

Objets polymorphes

Un objet est dit **polymorphe** s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;  
val vrai : 'a -> bool = <fun>
```

- Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;  
type 'a liste = Vide | Cellule of 'a * 'a liste
```

- Une **valeur polymorphe** est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé. P.ex.,

```
# Vide;;  
- : 'a liste = Vide
```

Polymorphisme paramétrique

En Caml, le polymorphisme est **paramétrique** : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type t bien défini, soit de tous les types possibles $'a$ (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre x d'une fonction, le système de typage fonctionne ainsi :

- 1 il recherche les occurrences de x dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;
- 2 si cette étape échoue (ou bien s'il n'y a aucune occurrence de x), alors x est du type le plus général $'a$.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

- | | |
|--|---|
| <code>(=) : 'a -> 'a -> bool</code> | Test d'égalité entre deux valeurs d'un même type. |
| <code>(<>) : 'a -> 'a -> bool</code> | Test de différence entre deux valeurs d'un même type. |
| <code>compare : 'a -> 'a -> int</code> | Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie <code>-1</code> (resp. <code>1</code>) si la 1 ^{re} est strict. inf. (resp. sup.) à la 2 ^e et <code>0</code> sinon. |
| <code>fst : 'a * 'b -> 'a</code> | Renvoie la 1 ^{re} coordonnée d'un couple dont les coordonnées sont de types possiblement différents. |
| <code>snd : 'a * 'b -> 'b</code> | Renvoie la 2 ^e coordonnée d'un couple dont les coordonnées sont de types possiblement différents. |

Exemple : exponentiation rapide

Le calcul de x^n , où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k + 1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
  if n = 0 then
    1
  else
    let tmp = (puiss x (n / 2)) in
    if n mod 2 = 0 then
      tmp * tmp
    else
      tmp * tmp * x;;
```

Il faut bien observer en l. 5 la liaison locale de `tmp` pour faire un seul appel récursif au lieu de deux.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération \times associative et un élément particulier **1**, unité pour l'opération \times .

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

$$('e \rightarrow 'e \rightarrow 'e) \rightarrow 'e \rightarrow 'e \rightarrow \text{int} \rightarrow 'e$$

où

- le 1^{er} paramètre de type $('e \rightarrow 'e \rightarrow 'e)$ est une fonction codant \times ;
- le 2^e paramètre de type $'e$ est l'unité **1** ;
- le 3^e paramètre de type $'e$ est l'élément **x** ;
- le 4^e paramètre de type `int` est l'entier **n**.

Le type de retour est $'e$.

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
  if n = 0 then
    unite
  else
    let tmp = (puiss_poly op unite x (n / 2)) in
    if n mod 2 = 0 then
      (op tmp tmp)
    else
      (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
# (puiss_poly (fun a b -> a * b) 1 2 10);;
- : int = 1024
# (puiss_poly (fun u v -> u ^ v) "" "abb" 4);;
- : string = "abbabbabbabb"
# (puiss_poly (fun a b -> a || b) false false 293898273);;
- : bool = false
```

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 2^e paramètre dans le 1^{er}.

```
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 à fournir en paramètre à la fonction).

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

$$('e \rightarrow 'e \rightarrow 'e) \rightarrow 'e \text{ liste} \rightarrow 'e$$

où

- le 1^{er} paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2^e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

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

Exemple introductif

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

Fonction rec f(x) :	Fonction g(x, y) :	Début :
Si x = 0 :	Si y est pair :	g(f(-1), 0)
0	y	Fin
Sinon :	Sinon :	
x + f(x - 1)	x	
Fin	Fin	
Fin	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.

P.ex., 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 a_1, \dots, a_n , à **substituer** les a_i **sans les évaluer** aux occurrences des paramètres de f correspondants.

P.ex., 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**.

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 :

```
(f (4 * 3) (2 * 1))  
  ~> (4 * 3) * (4 * 3) + (2 * 1)  
  ~> 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**.

Lors de l'application d'une fonction f à des expressions a_1, \dots, a_n , chaque a_i n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

De plus, l'évaluation d'un a_i , si elle a lieu, est enregistrée. Ainsi, toute occurrence d'un paramètre de f correspondant à un a_i 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.

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`.

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

La **liste vide** est notée `[]`.

```
# [];;  
- : 'a list = []
```

C'est une valeur polymorphe.

Opérateur de construction

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

Il peut être pensé comme étant de type

$$'a \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}$$

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 1^{er} élément et ceux de `lst` ensuite.

P.ex.,

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

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

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

L'expression `1 :: 2 :: 3 :: []` désigne en effet 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.

P.ex.,

```
# let tete lst =  
    match lst with  
        | [] -> (failwith "liste vide")  
        | e :: _ -> e;;  
val tete : 'a list -> 'a = <fun>
```

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

```
# let rec un_sur_deux lst =  
    match lst with  
        | [] -> []  
        | [e] -> [e]  
        | e1 :: e2 :: reste -> e1 :: (un_sur_deux reste);;  
val un_sur_deux : 'a list -> 'a list = <fun>
```

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

Note : ce sont des fonctions polymorphes.

Fonctions standard

La librairie de Caml propose dans le module `List` diverses fonctions de manipulation des listes. Parmi elles :

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

Exercice : donner (en examinant leurs implantations) les complexités en temps et en mémoire de ces fonctions relativement au nombre 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 **reconstruire** une liste en une autre.

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 **[reconstruction]** 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 d'utilisation :

```
# (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

$$('a \rightarrow \text{bool}) \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}$$

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 d'utilisation :

```
# (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 d'utilisation :

```
# (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

$$('a \rightarrow \text{bool}) \rightarrow 'a \text{ list} \rightarrow \text{bool}$$

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 d'utilisation :

```
# (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'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
- 2 d'une opération binaire associative `×` ;

et on souhaite calculer la valeur

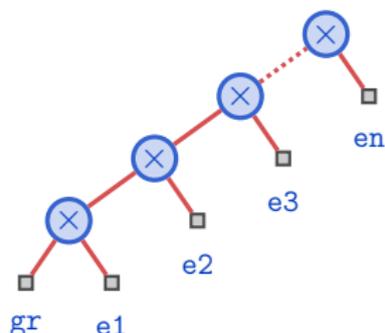
$$e1 \times e2 \times \dots \times en.$$

Ce problème admet de nombreuses instances :

- lorsque `lst` est une liste d'entiers et que `×` est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- lorsque `lst` est une liste de chaînes de caractères et `×` est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;
- lorsque `lst` est une liste dont les éléments sont comparables et `×` 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 $[e_1 ; e_2 ; e_3 ; \dots ; e_n]$ et une opération \times , on souhaite **évaluer** l'arbre syntaxique



Ici, gr est un 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 \times ;
- 2 la graine gr ;
- 3 la liste lst des opérandes.

Pliage à gauche

L'opération \times 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

$$('a \rightarrow 'a \rightarrow 'a) \rightarrow 'a \rightarrow 'a \text{ list} \rightarrow 'a$$

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 d'utilisation :

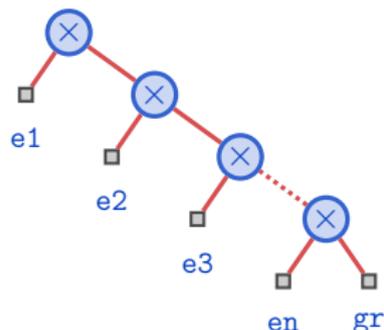
```
# (pliage_gauche (+) 0 [0; 2; 1; 5; 2; -2; 3]);;
- : int = 11

# let lst = [0; 2; 1; 5; 2; -2; 3] in
  (pliage_gauche max (List.hd lst) (List.tl lst));;
- : int = 5

# (pliage_gauche (^) "" ["re"; "la"; "re"; "sol"; "si"; "mi"]);;
- : string = "relaresolsimi"
```

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

$$('a \rightarrow 'a \rightarrow 'a) \rightarrow 'a \text{ list} \rightarrow 'a \rightarrow 'a$$

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 \times 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.