Types et polymorphisme

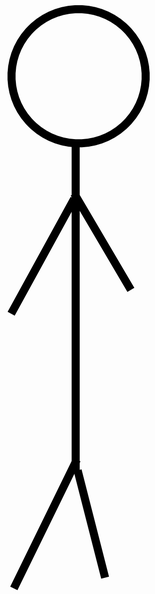
Vincent Archambault-B
IFT 2035 - Université de Montréal

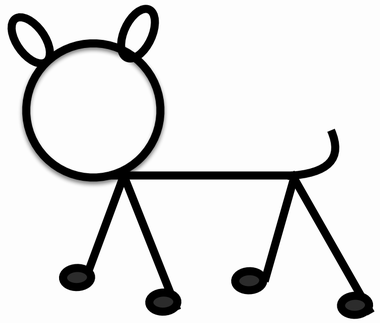
Ce document est dédié au domaine public via CCO

Pour obtenir le code source de ce document

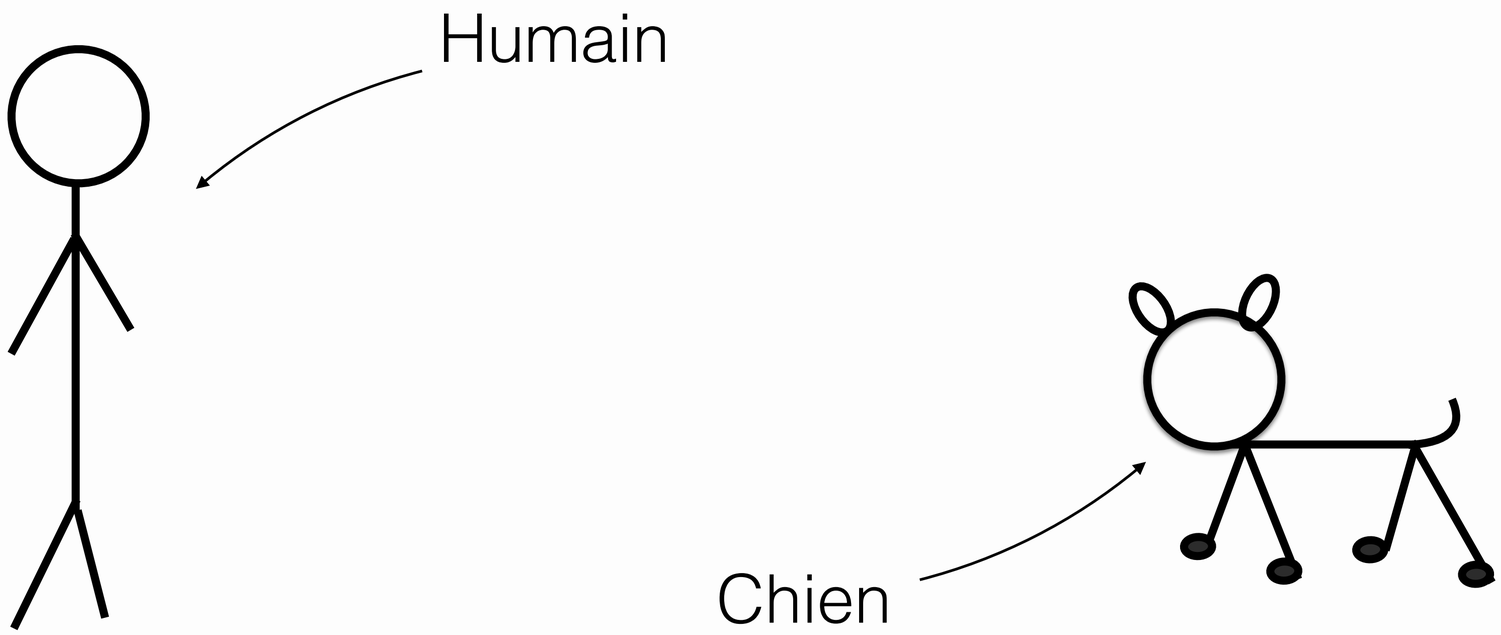
- https://github.com/archambaultv/IFT2035-UdeM
- vincent.archambault-bouffard@umontreal.ca

Comment caractériser les données ?

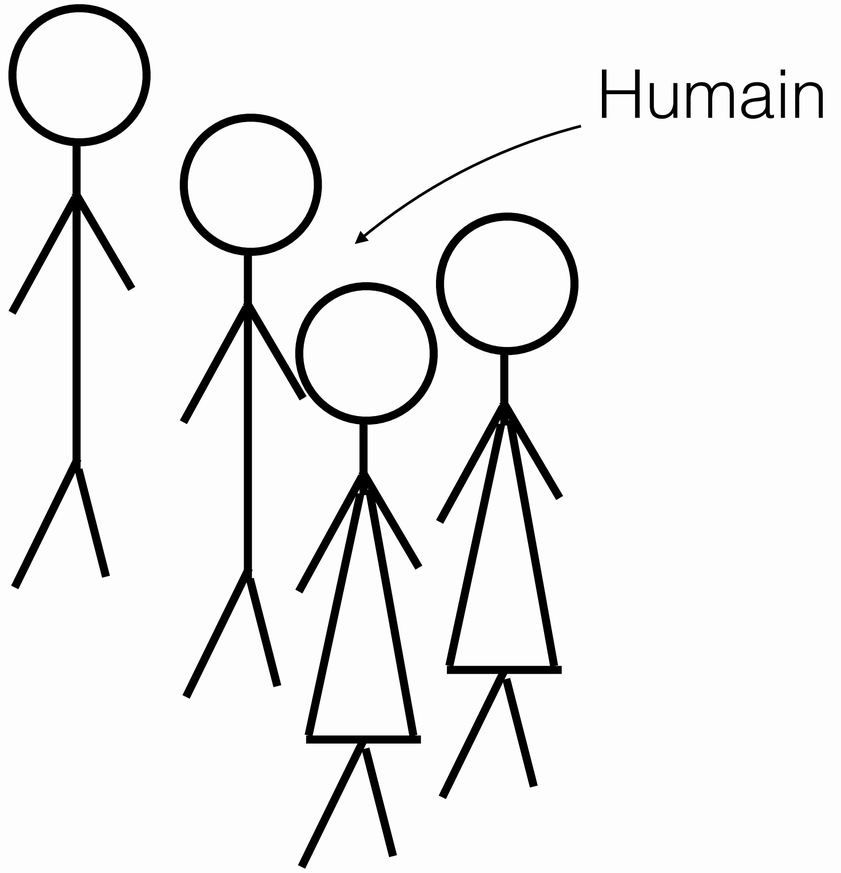


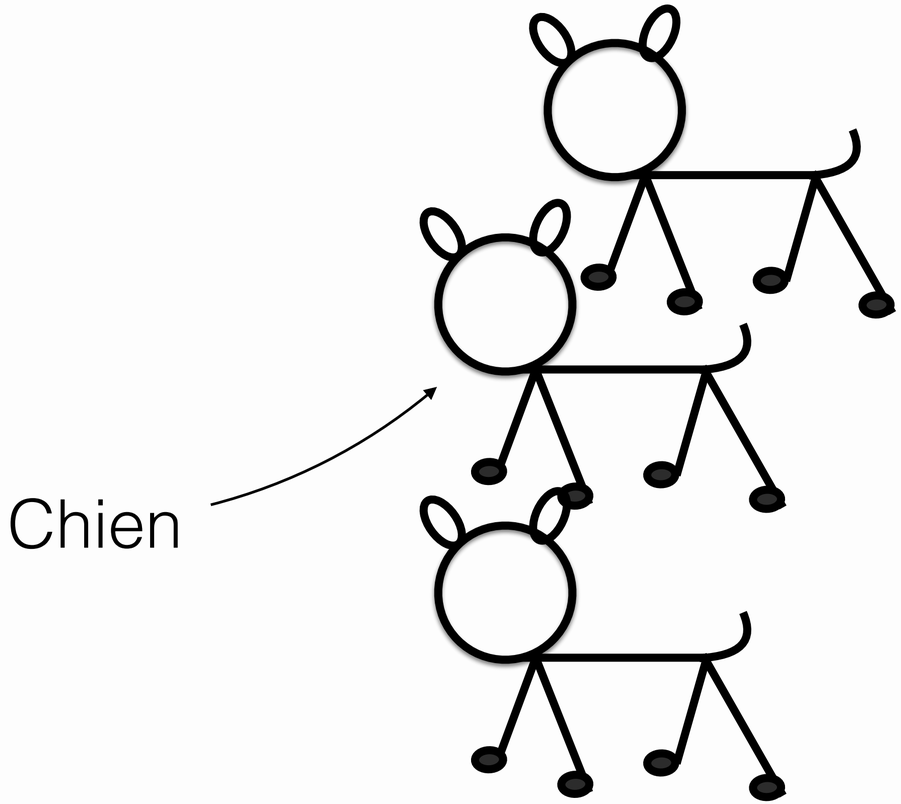


Type : exprime une propriété



Un type regroupe des données ayant une propriété en commun





Un type regroupe des données ayant une propriété en commun

```
1 :: Int
2 :: Int
3 :: Int
4 :: Int
```

```
"" :: [Char]
"Aude" :: [Char]
"Hello" :: [Char]
"World" :: [Char]
```

```
\x -> 1 + x :: Int -> Int
\y -> y + y :: Int -> Int
\y -> y - 3 :: Int -> Int
\z -> z * 5 :: Int -> Int
```

Les données d'un même type peuvent être utilisées au même endroit dans le programme

```
x + 1
x + 2
x + 3
x + 4
```

```
putStrLn ""
putStrLn "Aude"
putStrLn "Hello"
putStrLn "World"
```

```
(\x -> 1 + x) 1
(\y -> y + y) 1
(\y -> y - 3) 1
(\z -> z * 5) 1
```

Un système de types devrait fournir 2 garanties

sécurité Une expression bien typée est soit une valeur finale ou peut progresser dans son évaluation

progrès Une expression bien typée s'évalue vers une expression bien typée du même type

“Well typed programs do not go wrong”

– Robin Milner

“Un programme bien typé ne plante pas”

– Robin Milner

Un système de types évite les bugs qui feraient planter le programme

```
x + "Wrong"
```

```
putStrLn 1
```

```
(\x -> 1 : x) 1
-- x doit être une liste
```

Un système de types évite les bugs qui feraient planter le programme … sauf exception

```
div 1 0 :: Int
*** Exception: divide by zero
```

Types des fonctions Haskell

- Un type pour chaque argument
- Un type pour la valeur de retour
- Un flèche sépare les types

```haskell
foo :: Int -> [Char] -> Bool
foo x s = length s == x

bar :: Int -> Int
bar x = x * x - x

baz :: (Int -> Int) -> Int -> Bool
baz f x = f x == 3
```

Haskell

Polymorphisme (paramétré)

Comment exprimer des types génériques ?

```
x = "Hello World" //Liste de caractères  [Char]
y = [1, 2, 3]     //Liste de nombres     [Int]
z = [(+),(-)]     //Liste d’opérateurs   [Int -> Int -> Int]
```

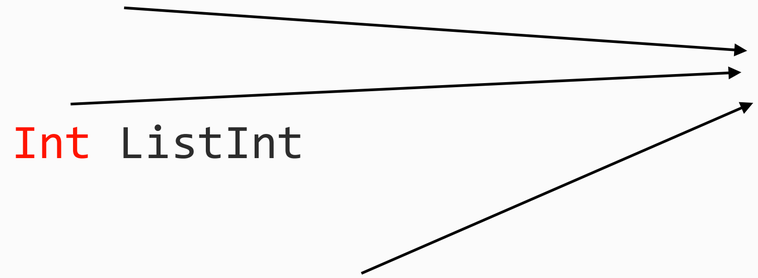
Haskell. (+) signifie la fonction \x y -> x + y. De façon générale, (op) signifie la fonction \x y -> x op y

Comment exprimer des types génériques ?

```
data ListChar = NilC
              | ConsC Char ListChar

data ListInt = NilI
             | ConsI Int ListInt

data ListFunc = NilF
              | ConsF (Int -> Int -> Int) ListFunc
```



Pourquoi ne pas passer le type en argument ?

Liste et polymorphisme

```
data List a = Nil
            | Cons a (List a)

type ListChar = List Char

type ListInt = List Int

type ListFunc = List (Int -> Int -> Int)
```

Haskell. type définit un synonyme, il ne s'agit pas vraiment d'un nouveau type.

Liste et polymorphisme

```
type String = [Char]
```

Haskell. type définit un synonyme, il ne s’agit pas vraiment d’un nouveau type.

Fonctions et polymorphisme

```
id :: a -> a
id x = x

foo :: a -> [a]
foo x = [x, x]

length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```

Haskell. Il suffit d'utiliser une lettre minuscule (variable) pour décrire une fonction générique.

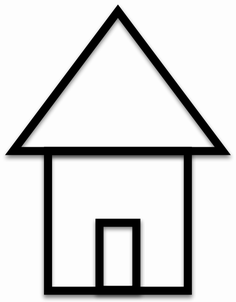
Typeclass (interface)

Typeclass (interface)

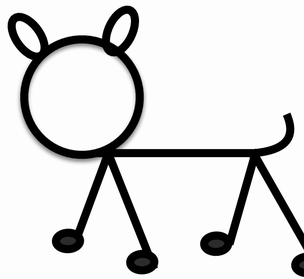
Polymorphisme de sous-typage, interface, typeclass. Chaque langage utilise son propre terme et donne une définition différente.

Le principe reste souvent le même: un type (interface) qui décrit certaines propriétés et la possibilité pour le programmeur de déclarer qu'un ou plusieurs types (data, class, etc.) implémentent ces propriétés.

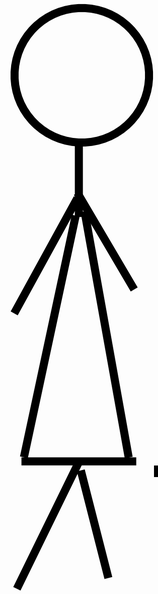
Typeclass (interface)



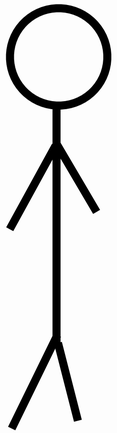
.nom



.nom



.nom



.nom

Typeclass en Haskell

```
-- Tout ce qui peut être convertie en String
class Show a where
  show :: a -> String
```

Typeclass en Haskell

```
-- Tout ce qui peut être convertie en String
class Show a where
  show :: a -> String

data Person = Person String Int

instance Show Person where
  show (Person name age) = name ++ ", " ++ show age ++ " ans"

x = show (Person "Paul" 23) -- "Paul, 23 ans"
```

Typeclass en Haskell

```
-- Possède une notion d’égalité
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Typeclass en Haskell

```
-- Possède une notion d’égalité
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

data Person = Person String Int

instance Eq Person where
  (Person name1 age1) == (Person name2 age2) =
     age1 == age2 && name1 == name2

  x /= y = not (x == y)

x = (Person "Paul" 23) == (Person  "Aude" 23) -- False
```

Typeclass en Haskell

```haskell
compare3 :: (Eq a) => a -> a -> a -> Bool
compare3 x y z = x == y && y == z

bonjour :: (Show a) => a -> String
bonjour x = "Bonjour " ++ show a ++ " !"
```

Vérification et inférence de types

Comment exprimer une règle de typage

$\Gamma \vdash e : \tau$ L'expression e, dans l'environnement Γ, est de type τ

Règle pour les variables

$\Gamma, x : \tau \vdash x : \tau$ Une variable x, dont le type τ est déjà connu dans l'environnement, conserve toujours son type. (Les variables ne changent pas de types)

Règle pour les fonctions

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x : \tau_1 \rightarrow e : \tau_1 \rightarrow \tau_2}$$

Une fonction avec paramètre x dont le type est τ_1 a pour type $\tau_1 \rightarrow \tau_2$ SI l'expression e a pour type τ_2 en supposant que l'environnement courant Γ est augmenté de la variable $x : \tau_1$

Règle pour les fonctions

$\Gamma \vdash (+) : \text{Int} \to \text{Int} \to \text{Int}$

Certaines primitives ont leur règle de typage déjà prédéfinie

Règle pour l'application de fonctions

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

Une application de la fonction e_1 de type $\alpha \to \beta$ avec le paramètre e_2 de type α a pour résultat une donnée de type β

Exemple

- Si la variable y a pour type String:
  - le corps de la fonction ne peut pas être typé car (+) n'accepte que des Int.
  - La condition de la règle de typage n'est pas respectée et le compilateur rapporte une erreur.

FAUX

$$\frac{\Gamma, y : String \vdash (y + 1) : Int}{\Gamma \vdash \lambda\, y : String \rightarrow y + 1 : String \rightarrow Int}$$

```
foo :: String -> Int
foo y = y + 1
```

Haskell : Mauvais type

Exemple

- Si le type de la variable foo n'est pas connu
  - le corps de la fonction doit retourner Int car (+) retourne un Int
  - y doit être de type Int, car il est un paramètre de (+)
  - Alors `foo :: Int -> Int`

$$\frac{\Gamma, y : ?? \vdash (y + 1) : \text{Int}}{\Gamma \vdash \lambda\, y : ?? \rightarrow y + 1 : ?? \rightarrow \text{Int}}$$

```
foo y = y + 1
```