Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

```
'e liste -> 'e -> bool
```

testant la présence de la valeur du 2e paramètre dans le 1er.

```
let rec appartient_liste lst x =
    match lst with
        |Vide -> false
        |Cellule (y, _) when y = x -> true
        |Cellule (_, reste) -> appartient_liste reste x
```

Ceci se base sur le fait que test d'égalité `=` est **polymorphe** (de type `'a -> 'a -> 'a` ; il n'est donc pas nécessaire de fournir en paramètre à la fonction une fonction de test d'égalité).

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant l'**opération « max »** d'une relation d'ordre totale ;
- le 2e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

```
let maximum_liste f_max lst =
    let rec aux lst max_prefixe =
        match lst with
            |Vide -> max_prefixe
            |Cellule (x, reste) -> aux reste (f_max max_prefixe x)
    in
    match lst with
        |Vide -> failwith "liste vide"
        |Cellule (x, reste) -> aux reste x
```

Plan

6. Notions

6.1 Récursivité
6.2 Filtrage
6.3 Fonctions d'ordre supérieur
6.4 Polymorphisme
6.5 Stratégies d'évaluation

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
# Fonction intermédiaire.
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
# Fonction intermédiaire.
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
# Point d'entrée de l'exécution.
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une **non-terminaison** ;
2. sinon, l'expression `f(-1)` n'est pas évaluée car le second argument, `0`, de l'appel à `g` est pair. L'exécution **termine** dans ce cas.

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à évaluer d'abord `a1`, …, `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

– Exemple –

Si l'on a une fonction

```
let f x y z = x + z
```

l'expression

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)
```

s'évalue au moyen des étapes suivantes :

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)  ⇝  f 1 (f 2 1 4) 12  ⇝  f 1 6 12  ⇝  13
```

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;

2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Beaucoup de langages utilisent cette stratégie, dont le CAML.

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

– Exemple –

Si l'on a une fonction

```
let f x y z = x + z
```

l'expression

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)
```

s'évalue au moyen des étapes suivantes :

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)  ⇝  (1 * 1) + (3 * 4)  ⇝  13
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

– Exemple –

En effet, si l'on a une fonction

```
let f x y = x * x + y
```

l'expression

```
f (4 * 3) (2 * 1)
```

s'évalue en appel par nom au moyen des étapes suivantes :

$$\texttt{f (4 * 3) (2 * 1)} \rightsquigarrow \texttt{(4 * 3) * (4 * 3) + (2 * 1)} \rightsquigarrow \texttt{146}$$

L'argument `4 * 3` est **évalué** ainsi **deux fois** (au lieu d'une seule que ferait un appel par valeur).

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

Une fonction est mémoïsée si, à chaque premier appel pour un jeu d'arguments donnés, la valeur qu'elle renvoie est enregistrée dans une table associant les arguments au résultat. Ainsi, tout second appel à la fonction avec le même jeu d'arguments ne provoque pas de réévaluation.

Lors de l'application d'une fonction `f` à des expressions `a1`, …, `an`, chaque `ai` n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

De plus, l'évaluation d'un `ai`, si elle a lieu, est enregistrée. Ainsi, toute occurrence d'un paramètre de `f` correspondant à un `ai` ne redemande pas d'être réévaluée.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le **principe de transparence référentielle** est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Le langage Haskell utilise cette stratégie.

7. Listes

Plan

7. Listes

7.1 Opérations
7.2 Non-mutabilité
7.3 Files

Les listes

Le langage CAML offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes (sans avoir à les redéfinir).

Une valeur de type `'a list` est une suite finie de valeurs de type `'a`.

Les listes sont notées avec des crochets et des points-virgules :

```
[e1 ; e2 ; ... ; en]
```

est une liste contenant, de gauche à droite, les éléments `e1`, `e2`, …, `en` tous d'un même type.

La liste vide est notée `[]`.

Le type `'a list` est un **type polymorphe** et `[]` est une **valeur polymorphe**.

– Exemples –

```
# [2 ; 4 ; 8 ; 16];;    - : int list = [2; 4; 8; 16]
# [];;    - : 'a list = []
```

Opérateur de construction

L'opérateur de construction `::` est un opérateur infixe d'arité deux.

Il peut être pensé comme étant de type

```
'a -> 'a list -> 'a list
```

Il agit de la manière suivante : si `e` est un élément et `lst` est une liste, l'expression `e :: lst` a pour valeur la liste qui contient `e` comme 1er élément et ceux de `lst` ensuite.

– Exemples –

```
# 2 :: [1 ; 2 ; 3];;
- : int list = [2; 1; 2; 3]
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

Il est **associatif de droite à gauche**.

– Exemple –

L'expression `1 :: 2 :: 3 :: []` désigne l'expression totalement parenthésée `(1 :: (2 :: (3 :: [])))`.

Déconstruction

L'opérateur de construction est également un opérateur de déconstruction lorsqu'on s'en sert avec un filtrage de motifs.

– Exemple –

```
let tete lst =
    match lst with
        |[] -> failwith "liste vide"
        |e :: _ -> e
```

déconstruit la liste en argument pour accéder à son 1er élément.

– Exemple –

```
let rec un_sur_deux lst =
    match lst with
        |[] -> []
        |[e] -> [e]
        |e1 :: e2 :: reste -> e1 :: (un_sur_deux reste)
```

renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Note : ce sont des **fonctions polymorphes**.

Fonctions élémentaires

La bibliothèque standard de CAML (`Stdlib`) propose à travers le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir
`append`	`'a list -> 'a list -> 'a list`	Concaténation des deux listes

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombres d'éléments dans les listes impliquées.

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété ;
4. **combiner** les éléments d'une liste pour calculer une valeur ;
5. **permuter** les éléments d'une liste.

Opérations habituelles sur les listes

– Exemples –

1. **[transformation]** multiplier les éléments d'une liste d'entiers par `3`, convertir une liste de caractères en la liste des codes ASCII correspondants ;
2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;
3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément `7` dans une liste d'entiers ;
4. **[combinaison]** calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;
5. **[permutation]** tri d'une liste, image miroir d'une liste.

Tout ceci se fait à l'aide de **fonctions d'ordre supérieur**.

Transformation de listes

Une bonne manière de spécifier la manière de **transformer** les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste)
```

– Exemples –

```
# transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6];;
- : int list = [3; 6; 9; 12; 15; 18]
# transformer int_of_char ['a' ; 'b' ; 'c' ; '1' ; '2' ; '3'];;
- : int list = [97; 98; 99; 49; 50; 51]
```

Nous avons réimplanté ici la fonction `map` du module `List`.

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

et sa définition est

```
let rec selectionner sel lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = selectionner sel reste in
            if sel e then e :: suite else suite
```

– Exemple –

```
# selectionner (fun x -> x mod 2 = 0) [13 ; 8 ; 9 ; 8 ; 6 ; 15 ; 2];;
- : int list = [8; 8; 6; 2]
```

Nous avons réimplanté ici la fonction `filter` du module `List`.

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ prop lst =
    match lst with
        |[] -> true
        |x :: _ when not (prop x) -> false
        |_ :: reste -> tester_univ prop reste
```

– Exemple –

```
# tester_univ (fun u -> u.[0] = 'a') ["a" ; "aba" ; "abacaba" ; "abacabadabacaba"];;
- : bool = true
```

Nous avons réimplanté ici la fonction `for_all` du module `List`.

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_exist prop lst =
    match lst with
        |[] -> false
        |x :: _ when prop x -> true
        |_ :: reste -> tester_exist prop reste
```

– Exemple –

```
# tester_exist (fun x -> x = 7) [0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8];;
- : bool = false
```

Nous avons réimplanté ici la fonction `exists` du module `List`.

Association des éléments d'une liste

Le problème est le suivant : on dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur `gr` $\times$ `e1` $\times$ `e2` $\times$ `...` $\times$ `en`.

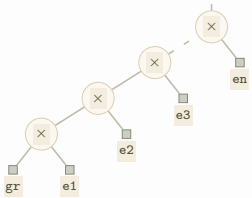
– Exemples –

Ce problème admet de nombreuses instances :

- lorsque `gr = 0`, `lst` est une liste d'entiers et $\times$ est la fonction d'**addition** des entiers, ceci calcule la **somme** des éléments de `lst` ;
- lorsque `gr = ""`, `lst` est une liste de chaînes de caractères et $\times$ est l'opération de **concaténation**, ceci calcule de gauche à droite la **concaténation** des chaînes de caractères de `lst` ;
- lorsque `gr = e1`, `lst` est une liste dont les éléments sont comparables et $\times$ est la fonction qui renvoie **le plus grand** de ses deux arguments, ceci calcule la **plus grande valeur** de `lst`.

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération `×`, on souhaite **évaluer** l'arbre syntaxique



`gr` est l'élément initial pour le calcul. On l'appelle graine.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération `×` ;
2. la graine `gr` ;
3. la liste `lst = [e1 ; e2 ; e3 ; ... ; en]` des opérandes.

Pliage à gauche

L'opération `×` est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> pliage_gauche op (op gr e) reste
```

– Exemples –

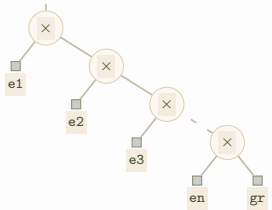
```
# pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3];;   - : int = 11

# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2] in pliage_gauche max (List.hd lst) (List.tl lst);;   - : int = 5

# pliage_gauche (^) "" ["mi" ; "la" ; "re" ; "sol" ; "do" ; "fa"];;   - : string = "milaresoldofa"
```

Pliage à droite

Il est possible de faire la même chose mais en considérant plutôt l'arbre syntaxique peigne droit (à la place du gauche) :



L'opération de pliage à droite est ainsi de type

```
('a -> 'a -> 'a) -> 'a list -> 'a -> 'a
```

et sa définition est

```
let rec pliage_droite op lst gr =
    match lst with
        |[] -> gr
        |e :: reste -> op e (pliage_droite op reste gr)
```

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

et

```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

D'un point de vue sémantique, lorsque l'opération `×` est **associative** et que la graine `gr` **commute** avec tous les éléments, les deux pliages donnent le même résultat.

Il y a une différence d'efficacité : le pliage à gauche est **récursif terminal** alors que le pliage à droite ne l'est pas. En effet, dans le pliage à gauche, la graine est l'accumulateur.

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =
    let rec aux lst acc =
        match lst with
            |[] -> acc
            |e :: reste -> aux reste (e :: acc)
    in
aux lst []
```

– Exemples –

```
# miroir ['a' ; 'b' ; 'c'];;
- : char list = ['c'; 'b'; 'a']
# miroir [1 ; 1 ; 2 ; 2 ; 2 ; 1];;
- : int list = [1; 2; 2; 2; 1; 1]
```

Nous avons réimplanté ici la fonction `rev` du module `List`.

Fonctions avancées

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

Fonction	Type
`map`	`('a -> 'b) -> 'a list -> 'b list`
`filter`	`('a -> bool) -> 'a list -> 'a list`
`for_all`	`('a -> bool) -> 'a list -> bool`
`exists`	`('a -> bool) -> 'a list -> bool`
`fold_left`	`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_right`	`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

En pratique, on utilise ces fonctions sans les réimplanter.

Il existe aussi la fonction `sort` de type

```
('a -> 'a -> int) -> 'a list -> 'a list
```

qui permet de renvoyer une version triée d'une liste d'éléments de type `'a` au moyen d'une fonction de comparaison (1er paramètre).

Opérateur d'application inversée

L'opérateur d'application inversée est l'opérateur binaire `|>`. Il est de type

`'a -> ('a -> 'b) -> 'b`.

Il permet, étant données une fonction `f` de type `'a -> 'b` et une expression `e` de type `'a`, d'écrire

`e |> f`

à la place de

`f e`.

Il sert à rendre les **applications successives de fonctions plus naturelles** : si `e` est une expression de type `'a0` et `f1` est de type `'a0 -> 'a1`, `f2` est de type `'a1 -> 'a2`, …, et `fn` est de type `'a`$_{n-1}$ `-> 'an`, à écrire

`e |> f1 |> f2 |> ... |> fn`

au lieu de

`fn (... (f2 (f1 e)) ...)`.

Application inversée et traitement des listes

– Exemples –

- Si `lst` est une liste de couples,

  ```
  lst |> List.map (fun (x, y) -> (y, x))
  ```

  est la liste obtenue en transposant les couples de `lst`.

- Si `lst` est une liste d'entiers,

  ```
  lst |> List.map (fun x -> x * (-2)) |> List.fold_left (+) 0
  ```

  est la somme des entiers de `lst` multipliés au préalable par `-2`.

- Si `lst` est une liste de listes,

  ```
  lst |> List.map List.length |> List.exists (fun x -> x mod 2 = 0)
  ```

  teste s'il existe une liste élément de `lst` qui est de longueur paire.

- Si `lst` est une liste de chaînes de caractères,

  ```
  lst |> List.filter (fun u -> u >= "ab") |> List.map String.lowercase_ascii |> List.fold_left (^) ""
  ```

  est la concaténation des chaînes de caractères de `lst` supérieures à `"ab"` puis converties en minuscules.

Plan

7. Listes

7.1 Opérations
7.2 Non-mutabilité
7.3 Files

Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet `x`, pour obtenir un objet `x'` calculé à partir de `x`, il faut **reconstruire** `x'`.

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée ;
2. démonstration de correction de programmes facilitée ;
3. gain de place mémoire.

Ces trois avantages s'appuient sur le fait que plusieurs grosses données peuvent **partager** des sous-données en commun, **sans aucune interférence**.

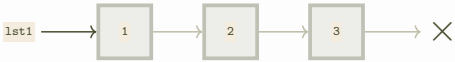
Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par la **construction d'une liste résultat**.

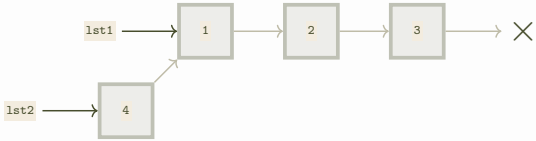
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;
# let lst2 = 4 :: lst1;;
```

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et **partage** les trois précédentes :



Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2)
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons les phrases

```
# let lst1 = [1 ; 2];;
# let lst2 = [3 ; 1];;
```

```
# let lst3 = 1 :: lst1;;
# let lst4 = concatener lst1 lst2;;
# let lst5 = concatener lst3 lst4;;
```

Nous obtenons en mémoire la configuration de partage suivante :