

Chapitre 6 - Tests et preuves

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



Spécification

Spécification

Une spécification est un *contrat* entre

- un *client* d'un système ou d'une unité de code
- la personne qui *implémente* le code

La spécification définit le système et ses propriétés.

Une spécification doit être :

- restrictive pour empêcher des implémentations inutiles ou qui ne font pas ce qui est désiré
- souple pour laisser place à différents choix d'implémentation

Exemple de spécification : avec des types

Une manière de définir une spécification est au travers des types :

```
val sqr : float -> float
```

```
val find : ('a -> bool) -> 'a list -> 'a
```

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

- Avantage : le compilateur s'assure de vérifier que notre implémentation est conforme à la spécification
 - pour autant qu'on ait un langage statiquement typé
- Désavantage : la spécification n'est pas très précise

Exemple de spécification : avec des commentaires

On peut compléter cela avec des commentaires :

```
(** [sqr x] is the square root of [x] *)  
val sqr : float -> float  
  
(** [find f l] returns the first element of the list [l]  
that satisfies the predicate [f].  
Raises [Not_found] if there is no value that satisfies  
[f] in the list [l]. *)  
val find : ('a -> bool) -> 'a list -> 'a
```

Exemple de spécification : avec des commentaires

```
(** [map f [a1; ...; an]] applies function [f] to [a1, ..., an],  
    and builds the list [[f a1; ...; f an]] with  
    the results returned by [f]. *)  
val map : ('a -> 'b) -> 'a list -> 'b list
```

- Avantage : c'est plus précis, et très souple
- Désavantage : on n'a aucune garantie que l'implémentation soit correcte

Conformité

Si un programme implémente une spécification de façon correcte, on dit qu'il est **conforme** à sa spécification.

Comment s'assurer qu'un programme soit conforme à sa spécification ?

- Revue de code
- Tests
- Preuves

Test

Exemple : median - spécification

On souhaite calculer la médiane d'une liste

Quelle spécification sous forme de type ?

Exemple : median - spécification

On souhaite calculer la médiane d'une liste

Quelle spécification sous forme de type ?

```
val median : int list -> int
```

```
val median : float list -> float option
```

```
val median : float non_empty_list -> float
```

```
...
```

Exemple : median - tests

Quels tests utiliser pour s'assurer qu'une implémentation soit conforme à la spécification ?

- liste vide
- liste à un élément
- liste à un nombre pair d'éléments
- liste à un nombre impair d'éléments
- liste non triée

Tests unitaires

```
open OUnit2
```

```
open Median
```

```
let tests = "test suite for sum" >::: [  
  "one element" >:: (fun _ -> assert_equal 1. (median [1.]));  
  "even length" >:: (fun _ -> assert_equal 1. (median [1.; 1.]));  
  "odd length" >:: (fun _ -> assert_equal 2. (median [1.; 2.; 3.]));  
]
```

```
let _ = run_test_tt_main tests
```

Tests unitaires

```
$ git clone https://gitlab.info.uqam.ca/inf6120/exemples # exemple dans median
$ cd examples
$ dune exec -- median_test
...
Ran: 3 tests in: 0.10 seconds.
OK
```

Tests unitaires et couverture

```
$ dune exec --instrument-with bisect_ppx -- median_test  
$ bisect-ppx-report html  
$ open _coverage/index.html
```

Tests unitaires et couverture

La couverture nous indique quelle portion du code a été exécutée lors des tests.

Plus elle est proche des 100%, mieux c'est.

Mais une couverture de 100% ne donne pas de garantie sur la conformité, juste un niveau de confiance.

Tests unitaires et couverture

```
let median (l : float list) : float =  
  let len = List.length l in  
  ((List.nth l ((len-1)/2)) +. (List.nth l (len/2))) /. 2.0
```

On prend l'élément au milieu de la liste... c'est incorrect !

Pourtant on passe tous les tests

Tests unitaires et couverture

On ajoute le test suivant :

```
"unsorted" >:: (fun _ -> assert_equal 4. (median [4.; 3.; 5.]));
```

Résultat :

```
Error: test suite for sum:3:unsorted.
```

```
File "...", line 2, characters 1-1:
```

```
Error: test suite for sum:3:unsorted (in the log).
```

```
not equal
```

Tests unitaires et couverture

```
let rec merge (xs : 'a list) (ys : 'a list) : 'a list =  
  match (xs, ys) with  
  | [], _ -> ys  
  | _, [] -> xs  
  | x :: xs', y :: _ when x <= y -> x :: merge xs' ys  
  | _ :: xs', y :: ys' -> y :: merge xs' ys' (* bug: on aurait du avoir x :: xs' *)
```

```
let rec sort (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | [x] -> [x]  
  | _ ->  
    let k = List.length l / 2 in  
    merge (sort (take k l)) (sort (drop k l))
```

Tests unitaires et couverture

```
open OUnit2
```

```
open Merge
```

```
let tests = "test suite for sort" >::: [  
  "sort empty list" >:: (fun _ -> assert_equal [] (sort []));  
  "sort single-element list" >:: (fun _ -> assert_equal (sort [1]) [1]);  
  "sort two-element sorted list" >:: (fun _ -> assert_equal (sort [1;2]) [1;2]);  
]
```

```
let _ = run_test_tt_main tests
```

Tests unitaires et couverture

```
$ dune exec --instrument-with bisect_ppx -- merge_unit_tests  
Ran: 3 tests in: 0.10 seconds.  
OK
```

Ici, atteindre 100% de couverture permettra de trouver le bug.

Tests de propriétés

De façon complémentaire, on peut aussi énoncer des **propriétés** sur les fonctions à tester.

Spécification :

```
(** [sort lst] is [lst] sorted according to the built-in operator [<=].  
    Performance:  $O(n \log n)$  time, where  $n$  is the number of elements in [lst].  
    Not tail recursive. *)  
let rec sort (l : 'a list) : 'a list = ...
```

Quelles sont des propriétés importantes pour une fonction de tri ?

Tests de propriété

$$\forall l : \text{isSorted}(\text{sort}(l))$$
$$\forall l : \text{sort}(l) = \text{sort}(\text{sort}(l))$$
$$\forall l : \text{length}(l) = \text{length}(\text{sort}(l))$$
$$\forall l, n : n \in l \implies n \in \text{sort}(l)$$
$$\forall l : \text{sort}(l) = \text{otherSort}(l)$$

C'est une autre forme de spécification.

Définir des propriétés

$$\forall l, \text{isSorted}(\text{sort}(l))$$

On peut définir une propriété par une fonction :

```
let rec is_sorted (l : 'a list) : bool = match l with
| [] -> true
| _ :: [] -> true
| x :: y :: rest -> x <= y && is_sorted (y :: rest)

let sort_is_sorted (l : 'a list) = is_sorted (sort l)
```

Définir des propriétés

$$\forall l, \text{sort}(l) = \text{sort}(\text{sort}(l))$$

```
let sort_twice_eq (l : 'a list) : bool =  
  sort l = sort (sort l)
```

Définir des propriétés

$$\forall l, \text{length}(l) = \text{length}(\text{sort}(l))$$

```
let sort_length (l : 'a list) : bool =  
  List.length l = List.length (sort l)
```

Définir des propriétés

$$\forall l, n, n \in l \implies n \in \text{sort}(l)$$

```
let preserves_element (l : 'a list) (n : int) : bool =  
  if List.mem n l  
  then List.mem n (sort l)  
  else true
```

Définir des propriétés

$$\forall l, \text{sort}(l) = \text{otherSort}(l)$$

```
let sort_other_sort (l : 'a list) : bool =  
  sort l = List.sort compare l
```

Tests de propriétés avec QCheck

On peut utiliser QCheck pour automatiser les tests de ces propriétés

```
let _ = QCheck_runner.run_tests [  
  QCheck.(Test.make ~name:"sort is sorted"  
    (list small_nat) sort_is_sorted);  
  QCheck.(Test.make ~name:"sort twice is the same"  
    (list small_nat) sort_twice_eq);  
  QCheck.(Test.make ~name:"sort preserves length"  
    (list small_nat) sort_length);  
  QCheck.(Test.make ~name:"sort preserves element"  
    (pair (list small_nat) small_nat)  
    (fun (l, n) -> preserves_element l n));  
  QCheck.(Test.make ~name:"sort is the same as another sort"  
    (list small_nat) sort_other_sort);  
]
```

Tests de propriétés avec QCheck

```
$ dune exec --instrument-with bisect_ppx -- merge_prop_tests
```

```
random seed: 157169349
```

```
--- Failure -----
```

```
Test sort preserves length failed (7 shrink steps):
```

```
[1; 0]
```

```
--- Failure -----
```

```
Test sort preserves element failed (13 shrink steps):
```

```
([4; 0], 4)
```

```
--- Failure -----
```

```
Test sort is the same as another sort failed (7 shrink steps):
```

```
[1; 0]
```

```
=====
```

Fonctionnement de QCheck

Inspiré de *Quickcheck* (bibliothèque Haskell), il existe des équivalents dans de nombreux langages.

- Génère des entrées aléatoires :
 - `small_nat` génère un petit nombre
 - `list small_nat` génère une liste de petits nombres
 - on peut définir des *générateurs* soi-même
- Appelle le test avec les valeurs
- Si le test retourne vrai, il passe
- Si le test retourne faux, il rate
 - QCheck réessaie avec une entrée plus petite, jusqu'à obtenir une entrée minimale. Plus pratique pour l'utilisateur.

Preuves

Problème principal des tests : on ne teste qu'un sous ensemble des comportements possibles.

Program testing can be used to show the presence of bugs, but never to show their absence.
– Dijkstra

- Si on a une erreur sur un chemin non couvert par un test, on ne la détectera pas
- On ne peut pas dans le cas général couvrir tous les chemins

Preuve d'égalité

On peut souhaiter prouver l'équivalence entre deux programmes.

Par exemple, on a défini `map` comme :

```
let rec map (f : 'a -> 'b) (l : 'a list) : 'b list =  
  match l with  
  | [] -> []  
  | head :: tail -> (f head) :: (map tail)
```

Et on a énoncé la propriété suivante :

```
map f l = List.fold_right (fun x xs -> (f x) :: xs) l []
```

On va **démontrer** que c'est bien le cas.

Égalité

Que signifie l'égalité entre deux éléments d'un langage ?

- ils sont **syntactiquement** identiques : 42 est égal à 42 syntactiquement
- ils sont **sémantiquement** équivalents : 42 est égal à $41+1$, car leur évaluation produit la même valeur
 - la sémantique de l'expression 42 est la valeur 42
 - la sémantique de l'expression $41+2$ est la valeur 42
 - la sémantique de l'expression `let x = 21 and y = 2 in x*y` est la valeur 42
 - toutes ces expressions sont donc sémantiquement équivalentes, même si elles sont syntactiquement différentes

Égalité entre fonctions

Que signifie l'égalité entre deux fonctions ?

- elles sont syntaxiquement identiques
 - $f = g$ si f et g ont la même définition
 - pas très intéressant
- $f = g$ si pour tout x , on a $f\ x = g\ x$
 - avec des tests, on pourrait montrer que c'est le cas pour *certaines* x
 - c'est l'*axiome d'extensionnalité*

Attention : les raisonnements que l'on va faire ne sont valides que quand on considère des expressions *pures* (sans effets de bords)

- `Random.bool ()` est syntaxiquement égal à lui même, mais pas sémantiquement !

Démonstration d'équivalence : `twice`

Soit :

```
let twice f x = f (f x)
```

```
let compose f g x = f (g x)
```

On souhaite démontrer que :

```
twice f = compose f f
```

Démonstration d'équivalence : `twice`

On peut appliquer des *réécritures* en utilisant les définitions

- on peut remplacer `twice f x` par `f (f x)`
- inversement, on peut remplacer `f (f x)` par `twice f x`

C'est ce qu'on appelle **transparence référentielle** : dans un langage pur (sans effet de bord), on peut remplacer un appel de fonction par sa définition appliquée

Démonstration d'équivalence : twice

Avec :

```
let twice f x = f (f x)
```

```
let compose f g x = f (g x)
```

Théorème : `twice f = compose f f`

Par extensionnalité, on souhaite donc montrer que pour tout x , on a :

$$(twice\ f)\ x = (compose\ f\ f)\ x$$

Cela se prouve en réécrivant les définitions :

```
(twice f) x = (compose f f) x    (* propriété à démontrer *)  
            = f (f x)           (* définition de compose *)  
            = twice f x         (* définition de twice *)
```



Démonstration d'équivalence : associativité de la composition

Associativité (rappel)

Une fonction f est associative ssi : $f (f x y) z = f x (f y z)$

Par exemple : $+$ est associatif :

$$(a + b) + c = a + (b + c)$$

Démonstration d'équivalence : associativité de la composition

On peut prouver que la composition de fonction est associative, c'est-à-dire :

`compose (compose f g) h = compose f (compose g h)`

En mathématiques :

$$(f \circ g) \circ h = f \circ (g \circ h)$$

Associativité de la composition

Théorème : associativité de la composition

On souhaite montrer que :

$$\text{compose} (\text{compose } f \ g) \ h = \text{compose } f \ (\text{compose } g \ h)$$

En utilisant l'axiome d'extensionnalité, cela revient donc à montrer que, pour tout x :

$$\text{compose} (\text{compose } f \ g) \ h \ x = \text{compose } f \ (\text{compose } g \ h) \ x$$

Associativité de la composition

Théorème : associativité de la composition (suite)

On va d'abord raisonner sur la partie à gauche :

$$\begin{aligned} \text{compose } (\text{compose } f \ g) \ h \ x &= (\text{compose } f \ g) \ (h \ x) && (* \text{ déf. de compose } *) \\ &= f \ (g \ (h \ x)) && (* \text{ déf. de compose } *) \end{aligned}$$

Ensuite sur la partie à droite :

$$\begin{aligned} \text{compose } f \ (\text{compose } g \ h) \ x &= f \ ((\text{compose } g \ h) \ x) && (* \text{ déf. de compose } *) \\ &= f \ (g \ (h \ x)) && (* \text{ déf. de compose } *) \end{aligned}$$

On obtient donc une égalité syntaxique. □

Démonstration : curryfication

Soit :

let $\text{curry } f \ x \ y = f \ (x, y)$

let $\text{uncurry } f \ (x, y) = f \ x \ y$

let $\text{id } x = x$

On souhaite montrer que :

$\text{compose uncurry curry} = \text{id}$

Démonstration : curryfication

Théorème : `compose uncurry curry = id`

Par extensionnalité, cela revient à montrer que, pour tout `f` :

```
(compose uncurry curry) f
= uncurry (curry f)           (* déf. compose *)
= fun (x, y) -> (curry f) x y (* déf. uncurry *)
= fun (x, y) -> f (x, y)      (* déf. curry *)
= f                           (* eta-réduction *)
= id f                        (* déf. id *)
```



Preuve par récurrence (*proof by induction*)

On définit la fonction :

```
let rec sumto n =  
  if n = 0 then  
    0  
  else  
    n + sumto (n - 1)
```

Si on souhaite prouver que cette implémentation est équivalente à la suivante :

```
let sumto' n = n * (n + 1) / 2
```

Cela peut se faire par récurrence

Preuve par récurrence (*proof by induction*)

Rappel (INF1132) : pour une propriété P (par exemple, $\text{sumto}' = \text{sumto}$)

- si P est vraie pour 0
 - $\text{sumto}'\ 0 = \text{sumto}\ 0$
- si, à partir du fait que P est vrai pour k , on peut montrer que P est vrai pour $k + 1$
 - $\text{sumto}'\ k = \text{sumto}\ k \implies \text{sumto}'\ (k+1) = \text{sumto}\ (k+1)$
- alors P est vrai pour tout n

Preuve par récurrence (*proof by induction*)

Démarche :

- on prouve la propriété pour $n = 0$, c'est le **cas initial**
- on prouve la propriété pour $n = k + 1$ en faisant l'hypothèse qu'elle est vraie pour $n = k$, c'est le **cas de récurrence**
 - on appelle l'hypothèse que P est vraie pour k l'**hypothèse de récurrence** (*induction hypothesis*, souvent dénoté IH)

Preuve par induction

```
let rec sumto n =  
  if n = 0 then 0 else n + sumto (n - 1)
```

est équivalent à

```
let sumto' n = n * (n + 1) / 2
```

La propriété P est $P(n) = (\text{sumto } n = \text{sumto}' n)$.

Preuve par induction

Équivalence $\text{sumto} = \text{sumto}'$

Cas initial $n = 0$:

$\text{sumto } 0 = \text{sumto}' 0$

$0 = \text{sumto}' 0$ (* déf. sumto *)

$0 = 0$ (* déf. sumto' *)

Récurrence $n = k + 1$, IH = $\text{sumto } k = \text{sumto}' k$:

$\text{sumto } (k+1) = (k+1) + \text{sumto } ((k+1)-1)$ (* déf. sumto *)

$= (k+1) + \text{sumto } k$ (* arithmétique *)

$= (k+1) + \text{sumto}' k$ (* IH *)

$= (k+1) + k * (k+1) / 2$ (* déf. sumto' *)

$= (k+1) * (k + 2) / 2$ (* arithmétique *)

$= \text{sumto}' (k+1)$ (* déf. sumto' *)



Utilité de la preuve de sumto

On peut donc remplacer la définition inefficace ($\mathcal{O}(n)$) :

```
let rec sumto n =  
  if n = 0 then 0 else n + sumto (n - 1)
```

Par une définition efficace ($\mathcal{O}(1)$)

```
let sumto' n = n * (n + 1) / 2
```

Et être assuré que cela soit correct.

On peut voir `sumto` comme une *spécification*, et `sumto'` comme une implémentation de cette spécification.

Factorielle efficace

On a une implémentation trivialement correcte mais inefficace de `fact` :

```
let rec fact n =  
  if n = 0 then 1 else n * fact (n - 1)
```

Et une implémentation efficace :

```
let rec fact_iter n acc =  
  if n = 0 then acc else fact_iter (n - 1) (acc * n)  
  
let fact' n = fact_iter n 1
```

On souhaite prouver que `fact = fact'`, autrement dit, `fact n = fact_iter n 1`

Factorielle efficace : essai de démonstration

À démontrer : $\text{fact } n = \text{fact_iter } n \ 1$

Cas de base : $n = 0$

```
fact 0 = fact_iter 0 1
      = 1           (* déf. de fact_iter *)
      = fact 0      (* déf. de fact *)
```

Factorielle efficace : essai de démonstration

À démontrer : $\text{fact } n = \text{fact_iter } n \ 1$

Cas de récurrence : $n = k+1$ avec $IH = \text{fact } k = \text{fact_iter } k \ 1$

- À gauche :

$$\begin{aligned}\text{fact } (k+1) &= (k+1) * \text{fact } k && (* \text{ déf. de fact } *) \\ &= (k+1) * \text{fact_iter } k \ 1 && (* IH *)\end{aligned}$$

- À droite :

$$\begin{aligned}\text{fact_iter } (k+1) \ 1 &= \text{fact_iter } k \ (1*(k+1)) && (* \text{ déf. de fact_iter } *) \\ &= \text{fact_iter } k \ (k+1) && (* \text{ arithmétique } *)\end{aligned}$$

Mais on est bloqué...

Factorielle efficace : démonstration

On généralise le théorème : pour tout p , $p * \text{fact } n = \text{fact_iter } n \ p$

$$p * \text{fact } n = \text{fact_iter } n \ p$$

Cas de base : $n = 0$

$$p * \text{fact } 0 = \text{fact_iter } 0 \ p$$

$$= p \quad (* \text{ déf. de fact_iter } *)$$

$$= p * 1 \quad (* \text{ arithmétique } *)$$

$$= p * \text{fact } 0 \quad (* \text{ déf. de fact } *)$$

Factorielle efficace : démonstration

```
p * fact n = fact_iter n p
```

Cas de récurrence : $n = k+1$ avec $IH = p * \text{fact } k = \text{fact_iter } k \ p$

- À gauche :

```
p * fact (k+1) = p * (k+1) * fact k (* déf. de fact *)  
               = fact_iter k (p*(k+1)) (* IH *)
```

- À droite :

```
fact_iter (k+1) p = fact_iter k (p*(k+1)) (* déf. de fact_iter *)
```



Factorielle efficace : démonstration

On sait donc que : $p * \text{fact } n = \text{fact_iter } n \ p$.

C'est donc vrai en particulier pour $p = 1$, ce qui démontre que :

$$\text{fact } n = \text{fact_iter } n \ 1 = \text{fact}' \ n$$

Récurrance structurelle

Quand on manipule des types algébriques (`list`, `tree`, ...), on peut également faire des démonstrations par récurrence.

On va parler alors de **récurrance structurelle** : la preuve se fait en considérant la structure du type de donnée.

Nombres de Peano

Soit la représentation suivante des nombres naturels (*représentation de Peano*) :

type nat = Z | S **of** nat

C'est-à-dire :

- 0 est nombre naturel, dénoté Z
- si k est un nombre naturel, $k+1$ l'est aussi, et est dénoté S k

Par exemple :

- 0 est Z
- 1 est S Z
- 2 est S (S Z)

Addition de nombres de Peano

On peut additionner deux naturels dans cette représentation :

```
let rec plus (a : nat) (b : nat) : nat =  
  match a with  
  | Z -> b  
  | S k -> S (plus k b)
```

Théorème $0 + n = n$

0 est neutre à gauche pour l'addition sur nat

On souhaite montrer que :

`plus Z n = n`

Cela se fait trivialement depuis la définition de `plus`

Théorème $n + 0 = n$

On souhaite montrer que :

$\text{plus } n \text{ Z} = n$

Cas $n = \text{Z}$

$\text{plus } \text{Z } \text{Z} = \text{Z}$ (** déf. de plus **)

Cas $n = \text{S } k$

$\text{plus } (\text{S } k) \text{ z} = \text{S } (\text{plus } k \text{ Z})$ (** déf. de plus **)

On est bloqué...

Hypothèse de récurrence pour nat

On peut définir l'hypothèse de récurrence pour nat, de façon similaire à ce qu'on a fait sur les entiers.

Pour une propriété P :

- si P est vrai pour Z
- si, en partant de l'hypothèse de récurrence que P est vrai pour k , on peut montrer que P est vrai pour $S\ k$
- alors P est vrai pour toute valeur de nat

Théorème $n + 0 = n$

0 est neutre à droite pour l'addition sur nat

Cas de base : $n = \mathbb{Z}$

plus $Z \ Z = Z$ (* déf. de plus *)

Cas de récurrence, $n = S k, IH = \text{plus } k \geq 0 = k :$

$$\begin{aligned} \text{plus } (S \ k) \ Z &= S \ (\text{plus } k \ Z) && (* \text{ d\'ef. de plus } *) \\ &= S \ k && (* \text{ IH } *) \end{aligned}$$


Récurrence sur les listes

On peut procéder de même pour les listes

```
type 'a list =  
  | [] (* cas de base *)  
  | (::) of 'a * 'a list (* récurrence *)
```

Le principe de récurrence pour les listes est le suivant, pour une propriété P :

- si P est vrai pour `[]`
- si, en faisant l'hypothèse de récurrence que P est vrai pour `1`, on peut montrer que P est vrai pour `x :: 1` (pour tout `x`)
- alors P est vrai pour toute liste

Associativité de append

```
let rec append (l1 : 'a list) (l2 : 'a list) : 'a list =  
  match l1 with  
  | [] -> l2  
  | head :: tail -> head :: append tail l2
```

```
let ( @ ) = append
```

On souhaite montrer l'associativité de @, c'est-à-dire :

$$a @ (b @ c) = (a @ b) @ c$$

Associativité de append

Théorème : $a @ (b @ c) = (a @ b) @ c$

Cas de base : $a = []$

$$\begin{aligned} [] @ (b @ c) &= b @ c && (* \text{ déf. de } @ *) \\ &= ([] @ b) @ c && (* \text{ déf. de } @ *) \end{aligned}$$

Associativité de append

Théorème : $a @ (b @ c) = (a @ b) @ c$

Cas de récurrence : $a = h :: t$, avec $IH = t @ (b @ c) = (t @ b) @ c$

À gauche :

$$\begin{aligned}(h :: t) @ (b @ c) &= h :: (t @ (b @ c)) \quad (* \text{ déf. de } @ *) \\ &= h :: ((t @ b) @ c) \quad (* IH *)\end{aligned}$$

À droite :

$$\begin{aligned}((h :: t) @ b) @ c &= (h :: (t @ b)) @ c \quad (* \text{ déf. de } @ *) \\ &= h :: ((t @ b) @ c) \quad (* \text{ déf. de } @ *)\end{aligned}$$



Démonstration d'équivalence (map et fold_right)

```
let rec map (f : 'a -> 'b) (l : 'a list) : 'b list =  
  match l with  
  | [] -> []  
  | head :: tail -> (f head) :: (map f tail)
```

```
let rec fold_right (f : 'b -> 'a -> 'a) (l : 'b list) (z : 'a) : 'a =  
  match l with  
  | [] -> z  
  | head :: tail -> f head (fold_right f tail z)
```

On avait énoncé que : $\text{map } f \ l = \text{fold_right } (\text{fun } x \ xs \rightarrow (f \ x) :: xs) \ l \ []$

On va dénoter $\text{fun } x \ xs \rightarrow (f \ x) :: xs$ par g pour simplifier la suite.

Démonstration d'équivalence (map et fold_right)

Théorème $\text{map } f \ l = \text{fold_right } g \ l \ []$

Cas de base : $l = []$

À gauche :

$\text{map } f \ l = []$ (** déf. de map **)

À droite :

$\text{fold_right } g \ [] \ [] = []$

Démonstration d'équivalence (map et fold_right)

Théorème $\text{map } f \ l = \text{fold_right } g \ l \ []$

Cas de récurrence, $l = h :: t$, avec $IH = \text{map } f \ t = \text{fold_right } g \ t \ []$

À gauche :

$$\begin{aligned} \text{map } f \ (h :: t) &= (f \ h) :: \text{map } t && (* \text{ déf. de map } *) \\ &= (f \ h) :: \text{fold_right } g \ t \ [] && (* IH *) \end{aligned}$$

À droite :

$$\begin{aligned} \text{fold_right } g \ (h :: t) \ [] &= g \ h \ (\text{fold_right } g \ t \ []) && (* \text{ déf. de fold_right } *) \\ &= (f \ h) :: \text{fold_right } g \ t \ [] && (* \text{ déf. de } g *) \end{aligned}$$


Démonstration d'équivalence (filter)

Exercice

Faire de même pour filter :

```
let rec filter (p : 'a -> bool) (l : 'a list) : 'a list = match l with
| [] -> []
| head :: tail when p head -> head :: (filter l)
| head :: tail -> filter l
```

```
let rec fold_right (z : 'a) (f : 'b -> 'a -> 'a) (l : 'b list) : 'a =
  match l with
  | [] -> z
  | head :: tail -> f head (fold_right f tail z)
```

Théorème :

```
filter p l = fold_right (fun x xs -> if p x then x :: xs else xs) l []
```

Concepts clés

- **Spécification** : description des fonctionnalités devant être implémentées par un logiciel, un module, une fonction
- **Couverture de tests** : la portion du code exercée par la suite de tests
- **Tests de propriétés** : complémentaires aux tests unitaires, ils testent des propriétés générales de l'implémentation
- **Transparence référentielle** : dans un langage fonctionnel pur, une expression peut être remplacée par sa valeur sans changer le comportement du programme
- **Axiome d'extensionnalité** : si deux fonctions se comportent de la même façon pour tout argument, elles sont égales
- **Preuve par récurrence structurelle** : un type de preuve par récurrence qui s'applique aux types inductifs