Algorithmique et Approche Fonctionnelle de la Programmation

Cours 3 : Récursivité et listes

21 septembre 2009

Louis Mandel
Université Paris-Sud 11
louis.mandel@lri.fr

D'après des transparents de Sylvain Conchon

Les fonctions récursives

Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle même

- il faut au moins un cas sans appel récursif si on veut que la fonction termine

```
# let rec fact n =
    if n = 0 then 1
    else n * fact (n - 1);;
val fact : int -> int = <fun>
```

En maths : $\begin{cases} u_0 & = & 1 \\ u_n & = & n * u_{n-1} \end{cases}$

Evaluation de `fact`

```
fact 3
⟶  if 3 = 0 then 1 else 3 * fact (3 - 1)
⟶  3 * fact (3 - 1)
⟶  3 * fact 2
⟶  3 * (if 2 = 0 then 1 else 2 * fact (2 - 1))
⟶  3 * (2 * fact (2 - 1))
⟶  3 * (2 * fact 1)
⟶  3 * (2 * (if 1 = 0 then 1 else 1 * fact (1 - 1)))
⟶  3 * (2 * (1 * fact (1 - 1)))
⟶  3 * (2 * (1 * fact 0))
⟶  3 * (2 * (1 * (if 0 = 0 then 1 else 0 * fact (0 - 1))))
⟶  3 * (2 * (1 * 1))
⟶  6
```

Avantages des définitions récursives

- L'écriture à l'aide d'une fonction récursive donne souvent un code plus lisible et plus susceptible d'être correct (car d'invariant plus simple) que son équivalent impératif utilisant une boucle

```
int fact(int n){
  int f = 1;
  int i = n;
  while (i>0){
    f = f * i;
    i--;
  };
  return f;
}
```

  - ▷ l'argument justifiant la correction de cette version est nettement plus complexe que la version récursive

Pourquoi le mot clé `rec` ?

- La règle de portée du `let`

```
# let f x = x + 1;;
val f : int -> int = <fun>
# let f y = if y = 0 then f 2 else 4 + y;;
val f : int -> int = <fun>
```

```
# let f x = x + 1;;
val f : int -> int = <fun>
# let rec f y = if y = 0 then f 2 else 4 + y;;
val f : int -> int = <fun>
```

Que se passe-t-il si `n` est négatif ?

- En mathématiques, la fonction factorielle n'est définie que sur les entiers naturels (positifs ou nuls)
- la fonction `fact` a pour type `int -> int` et ne termine pas si son paramètre est négatif
- Quelle valeur `fact` peut-elle rendre si `n < 0` ?
  - une solution raisonnable est de stopper le calcul en cours et d'expliquer le problème à l'utilisateur
  - on utilise pour cela la fonction prédéfinie `failwith m` qui termine l'exécution d'un programme en affichant le message `m` à l'écran

Implantation de `fact`

```
# let rec fact n =
    if n < 0 then failwith "argument negatif"
    else
      if n = 0 then 1 else n * fact (n - 1);;
val fact : int -> int = <fun>
```

On peut éviter les tests inutiles

```
# let rec fact_aux n =
    if n = 0 then 1 else n * fact_aux (n - 1);;
val fact_aux : int -> int = <fun>
# let fact n =
    if n < 0 then failwith "argument negatif"
    else fact_aux n;;
val fact : int -> int = <fun>
```

Implantation de `fact`

```
# let fact n =
    let rec fact_aux n =
      if n = 0 then 1 else n * fact_aux (n - 1)
    in
    if n < 0 then failwith "argument negatif"
    else fact_aux n
  ;;
val fact : int -> int = <fun>
```

La fonction `fact_aux` peut être définie localement à la fonction `fact`

Principes d'une définition récursive

- Une fonction récursive se définit par cas
  - ▷ les cas de bases (non récursifs)
  - ▷ les cas récursifs qui renvoient à la définition en cours

- La définition d'une fonction récursive revient toujours à identifier ces différents cas puis vérifier
  - ▷ que tous les cas sont couverts dans la définition
  - ▷ que les cas récursifs finissent par renvoyer aux cas de base (afin de garantir la terminaison)

Les listes

Les listes

Une liste est une suite (ordonnée) de valeurs de même type

```
# let liste_vide = [];;
val liste_vide : 'a list = []
# let liste_a_3_elements = [ 1; 2; 3; ];;
val liste_a_3_elements : int list = [1; 2; 3]
# let liste_de_booleens = [ false; 1 = 2; ];;
val liste_de_booleens : bool list = [false; false]
# let liste_de_listes = [ [ 1; 3; ]; []; [ 2; 4; ] ];;
val liste_de_listes : int list list = [[1; 3]; []; [2; 4]]

# let probleme = [ true; "faux"; ];;
Error: This expression has type string but an expression was expected of type
         bool
```

Les listes : Cons

L'opérateur prédéfini `::`

- se lit « cons » (prononcer le s final)
- est un opérateur infixe
- prend en argument
  - un élément `x`
  - une liste `l`
- `x :: l` retourne la liste où `x` est suivi de `l`

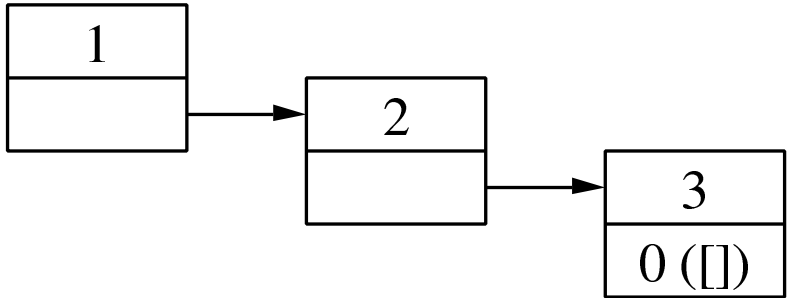
```
# let _ = 1 :: [ 2; 3; ];;
- : int list = [1; 2; 3]
# let _ = 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
# let _ = 'a' :: 'b' :: [] = [ 'a'; 'b' ];;
- : bool = true
```

Comparaison avec C

- Les listes prédéfinies en OCaml correspondent exactement aux listes chaînées définies habituellement en C par le type suivant :

```
typedef struct list{
  int elt;
  struct list* suivant;
} list;
```

- La représentation mémoire de ces listes correspond à un chaînage de blocs mémoire, par exemple, la liste `[1 ; 2 ; 3]` correspond à



Les listes : filtrage de motifs

expression ::= ...

| `match` *expression* `with` {`|` *pattern* `->` *expression*}$^{+}$

```
# let premier_element =
    match liste_a_3_elements with
    | [] -> 0
    | x :: xs -> x;;
val premier_element : int = 1
```

Les listes : filtrage de motifs

```
# let deuxieme_element =
    match liste_a_3_elements with
    | [] -> 0
    | x :: xs ->
        begin match xs with
        | [] -> 0
        | y :: ys -> y
        end;;
val deuxieme_element : int = 2
```

```
# let deuxieme_element =
    match liste_a_3_elements with
    | [] -> 0
    | x :: [] -> 0
    | x :: y :: ys -> y
  ;;
val deuxieme_element : int = 2
```

Fonction sur les listes

```
# let zero_en_tete l =
    match l with
    | [] -> false
    | x :: l' -> 0 = x
  ;;
val zero_en_tete : int list -> bool = <fun>
```

- Les fonctions sur les listes prennent généralement la forme d'une définition à deux cas
  - le cas où la liste est vide
  - le cas où la liste est composée d'un premier élément et du reste de la liste

Fonction sur les listes

```
# let zero_en_tete l =
    match l with
    | [] -> false
    | 0 :: l' -> true
    | x :: l' -> false
  ;;
val zero_en_tete : int list -> bool = <fun>
```

```
# let zero_en_tete l =
    match l with
    | 0 :: _ -> true
    | _ -> false
  ;;
val zero_en_tete : int list -> bool = <fun>
```

Fonction sur les listes : polymorphisme

```
# let tete l =
    match l with
    | [] -> failwith "tete: liste vide"
    | x :: l' -> x
  ;;
val tete : 'a list -> 'a = <fun>
```

Quel est le type de la fonction `tete` ?

- `tete` doit pouvoir s'appliquer à les listes d'entiers, comme par exemple

```
# let _ = tete [ 1; 2; 4; 8; 3; ];;
- : int = 1
```

- mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type

```
# let _ = tete [ [ 4.5; 0.3; 9.8 ]; []; [ 1.5e12; ]  ];;
- : float list = [4.5; 0.3; 9.8]
```

Fonction sur les listes : polymorphisme

- ... la fonction `tete` devrait avoir les deux types suivants :

```
val tete : int list -> int
val tete : (float list) list -> float list
```

- la fonction `tete` a une infinité de types
- le type inféré par OCaml est le plus général

```
val tete : 'a list -> 'a
```

  - `'a` (qui se lit « apostrophe a » ou encore « `alpha` ») est une variable de type
  - une variable de type veux dire n'importe quel type
- il faut donc lire le type de la fonction `tete` comme suit :
  - La fonction `tete` prend en argument une liste dont les éléments sont de n'importe quel type, que l'on notera `'a`. Elle retourne une valeur de ce même type `'a`.

Les listes : polymorphisme

La liste vide peut être considérée comme une liste d'entiers, de flottants, ...

```
# let _ = 1 :: [];;
- : int list = [1]
# let _ = 1.5e-10 :: [];;
- : float list = [1.5e-10]
# let _ = 1 :: [ 1.5e-10 ];;
Error: This expression has type float but an expression was expected of type
         int
# let _ = [];;
- : 'a list = []
```

Fonction sur les listes

```
# let a_2_elements l =
    match l with
    | [] -> false
    | _ :: l' ->
      begin match l with
      | [] -> false
      | x :: [] -> true
      | _ -> false
      end;;
val a_2_elements : 'a list -> bool = <fun>
```

```
# let a_2_elements l =
    match l with
    | [ x1; x2 ] -> true
    | _ -> false;;
val a_2_elements : 'a list -> bool = <fun>
```

fonctions récursives

La fonction `zeros`

La fonction `zeros` teste que tous les éléments d'une liste d'entiers sont des `0` (renvoie `true` si la liste est vide)

```
# let rec zeros l =
    match l with
    | [] -> true
    | x :: l' ->
      if x = 0 then zeros l'
      else false
  ;;
val zeros : int list -> bool = <fun>
# let _ = zeros [ 0; 0; 0; ];;
- : bool = true
# let _ = zeros [ 0; 1; 0; ];;
- : bool = false
```

La fonction `zeros` : evaluation

```
zeros (0 :: 0 :: 0 :: [])
```

$\longrightarrow$

```
  match [ 0; 1; 0] with
  | [] -> true
  | 0 :: (1 :: 0 :: []) ->
    if 0 = 0 then zeros (1 :: 0 :: [])
    else false
```

$\longrightarrow$

```
  if 0 = 0 then zeros (1 :: 0 :: [])
  else false
```

$\longrightarrow$

```
  zeros (1 :: 0 :: [])
```

$\longrightarrow$ ...

n-ième élément d'une liste

```
# let rec n_ieme n l =
    if n = 1 then List.hd l
    else n_ieme (n - 1) (List.tl l)
  ;;
val n_ieme : int -> 'a list -> 'a = <fun>
# let _ = n_ieme 3 [ 'a'; 'b'; 'c'; 'd'; 'e'; ];;
- : char = 'c'
# let _ = n_ieme 50 [ 'a'; 'b' ];;
Exception: Failure "tl".
```

Les fonctions `List.hd l` et `List.tl l` retournent respectivement la tête et le reste de la liste `l`

Appliquer une fonction à chaque élément d'une liste

```
# let rec succ_list l =
    match l with
    | [] -> []
    | x :: l' -> (x + 1) :: succ_list l'
  ;;
val succ_list : int list -> int list = <fun>
```

```
# let rec abs_list l =
    match l with
    | [] -> []
    | x :: l' -> abs x :: succ_list l'
  ;;
val abs_list : int list -> int list = <fun>
```

Recherche d'un élément

```
# let rec recherche_v1 x l =
    match l with
    | [] -> false
    | y :: l' -> if x = y then true else recherche_v1 x l'
  ;;
val recherche_v1 : 'a -> 'a list -> bool = <fun>
```

```
# let rec recherche_v2 x l =
    match l with
    | [] -> false
    | y :: l' -> if recherche_v2 x l' then true else x = y
  ;;
val recherche_v2 : 'a -> 'a list -> bool = <fun>
```

- dérouler l'évaluation des deux expressions suivantes :
  - ▷ `recherche_v1 4 [ 3; 4; 2; 1 ]`
  - ▷ `recherche_v2 4 [ 3; 4; 2; 1 ]`

Appel terminal

- Dans une fonction `f`, un appel à une fonction `g` est terminal si le résultat de cet appel est le résultat de `f`
  - autrement dit, dans une fonction `f`, un appel à `g` est terminal si cet appel est la dernière chose à faire pour le calcul de `f`
- Une fonction est récursive terminale si ses appels récursifs sont tous terminaux

l'exécution d'une fonction récursive terminale est plus efficace

Pile d'appel

- Pour exécuter les appels de fonctions, quelque soit le langage et le compilateur, il faut une pile d'appel
  - dans cette pile, les valeurs des arguments et l'adresse de retour de l'appel sont enregistrés
  - la taille de la pile croît en fonction du nombre d'appels « en attente »
  - la pile déborde quand il y a trop d'appels en attente (message `Stack overflow`)
- Un appel de fonction terminal peut réutiliser la place de l'appelant dans la pile pour l'appelé
- Une fonction récursive terminale (bien compilée) est aussi efficace qu'une boucle `for`

Appel terminal

- Dans les exemples suivants, les appels à g sont terminaux

```
let f x = g x

let f x = if ... then g x else ...

let f x = if ... then ... else g x

let f x = let y = ... in g y

let f x =
  match x with
  | ...
  | p -> g x
  | ...
```

Longueur d'une liste

```
# let rec longueur l =
    match l with
    | [] -> 0
    | _ :: l' -> 1 + (longueur l')
  ;;
val longueur : 'a list -> int = <fun>
```

Longueur d'une liste

```
# let longueur l =
    let rec longueur_aux acc l =
      match l with
      | [] -> acc
      | _ :: l' -> longueur_aux (1 + acc) l'
    in
    longueur_aux 0 l
  ;;
val longueur : 'a list -> int = <fun>
```

- Cette fonction est prédéfinie en OCaml : `List.length`

Récursion efficace

- Programmer avec des accumulateurs
  - ▷ principe analogue à l'ajout de variables auxiliaires en programmation impérative
  - ▷ ajout de fonctions auxiliaires avec des paramètres supplémentaires, appelés accumulateurs
- Exercice : transformer cette fonction en une fonction récursive terminale

```
# let rec somme l =
    match l with
    | [] -> 0
    | x :: l' -> x + somme l'
  ;;
val somme : int list -> int = <fun>
```

Concaténation de listes

```
# let rec append l1 l2 =
    match l1 with
    | [] -> l2
    | x :: l1' -> x :: (append l1' l2)
  ;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

- Exemple : `append [ 1; 2; ] [ 3; 4; 5; ] = [ 1; 2; 3; 4; 5; ]`
- Cette fonction est prédéfinie en OCaml : `List.append`
  - on peut aussi utiliser la notation : `l1 @ l2`

Concaténation de listes

```
# let rec append' l1 l2 =
    match l1 with
    | [] -> l2
    | x :: l1' -> append' l1' (x :: l2)
  ;;
val append' : 'a list -> 'a list -> 'a list = <fun>
```

- Exemple : `append' [ 1; 2; ] [ 3; 4; 5; ] = ?`
- Cette fonction est prédéfinie en OCaml : `List.rev_append`

Récursion sur deux listes

```
# let rec meme_longueur l1 l2 =
    match l1, l2 with
    | [], [] -> true
    | [], _ | _, [] -> false
    | _::l1', _::l2' -> meme_longueur l1' l2'
  ;;
val meme_longueur : 'a list -> 'b list -> bool = <fun>
```

Définitions simultanées

Définitions simultanées

```
# let _ =
    let a = 1 in
    let b = "hello" in
    let a = b in
    let b = a in
    (a, b);;
- : string * string = ("hello", "hello")
```

```
# let _ =
    let a = 1 in
    let b = "hello" in
    let a = b and b = a in
    (a, b);;
- : string * int = ("hello", 1)
```

Fonctions mutuellement récursives

```
# let rec pair n =
    if n = 0 then true
    else impair (n - 1)
  and impair n =
    if n = 0 then false
    else pair (n - 1)
  ;;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
```

Grâce à la construction `let/and`, les fonctions `pair` et `impair` sont chacune définie en fonction de l'autre