

Programmation Fonctionnelle Avancée

6 : Partage maximal de structures

Pierre Letouzey

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale
letouzey@irif.fr

13 mars 2023

Allocation de mémoire sur le tas

- ▶ Allocation de la mémoire, lors de la création de valeurs, dans la zone d'allocation dynamique (tas, en anglais : *heap*)
- ▶ Ne pas confondre avec la pile (anglais : *stack*) qui est utilisée lors des appels récursifs
- ▶ Le *Garbage Collector* récupère automatiquement la mémoire non référencée
- ▶ Les valeurs de type basique (bool, int) et les *constantes* des types algébriques sont représentées une seule fois dans la mémoire, les autres (string, constructeurs non-constants de types algébriques) sont nouvellement créées.

Deux opérateurs de test d'égalité

- ▶ = teste l'égalité *structurelle*
- ▶ == teste l'égalité *physique*
- ▶ Si $x == y$ alors $x = y$, mais le contraire n'est pas toujours vrai.
- ▶ = est normalement ce que le programmeur veut, mais cet opérateur a un coût linéaire en la taille des deux structures à comparer.
- ▶ == a un coût constant, mais il faut faire attention si ce test a du sens.

Exemples (equality.ml)

```
let _ = 1 = 1
```

```
let _ = 1 == 1
```

```
let _ = "ab" = "ab"
```

```
let _ = "ab" == "ab"
```

```
let s = "ab"
```

```
let t = s in s==t
```

Constructeurs

- ▶ Application d'un constructeur (d'un type algébrique) : coût constant, indépendant de la taille des arguments.
- ▶ Exemple des arbres binaires : Si t_1 et t_2 existent déjà sur le tas, `Noeud (t1,t2)` alloue de la mémoire pour une cellule *Noeud*, avec deux “pointeurs” vers les valeurs t_1 et t_2 .

Exemples (bintree1.ml)

```

type 'a tree =
  | Leaf of 'a
  | Node of 'a tree * 'a tree

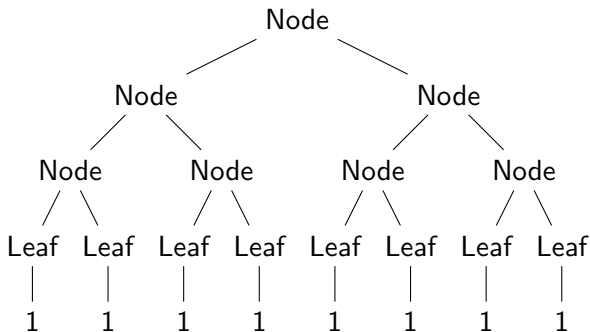
let rec bintree1 h c =
  if h=0 then Leaf c
  else
    Node (bintree1 (h-1) c, bintree1 (h-1) c)

let t1 = bintree1 5 42

```

Zéro partage : fonction bintree1

- ▶ L'appel `(bintree1 h c)` fait 2^{h+1} allocations en mémoire.
- ▶ `(bintree1 3 1)` :



Exemples (bintree2.ml)

```
let rec bintree2 h c =  
  if h=0 then Leaf c  
  else let t = bintree2 (h-1) c in Node (t,t)
```

```
let t2 = bintree2 5 42
```

```
let _ = t1=t2
```

```
let _ = t1==t2
```

```
let _ = match t1 with Node(x,y) -> x==y | _ -> false
```

```
let _ = match t2 with Node(x,y) -> x==y | _ -> false
```


Partage maximal : fonction bintree2

- L'appel `(bintree2 h c)` fait $h + 1$ allocations en mémoire.
- `(bintree2 3 1)` :

Node
()

Node
()

Node
()

Leaf
|
1

Égalité de valeurs d'un type algébrique

- ▶ Égalité physique (opérateur `==`) : teste simplement l'égalité des adresses mémoire des constructeurs de racine des deux valeurs. Coût constant.
- ▶ Égalité structurelle (opérateur `=`) : descend récursivement dans les deux structures, coût linéaire en la taille des deux valeurs.
- ▶ L'égalité structurelle générique de OCaml se comporte sur notre exemple comme la fonction `eqtree` suivante.

Exemples (bintree3.ml)

```
let rec eqtree t1 t2 = match t1,t2 with
  | Leaf v1, Leaf v2 -> v1=v2
  | Node (l1,r1), Node (l2,r2) ->
    eqtree l1 l2 && eqtree r1 r2
  | _ -> false

let _ = eqtree t1 t2
```

Objectif

- ▶ Dans l'exemple précédent on a pu construire t_2 tel que les sous-arbres sont partagés autant que possible.
- ▶ Mais la situation n'est pas toujours aussi simple : des structures peuvent être le résultat de calculs complexes.
- ▶ Notre but est de représenter quand même chaque structure une seule fois en mémoire (partage maximal). Cela aura deux avantages :
 1. minimiser la consommation de mémoire
 2. un test d'égalité plus efficace : on pourra remplacer `=` par `==`
- ▶ Nous avons besoin de deux ingrédients :
 1. les tables de hachage
 2. les tableaux faibles

Objectif

- ▶ Représenter des fonctions partielles finies $f : A \rightsquigarrow B$
- ▶ Cas d'usage : Le domaine *potentiel* A est très grand ou même infini ; par contre f est définie seulement pour un “petit” nombre de valeurs : son domaine *concret* est fini.
- ▶ (1) On souhaite un coût mémoire au plus linéaire dans la taille du domaine plus la taille du co-domaine de f .
- ▶ On pourrait utiliser des listes d'association, mais :
- ▶ (2) On souhaite une complexité constante pour accéder à la valeur de la fonction appliquée à un argument.
- ▶ (3) Fonctions modifiables : on veut pouvoir ajouter ou supprimer des paires (argument, résultat)

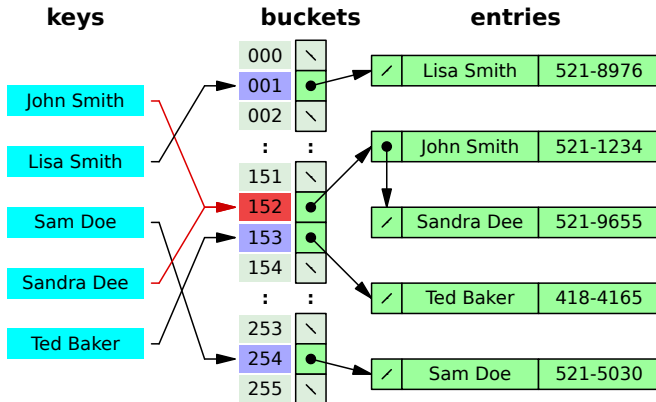
Réalisation

- ▶ Les tableaux répondent aux objectifs 2 (complexité constante) et 3 (fonctions modifiables), sauf qu'il faudrait utiliser comme indices des valeurs d'un type A .
- ▶ Solution : On utilise une *fonction de hachage* $h: A \rightarrow \text{int}$ pour mapper les arguments de la fonction f vers des entiers positifs (indices du tableau).
- ▶ Cela nous permet aussi de répondre à l'objectif 1 : on s'arrange pour que l'image de h soit un petit ensemble, par exemple l'intervalle $[0, \dots, d - 1]$.
- ▶ Le tableau peut donc avoir la taille $O(d)$.

Problème : conflits

- ▶ On ne peut pas exclure des conflits de la fonction de hachage : $x \neq y$ et $h(x) = h(y)$ (non-injectivité de h).
- ▶ En pratique, on essaye de les éviter par un bon choix de la fonction de hachage.
- ▶ Pour gérer les conflits résiduels, les entrées du tableau ne sont pas des valeurs de B , mais des fonctions partielles $A \rightsquigarrow B$. Ces fonctions devront avoir un domaine vraiment petit (quelques éléments seulement), on peut donc les représenter par une liste d'association par exemple.
- ▶ Conséquence : on a besoin d'un test d'égalité pour A .

Table de hachage avec listes chaînées



Structures avec partage maximal

- ▶ Problème : on veut travailler avec des structures dynamiquement allouées en mémoire, mais on veut à chaque moment un partage maximal de sous-structures.
- ▶ Aucune (sous-)structure ne doit être représentée plusieurs fois en mémoire.
- ▶ Avantages :
 - ▶ Moins de consommation mémoire
 - ▶ Le test d'égalité peut être remplacé par l'égalité physique
- ▶ Difficulté : Les structures peuvent être créées dynamiquement pendant l'exécution du programme, on doit reconnaître quand une structure existe déjà en mémoire.

La technique du hashconsing

- ▶ Le nom vient du langage LISP qui fournit un seul constructeur `cons` pour construire des structures (listes, arbres, ...).
- ▶ Première idée : utiliser une table de hachage pour stocker des paires (t, t) de structures.
- ▶ Quand on construit une nouvelle structure t on regarde si elle existe déjà dans la table : si oui on prend la structure existante à la place de t , si non on l'ajoute à la table.
- ▶ Pour chaque paire (t, t) dans la table de hachage :
 - ▶ le t de gauche sert de clé de recherche (a priori comparée via $=$)
 - ▶ le t de droite sert de *représentant canonique* : toutes les structures égales (via $=$) à t deviendront exactement ce t .

Termes arithmétiques hashconsés : version 1 (hc1.ml) |

Solution ad-hoc, en utilisant une table de hachage pour avoir accès à toutes les valeurs qu'on a déjà construites.

```
type term =
  | Const of int
  | Plus of term * term
  | Opp of term

let table = Hashtbl.create 251

let hashcons x =
  try Hashtbl.find table x
  with Not_found -> Hashtbl.add table x x; x

let const i = hashcons (Const i)
```

Termes arithmétiques hashconsés : version 1 (hc1.ml) II

```

let plus t1 t2 = hashcons (Plus (t1 , t2))
let opp t = hashcons (Opp t)

let t1 = plus (const 2) (const 42)
let t2 = plus (const (1 + 1)) (const (73-31))
let _ = t1=t2
let _ = t1==t2
    
```

Test d'égalité efficace

- ▶ La fonction `find` du module `Hashtbl` doit toujours tester l'égalité entre la clef donnée en argument, et la clef trouvée dans la table (à cause des conflits possibles).
- ▶ Pour cela, OCaml utilise par défaut `=`, la fonction polymorphe de test d'égalité structurelle, qui descend en profondeur dans la structure.
- ▶ Or, pour toutes les valeurs x et y *obtenues par la fonction `hashcons`* : $x = y$ ssi $x == y$. On peut en profiter pour tester l'égalité plus efficacement.
- ▶ On utilise le foncteur `Hashtbl.Make`, qui prend comme argument une structure avec un test d'égalité et une fonction de hachage.
- ▶ On s'assure par une interface que toutes les valeurs sont construites par le `hashconsing`.

Term arithmétiques hashconsés : version 2 (hc2.ml) I

Amélioration : on s'assure que toutes les valeurs sont créées à travers du hashconsing, ce qui nous permet d'utiliser dans la table de hachage un test d'égalité plus intelligent.

```
module type TERMHC = sig
  type term
  val const: int -> term
  val plus: term -> term -> term
  val opp: term -> term
end
```

```
module TermHC:TERMHC = struct
  module Term = struct
    type t =
      | Const of int
      | Plus of t*t
```

Term arithmétiques hashconsés : version 2 (hc2.ml) II

```

    | Opp of t
    (* shallow comparison, uses == on substructures *)
    let equal x y = match x,y with
    | Const i, Const j -> i == j
    | Plus (t1,t2), Plus (u1,u2) ->
        t1 == u1 && t2 == u2
    | Opp t, Opp u -> t == u
    | _ -> false
    let hash = Hashtbl.hash (* generic function *)
end

module H = Hashtbl.Make(Term)

type term = Term.t

let table = H.create 251
    
```

Term arithmétiques hashconsés : version 2 (hc2.ml) III

```

let hashcons x =
  try H.find table x
  with Not_found -> H.add table x x; x
let const i = hashcons (Term.Const i)
let plus t1 t2 = hashcons (Term.Plus (t1,t2))
let opp t = hashcons (Term.Opp t)
end

open TermHC
let t1 = plus (const 2) (const 42)
let t2 = plus (const (1 + 1)) (const (73-31))
let _ = t1=t2
let _ = t1==t2
  
```


Termes arithmétiques hashconsés - version 3 (`hc3.ml`)

- ▶ Pour calculer le hash d'une structure, notre implémentation utilise `Hashtbl.hash` qui traverse toute la structure.
- ▶ Pour éviter cela on peut stocker dans les nœuds les hashes des sous-structures.
- ▶ Attention, cela demande une modification du type.
- ▶ Au passage, on pourrait également associer à chaque structure hashconsée un entier unique, utilisable pour construire par exemple
 - ▶ des arbres équilibrés (qui nécessitent un ordre)
 - ▶ des arbres de Patricia (qui nécessitent une séquence de bit)
 - ▶ ...

Termes arithmétiques hashconsés - version 3 I

```
type 'a with_hashkey = {  
  node: 'a;  
  hkey: int (* hash key *)  
}
```

(type des termes avec clef de hachage *)*

```
type term = term_node with_hashkey  
and term_node =  
| Const of int  
| Plus of term*term  
| Opp of term
```

(structure parametre *)*

```
module Term_node = struct  
  type t = term_node  
  let equal x y = match x,y with
```

Termes arithmétiques hashconsés - version 3 II

```

| Const i , Const j -> i == j
| Plus (t1 , t2) , Plus (u1 , u2) ->
  t1 == u1 && t2 == u2
| Opp t , Opp u -> t == u
| _ -> false
let hash = function
| Const i -> i
| Plus (t1 , t2) ->
  abs(19*(19*t1.hkey+t2.hkey)+2)
| Opp t -> abs (19 * t.hkey + 1)
end

(* signature de la structure parametre *)
module type HashedType = sig
  type t
  val equal: t -> t -> bool

```

Termes arithmétiques hashconsés - version 3 III

```

    val hash: t-> int
end

(* signature de la structure obtenue *)
module type S =
  sig
    type value
    val hashcons : value -> value with_hashkey
  end
(* tres simplifiee *)

module Make(T : HashedType) : (S with type value = T.t)
struct
  module H = Hashtbl.Make(T)
  type value = T.t
  let table = H.create 251

```

Termes arithmétiques hashconsés - version 3 IV

```

let hashcons d =
  try H.find table d
  with Not_found ->
    let d_hc = {node = d; hkey = T.hash d}
    in H.add table d d_hc; d_hc
end
module H = Make(Term_node)

let const i = H.hashcons (Const i)
let plus t1 t2 = H.hashcons (Plus (t1,t2))
let opp t = H.hashcons (Opp t)

let t1 = const 2
let t2 = const (1 + 1)
let _ = t1==t2
  
```

Un problème ...

- ▶ L'implémentation actuelle risque d'épuiser la mémoire !
- ▶ En effet, toutes les structures de type `term`, une fois créées, restent accessibles au moins à travers la table de hachage.
- ▶ Donc le garbage collector ne peut jamais récupérer la mémoire occupée par une telle structure !

Tableaux faibles

- ▶ Module Weak de la bibliothèque standard
- ▶ Tableaux contenant des “pointeurs” faibles vers des structures
- ▶ Un objet vers lequel pointent seulement des pointeurs faibles peut être récupéré par le garbage collector (GC).
- ▶ Les valeurs stockées dans un tableau faible sont d'un type `'a option`. Quand une valeur est supprimée par le GC, toute entrée correspondante dans des tableaux faibles est remplacée par `None`.
- ▶ Fonctionne comme décrit seulement pour des types de valeurs pour lesquelles il y a une “allocation” de cellules de mémoire. Ceci exclut en particulier les entiers.

Exemples (weak1.ml)

```

let size = 1000
let a = Weak.create size
let _ = for i=0 to size-1 do
    Weak.set a i (Some [i;i+1;i+2;i+3;i+4])
done
let _ = Weak.get a 0
let _ = Weak.get a 1
let p = Weak.get a 1
let _ = Gc.compact ()
let _ = Weak.get a 0
let _ = Weak.get a 1

```


Fonctions de finalisation

- ▶ Fonctionnalité intéressante du Garbage Collector
- ▶ Le module Gc permet d'enregistrer une fonction de *finalisation* avec un objet de mémoire, cette fonction sera appliquée à l'objet avant la suppression de l'objet par le GC.
- ▶ Exemple d'application : on a un tableau faible d'objets qui représentent des connexions TCP/IP. Ces objets vont être détruits par le GC quand personne ne les utilise plus. Dans ce cas, on veut que le socket correspondant soit fermé.

Exemples (weak2.ml)

```

let size = 1000
let a = Weak.create size
let goodbye v = print_string "killing:␣";
    print_int (List.hd v); print_newline ()
let _ = for i=0 to size-1 do
    let v = [i;i+1;i+2;i+3;i+4] in
    Weak.set a i (Some v);
    Gc.finalise goodbye v
done
let _ = Gc.compact ()

```

Attentions aux “pointeurs” non intentionnés

- ▶ Attention : la création de “pointeurs” vers des objets fait que les objets stockés dans un tableau faible ne sont plus détruits par le GC.
- ▶ Cela inclus des clôtures créées lors la définition des finalisateurs (voir l'exemple suivant) !

Exemples (weak3.ml)

```

let size = 1000
let a = Weak.create size
let _ =
  for i=0 to size-1 do
    let v = [i;i+1;i+2;i+3;i+4] in
    let f x = print_string "killing:␣";
              print_int (List.hd v);
              print_newline ()
    in
    Weak.set a i (Some v);
    Gc.finalise f v
  done
let _ = Gc.compact ()

```

Améliorations du hashconsing

- ▶ Pour que des structures non référencées soient récupérées par le Gc, on utilise des tables de hachage faibles, voir Weak.Make.
- ▶ La vraie implémentation par Jean-Christophe Filliâtre n'utilise pas les tables de hachage faibles de OCaml, mais se base directement sur les tableaux faibles.
- ▶ On peut à plusieurs endroits éviter une application répétée de la fonction de hachage à la même donnée. Par exemple : une fois pour tester si une valeur est déjà dans la table, et puis une deuxième fois pour l'ajouter à la table.

Pour en savoir plus



Jean-Christophe Filliâtre and Sylvain Conchon

Type-safe modular hash-consing.

Proceedings of the ACM Workshop on ML.

Septembre 2006.

Code disponible à l'adresse

<https://github.com/backtracking/ocaml-hashcons>