Programmation Fonctionnelle Avancée

Séance 3 : Étude de cas - implémentation de `map`, modules, signatures

Alexandros Singh

Université Paris 8

15 octobre 2023

Problème : Somme d'une liste,

Rappelons notre implémentation naïve de la fonction `map` :

```
let rec map_naive f = function
    | [] -> []
    | h :: t -> (f h) :: map_naive f t
```

Que se passe-t-il lorsque nous exécutons le code suivant ?

```
# let f x = print_endline (string_of_int x);;
# map_naive f [1;2;3];;
```

Problème : Somme d'une liste,

Rappelons notre implémentation naïve de la fonction `map` :

```
let rec map_naive f = function
    | [] -> []
    | h :: t -> (f h) :: map_naive f t
```

Que se passe-t-il lorsque nous exécutons le code suivant ?

```
# let f x = print_endline (string_of_int x);;
# map_naive f [1;2;3];;
3
2
1
- : unit list = [(); (); ()]
```

Map

- La spécification du langage OCaml ne précise généralement pas l'ordre d'évaluation des sous-expressions !
- L'implémentation que nous utilisons utilise généralement une évaluation de droite à gauche !

```
# (* l'expression s'évalue à: *)
# f 1 :: map f [2;3];;
# f 1 :: (f 2 :: map f [3]);;
# f 1 :: (f 2 :: f 3 :: (map f []));;
# f 1 :: (f 2 :: f 3 :: []);;
# f 1 :: (f 2 :: () :: []);;
# f 1 :: () :: () [];;
# () :: () :: () :: [];;
```

Map

Comment forcer `naive-map` à évaluer l'application de `f` dès que possible ?

```
let rec map f = function
   | [] -> []
   | h :: t -> let r = f h in r :: map f t
```

C'est de cette manière que `map` est implémenté dans la Stdlib.

Map

Quelle est la complexité de `map` en termes d'appels à `f` ?

```
let rec map f = function
   | [] -> []
   | h :: t -> let r = f h in r :: map f t
```

Map

Quelle est la complexité de `map` en termes d'appels à `f` ?

```
let rec map f = function
   | [] -> []
   | h :: t -> let r = f h in r :: map f t
```

Réponse : pour une liste de taille n, n applications de `f` seront évaluées.

Map

Quelle est la complexité de `map` en termes d'appels à `f` ?

```
let rec map f = function
   | [] -> []
   | h :: t -> let r = f h in r :: map f t
```

Réponse : pour une liste de taille n, n applications de `f` seront évaluées.
On ne peut pas faire mieux ! Y a-t-il d'autres aspects sur lesquels nous pourrions faire des optimisations ?

Map

Quelle est la complexité de `map` en termes d'appels à `f` ?

```
let rec map f = function
   | [] -> []
   | h :: t -> let r = f h in r :: map f t
```

Réponse : pour une liste de taille n, n applications de `f` seront évaluées.
On ne peut pas faire mieux ! Y a-t-il d'autres aspects sur lesquels nous pourrions faire des optimisations ?
Réponse : nous pourrions rendre la fonction `map` récursive terminale !

Voici une première tentative :

```
let map_tr_append f =
    let rec map_tr_helper f acc = function
        | [] -> acc
        | h :: t -> map_tr_helper f (acc @ [f h]) t in
    map_tr_helper f []
```

Constatez-vous des problèmes ?

Voici une première tentative :

```
let map_tr_append f =
    let rec map_tr_helper f acc = function
        | [] -> acc
        | h :: t -> map_tr_helper f (acc @ [f h]) t in
    map_tr_helper f []
```

Constatez-vous des problèmes ? Rappelons la complexité de l'opération append : elle est linéaire par rapport à la taille de son premier argument !

Voici une première tentative :

```
let map_tr_append f =
    let rec map_tr_helper f acc = function
        | [] -> acc
        | h :: t -> map_tr_helper f (acc @ [f h]) t in
    map_tr_helper f []
```

Constatez-vous des problèmes ? Rappelons la complexité de l'opération append : elle est linéaire par rapport à la taille de son premier argument !
Par conséquent, cette implémentation effectuera $n \cdot O(n)$ nouvelles opérations dans le cadre de la construction de sa sortie !

Pour y remédier, nous pouvons utiliser `cons` qui est d'une complexité $O(1)$:

```
let map_tr_cons f =
    let rec map_tr_helper f acc = function
        | [] -> acc
        | h :: t -> map_tr_helper f (f h :: acc) t in
    map_tr_helper f []
```

Nous avons retrouvé la complexité temporelle initiale et la nouvelle fonction est récursive terminale ! Constatez-vous de nouveaux problèmes cette fois-ci ?

Pour y remédier, nous pouvons utiliser cons qui est d'une complexité $O(1)$:

```
let map_tr_cons f =
    let rec map_tr_helper f acc = function
        | [] -> acc
        | h :: t -> map_tr_helper f (f h :: acc) t in
    map_tr_helper f []
```

```
# map_tr_cons ((+) 1) [1;2;3;4;5];;
- : int list = [6; 5; 4; 3; 2]
# map_tr_cons (fun x -> x) [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
```

La sortie est inversée ! Écrivons une fonction pour la remettre dans le bon ordre :

```
let rec rev = function
    | [] -> []
    | h :: t -> (rev t) @ [h]
```

```
# map_tr_cons ((+) 1) [1;2;3;4;5] |> rev;;
- : int list = [2; 3; 4; 5; 6]
# map_tr_cons (fun x -> x) [1;2;3;4;5] |> rev;;
- : int list = [1; 2; 3; 4; 5]
```

Cela semble fonctionner ! Mais il y a encore un problème...

La sortie est inversée ! Écrivons une fonction pour la remettre dans le bon ordre :

```
let rec rev = function
    | [] -> []
    | h :: t -> (rev t) @ [h]
```

```
# map_tr_cons ((+) 1) [1;2;3;4;5] |> rev;;
- : int list = [2; 3; 4; 5; 6]
# map_tr_cons (fun x -> x) [1;2;3;4;5] |> rev;;
- : int list = [1; 2; 3; 4; 5]
```

Cela semble fonctionner ! Mais il y a encore un problème... `rev` utilise la fonction append, ce qui nous replonge dans le problème précédent !

La fonction récursive terminale suivante fait l'affaire et produit notre implémentation finale souhaitée de map (quasi-identique à celle de `stdlib`) :

```
let rev_tr l =
    let rec rev_helper l acc =
        match l with
        | [] ->  acc
        | h :: t -> rev_helper t (h :: acc) in
    rev_helper l []
```

Map

Encore une autre implémentation de map : via fold !

```
let map_via_fold f l =
    List.fold_right (fun x acc -> f x :: acc) l []
```

Map

Encore une autre implémentation de map : via fold !

```
let map_via_fold f l =
    List.fold_right (fun x acc -> f x :: acc) l []
```

- Nous constatons que pour un même algorithme (tel que map), nous pouvons avoir une variété d'implémentations, adaptées à différents cas d'utilisation.

- Nous constatons que pour un même algorithme (tel que map), nous pouvons avoir une variété d'implémentations, adaptées à différents cas d'utilisation.
- Le système de modules d'OCaml nous permet de regrouper des fonctionnalités, de gérer des espaces de noms et de fournir des abstractions qui cachent les détails et les spécificités des implémentations.

- Nous constatons que pour un même algorithme (tel que map), nous pouvons avoir une variété d'implémentations, adaptées à différents cas d'utilisation.
- Le système de modules d'OCaml nous permet de regrouper des fonctionnalités, de gérer des espaces de noms et de fournir des abstractions qui cachent les détails et les spécificités des implémentations.
- Il est construit à partir des notions suivantes :

- Nous constatons que pour un même algorithme (tel que map), nous pouvons avoir une variété d'implémentations, adaptées à différents cas d'utilisation.
- Le système de modules d'OCaml nous permet de regrouper des fonctionnalités, de gérer des espaces de noms et de fournir des abstractions qui cachent les détails et les spécificités des implémentations.
- Il est construit à partir des notions suivantes :
  - `struct`: Une structure est un ensemble de définitions connexes (telles que les définitions d'un type de données et les opérations associées sur ce type). Un nom est généralement attribué à la structure avec via `module`.

- Nous constatons que pour un même algorithme (tel que map), nous pouvons avoir une variété d'implémentations, adaptées à différents cas d'utilisation.
- Le système de modules d'OCaml nous permet de regrouper des fonctionnalités, de gérer des espaces de noms et de fournir des abstractions qui cachent les détails et les spécificités des implémentations.
- Il est construit à partir des notions suivantes :
  - `struct`: Une structure est un ensemble de définitions connexes (telles que les définitions d'un type de données et les opérations associées sur ce type). Un nom est généralement attribué à la structure avec via `module`.
  - `sig`: Les signatures sont des interfaces pour les structures. Une signature spécifie quels composants d'une structure sont accessibles de l'extérieur, et avec quel type. Elle permet également de cacher certains composants de la structure.

Exemple de structure

```
module Lifo = struct
    type 'a pile = 'a list;;
    (* pile vide *)
    let empty : 'a pile = []
    (* vérifier si la pile est vide *)
    let is_empty : 'a pile -> bool = function
        | [] -> true
        | _ -> false
    (* empiler *)
    let push (x : 'a) (s : 'a pile) = x :: s
    (* dépiler *)
    let pop : 'a pile -> 'a pile option = function
        | [] -> None
        | _ :: s -> Some s
    (* regarder l'élément au sommet de la pile *)
    let peek : 'a pile -> 'a option = function
        | [] -> None
        | x :: _ -> Some x
end
```

Signatures

Plus généralement, nous pouvons définir une signature pour ce que nous attendons des implémentations des structures lifo (par exemple en exigeant juste le type lui-même plus `empty`, `is_empty`, `push`, `pop`).
N'importe qui peut alors implémenter sa propre version dans un module et la rendre conforme à la signature. Par exemple :

```
module type LIFO =
    sig
        type 'a pile
        val empty : 'a pile
        val is_empty : 'a pile -> bool
        val push : 'a -> 'a pile -> 'a pile
        val pop : 'a pile -> 'a pile option
    end
module AbstractPile = (Lifo : LIFO);;
```

La signature cache la fonction peek :

```
# Lifo.(push 5 empty |> peek);;
- : int option = Some 5
# AbstractPile.(push 5 empty |> peek);;
Error: Unbound value peek
```