11X001 – Introduction à la programmation des algorithmes

3.3 Types de données mutables

François Fleuret

https://fleuret.org/francois

26 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

PGDC

Le plus grand diviseur commun à deux entiers (pgdc) vérifie la propriété suivante:

$$\forall a, b \in \mathbb{N}^*, a \leq b, \ \text{pgdc}(a, b) = \text{pgdc}(b \bmod a, a).$$

PGDC

Le plus grand diviseur commun à deux entiers (pgdc) vérifie la propriété suivante:

$$\forall a, b \in \mathbb{N}^*, a \leq b,\ \text{pgdc}(a, b) = \text{pgdc}(b \bmod a, a).$$

L'algorithme d'Euclide du calcul du pgdc utilise cette expression récursivement:

$$\forall a, b \in \mathbb{N}^*, a \leq b,\ \text{pgdc}(a, b) = \begin{cases} b \text{ si } a = 0 \\ \text{pgdc}(b \bmod a, a) \text{ sinon.} \end{cases}$$

PGDC (cont.)

```
func pgdc(_ a: Int, _ b: Int) -> Int {
  if a > b {
    return pgdc(b, a)
  } else {
    if a == 0 {
      return b
    } else {
      return pgdc(b%a, a)
    }
  }
}

print(pgdc(2 * 2 * 3 * 11 * 29, 2 * 5 * 5 * 17 * 29 * 59))
```

PGDC (cont.)

```
func pgdc(_ a: Int, _ b: Int) -> Int {
  if a > b {
    return pgdc(b, a)
  } else {
    if a == 0 {
      return b
    } else {
      return pgdc(b%a, a)
    }
  }
}

print(pgdc(2 * 2 * 3 * 11 * 29, 2 * 5 * 5 * 17 * 29 * 59))
```

affiche 58.

PGDC (cont.)

De plus,

$$\text{pgdc}(n_1, \ldots, n_k) = \text{pgdc}(\text{pgdc}(n_1, n_2), \ldots, n_k)$$

PGDC (cont.)

De plus,

$$\text{pgdc}(n_1, \ldots, n_k) = \text{pgdc}(\text{pgdc}(n_1, n_2), \ldots, n_k)$$

```
func pgdc(_ ns: [ Int ]) -> Int {
  return ns.reduce(0, pgdc)
}

print(pgdc([ 2 * 3 * 17, 2 * 3 * 19, 2 * 2 * 3 * 3 ]))
```

PGDC (cont.)

De plus,

$$\text{pgdc}(n_1, \ldots, n_k) = \text{pgdc}(\text{pgdc}(n_1, n_2), \ldots, n_k)$$

```
func pgdc(_ ns: [ Int ]) -> Int {
  return ns.reduce(0, pgdc)
}

print(pgdc([ 2 * 3 * 17, 2 * 3 * 19, 2 * 2 * 3 * 3 ]))
```

affiche 6.

Sémantique par référence

Sémantique par référence

Bien que par défaut les arguments en Swift sont passés par valeur et sont traités comme des constantes, on peut explicitement spécifier qu'un argument est passé par référence.

Dans ce cas la variable utilisée au moment de l'appel de la fonction et l'argument dans la fonction font référence à la même quantité, et toute modification du second modifie la première.

Sémantique par référence (cont.)

Pour indiquer qu'un argument est passé par référence:

1. Un tel argument doit être annoté avec `inout` dans la définition de la fonction.
2. La variable passée doit être préfixée par `&` quand la fonction est appelée.

On ne peut bien sûr passer ni une constante ni une expression littérale.

Sémantique par référence (cont.)

```
func incr( a: inout Int ) {
  a += 1
}

var x = 14

print(x)
incr(a: &x)
print(x)
```

affiche

```
14
15
```

Sémantique par référence (cont.)

Une constante ne peut pas être passée par référence:

```
func incr( a: inout Int ) {
  a += 1
}

let x = 14

print(x)
incr(a: &x)
print(x)
```

affiche

```
./a.swift:8:9: error: cannot pass immutable value as inout argument:
'x' is a 'let' constant
incr(a: &x)
        ^~
```

Sémantique par référence (cont.)

Et une constante littérale ne peut évidemment pas être passée par référence:

```
func incr( a: inout Int ) {
  a += 1
}

incr(a: &21)
```

affiche

```
./a.swift:5:9: error: cannot pass immutable value as inout argument:
literals are not mutable
incr(a: &21)
        ^~~
```

Sémantique par référence (cont.)

Une fonction peut évidemment avoir des paramètres passés par valeur et d'autres par référence:

```
func shift( _ c: Double, _ x: inout Double) {
  x = (c + x) / 2
}

var mu = 5.0
var nu = 7.0

print(mu, nu)
shift(mu, &nu)
print(mu, nu)
```

Sémantique par référence (cont.)

Une fonction peut évidemment avoir des paramètres passés par valeur et d'autres par référence:

```
func shift( _ c: Double, _ x: inout Double) {
  x = (c + x) / 2
}

var mu = 5.0
var nu = 7.0

print(mu, nu)
shift(mu, &nu)
print(mu, nu)
```

affiche

```
5.0 7.0
5.0 6.0
```

Tableaux mutables

Tableaux mutables

On a vu que l'on peut changer la valeur d'une variable.

Dans le cas d'un tableau, on peut modifier ses éléments individuellement. *E.g*

```
var xs = [1, 2, 3]
print(xs)  // Affiche [1, 2, 3]

xs[0] = 10
xs[2] += 3
print(xs) // Affiche [10, 2, 6]]
```

Sémantique par valeur

En Swift, un tableau est considéré comme une valeur, au même titre qu'un nombre ou un booléen.

Les tableaux sont copiés:

- À l'assignation, et
- au passage de fonction.

Sémantique par valeur (cont.)

```
var xs = [1, 2, 3]
var ys = xs       // Copie

print(xs)         // Affiche [1, 2, 3]
print(ys)         // Affiche [1, 2, 3]

xs[0] = 10
xs[2] += 3

print(xs)         // Affiche [10, 2, 6]]
print(ys)         // Affiche [1, 2, 3]
```

Sémantique par valeur (cont.)

```
var xs = [1, 2, 3]
var ys = xs       // Copie

print(xs)         // Affiche [1, 2, 3]
print(ys)         // Affiche [1, 2, 3]

ys[0] = 10
ys[2] += 3

print(xs)         // Affiche [1, 2, 3]
print(ys)         // Affiche [10, 2, 6]]
```

Exemple: tableau par référence

```
func incr( _ xs: inout [Int] ) {
  for i in 0..<xs.count {
    xs[i] += 1
  }
}

var data = [1, 2, 3]
incr( &data )

print(data)  // Affiche [2, 3, 4]
```

Sémantique par référence dans d'autres langages

La plupart des langages (*e.g.* Python, Javascript, C, Java, etc.) utilisent une sémantique par référence pour gérer les tableaux.

Dans ce cas, une variable tableau ne contient pas les valeurs du tableau elles-mêmes, mais une référence aux "variables" qui stockent ces valeurs.

Lorsqu'on assigne un tableau à un autre on crée une indirection entre les deux (référence, pointeur, alias, lien, etc.)

Les copies doivent être faites explicitement, pour éviter les modifications involontaires.

Sémantique par référence dans d'autres langages (cont.)

Assignement $Y = X$ par valeur

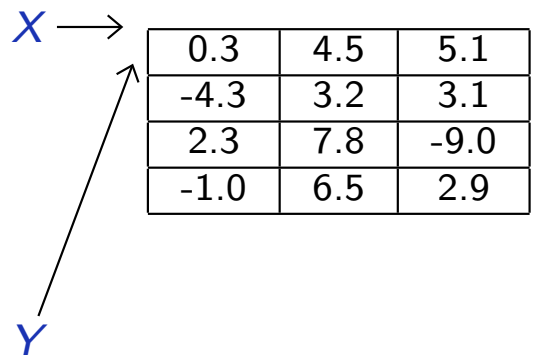
$X \longrightarrow$

0.3	4.5	5.1
-4.3	3.2	3.1
2.3	7.8	-9.0
-1.0	6.5	2.9

$Y \longrightarrow$

0.3	4.5	5.1
-4.3	3.2	3.1
2.3	7.8	-9.0
-1.0	6.5	2.9

Assignement $Y = X$ par référence



Sémantique par référence dans d'autres langages (cont.)

En Python par exemple:

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

print(xs)
print(ys)

xs[0] = 10
xs[2] += 3

print(xs)
print(ys)
```

Sémantique par référence dans d'autres langages (cont.)

En Python par exemple:

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

print(xs)
print(ys)

xs[0] = 10
xs[2] += 3

print(xs)
print(ys)
```

affiche

```
[1, 2, 3]
[1, 2, 3]
[10, 2, 6]
[10, 2, 6]
```

Efficacité du travail “sur place”

Imaginons que l’on veuille faire une fonction qui multiplie les valeurs négatives d’un tableau par une constante:

```
var x = [ -1.0, 2.0, 2.0, -2.0 ]

print(mulNeg(x, 5))
```

devrait afficher

```
[-5.0, 2.0, 2.0, -10.0]
```

Efficacité du travail “sur place” (cont.)

Une solution purement fonctionnelle serait

```
func mulNeg( _ x: [Double], _ k: Double) -> [Double] {
  func mulNegRec( _ i: Int, _ x: [Double]) -> [Double] {
    if i < x.count {
      var t = x
      if x[i] < 0 { t[i] *= k }
      return mulNegRec(i + 1, t)
    } else {
      return x
    }
  }

  return mulNegRec(0, x)
}
```

Efficacité du travail “sur place” (cont.)

```
func mulNeg( _ x: [Double], _ k: Double) -> [Double] {
  func mulNegRec( _ i: Int, _ x: [Double]) -> [Double] {
    if i < x.count {
      var t = x                    // !!! COPIE !!!
      if x[i] < 0 { t[i] *= k }
      return mulNegRec(i + 1, t)
    } else {
      return x
    }
  }

  return mulNegRec(0, x)
}
```

Dans cette version, `mulNegRec` est exécutée autant de fois que le tableau à traiter a d'éléments, et elle fait une copie de ce tableau à chaque fois.

Le nombre d'opérations et l'occupation mémoire sont donc proportionnelles à la taille du tableau au carré!

Efficacité du travail “sur place” (cont.)

Si on se permet de modifier le tableau lui-même, on évite la copie.

```
func mulNeg( _ x: inout [Double], _ k: Double) {
  func mulNegRec( _ i: Int) {
    if i < x.count {
      if x[i] < 0 { x[i] *= k }
      return mulNegRec(i + 1)
    }
  }
  mulNegRec(0)
}

var x = [ -1.0, 2.0, 2.0, -2.0 ]

mulNeg(&x, 5)

print(x)
```

affiche

```
[-5.0, 2.0, 2.0, -10.0]
```

Efficacité du travail “sur place” (cont.)

On peut même faire une version hybride qui a l’avantage de travailler “sur place” tout en ayant une signature purement fonctionnelle.

```
func mulNeg( _ x: [Double], _ k: Double) -> [Double] {
  var y = x
  func mulNegRec( _ i: Int) {
    if i < y.count {
      if y[i] < 0 { y[i] *= k }
      mulNegRec(i + 1)
    }
  }

  mulNegRec(0)
  return y
}

var x = [ -1.0, 2.0, 2.0, -2.0 ]

print(mulNeg(x, 5))
```

affiche

```
[-5.0, 2.0, 2.0, -10.0]
```

Efficacité du travail “sur place” (cont.)

Autre version avec une boucle

```
func mulNeg( _ x: [Double], _ k: Double) -> [Double] {
  var y = x
  for i in 0..<y.count {
    if y[i] < 0 { y[i] *= k }
  }
  return y
}
```

Efficacité du travail “sur place” (cont.)

Autre version avec une boucle

```
func mulNeg( _ x: [Double], _ k: Double) -> [Double] {
  var y = x
  for i in 0..<y.count {
    if y[i] < 0 { y[i] *= k }
  }
  return y
}
```

ou utilisant `map`

```
func mulNeg( _ x: [Double], _ k: Double) -> [Double] {
  return x.map({ a in if a < 0 { return a * k } else { return a }})
}
```

Fonctions de manipulations de tableaux

Plusieurs méthodes permettent de modifier un tableau mutable en ajoutant des éléments:

```
var x = [10, 20, 30]
x.append(40)  // Ajoute 40 à la fin du tableau
print(x)      // Affiche [10, 20, 30, 40]

x.append(contentsOf: [50,60]) // Ajoute 50 et 60
print(x) // Affiche [10, 20, 30, 40, 50, 60]

x.insert(35, at: 3)  // Insère 35 en position 3
print(x) // Affiche [10, 20, 30, 35, 40, 50, 60]
```

Fonctions de manipulations de tableaux (cont.)

Plusieurs méthodes permettent de modifier un tableau mutable en supprimant des éléments:

```
var x = [10, 20, 30, 35, 40, 50, 60]

x.remove( at: 0 ) // Supprime le premier élément
print(x) // Affiche [20, 30, 35, 40, 50, 60]

let y = x.removeLast() // Supprime le dernier élément
print(x) // Affiche [20, 30, 35, 40, 50]
print(y) // Affiche 60
```

Fonctions de manipulations de tableaux (cont.)

On peut créer un tableau composé de n éléments répétés de valeur x grâce à la fonction:

```
Array.init(repeating: x, count: n)
```

Par exemple:

```
let zeros = Array.init(repeating: 0, count: 6)

print(zeros) // Affiche [0, 0, 0, 0, 0, 0]
```

Exemple: Addition vectorielle

```
func add( _ x: [Double], _ y: [Double] ) -> [Double] {
  assert(x.count == y.count, "Dimensions do not match")
  var z = Array.init(repeating: 0.0, count: x.count)
  for i in 0..<x.count {
    z[i] = x[i] + y[i]
  }
  return z
}

let x = [  1.0, 2.0, 1.0 ]
let y = [ -1.0, 3.0, 4.0 ]

print(add(x, y))
```

affiche

```
[0.0, 5.0, 5.0]
```

Addition vectorielle: Impératif vs. fonctionnel

```
func add( _ x: [Double], _ y: [Double] ) -> [Double] {
  assert(x.count == y.count, "Dimensions do not match")
  var z = Array.init(repeating: 0.0, count: x.count)
  for i in 0..<x.count {
    z[i] = x[i] + y[i]
  }
  return z
}

let x = [  1.0, 2.0, 1.0 ]
let y = [ -1.0, 3.0, 4.0 ]

print(add(x, y))
```

```
func add( _ x: [Double], _ y: [Double] ) -> [Double] {
  assert(x.count == y.count, "Dimensions do not match")
  return zip(x, y).map( { i, j in i + j } )
}
```

Structures mutables

Propriétés mutables

Une structure peut contenir des propriétés mutables. Il suffit de les déclarer avec `var` au lieu de `let`.

Par exemple:

```
struct User {
  let name: String     // Immutable
  var password: String // Mutable
}
```

Structure mutable

Pour être mutable, une structure doit être déclarée en variable (comme un tableau):

```
struct User {
  let name: String
  var password: String
}

var alice = User(name: "Alice", password: "")

alice.password = "58Ul2dF" // Change le mot de passe

let bob = User(name: "Bob", password: "")

bob.password = "4Fg001e"  // Erreur de compilation
```

Sémantique par valeur

Comme les tableaux, les structures ont une sémantique par valeur:

```
struct User {
  let name: String
  var password: String
}

var alice1 = User(name: "Alice", password: "1234")
var alice2 = alice1

alice1.password = "58Ul2dF"

print( alice1.password ) // Affiche 58Ul2dF
print( alice2.password ) // Affiche 1234
```

Même comportement que pour les tableaux:

```
struct User {
  let name: String
  var password: String
}

func randomPassword( length: UInt ) -> String {
  let c = "abcdefghijklmnopqrstuvwxyz0123456789"
  return String( (0..<length).map( { _ in c.randomElement()! } ) )
}

func resetPassword( user: User ) {
  user.password = randomPassword(length: 16)
}

var alice = User(name: "Alice", password: "")

resetPassword(user: alice)
```

Même comportement que pour les tableaux:

```
struct User {
  let name: String
  var password: String
}

func randomPassword( length: UInt ) -> String {
  let c = "abcdefghijklmnopqrstuvwxyz0123456789"
  return String( (0..<length).map( { _ in c.randomElement()! } ) )
}

func resetPassword( user: User ) {
  user.password = randomPassword(length: 16)
}

var alice = User(name: "Alice", password: "")

resetPassword(user: alice)

./pwd.swift:12:8: error: cannot assign to property: 'user' is a 'let' constant
  user.password = randomPassword(length: 16)
  ~~~~ ^
```

Il faut donc spécifier que des arguments sont passés par référence pour pouvoir les modifier.

```
struct User {
  let name: String
  var password: String
}

func randomPassword( length: UInt ) -> String {
  let c = "abcdefghijklmnopqrstuvwxyz0123456789"
  return String( (0..<length).map( { _ in c.randomElement()! } ) )
}

func resetPassword( user: inout User ) {
  user.password = randomPassword(length: 16)
}

var alice = User(name: "Alice", password: "")

resetPassword(user: &alice)
```

Exemple (1)

```
struct Item {
  let description: String
  var price: UInt
  var amount: UInt
}

func discount( item: inout Item, of: Double ) {
  item.price = UInt( Double(item.price) * (1 - of) )
}

var apple = Item(description:"Apple",
                 price: 30, amount: 100)

print(apple)
discount(item: &apple, of: 0.1)
print(apple)
```

Exemple (1)

```
struct Item {
  let description: String
  var price: UInt
  var amount: UInt
}

func discount( item: inout Item, of: Double ) {
  item.price = UInt( Double(item.price) * (1 - of) )
}

var apple = Item(description:"Apple",
                 price: 30, amount: 100)

print(apple)
discount(item: &apple, of: 0.1)
print(apple)
```

affiche

```
Item(description: "Apple", price: 30, amount: 100)
Item(description: "Apple", price: 27, amount: 100)
```

Exemple (2)

```
func discount( item: inout Item, of: Double ) {
  item.price = UInt( Double(item.price) * (1-of) )
}

func discountAll( items: inout [Item], of: Double ) {
  for i in 0..<items.count {
    discount(item: &items[i], of: of)
  }
}
```

Exemple: Version fonctionnelle

```
func priceUpdate(_ f: (Item) -> UInt) -> (Item) -> Item {
   return { item in
     Item( description: item.description,
           price: f(item),
           amount: item.amount)}
}

func discount( of: Double ) -> (Item) -> UInt {
  return {item in UInt( Double(item.price) * (1-of) )}
}

func discountAll( items: [Item], of: Double ) -> [Item] {
  return items.map( priceUpdate( discount(of: of) ) )
}
```

Tuples

Même sémantique que les tableaux et les structures:

```
var x = (2, false)
var y = x

if !x.1 {
  x.0 += 1
}

print(x)
print(y)
```

Tuples

Même sémantique que les tableaux et les structures:

```
var x = (2, false)
var y = x

if !x.1 {
  x.0 += 1
}

print(x)
print(y)
```

affiche

```
(3, false)
(2, false)
```

Tuples (cont.)

Une fonction qui met une paire de `Double` par ordre croissant:

```
func ordonne( _ p: (Double, Double) ) -> (Double, Double) {
  if p.0 <= p.1 {
    return p
  } else {
    return (p.1, p.0)
  }
}

var a = (15.3, 15.2)

print(a)
a = ordonne(a)
print(a)
```

Tuples (cont.)

Une fonction qui met une paire de `Double` par ordre croissant:

```
func ordonne( _ p: (Double, Double) ) -> (Double, Double) {
  if p.0 <= p.1 {
    return p
  } else {
    return (p.1, p.0)
  }
}

var a = (15.3, 15.2)

print(a)
a = ordonne(a)
print(a)
```

affiche

```
(15.3, 15.2)
(15.2, 15.3)
```

Tuples (cont.)

Avec un argument par référence:

```
func ordonne( _ p: inout (Double, Double) ) {
  if p.0 > p.1 {
    let x = p.0
    p.0 = p.1
    p.1 = x
  }
}

var a = (15.3, 15.2)

print(a)
ordonne(&a)
print(a)
```

Tuples (cont.)

Avec un argument par référence:

```
func ordonne( _ p: inout (Double, Double) ) {
  if p.0 > p.1 {
    let x = p.0
    p.0 = p.1
    p.1 = x
  }
}

var a = (15.3, 15.2)

print(a)
ordonne(&a)
print(a)
```

affiche

```
(15.3, 15.2)
(15.2, 15.3)
```

Chaînes de caractères

Chaînes de caractères mutables

Comme les tableaux, les tuples et les structures, une chaîne de caractère devient mutable si on déclare comme une variable.

```
var greetings = "Hello"
greetings += ", "
greeeting += "world"

print( greetings) // Affiche Hello, world
```

Boucle sur les caractères

On peut itérer sur les caractères d'une chaîne de caractères avec une boucle `for`:

```
let abc = "ABC"
for char in abc {
  print( char )
}
// Affiche A
// Affiche B
// Affiche C
```

Index des caractères dans une chaîne

On peut accéder aux caractères individuels d'une chaîne grâce aux index. Ce ne sont pas des entiers mais sont du type spécifique `String.index`.

Index des caractères dans une chaîne

On peut accéder aux caractères individuels d'une chaîne grâce aux index. Ce ne sont pas des entiers mais sont du type spécifique `String.index`.

- L'index du premier caractères s'obtient grâce à la propriété `startIndex`.
- On accède aux suivants en utilisant un `offset`.
- Le dernier index s'obtient grâce à la propriété `endIndex`, qui correspond à la position après le dernier caractère.
- On peut itérer grâce à `indices`.

Index des caractères dans une chaîne (cont.)

```
let alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

let start = alphabet.startIndex
print( alphabet[start] ) // Affiche A

let e = alphabet.index( start, offsetBy: 4 )
print( alphabet[e])     // Affiche E

let end = alphabet.endIndex
print( alphabet[end])   // Erreur d'exécution
```

Index des caractères dans une chaîne (cont.)

Et on peut également itérer sur les index de tous les caractères d'une chaîne en utilisant la méthode `indices`:

```
let alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

for i in alphabet.indices {
  print(alphabet[i])
}
```

Index des caractères dans une chaîne (cont.)

On ne peut pas modifier un caractère à partir de son index!

```
var alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
let start = alphabet.startIndex
let e = alphabet.index( start, offsetBy: 4 )
alphabet[e] = "e"

main.swift:4:9: error: cannot assign through subscript:
                       subscript is get-only
alphabet[e] = "e"
```

Modification de chaîne de caractères

Les méthodes pour modifier les chaînes de caractères en rajoutant / enlevant des éléments sont similaires à celles pour les tableaux.

Les éléments sont toujours des caractères, et les indices sont aussi du type `String.index`.

```
var s = "ABCD"

s.append( "EF" )
print(s) // Affiche: ABCDEF

s.insert( "Z", at: s.startIndex )
print(s) // Affiche: ZABCDEF

s.removeLast()
print(s) // Affiche: ZABCDE
```

FIN