Programmation d'ordre supérieur et polymorphisme

CSI 3520

Amy Felty

University of Ottawa

L’abstraction

- *La paresse* peut être « utile » pour concevoir des programmes.
- N'écrivez jamais le même code deux fois.
  - Décomposez les parties communes en procédures réutilisables.
  - Mieux encore, utilisez des procédures de quelqu'un d’autre (bien testé, bien documenté et bien entretenu).
- Pourquoi est-ce une bonne idée?
  - Pourquoi ne pas simplement couper et coller des extraits de code (en utilisant l'éditeur) au lieu de créer de nouvelles fonctions?

L’abstraction

- *La paresse* peut être « utile » pour concevoir des programmes.
- N'écrivez jamais le même code deux fois.
  - Décomposez les parties communes en procédures réutilisables.
  - Mieux encore, utilisez des procédures de quelqu'un d’autre (bien testé, bien documenté et bien entretenu).
- Pourquoi est-ce une bonne idée?
  - Pourquoi ne pas simplement couper et coller des extraits de code (en utilisant l'éditeur) au lieu de créer de nouvelles fonctions?
  - Si l'on trouve et corrige un erreur dans une copie, il faut corriger toutes les copies.
  - Si l'on décide de changer de fonctionnalité, il faut rechercher tous les endroits où le code est utilisé.

Décomposition du code dans OCaml

Considérez les définitions suivantes:

```
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)
```

```
let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```

Décomposition du code dans OCaml

Considérez les définitions suivantes:

```
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)
```

```
let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```

Le code dans les deux exemples est presque identique - il faut le décomposer.

Décomposition du code dans OCaml

Une fonction *d'ordre supérieur* prend en compte la forme de récursivité:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)
```

Décomposition du code dans OCaml

Une fonction *d'ordre supérieur* prend en compte la forme de récursivité:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Applications de la fonction:

```
let inc x = x+1
let inc_all xs = map inc xs
```

Décomposition du code dans OCaml

Une fonction *d'ordre supérieur* prend en compte la forme de récursivité:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Applications de la fonction:

```
let inc x = x+1
let inc_all xs = map inc xs

let square y = y*y
let square_all xs = map square xs
```

On introduit de petites fonctions, pour appeler « map ». Ceci est gênant.

Décomposition du code dans OCaml

Une fonction *d'ordre supérieur* prend en compte la forme de récursivité:

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Au lieu de cela, nous pouvons utiliser une fonction *anonyme*.

À l'origine, Church a écrit cette fonction en utilisant λ au lieu de **fun**: (λx. x+1) or (λy. y*y)

Applications de la fonction:

```
let inc_all xs = map (fun x -> x + 1) xs


let square_all xs = map (fun y -> y * y) xs
```

Un détail gênant

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl);;
```

Que faire si on souhaite incrémenter une liste de nombres à virgule flottante?

On ne peut pas simplement appeler « map ». Cela ne fonctionne que sur des entiers!

Un détail gênant

```
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl);;
```

Que faire si on souhaite incrémenter une liste de nombres à virgule flottante?

On ne peut pas simplement appeler « map ». Cela ne fonctionne que sur des entiers!

```
let rec mapfloat (f:float->float) (xs:float list) :
            float list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(mapfloat f tl);;
```

Mais cela fonctionne

```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)

let ints = map (fun x -> x + 1) [1; 2; 3; 4]

let floats = map (fun x -> x +. 2.0) [3.1415; 2.718]

let strings = map String.uppercase [“sarah”; “joe”]
```

Quel est le type de cette version?

```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)

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

Quel est le type de cette version?

```
let rec map f xs =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)

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

On utilise souvent des lettres grecques comme α ou β comme noms de variables qui représentent des types.

Ce type pourrait être interprétée comme:

- pour tout type 'a et 'b,
- si on applique « map » à une fonction de type 'a -> 'b,
- le résultat est une fonction telle que
  - quand l'entrée est une liste de valeurs de type 'a
  - la fonction renvoie une liste de valeurs de type 'b.

Les types explicites

```
let rec map (f:'a -> 'b) (xs:'a list) : 'b list =
  match xs with
   | [] -> []
   | hd::tl -> (f hd)::(map f tl)

map : (‘a -> ‘b) -> ‘a list -> ‘b list
```

Le compilateur OCaml est assez intelligent pour déterminer le type *le plus général*.

On dit que « map » est *polymorphe* - une façon de dire que la fonction « map » peut être utilisée sur tous les types 'a et 'b.

Les « generics » dans Java sont dérivés du polymorphisme dans la famille de langages ML (mais ils ont été ajoutés plus tard et ils sont plus compliqués à cause du sous-typage dans Java)

Résumé

- « map » est une *fonction d'ordre supérieur* qui représente *une forme de récursivité* très courante.

- On peut écrire du code clair, concis et réutilisable si l'on utilise:
  - des fonctions d'ordre supérieur
  - des fonctions anonymes
  - des fonctions de première classe
  - le polymorphisme