Introduction à la Programmation Fonctionnelle avec Haskell

Yann Esposito

<2018-03-15 Thu>

Introduction à la Programmation Fonctionnelle avec Haskell

main :: IO ()

```
Initialiser l'env de dev
```

Install **stack** :

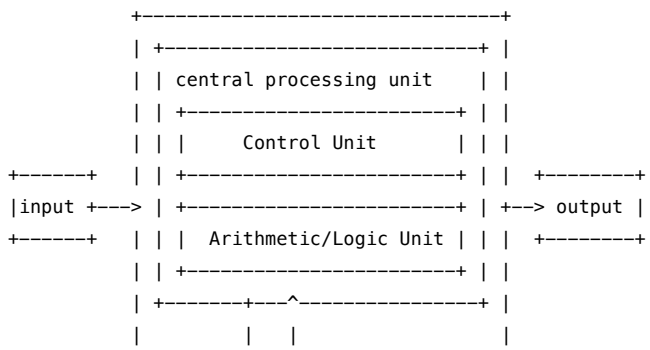
```
curl -sSL https://get.haskellstack.org/ | sh
```

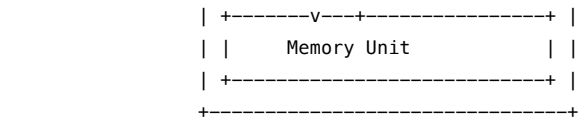
Install **nix** :

```
curl https://nixos.org/nix/install | sh
```

Programmation Fonctionnelle ?

Von Neumann Architecture (1945)





made with http://asciiflow.com

Von Neumann vs Church

- programmer à partir de la machine (Von Neumann)
  - tire vers l'optimisation
  - mots de bits, caches, détails de bas niveau
  - actions séquentielles
  - **1 siècle d'expérience**

. . .

- programmer comme manipulation de symbole (Alonzo Church)
  - tire vers l'abstraction
  - plus proche des représentations mathématiques
  - ordre d'évaluation non imposé
  - **4000 ans d'expérience**

Histoire

- λ-Calculus, Alonzo Church & Rosser 1936
  - Foundation, explicit side effect no implicit state

. . .

- LISP (McCarthy 1960)
  - Garbage collection, higher order functions, dynamic typing

. . .

- ML (1969-80)
  - Static typing, Algebraic Datatypes, Pattern matching

. . .

- Miranda (1986) → Haskell (1992☒)
  - Lazy evaluation, pure

Pourquoi Haskell ?

Simplicité par l'abstraction

/!\ SIMPLICITÉ ≠ FACILITÉ /!

- mémoire (garbage collection)
- ordre d'évaluation (non strict / lazy)
- effets de bords (pur)

- manipulation de code (referential transparency)

. . .

Simplicité : Probablement le meilleur indicateur de réussite de projet.

Production Ready™

- rapide
  - équivalent à Java (~ x2 du C)
  - parfois plus rapide que C
  - bien plus rapide que python et ruby

. . .

- communauté solide
  - 3k comptes sur Haskellers
  - >30k sur reddit *(35k rust, 45k go, 50k nodejs, 4k ocaml, 13k clojure)*
  - libs >12k sur hackage

. . .

- entreprises
  - Facebook (fighting spam, HAXL, ...)
  - beaucoup de startups, finance en général

. . .

- milieu académique
  - fondations mathématiques
  - fortes influences des chercheurs
  - tire le langage vers le haut

Tooling

- compilateur (GHC)
- gestion de projets ; cabal, stack, hpack, etc...
- IDE / hlint ; rapidité des erreurs en cours de frappe
- frameworks hors catégorie (servant, yesod)
- ecosystèmes très matures et inovant
  - Elm (☒ frontend)
  - Purescript (☒ frontend)
  - GHCJS (☒ frontend)
  - Idris (types dépendants)
  - Hackett (typed LISP avec macros)
  - Eta (☒ JVM)

Qualité

> *Si ça compile alors il probable que ça marche*

. . .

- tests unitaires : chercher quelques erreurs manuellements

. . .

- *test génératifs* : chercher des erreurs sur beaucoups de cas générés aléatoirements & aide pour trouver l'erreur sur l'objet le plus simple

. . .

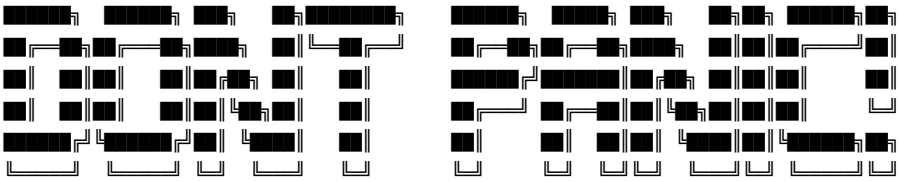
- *finite state machine generative testing* : chercher des erreurs sur le déroulement des actions entre différents agents indépendants

. . .

- **preuves** : chercher des erreur sur **TOUTES** les entrées possibles possible à l'aide du système de typage

Premiers Pas en Haskell

DON'T PANIC



- Haskell peut être difficile à vraiment maîtriser
- Trois languages en un :
  - Fonctionnel
  - Imperatif
  - Types
- Polymorphisme :
  - contexte souvent semi-implicite change le comportement du code.

Fichier de script isolé

Avec Stack : https://haskellstack.org

```
#!/usr/bin/env stack
{- stack script
   --resolver lts-12.10
   --install-ghc
   --package protolude
-}
```

Avec Nix : https://nixos.org/nix/

```
#! /usr/bin/env nix-shell
#! nix-shell -i runghc
#! nix-shell -p "ghc.withPackages (ps: [ ps.protolude ])"
#! nix-shell -I nixpkgs="https://github.com/NixOS/nixpkgs/archive/18.09.tar.gz"
```

Hello World ! (1/3)

```
-- hello.hs
main :: IO ()
main = putStrLn "Hello World!"
```

```
> chmod +x hello.hs
> ./hello.hs
Hello World!
```

```
> stack ghc -- hello.hs
> ./hello
Hello World!
```

Hello World ! (2/3)

```
main :: IO ()
main = putStrLn "Hello World!"
```

- `::` de type ;
- le type de `main` est `IO ()`.
- `=` égalité (la vrai, on peut interchanger ce qu'il y a des deux cotés) ;
- le type de `putStrLn` est `String -> IO ()` ;
- application de fonction `f x` pas `f(x)`, pas de parenthèse nécessaire ;

Hello World ! (3/3)

```
main :: IO ()
main = putStrLn "Hello World!"
```

- Le type `IO a` signifie : C'est une description d'une procédure qui quand elle est évaluée peut faire des actions d'IO qui retournera une valeur de type `a` ;
- `main` est le nom du point d'entrée du programme ;
- Haskell runtime va chercher pour `main` et l'exécute.

What is your name ?

What is your name ? (1/2)

```
main :: IO ()
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
```

```
  let output = "Nice to meet you, " ++ name ++ "!"
  putStrLn output
```

. . .

- l'indentation est importante !
- `do` commence une syntaxe spéciale qui permet de séquencer des actions `IO` ;
- le type de `getLine` est `IO String` ;
- `IO String` signifie : Ceci est la description d'une procédure qui lorsqu'elle est évaluée peut faire des actions IO et retourne une valeur de type `String`.

What is your name ? (2/2)

```
main :: IO ()
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  let output = "Nice to meet you, " ++ name ++ "!"
  putStrLn output
```

- le type de `getLine` est `IO String`
- le type de `name` est `String`
- `<-` est une syntaxe spéciale qui n'apparait que dans la notation `do`
- `<-` signifie : évalue la procédure et attache la valeur renvoyée dans le nom à gauche de `<-`
- `let <name> = <expr>` signifie que `name` est interchangeable avec `expr` pour le reste du bloc `do`.
- dans un bloc `do`, `let` n'a pas besoin d'être accompagné par `in` à la fin.

Erreurs classiques

Erreur classique #1

```
main :: IO ()
main = do
  putStrLn "Hello! What is your name?"
  let output = "Nice to meet you, " ++ getLine ++ "!"
  putStrLn output
```

```
/Users/yaesposi/.deft/pres-haskell/name.hs:6:40: warning: [-Wdeferred-type-errors]
    • Couldn't match expected type ‘[Char]’
                  with actual type ‘IO String’
    • In the first argument of ‘(++)’, namely ‘getLine’
      In the second argument of ‘(++)’, namely ‘getLine ++ "!"’
      In the expression: "Nice to meet you, " ++ getLine ++ "!"
  |
6 |   let output = "Nice to meet you, " ++ getLine ++ "!"
  |                                       ^^^^^^^
Ok, one module loaded.
```

Erreur classique #1

- `String` est `[Char]`
- Haskell n'arrive pas à faire matcher le type `String` avec `IO String`.
- `IO a` et `a` sont différents

Erreur classique #2

```
main :: IO ()
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn  "Nice to meet you, " ++ name ++ "!"
```

```
/Users/yaesposi/.deft/pres-haskell/name.hs:7:3: warning: [-Wdeferred-type-errors]
    • Couldn't match expected type ‘[Char]’ with actual type ‘IO ()’
    • In the first argument of ‘(++)’, namely
        ‘putStrLn "Nice to meet you, "’
      In a stmt of a 'do' block:
        putStrLn "Nice to meet you, " ++ name ++ "!"
      In the expression:
        do putStrLn "Hello! What is your name?"
           name <- getLine
           putStrLn "Nice to meet you, " ++ name ++ "!"
  |
7 |   putStrLn "Nice to meet you, " ++ name ++ "!"
```

Erreur classique #2 (fix)

- Des parenthèses sont nécessaires
- L'application de fonction se fait de gauche à droite

```
main :: IO ()
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn ("Nice to meet you, " ++ name ++ "!")
```

Concepts avec exemples

Concepts

- *style déclaratif & récursif*
- *immutabilité*
- *pureté* (par défaut)
- *evaluation paraisseuse* (par défaut)
- *ADT & typage polymorphique*

Style déclaratif & récursif

```
>>> x=0
... for i in range(1,11):
...     tmp = i*i
...     if tmp%2 == 0:
...       x += tmp
>>> x
220
```

```
-- (.) composition (de droite à gauche)
Prelude> sum . filter even . map (^2) $ [1..10]
220
Prelude> :set -XNoImplicitPrelude
Prelude> import Protolude
-- (&) flipped fn application (de gauche à droite)
Protolude> [1..10] & map (^2) & filter even & sum
220
```

Style déclaratif & récursif

```
>>> x=0
... for i in range(1,11):
...     j = i*3
...     tmp = j*j
...     if tmp%2 == 0:
...       x += tmp
```

```
Prelude> sum . filter even . map (^2) . map (*3) $ [1..10]
Protolude> [1..10] & map (*3) & map (^2) & filter even & sum
```

Style déclaratif & récursif

- Contrairement aux languages impératifs la récursion n'est généralement pas chère.
- tail recursive function, mais aussi à l'aide de la lazyness

Imutabilité

```
-- | explicit recursivity
incrementAllEvenNumbers :: [Int] -> [Int]
incrementAllEvenNumbers (x:xs) = y:incrementAllEvenNumbers xs
  where y = if even x then x+1 else x

-- | better with use of higher order functions
incrementAllEvenNumbers' :: [Int] -> [Int]
incrementAllEvenNumbers' ls = map incrementIfEven ls
  where
```

```haskell
incrementIfEven :: Int -> Int
incrementIfEven x = if even x then x+1 else x
```

Pureté : Function vs Procedure/Subroutines

- Une *fonction* n'a pas d'effet de bord
- Une *Procedure* ou *subroutine* but engendrer des effets de bords lors de son évaluation

Pureté : Function vs Procedure/Subroutines (exemple)

```haskell
dist :: Double -> Double -> Double
dist x y = sqrt (x**2 + y**2)

getName :: IO String
getName = readLine
```

- **IO a ☒ IMPUR** ; effets de bords hors evaluation :
  - lire un fichier ;
  - écrire sur le terminal ;
  - changer la valeur d'une variable en RAM est impur.

Pureté : Gain, paralellisation gratuite

```haskell
import Foreign.Lib (f)
--  f :: Int -> Int
--  f = ???

foo = sum results
  where results = map f [1..100]
```

. . .

`pmap` FTW !!!!! Assurance d'avoir le même résultat avec 32 cœurs

```haskell
import Foreign.Lib (f)
--  f :: Int -> Int
--  f = ???

foo = sum results
  where results = pmap f [1..100]
```

Pureté : Structures de données immuable

Purely functional data structures, *Chris Okasaki*

Thèse en 1996, et un livre.

Opérations sur les listes, tableaux, arbres de complexité amortie equivalent ou proche (pire des cas facteur log(n)) de celle des structures de données muables.

Évaluation parraisseuse : Stratégies d'évaluations

`(h (f a) (g b))` peut s'évaluer :

- `a` → `(f a)` → `b` → `(g b)` → `(h (f a) (g b))`
- `b` → `a` → `(g b)` → `(f a)` → `(h (f a) (g b))`
- `a` et `b` en parallèle puis `(f a)` et `(g b)` en parallèle et finallement `(h (f a) (g b))`
- `h` → `(f a)` seulement si nécessaire et puis `(g b)` seulement si nécessaire

Par exemple : `(def h (λx.λy.(+ x x)))` il n'est pas nécessaire d'évaluer `y`, dans notre cas `(g b)`

Évaluation parraisseuse : Exemple

```
quickSort [] = []
quickSort (x:xs) = quickSort (filter (<x) xs)
                   ++ [x]
                   ++ quickSort (filter (>=x) xs)

minimum list = head (quickSort list)
```

Un appel à `minimum longList` ne vas pas ordonner toute la liste. Le travail s'arrêtera dès que le premier élément de la liste ordonnée sera trouvé.

`take k (quickSort list)` est en `O(n + k log k)` où `n = length list`. Alors qu'avec une évaluation stricte : `O(n log n)`.

Évaluation parraisseuse : Structures de données infinies (zip)

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _  = []
zip _  [] = []
zip (x:xs) (y:ys) = (x,y):zip xs ys

zip [1..] ['a','b','c']
```

s'arrête et renvoie :

```
[(1,'a'), (2,'b'), (3, 'c')]
```

Évaluation parraisseuse : Structures de données infinies (2)

```
Prelude> zipWith (+) [0,1,2,3] [10,100,1000]
[10,101,1002]
Prelude> take 3 [1,2,3,4,5,6,7,8,9]
[1,2,3]

Prelude> fib = 0:1:(zipWith (+) fib (tail fib))
Prelude> take 10 fib
[0,1,1,2,3,5,8,13,21,34]
```

ADT & Typage polymorphique

Algebraic Data Types.

```haskell
data Void = Void Void -- 0 valeur possible!
data Unit = ()        -- 1 seule valeur possible

data Product x y = P x y
data Sum x y = S1 x | S2 y
```

Soit `#x` le nombre de valeurs possibles pour le type `x` alors :

- `#(Product x y) = #x * #y`
- `#(Sum x y)     = #x + #y`

ADT & Typage polymorphique : Inférence de type

À partir de :

```haskell
zip [] _  = []
zip _  [] = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

le compilateur peut déduire :

```haskell
zip :: [a] -> [b] -> [(a,b)]
```

Composabilité

Composabilité vs Modularité

Modularité : soit un `a` et un `b`, je peux faire un `c`. ex : x un graphique, y une barre de menu => une page `let page = mkPage ( graphique, menu )`

Composabilité : soit deux `a` je peux faire un autre `a`. ex : x un widget, y un widget => un widget `let page = x <+> y`

Gain d'abstraction, moindre coût.

Hypothèses fortes sur les `a`

Exemples

- **Semi-groupes** $\langle + \rangle$
- **Monoides** $\langle 0, + \rangle$
- **Catégories** $\langle \text{obj(C)}, \text{hom(C)}, \boxtimes \rangle$
- Foncteurs `fmap` (`(<$>)`)
- Foncteurs Applicatifs `ap` (`(<*>)`)
- Monades `join`

- Traversables `map`
- Foldables `reduce`

Catégories de bugs évités avec Haskell

Real Productions Bugs™

Bug vu des dizaines de fois en prod malgré :

1. specifications fonctionnelles
2. spécifications techniques
3. tests unitaires
4. 3 envs, dev, recette/staging/pre-prod, prod
5. Équipe de QA qui teste en recette

Solutions simples.

Null Pointer Exception : Erreur classique (1)

Au début du projet :

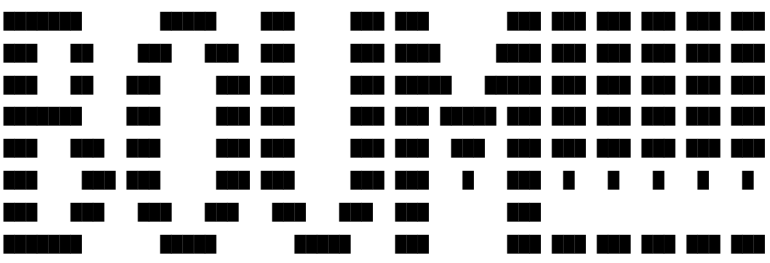
```
int foo( x ) {
  return x + 1;
}
```

Null Pointer Exception : Erreur classique (2)

Après quelques semaines/mois/années :

```
import do_shit_1 from "foreign-module";
int foo( x ) {
  ...
  var y = do_shit_1(x);
  ...
  return do_shit_20(y)
}
...
var val = foo(26/2334 - Math.sqrt(2));
```

. . .



Null Pointer Exception

Null Pointer Exception : Data type `Maybe`

```haskell
data Maybe a = Just a | Nothing
...
foo :: Maybe a
...
myFunc x = let t = foo x in
  case t of
    Just someValue -> doThingsWith someValue
    Nothing -> doThingWhenNothingIsReturned
```

Le compilateur oblige à tenir compte des cas particuliers ! Impossible d'oublier.

Null Pointer Exception : Etat

- Rendre impossibe de fabriquer un état qui devrait être impossible d'avoir.
- Pour aller plus loin voir, FRP, CQRS/ES, Elm-architecture, etc...

Erreur due à une typo

```haskell
data Foo x = LongNameWithPossibleError x
...
foo (LongNameWithPosibleError x) = ...
```

Erreur à la compilation : Le nom d'un champ n'est pas une string (voir les objets JSON).

Echange de parameters

```haskell
data Personne = Personne { uid :: Int, age :: Int }
foo :: Int -> Int -> Personne -- ??? uid ou age?
```

```haskell
newtype UID = UID Int deriving (Eq)
data Personne = Personne { uid :: UID, age :: Int }
foo :: UDI -> Int -> Personne -- Impossible de confondre
```

Changement intempestif d'un Etat Global

```haskell
foo :: GlobalState -> x
```

`foo` ne peut pas changer `GlobalState`

Organisation du Code

Grands Concepts

Procedure vs Functions :

Gestion d'une configuration globale
Gestion d'un état global
Gestion des Erreurs
Gestion des IO

Monades

Pour chacun de ces *problèmes* il existe une monade :

Gestion d'une configuration globale	`Reader`
Gestion d'un état global	`State`
Gestion des Erreurs	`Either`
Gestion des IO	`IO`

Effets

Gestion de plusieurs Effets dans la même fonction :

- MTL
- Free Monad
- Freer Monad

Idée : donner à certaines sous-fonction accès à une partie des effets seulement.

Par exemple :

- limiter une fonction à la lecture de la DB mais pas l'écriture.
- limiter l'écriture à une seule table
- interdire l'écriture de logs
- interdire l'écriture sur le disque dur
- etc...

Exemple dans un code réel (1)

```haskell
-- | ConsumerBot type, the main monad in which the bot code is written with.
-- Provide config, state, logs and IO
type ConsumerBot m a =
  ( MonadState ConsumerState m
  , MonadReader ConsumerConf m
  , MonadLog (WithSeverity Doc) m
  , MonadBaseControl IO m
```

```
  , MonadSleep m
  , MonadPubSub m
  , MonadIO m
  ) => m a
```

Exemple dans un code réel (2)

```
bot :: Manager
    -> RotatingLog
    -> Chan RedditComment
    -> TVar RedbotConfs
    -> Severity
    -> IO ()
bot manager rotLog pubsub redbots minSeverity = do
  TC.setDefaultPersist TC.filePersist
  let conf = ConsumerConf
             { rhconf = RedditHttpConf { _connMgr = manager }
             , commentStream = pubsub
             }
  void $ autobot
       & flip runReaderT conf
       & flip runStateT (initState redbots)
       & flip runLoggingT (renderLog minSeverity rotLog)
```

Règles pragmatiques

Organisation en fonction de la complexité

> Make it work, make it right, make it fast

- Simple : directement IO (YOLO !)
- Medium : Haskell Design Patterns : The Handle Pattern : https://jaspervdj.be/posts/2018-03-08-handle-pattern.html
- Medium (bis) : MTL / Free / Freeer / Effects...
- Gros : Three Layer Haskell Cake : http://www.parsonsmatt.org/2018/03/22/three_layer_haskell_cake.html
  - Layer 1 : Imperatif
  - Orienté Objet (Level 2 / 2')
  - Fonctionnel

3 couches

- **Imperatif** : ReaderT IO
  - Insérer l'état dans une `TVar`, `MVar` ou `IORef` (concurrence)
- **Orienté Objet** :
  - Handle / MTL / Free...
  - donner des access `UserDB`, `AccessTime`, `APIHTTP`...

- **Fonctionnel** : Business Logic `f : Handlers -> Inputs -> Command`

Services / Lib

Service : `init` / `start` / `close` + methodes... Lib : methodes sans état interne.

Conclusion

Pourquoi Haskell ?

- avantage compétitif : qualité & productivité hors norme
- change son approche de la programmation
- les concepts appris sont utilisables dans tous les languages
- permet d'aller là où aucun autre langage ne peut vous amener
- Approfondissement sans fin :
  - Théorie : théorie des catégories, théorie des types homotopiques, etc...
  - Optim : compilateur
  - Qualité : tests, preuves
  - Organisation : capacité de contraindre de très haut vers très bas

Avantage compétitif

- France, Europe du sud & Functional Programming
- Coût Maintenance » production d'un nouveau produit
- Coût de la refactorisation
- "Make it work, Make it right, Make it fast" moins cher.

Pour la suite

A chacun de choisir, livres, tutoriels, videos, chat, etc...

- Voici une liste de resources : https://www.haskell.org/documentation
- Mon tuto rapide : Haskell the Hard Way
- Moteurs de recherche par type : hayoo & hoogle
- Communauté & News : http://haskell.org/news & `#haskell-fr` sur freenode
- Libs : https://hackage.haskell.org & https://stackage.org

Appendix

STM : Exemple (Concurrence) (1/2)

```
class Account {
  float balance;
  synchronized void deposit(float amount){
    balance += amount; }
  synchronized void withdraw(float amount){
    if (balance < amount) throw new OutOfMoneyError();
```

```
    balance -= amount; }
  synchronized void transfert(Account other, float amount){
    other.withdraw(amount);
    this.deposit(amount); }
}
```

Situation d'interblocage typique. (A transfert vers B et B vers A).

STM : Exemple (Concurrence) (2/2)

```
deposit :: TVar Int -> Int -> STM ()
deposit acc n = do
  bal <- readTVar acc
  writeTVar acc (bal + n)
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do
  bal <- readTVar acc
  if bal < n then retry
  writeTVar acc (bal - n)
transfer :: TVar Int -> TVar Int -> Int -> STM ()
transfer from to n = do
  withdraw from n
  deposit to n
```

- pas de lock explicite, composition naturelle dans `transfer`.
- si une des deux opération échoue toute la transaction échoue
- le système de type force cette opération a être atomique : `atomically :: STM a -> IO a`