11X001 – Introduction à la programmation des algorithmes

2.4b Types Paramétriques et Opérations Collectives

François Fleuret

https://fleuret.org/francois

6 Octobre, 2020

Le contenu de ce document a été en grande partie repris du cours de Jean-Luc Falcone.

UNIVERSITÉ DE GENÈVE

FACULTY OF SCIENCE

Types Paramétriques

Utilité des types paramétriques

Avec ce que nous avons vu jusqu'ici, il serait nécessaire dans de nombreuses situations de re-écrire virtuellement la même fonction pour différents types.

Par exemple:

```
func last( x: [Int] ) -> Int {
  return x[x.count - 1]
}

func last( x: [Double] ) -> Double {
  return x[x.count - 1]
}

func last( x: [String] ) -> String {
  return x[x.count - 1]
}
```

Si le typage est dynamique, ce problème ne se pose pas:

Python

```
def last( x ):
  return x[ len(x) - 1 ]
```

Javascript

```
function last(x) {
  return x[ x.length - 1 ]
}
```

Utilité des types paramétriques (cont.)

```
func not( p: @escaping (Int) -> Bool ) -> (Int) -> Bool {
  return { i in ! p(i) }
}

func not( p: @escaping (Double) -> Bool ) -> (Double) -> Bool {
  return { i in ! p(i) }
}

func not( p: @escaping (User) -> Bool ) -> (User) -> Bool {
  return { i in ! p(i) }
}
```

Python

```
def notP( p ):
  return lambda x: not p(x)
```

Javascript

```
function not( p ) {
  return x => ! p(x)
}
```

Types génériques

Certains langages de programmation acceptent des types paramétriques.

Il s'agit de types définis à la compilation qui permettent de généraliser une implémentation, en la rendant indépendante du type finalement utilisé

Syntaxe des types génériques pour les fonctions

En Swift on introduit un type générique en utilisant des chevrons (<>) entre le nom de la fonction et la liste d'arguments.

Un type générique fonctionne comme un argument pour les types, qui ne change pas durant l'appel de la fonction.

On peut passer autant de types génériques que l'on veut.

Exemple: Trois éléments d'un tableau

```
func getThree<T>( _ x: [T] ) -> (T, T, T) {
  return (x[0], x[x.count/2], x[x.count-1])
}
```

Exemple: Trois éléments d'un tableau

```
func getThree<T>( _ x: [T] ) -> (T, T, T) {
  return (x[0], x[x.count/2], x[x.count-1])
}


print(getThree([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]))

print(getThree([ 1.0, 0.5, 0.25, 0.125, 0.0625 ]))

print(getThree([ "Prof", "Atchoum", "Dormeur", "Grincheux",
                 "Joyeux", "Timide", "Simplet" ]))
```

affiche

```
(1, 6, 11)
(1.0, 0.25, 0.0625)
("Prof", "Grincheux", "Simplet")
```

Exemple: Négation d'un prédicat

```
func not<T>( _ p: @escaping (T) -> Bool ) -> (T) -> Bool {
  return { i in !p(i) }
}

let isPos = { (i: Int) in i >= 0 }

let isNeg = not(isPos)

print(isPos(4), isPos(-1), isNeg(4), isNeg(-1))
```

affiche

```
true false false true
```

Exemple: Rajouter un élément à la fin d'un tableau

```
func append<T>( _ array: [T], _ element: T ) -> [T] {
  return array + [ element ]
}

let xs = append( [1, 2, 3], 4)

let bs = append( [true, true], false )

let mats = append( [[1, 2], [3, 4]], [5, 6])
```

Contraintes sur les types

On peut également contraindre les types possibles. Nous limiterons notre usage aux types numériques pour indiquer à Swift que nous pouvons utiliser les opérations arithmétiques.

En effet:

```
func difference<T>( _ a: T, _ b: T) -> T {
  return b - a
}
```

```
./test.swift:2:12: error: protocol 'FloatingPoint' requires that 'T'
conform to 'FloatingPoint'
  return b - a
```

Contraintes sur les types (cont.)

Nous pouvons indiquer que `T` supportera le “protocole” `Numeric`:

```
func difference<T: Numeric>( _ a: T, _ b: T) -> T {
  return b - a
}

print(difference(5, 3))
print(difference(2.25, 1.15))
```

Contraintes sur les types (cont.)

Nous pouvons indiquer que `T` supportera le “protocole” `Numeric`:

```
func difference<T: Numeric>( _ a: T, _ b: T) -> T {
  return b - a
}

print(difference(5, 3))
print(difference(2.25, 1.15))
```

Pour utiliser des fonctions qui ne sont applicables qu’à des quantités à virgule flottante (*e.g.* `sqrt`), on peut préciser le protocol `FloatingPoint`.

Nous n’approfondirons pas plus ici cette notion.

Inférence de types génériques (1)

```
func applique<T, U>( _ t: T, _ f: (T) -> U ) -> U {
  return f(t)
}

let res = applique( 2, { i in Double(i + 1) } )

print(res)
```

affiche `3.0`.

Inférence de types génériques (2)

```
// Définition initiale
let res = applique( 2, { i in Double(i+1) } )

// Types génériques
let res: U = applique<T, U>( 2: T, { i in Double(i+1) }: (T) -> U )

// Inférence des types de la fonction anonyme
let res: U = applique<T, U>( 2: T, { i: T in Double(i+1): U })

// Unification T = Int
let res: U = applique<U>( 2: Int, {i: Int in Double(i+1): U })

// Unification U = Double
let res: Double = applique(2: Int, {i: Int in Double(i+1): Double })
```

Composition de fonctions

```
func compose<A, B, C>( _ f: @escaping (A) -> B,
                       _ g: @escaping (B) -> C ) -> (A) -> C {
  return { a in g(f(a)) }
}

let size = { (s: String) -> Int in s.count }
let large = { (i: Int) -> Bool in i > 6 }
let largeWord = compose( size, large )

print( largeWord("Hello") )
print( largeWord("Large Word") )
```

affiche

```
false
true
```

Composition de fonction: Inférence de types (compilation)

```
let largeWord = compose( size, large )

// Définition de compose
let largeWord: (A) -> C =
      compose<A, B, C>( size: (A) -> B, large: (B) -> C )

// size: (String) -> Int, donc A = String et B = Int
let largeWord: (String) -> C =
      compose<C>( size: (String) -> Int, large: (Int) -> C )

// large: (Int) -> Bool, donc C = Bool
let largeWord: (String) -> Bool =
      compose( size: (String) -> Int, large: (Int) -> Bool )
```

Composition de fonction: Évaluation (exécution)

```
compose( size, large )
```

Composition de fonction: Évaluation (exécution)

```
compose( size, large )
{ return { a in large(size(a)) } }
```

Composition de fonction: Évaluation (exécution)

```
compose( size, large )
{ return { a in large(size(a)) } }
{ a in large(size(a)) }
```

Composition de fonction: Évaluation (exécution)

```
compose( size, large )
{ return { a in large(size(a)) } }
{ a in large(size(a)) }
{ a in large( {s in s.count}(a) ) }
```

Composition de fonction: Évaluation (exécution)

```
compose( size, large )
{ return { a in large(size(a)) } }
{ a in large(size(a)) }
{ a in large( {s in s.count}(a) ) }
{ a in large( a.count ) }
```

Composition de fonction: Évaluation (exécution)

```
compose( size, large )
{ return { a in large(size(a)) } }
{ a in large(size(a)) }
{ a in large( {s in s.count}(a) ) }
{ a in large( a.count ) }
{ a in { i in i > 6 }( a.count ) }
```

Composition de fonction: Évaluation (exécution)

```
compose( size, large )
{ return { a in large(size(a)) } }
{ a in large(size(a)) }
{ a in large( {s in s.count}(a) ) }
{ a in large( a.count ) }
{ a in { i in i > 6 }( a.count ) }
{ a in a.count > 6 }
```

Currying

Le currying consiste à transformer une fonction qui prend plusieurs arguments en une séquence de fonctions ne prenant qu'un argument:

Par exemple, avec

$$f : (x, y) \mapsto x + y$$
$$\psi : x \mapsto (u \mapsto x + u)$$

on a

$$\forall x, y, \ f(x, y) = (\psi(x))\,(y).$$

Cette transformation s'appelle Currying en hommage au logicien Haskell Curry (1900-1982).

Currying (cont.)

```
func curry<A, B, C>(_ f: @escaping (A, B) -> C ) -> (A) -> (B) -> C {
  return { a in { b in f(a, b) } }
}

func larger( than: Int, x: Int ) -> Bool {
  return x > than
}

let isLargerThan100 = curry( larger )( 100 )

print(isLargerThan100(-4), isLargerThan100(99), isLargerThan100(102))
```

affiche

```
false false true
```

Currying (cont.)

```
func curry<A, B, C>(_ f: @escaping (A, B) -> C )-> (A) -> (B) -> C {
  return { a in { b in f(a, b) } }
}

func compare( x: Int, than: Int ) -> Bool {
  return x > than
}
```

Currying (cont.)

```
func curry<A, B, C>(_ f: @escaping (A, B) -> C )-> (A) -> (B) -> C {
  return { a in { b in f(a, b) } }
}

func compare( x: Int, than: Int ) -> Bool {
  return x > than
}

func swap<A, B, C>( _ f: @escaping (A, B) -> C ) -> (B, A) -> C {
  return { (b, a) in f(a, b)}
}
```

Currying (cont.)

```
func curry<A, B, C>(_ f: @escaping (A, B) -> C )-> (A) -> (B) -> C {
  return { a in { b in f(a, b) } }
}

func compare( x: Int, than: Int ) -> Bool {
  return x > than
}

func swap<A, B, C>( _ f: @escaping (A, B) -> C ) -> (B, A) -> C {
  return { (b, a) in f(a, b)}
}

let isLargerThan100 = curry( swap( compare ) )( 100 )

print(isLargerThan100(-4), isLargerThan100(99), isLargerThan100(102))
```

affiche

```
false false true
```

On peut aussi utiliser les génériques pour définir des types:

```
struct Pair<T> {
  let first, second: T
}

func mixup<T>( p: Pair<T>, q: Pair<T> ) -> (Pair<T>, Pair<T>) {
  return (Pair(first: p.first, second: q.second),
          Pair(first: q.first, second: p.second))
}

print(mixup(p: Pair(first: 11, second: 12),
            q: Pair(first: 21, second: 22)))
```

affiche

```
(Pair<Swift.Int>(first: 11, second: 22), Pair<Swift.Int>(first: 21, second: 12))
```

Opérations collectives

Opérations collectives

De nombreuses opérations sur les collections (*e.g.* tableaux) peuvent être décrites comme:

1. des transformations qui appliquent une même opération à chaque élément,
2. des filtres qui sélectionnent un sous-ensemble d'éléments, ou
3. des réductions qui calculent un résultat unique à partir de tous les éléments.

Transformations (`map`):

- Calculer le montant dû par chaque utilisateur.
- Appliquer un filtre sur chaque image d'une collection.
- Transformer chaque couleur d'un thème en niveaux de gris.

Opérations collectives, exemples

Transformations (`map`):

- Calculer le montant dû par chaque utilisateur.
- Appliquer un filtre sur chaque image d'une collection.
- Transformer chaque couleur d'un thème en niveaux de gris.

Filtres (`filter`):

- Quels sont les membres qui n'ont pas payé leur cotisation ?
- Supprimer les articles épuisés de l'inventaire.
- Garder uniquement les objets sélectionnés.

Opérations collectives, exemples

Transformations (`map`):

- Calculer le montant dû par chaque utilisateur.
- Appliquer un filtre sur chaque image d'une collection.
- Transformer chaque couleur d'un thème en niveaux de gris.

Filtres (`filter`):

- Quels sont les membres qui n'ont pas payé leur cotisation ?
- Supprimer les articles épuisés de l'inventaire.
- Garder uniquement les objets sélectionnés.

Réductions (`reduce`):

- Calculer le montant total.
- Quelle est la plus grosse facture ?
- Calculer le centre de masse d'un nuage de points.

Transformations, fonction `map`

La fonction `map` permet de transformer un tableau en appliquant une fonction sur chaque élément.

Le tableau d'origine n'est pas modifié.

Transformations, fonction `map`

La fonction `map` permet de transformer un tableau en appliquant une fonction sur chaque élément.

Le tableau d'origine n'est pas modifié.

Par exemple:

```
let xs = [ 1, 2, 3 ]

let ys = xs.map( { x in x * 2 } )

print(ys)
```

affiche

```
[2, 4, 6]
```

Transformations (exemples, cont.)

Le type des éléments peut changer:

```
let words = [ "machin", "chose", "truc" ]

let ns = words.map( { w in w.count } )

print(ns)
```

affiche

```
[6, 5, 4]
```

Transformations (exemples, cont.)

```
func grayLevels( color: Color ) -> Color {
  let avg = ( UInt16(color.red) +
              UInt16(color.green) +
              UInt16(color.blue) ) / 3
  let level = UInt8(avg)
  return Color( red: level, green: level, blue: level )
}

let colors: [Color] = /*...*/
let grays = colors.map( grayLevels )
```

Transformations (exemples, cont.)

```
struct Item {
  let name: String
  let unitPrice: UInt
  let amount: UInt
}

let items: [Item] = [
  Item( name: "apple", unitPrice: 45, amount: 80 ),
  Item( name: "orange", unitPrice: 75, amount: 15 )
]

let totals: [UInt] = items.map( { a in a.unitPrice * a.amount } )
```

Exemple: map

```
func rebate( _ item: Item) -> Item {
  if item.amount >= 20 {
    return Item(
      name: item.name,
      unitPrice: item.unitPrice * 95 / 100,
      amount: item.amount
    )
  } else {
    return item
  }
}

let totals: [UInt] = items.map(rebate).map( { a in a.unitPrice * a.amount } )
```

Remarque `map`

Ces trois expressions sont équivalentes:

```
xs.map(f).map(g)
xs.map( {x in g(f(x)) } )
xs.map( compose(f, g) )
```

- On peut combiner plusieurs `map` consécutifs en composant les opérations
- `map` me permet de transformer toute fonction `A->B` en une fonction `[A]->[B]`

Filtrage, fonction `filter`

La fonction `filter` permet de garder uniquement les éléments qui satisfont un prédicat.

Le tableau d'origine n'est pas modifié

Filtrage, fonction `filter`

La fonction `filter` permet de garder uniquement les éléments qui satisfont un prédicat.

Le tableau d'origine n'est pas modifié

Par exemple:

```
let words = ["machin", "chose", "truc"]

let ns = words.filter( { w in w.count < 6 } )

print(ns)
```

affiche

```
["chose", "truc"]
```

Exemple: `filter`

```
struct Item {
  let name: String
  let unitPrice: UInt
  let amount: UInt
}

let items: [Item] = [
  Item( name: "apple", unitPrice: 45, amount: 80 ),
  Item( name: "orange", unitPrice: 75, amount: 0 ),
  Item( name: "peer", unitPrice: 75, amount: 15 )
]

print("All:", items)

print("Cleaned up:", items.filter({ i in i.amount > 0}))
```

affiche

```
All: [ Item(name: "apple", unitPrice: 45, amount: 80),
       Item(name: "orange", unitPrice: 75, amount: 0),
       Item(name: "peer", unitPrice: 75, amount: 15)]

Cleaned up: [ Item(name: "apple", unitPrice: 45, amount: 80),
              Item(name: "peer", unitPrice: 75, amount: 15)]
```

Exemple: `filter`

```
func isGray( color c: Color ) -> Bool {
  return c.red == c.green && c.red == c.blue
}

func not<T>( _ p: (T) -> Bool ) -> (T) -> Bool {
  return { i in !p(i) }
}

let vividColors = colors.filter( not(isGray) )
```

Remarque `filter`

Chaque élément est filtré indépendamment

La taille du tableau produit est inférieure ou égale à celle du tableau de départ

On peut combiner plusieurs `filter` consécutifs en fusionnant les prédicats.

Avec

```
func and<T>( p1: (T) -> Bool, p2: (T) -> Bool ): (T) -> Bool {
  return { t in p1(t) && p2(t) }
}

ys1 = xs.filter(p).filter(q)
ys2 = xs.filter( and(p, q) )
```

Les tableaux `ys1` et `ys2` sont identiques.

Propriétés `filter`: Exemple

```
struct Item {
  let name: String
  let unitPrice: UInt
  let amount: UInt
}

let items: [Item] = /*...*/

items.filter(
  and(
    {item in item.uniPrice >= 100_00},
    {item in item.amount > 1 }
  )
)
```

Réduction

Étant données une valeur

$$y_0 \in \mathcal{Y}$$

et une fonction

$$f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Y},$$

la réduction appliquée au tableau de valeurs

$$x_n \in \mathcal{X},\ n = 1, \ldots, N$$

calcule la suite

$$y_n = f(x_n, y_{n-1}),\ n = 1, \ldots, N$$

et retourne y_N.

Réduction, fonction `reduce`

La méthode `reduce` implémente cette opération, et prend comme arguments y_0 et f. Elle ne modifie pas le tableau auquel elle est appliquée.

Réduction, fonction `reduce`

La méthode `reduce` implémente cette opération, et prend comme arguments y_0 et f. Elle ne modifie pas le tableau auquel elle est appliquée.

Par exemple:

```
let ns = [1, 2, 3, 4, 5]
let n = ns.reduce( 0, { y, x in x + y } )
print(n)
```

affiche

```
15
```

Exemple de réduction (1)

Longueur moyenne des mots d'une liste:

```
let words = [ "machin", "chose", "truc" ]

let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )

let avg = Double(sum) / Double(words.count)

print(avg)
```

Exemple de réduction (1)

Longueur moyenne des mots d'une liste:

```
let words = [ "machin", "chose", "truc" ]

let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )

let avg = Double(sum) / Double(words.count)

print(avg)
```

Ou, de manière équivalente

```
let sum = words.reduce( 0, { y, w in w.count + y } )
```

Exemple de réduction (1, cont.)

```
let words = [ "machin", "chose", "truc" ]
let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )
```

Évaluation de la réduction:

```
[6, 5, 4].reduce( 0, { y, x in x + y } )
```

Exemple de réduction (1, cont.)

```
let words = [ "machin", "chose", "truc" ]
let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )
```

Évaluation de la réduction:

```
[6, 5, 4].reduce( 0, { y, x in x + y } )
[5, 4].reduce( 0+6, { y, x in x + y } )
```

Exemple de réduction (1, cont.)

```
let words = [ "machin", "chose", "truc" ]
let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )
```

Évaluation de la réduction:

```
[6, 5, 4].reduce( 0, { y, x in x + y } )
[5, 4].reduce( 0+6, { y, x in x + y } )
[4].reduce( (0+6)+5, { y, x in x + y } )
```

Exemple de réduction (1, cont.)

```
let words = [ "machin", "chose", "truc" ]
let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )
```

Évaluation de la réduction:

```
[6, 5, 4].reduce( 0, { y, x in x + y } )
[5, 4].reduce( 0+6, { y, x in x + y } )
[4].reduce( (0+6)+5, { y, x in x + y } )
[].reduce( ((0+6)+5)+4, { y, x in x + y } )
```

Exemple de réduction (1, cont.)

```
let words = [ "machin", "chose", "truc" ]
let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )
```

Évaluation de la réduction:

```
[6, 5, 4].reduce( 0, { y, x in x + y } )
[5, 4].reduce( 0+6, { y, x in x + y } )
[4].reduce( (0+6)+5, { y, x in x + y } )
[].reduce( ((0+6)+5)+4, { y, x in x + y } )
((0+6)+5)+4
```

Exemple de réduction (1, cont.)

```
let words = [ "machin", "chose", "truc" ]
let sum = words.map( { w in w.count } ).reduce( 0, { y, x in x + y } )
```

Évaluation de la réduction:

```
[6, 5, 4].reduce( 0, { y, x in x + y } )
[5, 4].reduce( 0+6, { y, x in x + y } )
[4].reduce( (0+6)+5, { y, x in x + y } )
[].reduce( ((0+6)+5)+4, { y, x in x + y } )
((0+6)+5)+4
15
```

Exemple de réduction (2)

```
let words = ["le", "machin", "chose", "truc"]

func h(y: String, x: String) -> String {
  if y.count >= x.count {
    return y
  } else {
    return x
  }
}

print(words.reduce( "", h ))
```

Exemple de réduction (2, cont.)

Évaluation de `words.reduce( "", h )`:

```
["le", "machin", "chose"].reduce( "", h )
```

Exemple de réduction (2, cont.)

Évaluation de `words.reduce( "", h )`:

```
["le", "machin", "chose"].reduce( "", h )
["machin", "chose"].reduce( h("", "le"), h )
```

Exemple de réduction (2, cont.)

Évaluation de `words.reduce( "", h )`:

```
["le", "machin", "chose"].reduce( "", h )
["machin", "chose"].reduce( h("", "le"), h )
["chose"].reduce( h( h("", "le"), "machin" ), h )
```

Exemple de réduction (2, cont.)

Évaluation de `words.reduce( "", h )`:

```
["le", "machin", "chose"].reduce( "", h )
["machin", "chose"].reduce( h("", "le"), h )
["chose"].reduce( h( h("", "le"), "machin" ), h )
[].reduce( h( h( h("", "le"), "machin" ), "chose" ), h )
```

Exemple de réduction (2, cont.)

Évaluation de `words.reduce( "", h )`:

```
["le", "machin", "chose"].reduce( "", h )
["machin", "chose"].reduce( h("", "le"), h )
["chose"].reduce( h( h("", "le"), "machin" ), h )
[].reduce( h( h( h("", "le"), "machin" ), "chose" ), h )
h( h( h("", "le"), "machin" ), "chose" )
```

Exemple d'évaluation (1)

```
func h(x: String, y: String) -> String {
  if y.count >= x.count {
    return y
  } else {
    return x
  }
}
```

```
h( h( h("", "le"), "machin" ), "chose" )
```

Exemple d'évaluation (1)

```
func h(x: String, y: String) -> String {
  if y.count >= x.count {
    return y
  } else {
    return x
  }
}
```

```
h( h( h("", "le"), "machin" ), "chose" )
h( h( "le", "machin" ), "chose" )
```

Exemple d'évaluation (1)

```
func h(x: String, y: String) -> String {
  if y.count >= x.count {
    return y
  } else {
    return x
  }
}
```

```
h( h( h("", "le"), "machin" ), "chose" )
h( h( "le", "machin" ), "chose" )
h( "machin", "chose" )
```

Exemple d'évaluation (1)

```
func h(x: String, y: String) -> String {
  if y.count >= x.count {
    return y
  } else {
    return x
  }
}
```

```
h( h( h("", "le"), "machin" ), "chose" )
h( h( "le", "machin" ), "chose" )
h( "machin", "chose" )
"machin"
```

forall et exists comme réductions

```
func forall<T>( _ xs: [T], _ p: (T) -> Bool ) -> Bool {
  return xs.reduce( true, { y, x in y && p(x) } )
}

func exists<T>( _ xs: [T], _ p: (T) -> Bool ) -> Bool {
  return xs.reduce( false, { y, x in y || p(x) } )
}
```

Centre de masse d'un nuage de points

Un point est un vecteur dans $\mathbb{R}^3$.

Le centre de masse d'un nuage de N points est:

$$\mathbf{c} = \frac{1}{N} \sum_{i=0}^{N-1} \mathbf{p}_i$$

Centre de masse d'un nuage de points

Un point est un vecteur dans $\mathbb{R}^3$.

Le centre de masse d'un nuage de N points est:

$$\mathbf{c} = \frac{1}{N} \sum_{i=0}^{N-1} \mathbf{p}_i$$

On a donc besoin de:

1. additionner deux points
2. diviser un point par une constante:

Centre de masse d'un nuage de points (cont.)

```
struct Point {
  let x, y, z: Double
}

func sum( p: Point, q: Point ) -> Point {
  return Point( x: p.x + q.x, y: p.y + q.y, z: p.z + q.z )
}

func div( p: Point, a: Double ) -> Point {
  return Point( x: p.x / a, y: p.y / a, z: p.z / a )
}
```

Centre de masse d'un nuage de points (cont.)

```
let points: [Point] = [
  Point( x: 1, y: 1, z: 1),
  Point( x: 1, y: 9, z: 99)
]

let center = div(
  p: points.reduce( Point(x: 0, y: 0, z: 0), sum ),
  a: Double(points.count)
)

print(center)
```

Généralisation de la moyenne

```
func average<T>(zero: T,
                sum: @escaping (T, T) -> T,
                div: @escaping (T, Double) -> T) -> ([T]) -> T {
  return { ts in
    div( ts.reduce(zero, sum), Double(ts.count) )
  }
}

let centerOfMass = average(zero: Point(x:0, y:0, z:0), sum: sum, div: div)

print(centerOfMass( points ))
```

FIN