mémoire de* `t1` *et* `t2` *lorsque* `n = 3`*.*

2 Tableaux Persistants

Téléchargez le code de démarrage : https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/01-tableaux/td01.zip

Instructions :

- Le code à modifier est dans `lib/persistance.ml`
- Pour lancer les tests, exécuter `dune exec bin/test.exe`
- Pour charger votre code dans un toplevel ocaml (par exemple pour le tester interactivement et utiliser `memgraph`) :
  - lancer un toplevel depuis le dossier du projet : `ocaml`
  - importer le module `memgraph_kitty` si besoin (voir plus haut)
  - lancer `#use_output "dune ocaml top-module lib/persistance.ml";;`
  - puis `open Persistance;;`

Dans la suite de ce TP nous allons explorer l'implémentation et l'utilisation de structures de données *persistantes*. Comme vu en cours, une structure persistante est une structure qui *apparaît comme immuable* : une opération sur la structure renvoie une nouvelle version de la structure, tandis que l'ancienne version de la structure reste accessible. Ceci n'implique pas forcément une perte en efficacité ou consommation de mémoire, car il est souvent possible de *partager* une partie de la mémoire commune entre différentes versions de la structure.

On va ici s'intéresser à l'implémentation de **tableaux persistants** s'appuyant sur des *effets de bord « sous le capot »*. En effet, tant que l'on préserve l'illusion que la structure est persistante (on dit que la structure est « observationnellement immuable »), tous les coups sont permis ! L'utilisation d'effets de bords contrôlés permet alors de gagner en performances sans compromettre la facilité d'utilisation de la structure.

Un tableau (*array* en anglais) est une structure de données fondamentale. Traditionnellement, un tableau est une structure que l'on modifie en place, et donc non-persistante (on dit aussi *éphémère*). C'est notamment le cas des tableaux fournis en OCaml, et que l'on manipule avec les fonctions du module `Array` de la bibliothèque standard :

```
module Array : sig
  type 'a t = 'a array
  val get : 'a t -> int -> 'a
  val set : 'a t -> int -> 'a -> unit
  ...
end
```

On voit au type de la fonction `set` que celle-ci modifie le tableau en place : son type de retour est `unit`, indiquant que le tableau a été modifié en place.

Dans ce TD, on va s'intéresser à l'implémentation de tableaux *persistants* satisfaisant la signature de module suivante :

```
module type PARRAY = sig
  type 'a t (* le type du tableau *)

  (* [make n x] renvoie un tableau de taille [n]
     initialisé avec la valeur [x] *)
  val make : int -> 'a -> 'a t
```

```
  (* [get a n] renvoie la valeur à indice [n] du tableau [a] *)
  val get : 'a t -> int -> 'a

  (* [set a n x] renvoie un nouveau tableau mettant à jour [a]
     avec la valeur [x] à l'indice [n] *)
  val set : 'a t -> int -> 'a -> 'a t
end
```

Rappel: On peut *implémenter* une signature en utilisant la syntaxe suivante. Le compilateur vérifiera alors que l'implémentation fournie satisfait bien la signature indiquée (`PARRAY`).

```
module MyModule : PARRAY = struct
   (* déclarations *)
end
```

Exercice 6 (Tableau naïf). *Un moyen simple (mais très inefficace) d'implémenter des tableaux persistants est d'effectuer une copie du tableau entier à chaque modification. Complétez la définition de `NaiveArray` sur ce principe.*

```
(* Dans le fichier `lib/persistance.ml` *)
module NaiveArray : PARRAY = struct
  type 'a t = 'a array

  let make n v = Array.make n v

  let get arr i = (* TODO *)
  let set arr i v = (* TODO *)
end
```

N'oubliez pas que notre implémentation doit être *persistante* : les anciennes versions des tableaux doivent être toujours utilisables.

En particulier, assurez vous que votre code passe bien le test défini dans `bin/test.ml`, grâce à la commande

```
dune exec bin/test.exe
```

Cette implémentation naïve est certes persistante, mais très inefficace, car elle ne cherche pas à partager de mémoire entre différentes versions d'un tableau. Pour chaque opération sur un tableau, on se retrouve à copier le contenu entier du tableau (potentiellement des millions d'éléments !).

Pour faire mieux, on considère l'idée suivante : plutôt que de dupliquer entièrement le tableau OCaml sous-jacent à chaque opération, on va partager ce tableau entre les différentes versions persistantes du tableau. De plus, chaque tableau persistant maintient additionnellement une liste de changements à prendre en compte par rapport au tableau OCaml sous-jacent.

Par exemple, si l'on commence avec un tableau `a1` contenant les valeurs $1, 2, 3, 4, 5$, puis que l'on exécute :

```
let a2 = set a1 0 8 (* a1.(0) <- 8 *)
let a3 = set a2 2 0 (* a2.(2) <- 0 *)
```

le résultat est la structure en mémoire affichée en Figure 1. Afin d'optimiser les accès, la dernière version du tableau `a3` pointe directement vers le tableau modifié, pendant que les versions précédents `a2` et `a1` indiquent les modifications qui doivent être effectués pour restaurer les états précédents.

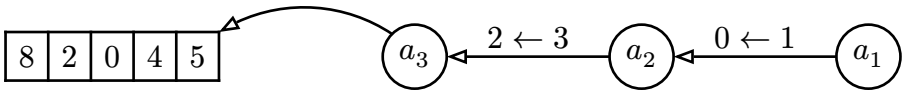


Figure 1. – Représentation mémoire des modifications de a_0

Exercice 7 (Tableaux persistants (1)). *Implémentez des tableaux persistants en utilisant la stratégie décrite précédemment.*

On représentera un tableau persistant par une référence (c'est important!) sur un type somme `'a data` *qui indique comment il est représenté : soit par une valeur immédiate* `Arr a` *pour un tableau OCaml* `a`*; soit par une indirection* `Diff (i, v, t)`*, représentant le tableau persistant identique au tableau* `t` *excepté pour la case d'indice* `i` *où il vaut* `v`*.*

L'utilisation d'une référence est cruciale. Pourquoi ? Pourquoi n'a t-on pas défini `type 'a t = 'a data` *?*

L'implémentation de la fonction `set` *est le cas le plus subtil. On pourra s'aider d'un dessin de la représentation de la mémoire dans les différents cas !*

```
module FuncArray1 : PARRAY = struct
  type 'a t = ('a data) ref
  and 'a data =
    | Arr of 'a array
    | Diff of int * 'a * 'a t

  let make n v = ref (Arr (Array.make n v))

  let get a i = (* TODO *)

  let set a i v = (* TODO *)
end
```

Vérifiez que le test passe pour cette nouvelle implémentation de tableaux persistants (via `dune exec bin/test.exe`). Vérifiez également qu'elle est en effet plus efficace que la version naïve, grâce à la commande

```
dune exec bin/bench.exe
```

La performance de notre implémentation de tableaux persistants peut encore être améliorée. Un problème de l'implémentation actuelle est qu'accéder aux anciennes versions d'un tableau est de plus en plus coûteux. Lors de la lecture d'une ancienne version, il faut en effet traverser une liste de modifications d'autant plus longue que la version est ancienne, et ce à chaque accès lecture.

C'est particulièrement fâcheux pour les algorithmes effectuant du rebroussement (ou *backtracking*), pour lesquels on veut revenir à une ancienne version de la structure et poursuivre les calculs à partir de cette ancienne version.

Pour améliorer ces performances on peut effectuer une opération de « re-racinage » (*re-rooting*) à chaque fois que l'on accède à une ancienne version du tableau. Cette opération ré-organisera notre tableau pour placer l'ancienne version à la racine du tableau. Si on reracine le tableau de la Figure 1 à la version a_2, on obtient alors la mémoire comme représentée en Figure 2.

Après le re-racinage, a_2 pointe directement sur le tableau OCaml sous-jacent, tandis que a_3 pointe vers a_2 avec une nouvelle indirection.

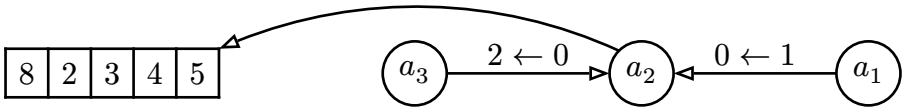


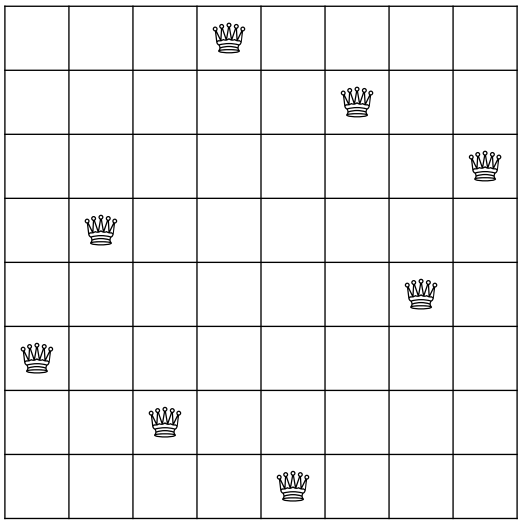
Figure 2. – Mémoire après re-racinage à a_2

Exercice 8 (Reracinage). *Copiez* `FuncArray1` *dans le module* `FuncArray2`*, et implémentez-y une fonction* `reroot : 'a t -> 'a t`*. Un appel à* `reroot t` *doit re-raciner l'arbre à partir de* `t`*, afin que des accès ultérieurs à* `t` *correspondent à des accès immédiats au tableau OCaml sous-jacent.*

Ensuite, utilisez la fonction `reroot` *dans l'implémentation de* `get` *et* `set` *afin d'effectuer un re-racinage lors de l'accès à une ancienne version.*

3 Application : le problème des n-reines

Le problème des n-reines consiste à trouver un placement de n reines sur un échiquier de taille $n \times n$ telle que aucune reine n'attaque une autre. Par exemple, voici une solution valide pour 8 reines.



Il existe une procédure simple et efficace pour trouver une solution en utilisant le backtracking :

- Choisir une rangée de la colonne de gauche
- Pour chaque colonne successive:
  - Choisir une rangée dans la colonne
  - Tester si cette position est valide (aucun conflits avec choix précédents)
  - Si oui, continuer a la prochaine colonne
  - Si non, backtracker et tenter un autre choix

On peut donc utiliser notre implémentation de tableaux persistants pour représenter la matrice correspondant à l'échiquier, et ainsi pouvoir revenir à une ancienne version très facilement, en tirant parti de la nature persistante de l'état de l'échiquier.

Exercice 9 (n-reines). *En partant du code fourni, implémentez la fonction* `nqueens_inner`.

Lancez `dune exec bin/nqueens.exe` pour tester votre code.

4 Bonus : tableaux semi-persistants

Dans certains algorithmes, on n'a pas besoin de la persistance complète. Par exemple, quand on fait du *backtracking*, on remonte une branche de décisions pour en commencer une nouvelle, fraîche. Dans ce cas on a besoin d'accéder à un *ancêtre* de notre tableau, mais jamais à un cousin.

Une structure de données qui permet de modifier seulement les ancêtres de la dernière version est appelé *semi-persistante*[1]. Dans le cas des tableaux il suffit de changer la fonction `reroot`, pour oublier les changements pendant un backtrack.

La Figure 3 illustre ce qui se passe si on utilise un re-racinage semi-persistant de a_3 à a_2 (en partant de la configuration de Figure 1).

Figure 3. – Re-racinage semi-persistant a a_2

Pendant le re-racinage on oublie complètement le changement de a_3, car cette version n'est pas un *ancêtre* de a_2.

Exercice 10 (Semi-persistance). *Réimplémentez l'opération de re-racinage de manière semi-persistante. Il suffit de modifier la version précédente de* `reroot`.

Rustlings pour la semaine prochaine

Aller jusqu'à `04_primitive_types` inclus.

1. Conchon et Filliâtre, « Semi-Persistent Data Structures ».