ENS Paris-Saclay 2023-2024

Programmation avancée

Examen

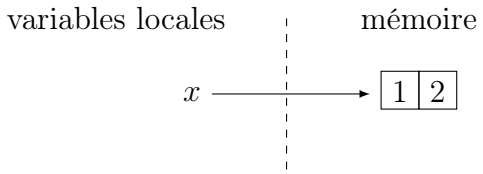
Tous les documents sur papier sont autorisés, dans la limite du raisonnable (50 pages maximum). L'usage d'un dispositif électronique est interdit.

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 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.*

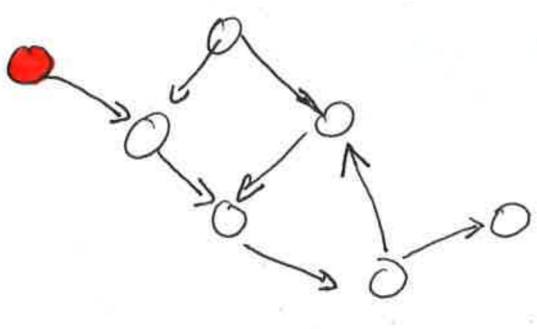
```
let x = ref 0
let y = (x, x)
x := 1
let a = [| [|1;2|]; [|3; !x|] |]
```

Solution ...

Question 2. *Les fonctions `malloc()` et `free()` du langage C permettent respectivement d'obtenir accès à un nouveau bloc mémoire de la taille souhaitée, et de le libérer lorsque l'on n'en a plus besoin. (OCaml et Rust utilisent en interne des fonctions similaires pour gérer la mémoire et allouer de nouveaux blocs par exemple lors de l'appel à `Array.make` ou `Vec::new()`.)*

Expliquer succintement le fonctionnement interne de `malloc()` et `free()` (ou un fonctionnement possible). Quels mécanismes mettent-t-il en place pour pouvoir faire efficacement leur travail ? (réponse en 10 lignes maximum)

On considère le graphe suivant, qui représente une collection d'objets dans la mémoire d'un programme à un instant t de son exécution. Un nœud du graphe correspond à un bloc mémoire, et une arête du graphe à une référence d'un bloc à un autre. Le nœud coloré en rouge correspond à une *racine*.



Question 3. *Expliquer succintement le principe de gestion mémoire par* comptage de références *(5 lignes maximum).*

Question 4. *Reproduire le graphe ci-dessus en indiquant pour chaque nœud le compte de références pour ce nœud. Puis indiquer quels nœuds peuvent être désalloués (selon la stratégie de comptage de références) en les barrant d'une croix.*

Question 5. *Quels nœuds seraient désalloués par une stratégie de gestion mémoire par "ramasse miette traçant" (par exemple de type* mark and sweep *ou* stop and copy*) ? Pourquoi ? (réponse en 8 lignes maximum)*

On considère le code OCaml suivant, qui définit un module de manipulation de nombres entiers pairs.

```
module Even = struct
  type t = int
  let double x = 2 * x
  let plus2 x = x + 2
  let of_int x = if x mod 2 = 0 then Some x else None
  let check x = assert (x mod 2 = 0)
end
```

Question 6. *Donner une signature de module* `S` *pour* `Even`*, telle que :*

- *quel que soit le code client utilisant* `Even` *restreint à la signature* `S`*, l'assertion dans la fonction* `check` *soit toujours satisfaite ;*
- *la signature donne le plus de liberté possible au code cliant utilisant le module (par exemple, on veut mieux que la signature vide qui cache l'entièreté du module).*

Solution

```
module type S = sig
 type t
 val double : int -> t
 val plus2 : t -> t
 val of_int : int -> t option
 val check : t -> unit
end
```

Le mécanisme de types algébriques généralisés (GADTs) disponible en OCaml permet d'écrire la déclaration suivante :

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

Question 7. *Implémenter une fonction OCaml nommée* `transitivity` *en complétant la déclaration ci-dessous.*

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

Solution

```
let transitivity : type a b c. (a, b) eq -> (b, c) eq -> (a, c) eq =
  fun Eq Eq -> Eq
```

Question 8. *Parmis les types Rust suivants, lesquels implémentent le trait* `Copy` *(pas de justification nécessaire) :* `u32`, `Box<u32>`, `&u32`, `&Box<u32>`, `Vec<&u32>`, `(&mut i32, bool)`, `Option<(u32, i32)>` *?*

Solution `u32`, `&u32`, `&Box<u32>`, `Option<(u32, i32)>` implémentent le tait `Copy`.

Question 9. *En Rust, à quoi servent les* règles d'orphelinat *(*orphan rules*) ?*

Solution Ces règles permettent de garantir la *cohérence* des instances des traits : il n'existe jamais deux instances d'un même trait pour un même type.

Question 10. *Qu'est-ce que la* mutabilité intérieure *en Rust ? Donner l'exemple d'un type de la bibliothèque standard de Rust ayant cette fonctionnalité.*

Solution Il s'agit de la capacité de modifier la mémoire avec uniquement un emprunt partagé. Par exemple, les types `Cell`, `RefCell`, `Rc` ont cette fonctionnalité.

Question 11. *Que permettent les traits* `Send` *et* `Sync` *? Quel lien y a-t-il entre les instances de ces deux traits ?*

Solution Lorsqu'un type implémente le trait `Send`, on peut envoyer une valeur de ce type à un autre thread. Lorsqu'un type implémente le trait `Sync`, on peut partager un pointeur vers une valeur de ce type avec un autre thread. Un type `T` implémente `Sync` ssi `&T` implémente `Send`.

2 Itération, de bout en bout

Dans cette partie du sujet, on s'intéresse à différentes notions d'*itérateurs*. Un itérateur est une valeur permettant de calculer sur une séquence de valeurs, par exemple les valeurs contenues dans une structure de données. On écrira par exemple du code consommant les éléments d'un itérateur quelconque, s'abstrayant ainsi de la manière dont sont produits ces éléments. On pourra alors appeler ce code avec différents itérateurs provenant de différentes structures de données.

On considère les types suivants en OCaml. Chaque type correspond à une notion différente d'itérateur, que l'on va considérer dans la suite du sujet.

1. Traverse :

```
type 'a traverse = ('a -> unit) -> unit
```

2. Seq :

```
type 'a seq = unit -> 'a node
and 'a node =
  | Step of 'a * 'a seq
  | Done
```

3. Cursor :

```
type ('a, 's) cursor = 's * ('s -> ('a * 's) option)
```

4. Gen :

```
type 'a gen = unit -> 'a option
```

Pour les itérateurs 2 et 3, on considère de plus que **les fonctions contenues dans l'itérateur sont pures** (en particulier, pas de mutation).

— Un itérateur « Traverse » prend une fonction en argument et l'appelle sur chaque élément de la séquence ;

— Un itérateur « Seq » appliqué à `()` renvoie l'élément suivant ainsi qu'un nouveau « Seq » pour continuer, ou indique que l'itération est terminée ;

— Un itérateur « Cursor » consiste en la paire d'un « état interne », et d'une fonction permettant d'accéder à l'élément suivant ainsi que l'état interne suivant, s'il existe.

— Un itérateur « Gen » renvoie un nouvel élément à chaque fois qu'on l'appelle en l'appliquant à `()` (ou `None` si l'itération est terminée).

Pour chaque type `'a t` de la liste ci-dessus, et étant donnée une séquence d'éléments, il est possible : 1) d'*implémenter* une valeur de type `'a t` produisant cette séquence d'éléments ; 2) d'*utiliser* une valeur de type `'a t` pour itérer sur cette séquence d'éléments.

```
let print_traverse (iter: string traverse): unit =
  iter print_endline

let rec print_seq (iter: string seq): unit =
  match iter () with
  | Done -> ()
  | Step (str, next) -> print_endline str; print_seq next

let rec print_cursor (iter: (string, 's) cursor): unit =
  let (state, next) = iter in
  match next state with
  | None -> ()
  | Some (str, state') -> print_endline str; print_cursor (state', next)

let rec print_gen (iter: string gen): unit =
  match iter () with
  | None -> ()
  | Some str -> print_endline str; print_gen iter
```

FIGURE 1 – Exemples de fonctions affichant le contenu d'un itérateur de chaînes de caractères, pour chaque type d'itérateur.

On donne en Figure 1 des exemples de code *utilisant* des itérateurs pour chaque type ci-dessus ; en l'occurence ceux-ci itèrent sur une séquence de chaînes de caractères et les affichent à la suite avec `print_endline`.

Note : dans la suite du sujet, lorsqu'une définition de fonction est à compléter, on s'autorisera à rajouter `rec` si nécessaire.

2.1 Implémentation d'itérateurs

On s'intéresse maintenant, pour une structure de données concrète, à *implémenter* des itérateurs produisant la séquence des éléments contenus dans cette structure.

Question 12. *Implémenter pour chaque type d'itérateur une fonction produisant un itérateur à partir d'une liste d'éléments (de type `'a list`).*

```
let list_traverse (l: 'a list): 'a traverse = (* TODO *)
let list_seq (l: 'a list): 'a seq = (* TODO *)
let list_cursor (l: 'a list): ('a, (* TODO *)) cursor = (* TODO *)
let list_gen (l: 'a list): 'a gen = (* TODO *)
```

Solution

```
let list_traverse (l: 'a list): 'a traverse =
  fun f -> List.iter f l

let rec list_seq (l: 'a list): 'a seq = fun () ->
```

```
  match l with
  | [] -> Done
  | x :: xs -> Step (x, list_seq xs)

let rec list_cursor (l: 'a list): ('a, 'a list) cursor =
  (l, function
     | [] -> None
     | x :: xs -> Some (x, xs))

let list_gen (l: 'a list): 'a gen =
  let r = ref l in
  fun () ->
    match !r with
    | [] -> None
    | x :: xs -> r := xs; Some x
```

On définit maintenant un type d'arbres binaires dont les éléments sont stockés aux feuilles :

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

Question 13. *Implémenter un fonction produisant un itérateur « Traverse » à partir d'un arbre binaire.*

```
let tree_traverse (t: 'a tree): 'a traverse = (* TODO *)
```

Solution

```
let rec tree_traverse t f =
  match t with
  | Leaf x -> f x
  | Node (t1, t2) -> tree_traverse t1 f; tree_traverse t2 f
```

Question 14. *Implémenter un fonction produisant un itérateur « Cursor » à partir d'un arbre binaire.*

```
let tree_cursor (t: 'a tree): ('a, (* TODO *)) cursor = (* TODO *)
```

Solution

```
let tree_cursor (t: 'a tree): ('a, 'a tree list) cursor =
  let rec next = function
    | [] -> None
    | Leaf x :: stk -> Some (x, stk)
    | Node (t1, t2) :: stk -> next (t1::t2::stk) in
  ([t], next)
```

2.2 Conversion entre itérateurs

Étant données nos différentes notions d'itérateur, on cherche maintenant à les comparer en terme d'expressivité : peut-on implémenter un itérateur d'un certain type en fonction d'un itérateur d'un autre type ? (Tout en préservant l'ordre des éléments et en gardant la même complexité asymptotique.)

Rappel : pour les itérateurs « Seq » et « Cursor » les fonctions contenues dans l'itérateur doivent êtres **pures**.

Question 15. *Peut-on implémenter un itérateur « Cursor » en fonction d'un itérateur « Gen » inconnu ? Justifier : si oui, implémenter une fonction de type* `'a gen -> 'a cursor` *; si non, expliquer pourquoi.*

Solution Non : Gen utilise de l'état mutable et Cursor non.

Question 16. *Peut-on implémenter un itérateur « Gen » en fonction d'un itérateur « Cursor » inconnu ? Justifier : si oui, implémenter une fonction de type* `'a cursor -> 'a gen` *; si non, expliquer pourquoi.*

Solution

```
let cursor_to_gen ((st, next): 'a cursor): 'a gen =
  let r = ref st in
  fun () ->
    match next !r with
    | None -> None
    | Some (x, st) -> r := st; Some x
```

Question 17. *Peut-on implémenter un itérateur « Traverse » en fonction d'un itérateur « Gen » inconnu ? Justifier : si oui, implémenter une fonction de type* `'a gen -> 'a traverse` *; si non, expliquer pourquoi.*

Solution

```
  let gen_to_traverse (g: 'a gen): 'a traverse =
    let rec loop f =
      match g () with
      | None -> ()
      | Some x -> f x; loop f
   in loop
```

Question 18. *Peut-on implémenter un itérateur « Gen » en fonction d'un itérateur « Traverse » inconnu ? Justifier : si oui, implémenter une fonction de type* `'a traverse -> 'a gen` *; si non, expliquer pourquoi.*

Solution Non, sauf si on a une primitive d'inversion de contrôle dans le langage.

Question 19. *Même question pour chaque combinaison de deux itérateurs possibles, listée ci-dessous.* ***Ne pas justifier****, simplement indiquer par « oui » ou « non » s'il est possible d'implémenter une fonction transformant un itérateur du premier type en fonction d'un itérateur inconnu du deuxième type.*

- *Implémenter Traverse en fonction de Seq*
- *Implémenter Traverse en fonction de Cursor*
- *Implémenter Traverse en fonction de Gen*
- *Implémenter Seq en fonction de Traverse*
- *Implémenter Seq en fonction de Cursor*
- *Implémenter Seq en fonction de Gen*
- *Implémenter Cursor en fonction de Traverse*
- *Implémenter Cursor en fonction de Seq*
- *Implémenter Cursor en fonction de Gen*
- *Implémenter Gen en fonction de Traverse*
- *Implémenter Gen en fonction de Seq*
- *Implémenter Gen en fonction de Cursor*

Solution

- Oui
- Oui
- Oui
- Non
- Oui
- Non
- Non
- Oui
- Non
- Non
- Oui
- Oui

2.3 Comparaison avec les itérateurs de Rust

On rappelle le trait `Iterator` de la bibliothèque standard de Rust :

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // ...
}
```

Question 20. *À quelle notion d'itérateur décrite ci-dessus les itérateurs définis dans la bibliothèque standard de Rust ressemblent-ils le plus ? Pourquoi ?*

Solution Les itérateurs de Rust agissent donc en utilisant un état en mémoire qu'ils mutent à chaque appel. Cela correspond aux itérateurs « Gen ».

Question 21. *Quel serait, en Rust, le prototype*[1] *d'une fonction implémentant un itérateur « Traverse » manipulant des valeurs de type* `T` *?*

Solution

```
fn traverse<F: FnMut(T)>(f: F)
```

1. Rappel : le *prototype* d'une fonction correspond à la partie de sa déclaration située avant le corps de la fonction. Il inclut le nom de la fonction, les paramètres génériques, le type de ses paramètres, le type de retour et les contraintes de traits. Par exemple, le prototype de la fonction `main` est `fn main()`.

2.4 Itérateur monadique

On suppose qu'est défini un type `st`. On donne la signature de module `StateMonad` suivante :

```
module type StateMonad = sig
  type 'a t
  val return : 'a -> 'a t
  val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
  val get : st t
  val put : st -> unit t
end
```

On suppose qu'on a un foncteur `MonadicIter` de signature :

```
module MonadicIter(M: StateMonad) : sig
  val init_st : st
  val monadic_it : unit -> 'a option M.t
end
```

Ce foncteur donne un itérateur « Gen », utilisant la monade d'état `M.t` pour manipuler son état interne (plutôt que de l'état mutable). La valeur `init_st` contient la valeur initiale de l'état géré par la monade. On suppose que l'itérateur `it` ci-dessus est pur : il n'utilise que la monade pour son état interne, et n'utilise pas, par exemple, de référence ou d'enregistrement mutable.

Question 22. *En instanciant ce foncteur, écrire un itérateur « Cursor » produisant la même séquence. Rappel : les itérateurs « Cursor » sont toujours supposés purs ; votre implémentation ne peut donc pas utiliser de références ou d'enregistrements mutables.*

Solution

```
  module StateMonad = struct
    type 'a t = st -> ('a * st)
    let return x = fun s -> (x, s)
    let (>>=) x f = fun s -> let (y, s) = x s in f y s
    let get = fun s -> (s, s)
    let put x = fun _ -> ((), x)
  end

  module It = MonadicIter(StateMonad)
  let it : ('a, st) cursor =
    (It.init_st,
     fun s -> let (o, s) = It.monadic_it () s in Option.map (fun x -> (x, s)) o)
```

2.5 Sérialisation

On s'intéresse dans cette section à la *sérialisation* d'un type de données structurées (représenté par un type algébrique) en une séquence (représentée par un itérateur) de *tokens*.

Le type `data` de nos données structurées est défini ci-dessous.

```
type data =
  | Num of int
  | Str of string
  | Pair of data * data
```

Une valeur de type `data` représente une structure paires imbriquées, contenant des entiers ou chaines de caractères à leurs feuilles.

Une représentation alternative est comme une séquence de valeurs du type `token` défini ci-dessous.

```
type token =
  | TokNum of int
  | TokStr of string
  | TokLeftParen
  | TokRightParen
```

Une valeur de type `data` peut alors être représentée par une séquence de tokens, de type `token seq`, en encadrant les deux arguments d'un constructeur `Pair` par les tokens `TokLeftParen` (avant) et `TokLeftParen` (après), qui représentent une paire de parenthèses.

Par exemple, la valeur OCaml de type `data` : `Pair (Pair (3, "abc"), 4)` (qui pourrait s'écrire "`((3 "abc") 4)`") est représentée par la séquence des tokens suivants : `TokLeftParen`, `TokLeftParen`, `TokNum 3`, `TokStr "abc"`, `TokRightParen`, `TokNum 4`, `TokRightParen`.

Question 23. *Définir les fonctions auxiliaires* ***`singleton`*** *et* `(++)` *qui renvoient respectivement un itérateur "Seq" contenant un seul élément, et la concaténation de deux itérateurs.*

```
let singleton (x: 'a): 'a seq = (* TODO *)
let (++) (s1: 'a seq) (s2: 'a seq): 'a seq = (* TODO *)
```

Solution

```
let singleton (x: 'a) : 'a seq = fun () -> Step (x, fun () -> Done)

let rec (++) (s1: 'a seq) (s2: 'a seq): 'a seq = fun () ->
  match s1() with
  | Done -> s2()
  | Step (x, next) -> Step (x, next ++ s2)
```

Question 24. *Définir une fonction de sérialisation transformant un* ***`data`*** *en la séquence de tokens le représentant.*

```
let serialize_data (d: data): token seq = (* TODO *)
```

Solution

```
let rec serialize_data = function
  | Num n -> singleton (TokNum n)
  | Str s -> singleton (TokStr s)
  | Pair (d1, d2) ->
    singleton TokLeftParen ++ serialize s1 ++ serialize s2 ++
    singleton TokRightParen
```

Une valeur de type `data` contient à la fois des données (entiers et chaines de caractères) et une structure dans laquelle sont stockées ces données (l'arbre de paires imbriquées). On cherche maintenant à séparer ces deux aspects. Pour ce faire, on définit un GADT `'a schema` qui contient seulement la description d'une structure arborescente, annoté par un paramètre de type `'a` décrivant le type des données à associer à cette structure.

```
type _ schema =
  | SNum : int schema
  | SStr : string schema
  | SPair : 'a schema * 'b schema -> ('a * 'b) schema
```

Question 25. *Écrire une fonction de sérialisation prenant en argument un schéma et les données correspondant comme deux arguments séparés.*

```
let serialize_schema : type a. a schema -> a -> token seq = (* TODO *)
```

Solution

```
let rec serialize_schema : type a. a schema -> a -> token Seq.t = fun sch x ->
  match sch with
  | SNum -> singleton (TokInt x)
  | SStr -> singleton (TokString x)
  | SPair (s1, s2) ->
    singleton TokLeftParen ++ serialize_schema s1 (fst x) ++
      serialize_schema s2 (snd x) ++ singleton TokRightParen
```

À partir d'une valeur de type `data`, on peut ensuite extraire la paire du schéma correspondant d'une part, et des données d'autre part.

Question 26. *Définir un type `edata` représentant la décomposition d'un `data` en une paire d'un schéma et des données correspondantes. (Indice : utiliser un GADT.) L'utiliser ensuite pour définir les fonctions ci-dessous.* ***Note :*** *la fonction `serialize_edata` doit réutiliser `serialize_schema`.*

```
type edata = (* TODO *)
let data_to_edata : data -> edata = (* TODO *)
let serialize_edata : edata -> token seq =
  (* TODO (indication: deux lignes de code maximum *)
```

Solution

```
type edata =
  | E : 'a schema * 'a -> edata

let rec data_to_edata : data -> edata = fun dat ->
  match dat with
  | Num n -> E (SNum, n)
  | Str s -> E (SStr, s)
  | Pair (d1, d2) ->
    let E (s1, x1) = data_to_edata d1 in
    let E (s2, x2) = data_to_edata d2 in
    E (SPair (s1, s2), (x1, x2))
```

Alternativement, on peut remplacer le type `data` et définir un type GADT équivalent `'a tdata` suivant la même structure, dont le paramètre de type indique le type des données contenu. On peut alors typer plus simplement la fonction de décomposition d'un `tdata` en une paire schéma + données.

Question 27. *Définir `tdata` et la fonction `decompose_tdata` en complétant le code ci-dessous. **Note :** ne pas réutiliser `edata` ou aucune des fonctions ci-dessus (seulement la définition du type `schema`).*

```
type _ tdata = (* TODO *)
let decompose_tdata : type a. a tdata -> a schema * a = (* TODO *)
```

Solution

```
type _ tdata =
  | Num : int -> int tdata
  | Str : string -> string tdata
  | Pair : 'a tdata * 'b tdata -> ('a * 'b) tdata

let rec decompose_data : type a. a tdata -> a schema * a = fun dat ->
  match dat with
  | Num n -> (SNum, n)
  | Str s -> (SStr, s)
  | Pair (d1, d2) ->
    let (s1, x1) = decompose_data d1 in
    let (s2, x2) = decompose_data d2 in
    (SPair (s1, s2), (x1, x2))
```

2.6 Itérateurs et *lifetimes*

On rappelle que la bibliothèque de vecteurs de Rust contient un type `IterMut<'a, T>` qui implémente le trait `Iterator` et une fonction `iter_mut` permettant de construire une valeur de ce type :

```rust
struct IterMut<'a, T> { /* ... */ }

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn<'b> next(&'b mut IterMut<'a, T>) -> Option<&'a mut T>;
    /* ... */
}

impl<T> Vec<T> {
  fn<'a> iter_mut(&'a mut self) -> IterMut<'a, T> { /* ... */ }
}
```

Question 28. *À quoi servent ces déclarations ?*

Solution Ces déclarations permettent d'itérer de manière mutable sur un vecteur : l'itérateur produit renvoie des emprunts mutables vers le contenu du vecteur, permettant ainsi de modifier le contenu du vecteur pendant l'itération.

Question 29. *Est-il possible d'utiliser un emprunt renvoyé par un appel de* `IterMut::next` *après d'autres appels futurs à* `IterMut::next` *? Expliquer.*

Solution Tous les emprunts renvoyés par `IterMut::next` sont utilisables jusqu'à la fin de l'itération, y compris après d'autres appels à `IterMut::next`. La raison est que la lifetime des emprunts renvoyés par `IterMut::next` est `'a`, la lifetime qui annote le type `IterMut`, et qui correspond à la lifetime de l'emprunt initial du vecteur. Ainsi, la lifetime commune de ces emprunts est la lifetime de l'emprunt du vecteur, qui dure pendant toute l'itération.

Question 30. *Il est impossible d'implémenter un itérateur qui renverrait des paires d'emprunts mutables vers les cases successives d'un vecteur (i.e.,* `(&mut v[0], &mut v[1])`*, puis* `(&mut v[1], &mut v[2])`*, etc...).*

Cela serait-il sûr ? Pourquoi ?

Solution Tous les emprunts ainsi générés auraient le même type (`Item`), et donc la même durée de vie. Donc cette durée de vie sera toujours valide au dernier appel à `next`. Or, certains de ces emprunts pointent sur la même région mémoire. Donc plusieurs emprunts pointant sur la même région mémoire existeront en même temps avec un lifetime toujours valide. Ceci casse l'invariant « mutation XOR aliasing » de Rust, et donc sa sûreté.