Programmation avancée

Examen

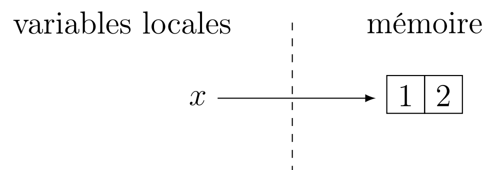
Tous les documents sur papier sont autorisés, dans la limite du raisonnable (50 pages maximum). L'usage d'un dispositif électronique est interdit.

La durée de l'épreuve est de 2 heures. Il n'est pas nécessaire de traiter l'intégralité du sujet pour avoir une bonne note.

Les parties sont indépendantes. Au sein d'une même partie, il est préférable de traiter les questions dans l'ordre, mais chaque question est indépendante. On pourra toujours utiliser le résultat d'une question (fonction à implémenter, par exemple) dans les questions qui suivent, même si l'on a pas traité la question.

1 Questions

Rappel : le programme `let x = (1, 2)` alloue un bloc en mémoire pointé par la variable `x` et contenant les entiers 1 et 2, que l'on représente comme suit :



Question 1. *Dessiner le contenu de la mémoire après l'exécution du programme donné ci-dessous.*

```
let x = ref 0
let y = (x, x)
x := 1
let a = [| [|1;2|]; [|3;4|] |]
```

Question 2. *Comment un programme OCaml fait-il pour libérer la mémoire correspondant à des blocs qui ont été alloués mais ne sont plus utilisés par le programme ? (réponse en 15 lignes maximum)*

Le mécanisme de types algébriques généralisés (GADTs) disponible en OCaml permet d'écrire la déclaration suivante, qui définit un témoin d'égalité entre types.

```
type (_, _) eq =
  | Equal : ('a, 'a) eq
```

Question 3. *Implémenter une fonction OCaml nommée* `cast` *en complétant la déclaration ci-dessous.*

```
let cast : type a b. (a, b) eq -> a -> b =
  (* TODO *)
```

On définit ci-dessous le type algébrique `'a result` représentant soit une valeur `'a`, soit une erreur avec un message d'erreur associé.

```
type 'a result =
  | Success of 'a
  | Error of string
```

La monade des calculs pouvant renvoyer une erreur est un module `ErrM` avec la signature suivante :

```
module ErrM : sig
  type 'a t
  val return : 'a -> 'a t
  val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
  val fail : string -> 'a t
  val run : 'a t -> 'a result
end
```

Voici un exemple de programme l'utilisant :

```
open ErrM

let res = run (
  return 40 >>= fun n ->
  return (n + 2)
) (* res = Success 42 *)

let res = run (
  return 40 >>= fun n ->
  fail "oops"
) (* res = Error "oops" *)
```

Question 4. *Donner une implémentation pour le module `ErrM`, satisfaisant la signature donnée ci-dessus. **NB** : cette implémentation ne doit utiliser que le fragment fonctionnel pur d'OCaml (pas d'exceptions, références, etc).*

Question 5. *On considère le programme Rust suivant :*

```
fn main() {
  let mut v = vec![1, 2, 3];
  for x in v.iter_mut() {
    *x += 1;
    if *x == 4 {
      v.push(5);
      // break
    }
  }
  println!("{}", v.iter().sum::<i32>());
}
```

(a) *Ce programme n'est pas accepté par le compilateur Rust. Pourquoi ? Expliquez quel(les) sont les durée(s) de vie considérées par le compilateur, les contraintes qu'il considère et expliquez où est le conflit.*

(b) *Si on décommente l'instruction* `break`*, il est accepté. Pourquoi cela fait-il une différence ?*

(c) *Qu'affiche alors le programme lorsqu'il est exécuté ?*

Question 6. *Le trait suivant est défini dans la bibliothèque standard de Rust :*

```
pub trait Add<Rhs = Self> {
  type Output;
  fn add(self, rhs: Rhs) -> Self::Output;
}
```

(a) *À quoi sert-il ? Quels sont les programmes qui l'utilisent ?*

(b) *Donner un exemple d'instances déjà définie dans la bibliothèque standard. On donnera le contenu de l'instance, sans donner le corps de la méthode* `add`*.*

Question 7. *On considère le programme Rust suivant :*

```
use std::rc::Rc;
use std::cell::{RefCell, RefMut};

fn main() {
    let x = Rc::new(RefCell::new(0));
    let y = x.clone();

    //{
    let mut b : RefMut<i32> = x.borrow_mut();
    *b += 1;
    //}

    *y.borrow_mut() += 1;
    println!("{}", x.borrow());
}
```

(a) *Que se passe-t-il si on l'exécute ? Pourquoi ?*

(b) *Même question si on décommente les accolades.*

Question 8.

(a) *Qu'est-ce qu'une* race condition *?*

(b) *Dans un langage comme C++, que se passe-t-il si un programme contient une race condition ?*

(c) *Est-ce un problème pour un programme Rust sans code* `unsafe` *?*

```
module Map : sig
  module type OrderedType = sig
    type t
    val compare : t -> t -> int
    (* [compare x y] renvoie -1 si x < y, 0 si x = y, et 1 si x > y *)
  end

  module type S = sig
    type key
    type 'a t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a option
  end

  module Make (Ord : OrderedType) : S with type key = Ord.t
end
```

FIGURE 1 – Interface minimale du module `Map` de la bibliothèque standard

2 Une table associative hétérogène

La bibliothèque standard OCaml définit un module `Map` fournissant une implémentation de tables associatives fonctionnelles (à l'aide d'arbres équilibrés). L'implémentation est fournie par un foncteur `Map.Make`, renvoyant une implémentation de la table étant donné un module spécifiant le type ordonné à utiliser pour les clefs.

On donne en figure 2 la *signature* d'une version minimale de ce module `Map`. Dans la suite du sujet, on supposera que l'on a accès à un tel module `Map`.

Question 9. *Instancier le foncteur `Map.Make` avec le module `String` de la bibliothèque standard, afin d'obtenir un module contenant une implémentation de tables associatives indicées par des chaînes de caractère. Appeler ce module `StringMap`.*

Question 10. *`StringMap` a le type `Map.S with type key = string`. Utiliser `StringMap` pour créer une table associative `m` associant la clef `"a"` à l'entier 1, et `"b"` à 2.*

Les tables associatives fournies par le module `Map` ont une limitation : une table donnée ne peut stocker que des valeurs d'un même type. Par exemple, une table de type `int StringMap.t` stocke des valeurs de type `int` (associées à des clefs de type `string`). De même, une table de type `float StringMap.t` stocke des valeurs de type `float`. Et il est par conséquent impossible d'avoir une table stockant simultanément des valeurs de type `int` et de type `float`.

Dans cette section, on va voir qu'il est en fait possible d'implémenter une structure de tables associatives *hétérogènes*, qui permettent de stocker des valeurs de différents types au sein de la même table. Pour cela, on fera usage de types algébriques généralisés (GADTs) afin d'instancier astucieusement le foncteur `Map.Make`.

La figure 2 donne un début d'implémentation pour une telle structure (qu'on complétera en section 2.2). Comme pour `Map`, l'implémentation consiste en un foncteur (`MakeHet`)

```ocaml
(* Le résultat d'une comparaison sur des clefs hétérogènes.
   [Lt] indique "strictement plus petit", [Eq] indique "égal",
   et [Gt] "strictement plus grand". *)
type (_, _) order =
  | Lt : ('a, 'b) order
  | Eq : ('a, 'a) order
  | Gt : ('a, 'b) order

(* "Clefs hétérogènes" *)
module type KeyType = sig
  (* Une clef de type ['a t] permet de stocker des valeurs de type ['a]. *)
  type 'a t
  (* Fonction de comparaison sur les clefs. D'après la définition de [order],
     si [compare x y] renvoie [Eq] alors on sait que [x] et [y] sont
     paramétrées par le même type. *)
  val compare : 'a t -> 'b t -> ('a, 'b) order
end

(* Signature pour une structure de table hétérogène. *)
module type S = sig
  type 'a key
  type t
  val empty : t
  val add : 'a key -> 'a -> t -> t
  val find : 'a key -> t -> 'a option
end

(* Foncteur implémentant la signature [S] à partir d'un module [Key]
   de signature [KeyType]. *)
module MakeHet (Key : KeyType) : S with type 'a key = 'a Key.t =
struct
  type 'a key = 'a Key.t
  (* TODO: implémentation *)
end
```

Figure 2 – Implémentation à compléter pour des tables associatives hétérogènes

paramétré par un module spécifiant le type ordonné des clefs. *La différence principale est que le type des clefs est désormais un type paramétré (`'a t` dans `KeyType`).* Une clef de type `'a key` permet de stocker une valeur de type `'a` dans la table associative.

2.1 Un exemple de clefs hétérogènes

Le foncteur `MakeHet` prend en argument un module de signature `KeyType`. Dans cette section, on implémente un module `NumKey` satisfaisant cette signature et fournissant un type de clefs pouvant être associées à des nombres (entiers ou flottants) dans une table hétérogène. On donne ci-dessous une implémentation partielle pour le module `NumKey` :

```
module NumKey = struct
  type _ t =
    | IntK : string -> int t
    | FloatK : string -> float t

  let compare : type a b. a t -> b t -> (a, b) order =
    (* TODO *)
end
```

Une valeur de type `'a NumKey.t` est soit de la forme `IntK "abc"`, soit de la forme `FloatK "abc"`. Le choix du constructeur (`IntK` ou `FloatK`) indique le type attendu pour la donnée associée à la clef dans la table : dans le premier cas, il s'agit d'un entier, et dans le deuxième cas d'un flottant. La chaîne de caractères est le *nom* de la clef ; on peut donc avoir des clefs de noms différents permettant de stocker de multiples entiers et flottants dans la table.

Question 11. *Compléter le code en donnant une implémentation pour la fonction `compare` du module `NumKey`. Il doit s'agir d'un ordre total sur les clefs.*

On pose `module NumHMap = MakeHet (NumKey)`.

Question 12. *Écrire un programme créant une table associative `m` associant une clef de nom `"a"` à l'entier 1 et une clef de nom `"b"` au flottant 2.5. Quel est la valeur renvoyée par `NumHMap.find (FloatK "a") m` ? Quel est son type ?*

2.2 Structure associative hétérogène : implémentation

On s'intéresse maintenant à implémenter le foncteur de la figure 2 (pour remplacer le commentaire marqué "TODO"). Dans la suite de la section, on considère que le code que l'on écrit est à insérer à la place du TODO ; on suppose donc que l'on a accès à un module `Key` de signature `KeyType` (sur lequel on ne sait rien de plus).

On cherche à instancier `Map.Make` avec un type de clefs et de valeurs bien choisi. La difficulté est la suivante : on a un type de clef polymorphe (`'a Key.t`) et des valeurs dont le type dépend de celui de la clef, tandis que `Map.Make` veut un unique type de clefs et de valeurs pour la map entière.

2.2.1 Clefs

On s'intéresse d'abord au cas des clefs. On définit ci-dessous le type `ekey` comme un GADT. Une valeur de type `ekey` contient un `'a Key.t` où le type `'a` est quantifié existentiellement. Le type `ekey` n'est plus paramétré : on peut l'utiliser pour instancier `Map.Make`.

```
type ekey =
  | E : 'a Key.t -> ekey
```

Question 13. *Définir un module `EKey` en complétant le code ci-dessous. Le module `EKey` ainsi défini doit satisfaire la signature `OrderedType`.*

```
module EKey = struct
  type t = ekey
  (* TODO *)
end
```

On pose `module EKeyMap = Map.Make (EKey)`.

2.2.2 Packs

On s'intéresse maintenant au cas des données stockées dans la map. On définit un *pack* comme la paire d'une valeur de type `'a Key.t` et d'une valeur de type `'a`. On veut que le type `pack` ne dépende pas de `'a` : comme pour les clefs, on veut quantifier existentiellement sur le type `'a`.

Question 14. *Définir un type GADT nommé `pack`. Une valeur de type `pack` stocke la paire d'un `'a Key.t` et d'un `'a`, pour un `'a` quantifié existentiellement. Implémenter ensuite la fonction `pack` en complétant le code ci-dessous.*

```
let pack : type a. a Key.t -> a -> pack =
  (* TODO *)
```

Question 15. *En utilisant la fonction `Key.compare` et le type `eq` tel que défini en section 1, implémenter une fonction `key_eq` en complétant le code ci-dessous. Cette fonction teste si deux clefs sont égales, et si oui renvoie un témoin d'égalité pour leur paramètres de type.*

```
let key_eq : type a b. a Key.t -> b Key.t -> (a, b) eq option =
  (* TODO *)
```

Question 16. *Étant donné le* pack *d'une clef et d'une valeur, et étant donné une seconde clef, on peut extraire la valeur contenue dans le pack si les deux clefs sont égales. Implémenter une fonction `unpack` implémentant cette opération, en complétant le code ci-dessous.*

```
let unpack : type a. a Key.t -> pack -> a option =
  (* TODO *)
```

2.2.3 Fonctions

On définit le type `t` des tables associatives hétérogènes comme ci-dessous.

```
type t = pack EKeyMap.t
```

Il ne reste plus qu'à implémenter les opérations de notre table hétérogène. On utilisera les fonctions définies dans les questions précédentes lorsque celles-ci sont pertinentes.

Question 17. *Donner l'implémentation pour la valeur* `empty` *représentant la table vide.*

```
let empty : t = (* TODO *)
```

Question 18. *Donner l'implémentation de la fonction* `add` *qui ajoute un élément à une table associative hétérogène.*

```
let add : 'a Key.t -> 'a -> t -> t =
  (* TODO *)
```

Question 19. *Donner l'implémentation de la fonction* `find` *qui renvoie la valeur associée à une clef dans une table hétérogène.*

```
let find : 'a Key.t -> t -> 'a option =
  (* TODO *)
```