Programmation Avancée (INF441)
Contrôle classant

7 juin 2016

Les parties 1 et 2 sont indépendantes l'une de l'autre. Elles peuvent être traitées dans l'ordre de votre choix. Dans chaque partie, il est souvent possible de traiter une question sans avoir répondu à celles qui précèdent.

1 Une table de hachage et son itérateur

Dans cette partie, on demande d'écrire **du code Java** aussi **correct** que possible.

On souhaite écrire une classe `HashSet<E>` telle qu'un objet de type `HashSet<E>` représente un ensemble (modifiable) d'éléments de type `E`.

On décide de représenter cet ensemble en mémoire sous forme d'une table de hachage. On sait que le type `E` est muni de méthodes `hashCode` et `equals`, héritées de la classe `Object` (figure 1). On suppose que la méthode `equals` implémente une relation d'équivalence, donc est réflexive, symétrique, et transitive. On suppose que la méthode `hashCode` est compatible avec `equals`, c'est-à-dire que deux éléments de type `E` « égaux » au sens de `equals` ont le même code de hachage. On souligne le fait que la méthode `hashCode` renvoie un entier arbitraire, qui peut donc être positif, négatif, ou nul.

On a besoin d'abord de listes simplement chaînées, immuables, d'éléments de type `E`. On définit une classe abstraite `List<E>` (figure 2), où on déclare quatre méthodes abstraites :

1. `isEmpty()` indique si la liste **`this`** est vide ou non.
2. `head()` lance une exception si la liste **`this`** est vide ;
   sinon, elle renvoie le premier élément de cette liste.
3. `tail()` lance une exception si la liste **`this`** est vide ;
   sinon, elle renvoie cette liste privée de son premier élément.
4. `contains(e)` indique si la liste **`this`** contient un élément « égal » à `e`.

On définit ensuite deux classes `Nil<E>` et `Cons<E>`, qui sont sous-classes de `List<E>` (figure 2). Un objet de classe `Nil<E>` représente une liste vide ; un objet de classe `Cons<E>` représente une liste non vide. On souligne le fait que la liste vide **n'est pas** représentée par un pointeur **`null`**.

Question 1 Implémentez les quatre méthodes décrites ci-dessus dans chacune des deux classes `Nil<E>` et `Cons<E>`. ◇

Question 2 Donnez (sans preuve) la complexité asymptotique en temps, dans le pire cas, de chacune de ces quatre méthodes. Expliquez le sens des paramètres qui apparaissent dans vos bornes de complexité. ◇

On s'intéresse maintenant à la classe `HashSet<E>` (figure 3).

```
class Object {
  int hashCode ();
  boolean equals (Object that);
}
interface Iterable<T> {
  Iterator<T> iterator ();
}
interface Iterator<T> {
  boolean hasNext ();
  T next () throws NoSuchElementException;
}
```

FIGURE 1 – Définitions (simplifiées) issues de la bibliothèque de Java

```
abstract class List<E> {
  abstract boolean isEmpty ();
  abstract E head () throws NoSuchElementException;
  abstract List<E> tail () throws NoSuchElementException;
  abstract boolean contains (E e);
}

class Nil<E> extends List<E> {
  // À COMPLÉTER
}

class Cons<E> extends List<E> {
  final E       head;
  final List<E> tail;
  Cons (E head, List<E> tail) {
    this.head = head;
    this.tail = tail;
  }
  // À COMPLÉTER
}
```

FIGURE 2 – Les classes `List<E>`, `Nil<E>`, et `Cons<E>` (à compléter)

```
public class HashSet<E> implements Iterable<E> {
  private final int N = 1024;
  private final List<E>[] table;
  public HashSet () {
    table = (List<E>[]) new List [N];
    for (int i = 0; i < table.length; i++)
      table[i] = new Nil<E> ();
  }
  // À COMPLÉTER
}
```

FIGURE 3 – La classe `HashSet<E>` (à compléter)

Cette classe a un champ `table` de type `List<E>[]` : c'est un tableau de listes d'éléments. Le constructeur de la classe `HashSet<E>` alloue ce tableau et l'initialise de telle façon que chaque case du tableau contient initialement la liste vide. La taille N du tableau est fixée arbitrairement ; on prendra soin d'écrire du code qui fonctionne pour toute valeur (strictement positive) de N.

Le tableau `table` est organisé en une table de hachage. Un élément `e` doit donc être stocké dans le tableau à un indice `index(e)` qui ne dépend que du code de hachage de `e`.

Question 3 Implémentez dans la classe `HashSet<E>` une méthode `int index (E e)` qui, étant donné un élément `e` quelconque, renvoie l'indice où cet élément doit être stocké. Cet entier doit être un indice valide dans le tableau `table`. ◇

Question 4 Donnez (sans preuve) la complexité asymptotique en temps, dans le pire cas, de la méthode `index`. Expliquez le sens des paramètres qui apparaissent dans votre borne de complexité.◇

La méthode `index` n'est pas injective : elle peut attribuer le même indice à plusieurs éléments distincts. C'est pourquoi dans le tableau `table`, à l'indice `i`, on stocke la liste des éléments `e` tels que `index(e)` est égal à `i`.

Question 5 Implémentez dans la classe `HashSet<E>` une méthode `boolean contains (E e)` qui, étant donné un élément `e` quelconque, indique si cet élément appartient ou non à l'ensemble représenté par l'objet `this`. ◇

Question 6 Donnez (sans preuve) la complexité asymptotique en temps, dans le pire cas, de la méthode `contains`. Expliquez le sens des paramètres qui apparaissent dans votre borne de complexité. ◇

Question 7 Implémentez dans la classe `HashSet<E>` une méthode `boolean add (E e)` qui, étant donné un élément `e` quelconque, ajoute cet élément à l'ensemble représenté par l'objet `this`, et indique si l'élément appartenait déjà à l'ensemble. ◇

Question 8 Donnez (sans preuve) la complexité asymptotique en temps, dans le pire cas, de la méthode `add`. Expliquez le sens des paramètres qui apparaissent dans votre borne de complexité.◇

Question 9 Implémentez dans la classe `HashSet<E>` une méthode `Iterator <E> iterator ()` qui renvoie un nouvel itérateur sur l'ensemble représenté par l'objet `this`. On conseille d'écrire la méthode `hasNext` de l'itérateur avant d'écrire la méthode `next`. ◇

Question 10 Donnez (sans preuve) la complexité asymptotique en temps, dans le pire cas, de la méthode `iterator`. Donnez ensuite la complexité en temps d'une énumération complète, c'est-à-dire le coût total des appels à `hasNext` et à `next` nécessaires pour énumérer tous les éléments de l'ensemble. Donnez enfin la complexité en espace d'une énumération : combien d'espace (au-delà de l'espace occupé par l'ensemble lui-même) une énumération exige-t-elle ? Expliquez le sens des paramètres qui apparaissent dans vos bornes de complexité. ◇

Question 11 On souhaite écrire en dehors de la classe `HashSet<E>` une méthode `static <E> void add (HashSet<E> set1, HashSet<E> set2)` qui ajoute à l'ensemble `set1` tous les éléments de l'ensemble `set2`. Implémentez cette méthode. Pourrait-on lui donner un type plus général ? ◇

2 Arbres et diagrammes binaires de décision

Dans cette partie, on demande d'écrire **du code OCaml** aussi **correct** que possible.

2.1 Arbres binaires de décision

On s'intéresse à des formules logiques construites à partir :

1. des deux constantes *faux* et *vrai* ;
2. de n variables prises dans un ensemble fini $\{x_0, x_1, \ldots, x_{n-1}\}$;
3. des deux connecteurs logiques $\neg$ (négation, « NON ») et $\wedge$ (conjonction, « ET »).

Voici quelques exemples de formules :

1. $\mathit{vrai} \wedge x_2$.
2. $\neg(x_0 \wedge \neg x_0)$.
3. $\neg(x_0 \wedge x_1) \wedge \neg x_0$.

Notons $\mathbb{B}$ l'ensemble $\{\mathit{faux}, \mathit{vrai}\}$. Si une variable est considérée comme une inconnue à valeurs dans $\mathbb{B}$, alors une formule peut être interprétée comme une fonction de $\mathbb{B}^n$ dans $\mathbb{B}$. En termes plus simples, une formule est une manière de décrire une table de vérité.

Deux formules en apparence différentes peuvent avoir la même table de vérité. On dit alors qu'elles sont équivalentes. Par exemple, il n'est pas difficile de vérifier que :

1. La première formule ci-dessus est équivalente à x_2.
2. La deuxième formule ci-dessus est équivalente à *vrai*.
3. La troisième formule ci-dessus est équivalente à $\neg x_0$.

Pour déterminer si deux formules F et F' sont équivalentes, l'algorithme le plus simple serait de construire leurs tables de vérité, puis de comparer ces tables.

Question 12 Cela serait-il un bon algorithme ? Pourquoi ? Justifiez brièvement. ◇

Plutôt que de construire et de comparer des tables de vérité, on préfère une autre approche, à savoir la mise en forme canonique. La notion de forme canonique est définie de telle manière que deux formules sont équivalentes si et seulement si leurs formes canoniques sont identiques.

```
type variable = int
type bdt =
  | C of bool
  | D of variable * bdt * bdt
```

FIGURE 4 – Arbres binaires de décision

La forme canonique choisie est un **arbre binaire de décision**, ou « binary decision tree ». Elle est décrite par le type `bdt` (figure 4). Ce type algébrique n'a que deux constructeurs, nommés `C` et `D`, comme « constante » et « décision » :

1. L'arbre `C c` représente la formule *faux* ou la formule *vrai*, suivant la valeur du Booléen `c`.
2. Si les arbres `t` et `f` représentent respectivement les formules T et F, alors l'arbre `D(i, t, f)` représente la formule « *si* x_i *alors* T *sinon* F », que l'on pourrait écrire aussi $(x_i \wedge T) \vee (\neg x_i \wedge F)$.

Un arbre représente en fait une classe d'équivalence de formules. Si l'arbre `b` représente la formule B_1 et si les formules B_1 et B_2 sont équivalentes, alors `b` représente également la formule B_2.

On impose à ces arbres deux **invariants**. Pour tout nœud de la forme `D(i, t, f)`, on exige :

1. Les sous-arbres `t` et `f` sont différents l'un de l'autre. On écrit `t` $\neq$ `f`.
   (Les arbres qui respectent cette condition sont dits **réduits**.)
2. Toutes les variables qui apparaissent (portées par un nœud `D`) dans les sous-arbres `t` et `f` sont d'indice strictement supérieur à `i`. On écrit `i` $<$ `t` et `i` $<$ `f`.
   (Les arbres qui respectent cette condition sont dits **ordonnés**.)

Dans la suite, on n'utilise et on ne construit que des arbres réduits et ordonnés.

Question 13 Définissez un arbre `zero` de type `bdt` qui représente la formule *faux*. Définissez un arbre `one` de type `bdt` qui représente la formule *vrai*. ◇

Question 14 Écrivez une fonction `var` de type `int -> bdt` telle que l'arbre `var i` représente la formule x_i. ◇

Question 15 Écrivez une fonction `neg` de type `bdt -> bdt` telle que, si l'arbre `b` représente une certaine formule B, alors l'arbre `neg b` représente la formule $\neg B$. ◇

Question 16 Écrivez une fonction `d` de type `int -> bdt -> bdt -> bdt` telle que, si les arbres `t` et `f` représentent respectivement les formules T et F, et si les conditions `i` $<$ `t` et `i` $<$ `f` sont satisfaites, alors `d i t f` représente la formule « *si* x_i *alors* T *sinon* F ». On souligne le fait que la condition `t` $\neq$ `f` n'est pas nécessairement satisfaite ; néanmoins, l'arbre `d i t f` doit être réduit et ordonné. ◇

Question 17 Écrivez une fonction `conj` de type `bdt -> bdt -> bdt` telle que, si les arbres `b1` et `b2` représentent respectivement les formules B_1 et B_2, alors `conj b1 b2` représente la formule $B_1 \wedge B_2$.◇

Question 18 On souhaite présenter le type `bdt` à l'utilisateur comme un type abstrait. Peut-on sans danger lui donner accès aux constantes `zero` et `one` et aux fonctions `var`, `neg`, `d`, et `conj` décrites ci-dessus ? Si non, quel danger court-on ? Justifiez brièvement. ◇

Question 19 Quelle(s) autre(s) fonction(s) devrait-on donner à l'utilisateur pour que le type `bdt` et ses opérations soient utiles ? Justifiez brièvement. ◇

2.2 Diagrammes binaires de décision

Afin d'économiser de la mémoire et du temps, on souhaite maintenant imposer un partage maximal aux arbres binaires de décision. En d'autres termes, on veut éviter d'allouer dans le tas deux objets identiques : si deux nœuds ont la même étiquette (`C` ou `D`) et les mêmes champs, alors ils doivent être en fait un seul et même nœud.

On définit pour cela une nouvelle structure de données, légèrement plus complexe, appelée **diagramme binaire de décision**, ou « binary decision diagram ».

```
type variable = int
type identity = int
type bdd = {
    identity: identity;
    nature: nature;
  }
and nature =
  | C of bool
  | D of variable * bdd * bdd
```

FIGURE 5 – Diagrammes binaires de décision

Un « nœud » est un objet de type `bdd`. Chaque nœud a maintenant une « identité », c'est-à-dire un entier, stocké dans le champ `identity`. Chaque nœud a également une « nature », qui indique quelle formule logique est représentée par ce nœud. Elle est stockée dans le champ `nature`.

On impose deux **invariants** globaux :

1. L'identité d'un nœud est unique. En d'autres termes, deux nœuds distincts dans le tas doivent avoir des identités distinctes. Donc, l'égalité `bdd1.identity = bdd2.identity` implique l'égalité d'adresses `bdd1 == bdd2`.
2. Le partage est maximal. En d'autres termes, deux nœuds de même nature doivent être en fait un seul et même nœud. Donc, l'égalité `bdd1.nature = bdd2.nature` implique l'égalité d'adresses `bdd1 == bdd2`.

Dans la suite, on suppose et on fait en sorte que ces invariants soient respectés à tout instant.

Pour imposer le partage maximal, à chaque fois que l'on s'apprête à allouer un nouveau nœud, il faut être capable de déterminer si on a déjà alloué dans le passé un nœud de même nature. Pour cela, on utilise des tables de hachage, d'où les questions qui suivent.

Question 20 Écrivez une fonction `equal` de type `bdd -> bdd -> bool` qui détermine si deux nœuds représentent la même formule. Écrivez une fonction `hash` de type `bdd -> int` qui à un nœud associe un code de hachage. Ces deux fonctions doivent être très courtes et très efficaces. Elles peuvent exploiter les invariants décrits ci-dessus. ◇

Question 21 À l'aide de la fonction `equal` écrite lors de la question précédente, écrivez une fonction `nature_equal` de type `nature -> nature -> bool` qui détermine si deux valeurs de type `nature` sont égales (i.e., décrivent la même formule). À l'aide de la fonction `hash` écrite lors de la question précédente, écrivez une fonction `nature_hash` de type `nature -> int` qui à une valeur de type `nature` associe un code de hachage. Pour combiner deux entiers x et y en un seul, on peut utiliser la fonction `combine : int -> int -> int` définie par `let combine x y = x * 65599 + y`. ◇

```
module Make (H : sig
  type t
  val equal : t -> t -> bool
  val hash : t -> int
end) : sig
  type key = H.t
  type 'a t
  val create : int -> 'a t
  val add : 'a t -> key -> 'a -> unit
  val find : 'a t -> key -> 'a
end
```

FIGURE 6 – Signature du foncteur `Hashtbl.Make`

Question 22 À l'aide des fonctions `nature_equal` et `nature_hash` de la question précédente, et à l'aide du foncteur `Hashtbl.Make` fourni par la librairie d'OCaml (figure 6), définissez un module `N` qui implémente des tables de hachage dont les clefs sont de type `nature`. ◇

Question 23 Écrivez une fonction `memo` de type `(nature -> 'a) -> (nature -> 'a)` telle que, si `f` est une fonction quelconque de type `nature -> 'a`, alors `let f' = memo f` définit une fonction `f'` de type `nature -> 'a` qui est une version mémoisante de la fonction `f`. En d'autres termes, si on applique `f'` pour la première fois à un certain `x`, le résultat `y` doit être le même que si on avait appliqué `f` à `x`. Si plus tard on applique à nouveau `f'` à ce même `x`, alors le résultat doit être ce même `y`, et il doit être obtenu sans que `f` soit appelée à nouveau. La fonction `f'` utilisera, de façon interne, une table de hachage fournie par le module `N` de la question précédente. ◇

Pour attribuer facilement à chaque nouveau nœud une identité distincte, on suppose donnée une fonction `new_counter` de type `unit -> (unit -> int)`. L'appel `new_counter()` crée un nouveau compteur, c'est-à-dire une fonction `next` de type `unit -> int` qui, à chaque appel, renvoie un nouvel entier.

Question 24 À l'aide des fonctions `memo` et `new_counter`, définissez une fonction `construct` de type `nature -> bdd` qui, étant donnée une nature, renvoie un nœud de cette nature. Afin de respecter l'invariant de partage maximal, s'il existe déjà un nœud de cette nature, ce nœud existant doit être renvoyé. Dans le cas contraire, un nouveau nœud, doté d'une nouvelle identité, doit être alloué et renvoyé. ◇

Question 25 À l'aide de la fonction `construct` et en vous inspirant des questions 13, 14 et 16, définissez les constantes `zero` et `one` de type `bdd` ainsi que les fonctions `var` de type `variable -> bdd` et `d` de type `int -> bdd -> bdd -> bdd`. ◇