Programmation Avancée (INF441)

Itération

François Pottier

17 mai 2016

Itérer sur une liste (1)

Pour calculer la somme des éléments d'une liste, on peut écrire en OCaml une fonction récursive :

```
let rec sum (xs : int list) : int =
  match xs with
  | []      -> 0
  | x :: xs -> x + sum xs
```

Dans quel ordre les additions sont-elles effectuées ?

Itérer sur une liste (1)

Pour calculer la somme des éléments d'une liste, on peut écrire en OCaml une fonction récursive :

```
let rec sum (xs : int list) : int =
  match xs with
  | []      -> 0
  | x :: xs -> x + sum xs
```

Dans quel ordre les additions sont-elles effectuées ?

- de la droite vers la gauche,
  car avant d'effectuer l'addition, il faut évaluer son argument `sum xs`

Quel est le coût en temps et en espace de ce calcul ?

Itérer sur une liste (1)

Pour calculer la somme des éléments d'une liste, on peut écrire en OCaml une fonction récursive :

```
let rec sum (xs : int list) : int =
  match xs with
  | []      -> 0
  | x :: xs -> x + sum xs
```

Dans quel ordre les additions sont-elles effectuées ?

- de la droite vers la gauche,
  car avant d'effectuer l'addition, il faut évaluer son argument `sum xs`

Quel est le coût en temps et en espace de ce calcul ?

- temps $O(n)$ et espace $O(n)$ (sur la pile)

Itérer sur une liste (2)

On peut aussi écrire une boucle :

```
let sum (xs : int list) : int =
  let accu = ref 0 in
  let remainder = ref xs in
  let finished = ref false in
  while not !finished do
    match !remainder with
    | [] -> finished := true
    | x :: xs -> accu := !accu + x; remainder := xs
  done;
  !accu
```

Dans quel ordre les additions sont-elles effectuées ?

Itérer sur une liste (2)

On peut aussi écrire une boucle :

```
let sum (xs : int list) : int =
  let accu = ref 0 in
  let remainder = ref xs in
  let finished = ref false in
  while not !finished do
    match !remainder with
    | [] -> finished := true
    | x :: xs -> accu := !accu + x; remainder := xs
  done;
  !accu
```

Dans quel ordre les additions sont-elles effectuées ?

- de la gauche vers la droite

Quel est le coût en temps et en espace ?

Itérer sur une liste (2)

On peut aussi écrire une boucle :

```
let sum (xs : int list) : int =
  let accu = ref 0 in
  let remainder = ref xs in
  let finished = ref false in
  while not !finished do
    match !remainder with
    | [] -> finished := true
    | x :: xs -> accu := !accu + x; remainder := xs
  done;
  !accu
```

Dans quel ordre les additions sont-elles effectuées ?

- de la gauche vers la droite

Quel est le coût en temps et en espace ?

- temps $O(n)$ et espace $O(1)$

Deux styles différents

Le style fonctionnel et récursif est plus simple, plus clair.

- le code peut être lu comme une définition mathématique !

Le style impératif et itératif semble ici moins coûteux en espace, mais :

- l'emploi de variables modifiables le rend moins clair,
- l'emploi de `finished` est lourd, et il faut deux tests par itération.

Peut-on marier l'élégance du premier et l'économie d'espace du second ?

Itérer sur une liste (3)

On peut écrire le calcul sous cette troisième forme :

```
let rec sum_accu (accu : int) (xs : int list) : int =
  match xs with
  | []      -> accu
  | x :: xs -> sum_accu (accu + x) xs

let sum (xs : int list) : int =
  sum_accu 0 xs
```

On voit que `sum_accu accu xs` est égal à `accu + sum xs`.

Dans quel ordre les additions sont-elles effectuées ?

Itérer sur une liste (3)

On peut écrire le calcul sous cette troisième forme :

```
let rec sum_accu (accu : int) (xs : int list) : int =
  match xs with
  | []      -> accu
  | x :: xs -> sum_accu (accu + x) xs

let sum (xs : int list) : int =
  sum_accu 0 xs
```

On voit que `sum_accu accu xs` est égal à `accu + sum xs`.

Dans quel ordre les additions sont-elles effectuées ?

- de la gauche vers la droite,
  car `accu + x` est évalué avant l'appel récursif

Quel est le coût en temps et en espace ?

Itérer sur une liste (3)

On peut écrire le calcul sous cette troisième forme :

```
let rec sum_accu (accu : int) (xs : int list) : int =
  match xs with
  | []      -> accu
  | x :: xs -> sum_accu (accu + x) xs

let sum (xs : int list) : int =
  sum_accu 0 xs
```

On voit que `sum_accu accu xs` est égal à `accu + sum xs`.

Dans quel ordre les additions sont-elles effectuées ?

- de la gauche vers la droite,
  car `accu + x` est évalué avant l'appel récursif

Quel est le coût en temps et en espace ?

- temps $O(n)$ et espace $O(1)$, car l'appel récursif est terminal

Appels terminaux

Qu'est-ce qu'un appel terminal ?

- à quoi le reconnaît-on ?
- pourquoi ne consomme-t-il pas d'espace sur la pile ?
- comment l'utilise-t-on habituellement ?

C'est l'un des sujets de cette séance. J'y reviendrai en dernière partie.

Itérer sur un tableau (1)

Et si les éléments sont stockés non dans une liste mais dans un tableau ?

Pour calculer leur somme, on peut écrire une boucle, comme en Java :

```
let sum (xs : int array) : int =
  let accu = ref 0 in
  for i = 0 to Array.length xs - 1 do
    accu := !accu + xs.(i)
  done;
  !accu
```

Cela demande un temps $O(n)$ et un espace $O(1)$.

Le code n'est pas le même que dans le cas d'une liste.

Itérer sur un tableau (2)

On peut aussi écrire une fonction récursive terminale :

```
let rec sum_accu (xs : int array) accu i : int =
  if i < Array.length xs then
    sum_accu xs (accu + xs.(i)) (i + 1)
  else
    accu

let sum (xs : int array) : int =
  sum_accu xs 0 0
```

Cela demande également un temps $O(n)$ et un espace $O(1)$.

Ici non plus, le code n'est pas le même que dans le cas d'une liste.

Itérer sur un ensemble ?

Et si les éléments sont stockés dans un ensemble ?

Comme dans la séance 5, donnons-nous une implémentation des ensembles d'entiers, en nous appuyant sur la bibliothèque `Set` :

```
module IntegerSet = Set.Make(struct type t = int ... end)
```

Ce module offre des fonctions `add`, `choose`, `remove`, etc.

Comment calculer la somme des éléments d'un ensemble ?

Itérer sur un ensemble ?

Et si les éléments sont stockés dans un ensemble ?

Comme dans la séance 5, donnons-nous une implémentation des ensembles d'entiers, en nous appuyant sur la bibliothèque `Set` :

```
module IntegerSet = Set.Make(struct type t = int ... end)
```

Ce module offre des fonctions `add`, `choose`, `remove`, etc.

Comment calculer la somme des éléments d'un ensemble ?

À première vue, parce que le type `IntegerSet.t` est abstrait,

- nous ne savons pas qu'un ensemble est représenté par un arbre,
- donc nous n'avons pas accès à ses éléments.

Itérer sur un ensemble ?

Certes, on peut imaginer d'utiliser `choose` et `remove` :

```
val choose: t -> elt (* raises Not_found if empty *)
val remove: elt -> t -> t
```

Ils suffisent à retirer les éléments, les uns après les autres.

Itérer sur un ensemble (1)

Cela donnerait par exemple :

```
let try_choose xs : int option =
  try Some (IntegerSet.choose xs) with Not_found -> None

let rec sum_accu xs accu =
  match try_choose xs with
  | None   -> accu
  | Some x -> sum_accu (IntegerSet.remove x xs) (accu + x)

let sum (xs : IntegerSet.t) : int =
  sum_accu xs 0
```

Ça marche, mais ce n'est pas naturel, et le coût en temps est $O(n \log n)$.

Quel est le problème ?

Nous venons de constater que :

- pour une structure donnée, il peut y avoir plusieurs façons d'itérer ;
- pour différentes structures, l'itération s'écrit de manière différente ;
- l'utilisateur aimerait ne pas savoir comment on itère ;
  - le choix entre liste, tableau, ensemble... ne l'intéresse parfois pas
- parfois, l'utilisateur ne doit pas savoir comment on itère.
  - le type des ensembles est abstrait

Quel est le problème ?

Comment séparer modulairement producteur et consommateur ?

Idéalement,

- le producteur sait comment produire une suite d'éléments,
  pas ce qu'on veut en faire ;
- le consommateur sait quoi faire de ces éléments,
  pas comment ils sont produits.

Quel est le problème ?

Quels sont les avantages de cette séparation ?

On respecte la barrière d'abstraction :

- le producteur de la suite des éléments d'un ensemble est écrit dans le module `Set`, donc a accès à la représentation interne ;
- les consommateurs peuvent être situés à l'extérieur.

On peut assembler producteurs/consommateurs de façon plus flexible.

- un producteur peut être combiné avec différents consommateurs,
- et vice-versa.

Quelles sont les solutions ?

Nous sommes face à un problème typique de décomposition modulaire.

Pouvons-nous le résoudre avec les outils dont nous disposons ?

- abstraction ($\exists$),
- paramétrisation ($\forall$),
- services (fonctions et objets)...

Oui.

La solution n'est d'ailleurs pas unique...

Contrairement aux apparences, ce problème est subtil et fondamental.

Il va nous occuper aujourd'hui et les deux prochaines semaines.

Quelles sont les solutions ?

Je vais mentionner d'abord une solution insatisfaisante :

- les listes ;

puis nous étudierons trois solutions viables :

- les réducteurs ou « fold » ;
- les itérateurs, modifiables ou non ;
- les flots.

On pourrait citer au moins trois autres solutions.

Les listes, un format universel ?

Le producteur doit communiquer au consommateur une suite d'éléments.

On pourrait adopter la liste comme format d'échange universel.

- c'est un type concret, connu de tous ;
- les producteurs construiraient des listes (dans le tas) ;
- les consommateurs accéderaient à ces listes.

La fonction `IntegerSet.elements`, de type `IntegerSet.t -> int list`, est un producteur conforme à ce modèle.

Quels problèmes cela poserait-il ?

Inconvénients

Les défauts de cette approche sont nombreux :

- l'espace requis est $O(n)$,
  même si le consommateur n'a besoin que d'un élément à la fois ;
- l'espace et le temps requis sont $O(n)$,
  même si le consommateur n'a besoin que des k premiers éléments ;
- le temps requis pour obtenir le premier élément est $O(n)$;
- si la suite est infinie, cette approche est inapplicable.

En bref, elle n'est jamais satisfaisante.

Comment faire mieux ?

Il faut que producteur et consommateur dialoguent sans attendre que la suite toute entière soit construite.

Quel mécanisme de communication peut-on utiliser pour cela ?

Le plus élémentaire est l'appel de fonction.

Il permet un transfert de contrôle entre producteur et consommateur.

Qui appelle qui ?

Mais qui doit être l'appelant, et qui doit être l'appelé ?

Si le producteur est l'appelant, il appelle le consommateur lorsqu'un élément est disponible :

- « tiens, voilà un élément ; traite-le et rends-moi la main. »

Si le consommateur est l'appelant, il appelle le producteur lorsqu'il a besoin d'un élément :

- « hep, trouve le prochain élément et donne-le moi. »

Qui appelle qui ?

Si le producteur est l'appelant, il prend la forme d'une fonction d'ordre supérieur, paramétrée par le consommateur.

- `map`, `iter`, `fold`, etc.
- c'est le style habituel de la bibliothèque d'OCaml.

Si le producteur est l'appelé, il prend la forme d'un itérateur.

- c'est le style habituel de la bibliothèque de Java.

Les deux styles sont possibles dans les deux langages.

Quel style préférer ?

Dans l'absolu, ce choix ne fait que déplacer la difficulté.

Celui qui joue le rôle de l'appelant est avantagé :

- Il bénéficie de suspension et redémarrage automatiques,
- avec sauvegarde automatique de son état (sur la pile).

Celui qui joue le rôle de l'appelé est potentiellement désavantagé :

- Il doit sauvegarder explicitement son état entre deux appels.

Contrôle au producteur

Donnons le contrôle au producteur : il est l'appelant. (Poly, §4.1.)

Le consommateur est donc une fonction `consume` de type `'a -> unit`, censée savoir traiter un élément.

Le producteur est paramétré par cette fonction. Ainsi, un même producteur peut être combiné avec différents consommateurs.

Itérer sur une liste

Voici, dans ce style, comment on produit les éléments d'une liste :

```
let rec iter consume xs =
  match xs with
  | [] ->
      ()
  | x :: xs ->
      consume x;
      iter consume xs
```

Ici, on itère de gauche à droite. Le type de `iter` est :

```
val iter: ('a -> unit) -> 'a list -> unit
```

L'appel récursif est terminal. La complexité en espace est $O(1)$.

Cette fonction est définie dans le module `List` de la bibliothèque.

Itérer sur une liste

Pour utiliser `List.iter`, on l'applique à un consommateur et à une liste.

Voici par exemple le calcul de la somme des éléments :

```
let sum xs =
  let accu = ref 0 in
  List.iter (fun x -> accu := !accu + x) xs;
  !accu
```

On pourrait remplacer `List.iter` par `Array.iter`, `IntegerSet.iter`, etc.

Si tous les producteurs se présentent sous forme d'une fonction `iter`, ils deviennent facilement interchangeables.

Le producteur s'écrit facilement...

On pourrait tout aussi aisément :

- itérer de droite à gauche sur une liste ;
- itérer sur un arbre (dans l'ordre préfixe, infixe, ou postfixe) ;
- itérer sur une table de hachage, etc.

Le producteur s'écrit facilement, car :

- la récursivité facilite le parcours ;
- l'appel à `consume` suspend puis reprend l'exécution du producteur.

... mais le consommateur pas toujours

Le consommateur offre un service au producteur : traiter, à la demande, un élément. Il est subordonné au producteur.

Cette approche n'est pas viable lorsqu'un consommateur souhaite obtenir des éléments alternativement en provenance de plusieurs producteurs.

Par exemple, il est impossible (à moins d'allouer une liste) de :

- comparer deux séquences issues de producteurs distincts (cf. TD6) ;
- fusionner deux séquences issues de producteurs distincts.

Contrôle au consommateur

Étudions l'approche duale, où le producteur offre un service au consommateur : produire, à la demande, un élément.

Le producteur est donc un objet (ou une fonction) que l'on interroge pour obtenir le prochain élément de la séquence, s'il en reste.

On appelle cela un itérateur. (Poly, §4.2. Liskov et Guttag, chapitre 6.)

L'interface des itérateurs étant standardisée, on pourra (ici aussi) facilement remplacer un producteur par un autre.

Itérateur modifiable ou immuable ?

Si l'on appelle deux fois l'itérateur, doit-il :

- produire et renvoyer deux éléments successifs de la séquence,
- ou bien renvoyer à chaque appel le même élément ?

Dans le premier cas, l'itérateur est modifiable donc éphémère.

Dans le second cas, il est immuable donc persistant.

Interface

En OCaml, le type d'un itérateur modifiable pourrait être tout simplement :

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

Exercice : construire un `int iterator` qui produit la suite infinie $0, 1, \ldots$
(Solution.)

Interface

En Java, le type d'un itérateur modifiable est défini dans la bibliothèque :

```
public interface Iterator<E> {
  boolean hasNext ();
  E next () throws NoSuchElementException;
}
```

`hasNext()` indique s'il reste un élément. Elle ne modifie pas l'état.

`next()` produit le prochain élément, s'il existe. Elle modifie l'état.

Création de nouveaux itérateurs

Un itérateur modifiable est éphémère. Il ne sert qu'une fois.

Il faut donc pouvoir en créer de nouveaux à la demande :

```
public interface Iterable<E> {
  Iterator<E> iterator ();
}
```

Si une structure de données implémente `Iterable<E>`, alors on peut itérer sur elle autant de fois qu'on le souhaite.

Utilisation d'un « itérable »

Supposons que `c` soit une structure de type `Iterable<E>`.

```
Iterator<E> it = c.iterator(); // create a new iterator
while (it.hasNext()) {
  E el = it.next();            // obtain the next element
  ...                          // use it
}
```

Cet idiome est tellement fréquent qu'un sucre a été introduit en Java 1.5.

```
for (E el : c) { ... }
```

Utilisation d'un « itérable »

La plupart des « collections » de la bibliothèque de Java sont « itérables ».

Par exemple, la classe `TreeSet<E>` implémente l'interface `Iterable<E>`.

C'est pratique :

```
TreeSet<Integer> s = ...;
for (int i : s) { ... }
```

Définition d'un « itérable »

On souhaite définir un `Iterable<Integer>` qui représente la suite $[i, j[$.

```
public static Iterable<Integer> interval (int i, int j) {
  return () ->
    new Iterator<Integer> () {
      private int next = i; // our mutable internal state
      public boolean hasNext () { return next < j; }
      public Integer next () {
        if (hasNext()) return next++;
        throw new NoSuchElementException ();
      }
    };
}
```

Cela s'écrit plutôt bien, grâce aux clôtures et classes anonymes.

L'itérateur doit explicitement mémoriser et maintenir à jour son état.

Le problème des modifications concurrentes

Supposons qu'une collection modifiable implémente `Iterable<E>`.

Que faire si la collection est modifiée pendant qu'un itérateur est actif ?

```
List<String> cities = new ArrayList<> ();
cities.add("Paris");
cities.add("Roma");
for (String city : cities) {
  System.out.println(city);
  cities.add("Palaiseau");
}
```

Qu'attend-on de ce code ?

Le problème des modifications concurrentes

D'après la documentation de la classe `Iterator`, le comportement de ce code est non spécifié.

> *The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress [...]*

En pratique, l'itérateur lance une `ConcurrentModificationException` s'il détecte que la collection a été modifiée après la création de l'itérateur.

On utilise pour cela des « numéros de version ».

Interface

Un itérateur immuable doit produire à la demande :

- non seulement un élément,
- mais aussi un nouvel itérateur qui représente le reste des éléments.

On pourrait modéliser cela en OCaml à l'aide d'une abréviation de types :

```
type 'a immutable_iterator =
  unit -> ('a * 'a immutable_iterator) option
```

Toutefois, OCaml refuse les abréviations récursives. (Poly, §4.3.)

Interface

On peut exprimer la même idée comme une définition de type récursive :

```
type 'a immutable_iterator =
  unit -> 'a head

and 'a head =
| Nil
| Cons of 'a * 'a immutable_iterator
```

Celle-ci est acceptée par OCaml.

Exercice : écrire un `int immutable_iterator` qui représente la suite $[i, j[$. (Solution.)

Analogie entre itérateur immuable et liste

En réalité, nous venons de découvrir une variante du type des listes !

Si nous remplaçons `'a immutable_iterator` par sa définition, à savoir `unit -> 'a head`, nous obtenons :

```
type 'a head =
| Nil
| Cons of 'a * (unit -> 'a head)
```

C'est une liste dont les éléments sont produits à la demande.

J'appelle cela une cascade.

Nous les étudierons plus en détail lors de la séance 7.

Flots

Les flots sont une autre variante des listes, dont les éléments sont produits à la demande et mémoisés.

```
type 'a head =
| Nil
| Cons of 'a * ('a head Lazy.t)
```

Nous les étudierons plus en détail lors de la séance 7. (Poly, §4.3.)

À quoi reconnaît-on un appel terminal ?

Intuitivement :

- un appel est terminal si l'appelant n'a plus rien à faire après l'appel.

On peut dire dans ce cas que l'appelant délègue à l'appelé la responsabilité de terminer le calcul et renvoyer un résultat.

Exemples

L'appel de `gcd` à elle-même est terminal :

```
let rec gcd x y =
  if y = 0 then
    x
  else
    gcd y (x mod y)
```

Exemples

L'appel de `fact_accu` à elle-même est terminal :

```
let rec fact_accu accu n =
  if n = 0 then accu
  else fact_accu (n * accu) (n - 1)
```

L'appel de `fact` à elle-même n'est pas terminal :

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)
```

À quoi reconnaît-on un appel terminal ?

On distingue expressions e et expressions en position terminale ept :

$$
\begin{array}{rcll}
e & ::= & & \\
 & | & \mathit{fun}\ x \to \mathit{ept} & \\
 & | & \mathit{let\ (rec)}\ x = e\ \mathit{in}\ e & \\
 & | & \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e & \\
 & | & e(e) & \text{appel non terminal} \\
\mathit{ept} & ::= & & \\
 & | & \mathit{fun}\ x \to \mathit{ept} & \\
 & | & \mathit{let\ (rec)}\ x = e\ \mathit{in}\ \mathit{ept} & \\
 & | & \mathit{if}\ e\ \mathit{then}\ \mathit{ept}\ \mathit{else}\ \mathit{ept} & \\
 & | & e(e) & \text{appel terminal}
\end{array}
$$

En analysant le programme à l'aide de cette grammaire, on voit quelles expressions sont en position terminale, donc quels appels sont terminaux.

À quoi est-ce utile ?

En OCaml, lors d'un appel terminal,

- l'exécution de l'appelant se termine
- (et ses variables locales disparaissent de la pile)
- avant que le contrôle soit transféré à l'appelé.

La règle générale donnée au début du cours (séance 1) reste valide :

> *La taille de la pile est proportionnelle au nombre de fonctions en cours d'exécution.*

Les appels terminaux n'ont donc aucun coût en espace sur la pile.

Exemples

Un appel terminal ressemble à un « goto », car l'appelant ne reprend jamais le contrôle.

Une fonction récursive terminale est compilée comme une boucle :

```
static int gcd (int x, int y) {
  while (true) {
    if (y == 0)
      return x;
    else {
      int tmp = x; x = y; y = tmp % y;
    }
  }
}
```

Sa complexité en espace est $O(1)$.

Exemples

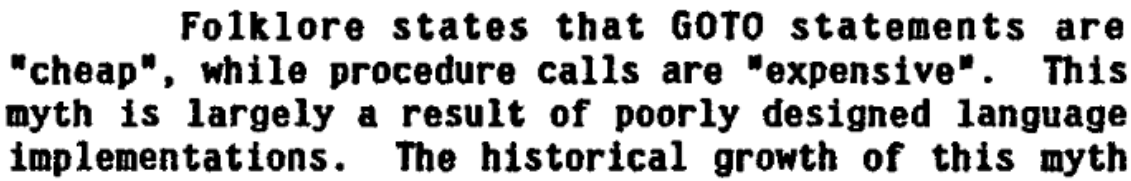
Ci-dessous, l'appel à `consume` n'est pas terminal, tandis que l'appel à `iter` est terminal. La complexité en espace est $O(1)$.

```
let rec iter consume xs =
  match xs with
  | []      -> ()
  | x :: xs -> consume x; iter consume xs
```

Ci-dessous, c'est l'inverse. La complexité en espace est $O(n)$.

```
let rec iter consume xs =
  match xs with
  | []      -> ()
  | x :: xs -> iter consume xs; consume x
```

Historique



```
        Folklore states that GOTO statements are
"cheap", while procedure calls are "expensive".  This
myth is largely a result of poorly designed language
implementations.  The historical growth of this myth
```

La notion d'appel terminal remonte à Steele (1977), qui l'a implémentée dans Scheme.

Steele souligne que l'on comprend mieux un code exprimé en style récursif.

Scheme, OCaml, Haskell, Scala, C# garantissent que les appels terminaux n'ont pas de coût en espace, donc favorisent ce style.

C et Java n'offrent pas cette garantie, ce qui impose le style itératif.

Interlude

Reconstruction de l'algorithme de Knuth-Morris-Pratt.
(Code en ligne) (Explication)
(Pas dans le poly.)

À retenir

À propos de l'itération :

- séparer producteur et consommateur est souhaitable,
- mais il existe plusieurs façons de le faire ;
- la question « qui contrôle ? » est centrale.

À propos des appels terminaux :

- ils permettent de raisonner en termes de fonctions,
- tout en offrant l'efficacité des « goto ».

Rappels

- TD aujourd'hui : itérateurs immuables sur des arbres binaires.
- Date limite pour rendre votre projet : **31 mai 2016** à 23h59.
- Contrôle classant le **mardi 7 juin 2016**.

Solution aux exercices

`from 0` crée un itérateur modifiable qui produit la suite infinie $0, 1, \ldots$

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

let from (i : int) : int iterator =
  let next = ref i in
  fun () ->
    let n = !next in
    next := n + 1;
    Some n
```

Solution aux exercices

`interval i j` crée un itérateur immuable qui représente la suite $[i, j[$.

```
type 'a immutable_iterator =
  unit -> 'a head

and 'a head =
| Nil
| Cons of 'a * 'a immutable_iterator

let rec interval i j : int immutable_iterator =
  fun () ->
    if i < j then
      Cons (i, interval (i + 1) j)
    else
      Nil
```