Introduction à la Programmation Fonctionnelle en Haskell

Yann Esposito

<2018-03-15 Thu>

Contents

1 Courte Introduction

1.1 Prelude

Initialiser l'env de dev:

```
curl -sSL https://get.haskellstack.org/ | sh
stack new ipfh https://git.io/vbpej && \
cd ipfh && \
stack setup && \
stack build && \
stack test && \
stack bench
```

1.2 Parcours jusqu'à Haskell

1.2.1 Parcours Pro

- Doctorat (machine learning, hidden markov models) 2004
- Post doc (écriture d'un UI pour des biologistes en Java). 2006
- Dev Airfrance, (Perl, scripts shell, awk, HTML, CSS, JS, XML…) 2006 → 2013
- Dev (ruby, C, ML) pour GridPocket. (dev) 2009 → 2011, (impliqué) 2009 →
- Clojure dev & Machine Learning pour Vigiglobe. 2013 → 2016
- Senior Clojure développeur chez Cisco. 2016 →

1.2.2 Langages de programmations basiques

1. BASIC (MO5, Amstrad CPC 6129, Atari STf)
2. Logo (école primaire, + écriture d'un cours en 1ère année de Fac)
3. Pascal (lycée, fac)
4. C (fac)
5. ADA (fac)

1.2.3 Langages de programmations orientés objet

1. C++ (fac + outils de recherche pour doctorat)
2. Eiffel (fac)
3. Java (fac, UI en Java 1.6, Swing pour postdoc)
4. Objective-C (temps personnel, app iPhone, app Mac, Screensavers)

1.2.4 Langages moderne de script

1. PHP (fac, site perso)
2. Python (fac, projets perso, jeux, etc…)
3. Awk (fac, Airfrance, …)
4. Perl (Airfrance…)
5. Ruby (GridPocket, site perso v2)

6. Javascript:
    - *Airfrance* basic prototype, jquery, etc..,
    - spine.js
    - backbone.js
    - Coffeescript
    - Cappuccino (Objective-J)
    - Sproutcore
    - *Vigiglobe* actionhero (nodejs), angularjs v1

1.2.5 Langage peu (re)connus

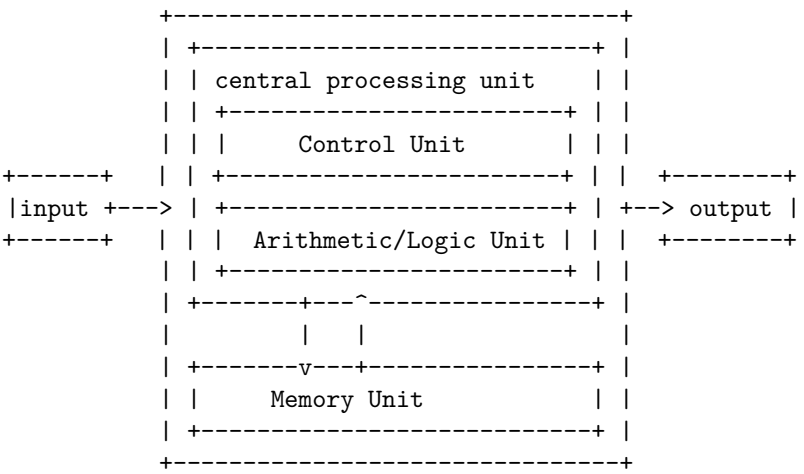
1. Metapost
2. zsh (quasi lang de prog)
3. prolog

1.2.6 Langages fonctionnels

1. CamL
2. Haskell (Vigiglobe, personnal)
3. Clojure (Vigiglobe, Cisco)

1.3 Qu'est-ce que la programmation fonctionnelle?

1.3.1 Von Neumann Architecture



made with http://asciiflow.com

1.3.2 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**

1.3.3 Histoire

- -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

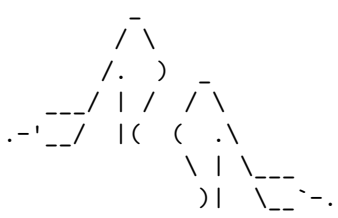
. . .

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

1.3.4 Retour d'expérience subjectif

pieds nus (code machine, ASM)

. . .



Talons hauts (C, Pascal, Java, C++, Perl, PHP, Python, Ruby, etc…)

. . .

Tennis (Clojure, Scheme, LISP, etc…)

. . .

Voiture (Haskell, Purescript, etc…)

1.4 Pourquoi Haskell?

1.4.1 Simplicité par l'abstraction

/!\ SIMPLICITÉ FACILITÉ /!

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

1.4.2 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

1.4.3 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)

1.4.4 Qualité

Si ça compile alors il probable que ça marche

. . .

- test unitaires : chercher quelques erreurs manuellements

. . .

- *test génératifs* : chercher des erreurs sur beaucoups de cas générés aléatoirement & 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

2 Premiers Pas en Haskell

2.0.1 Hello World! (1/3)

```
module Main where

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

file:~/.deft/pres-haskell/hello.hs

2.0.2 Hello World! (2/3)

```
module Main where

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

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

2.0.3 Hello World! (3/3)

```
module Main where

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 et finalement retourne une valeur de type `a` ;
- `main` est le nom du point d'entrée du programme ;
- Haskell runtime va chercher pour `main` et l'exécute.

2.1 What is your name?

2.1.1 What is your name? (1/3)

```haskell
module Main where

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

file:pres-haskell/name.hs

2.1.2 What is your name? (2/3)

```haskell
module Main where

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 à la fin retourne une valeur de type `String`.

2.1.3 What is your name? (3/3)

```haskell
module Main where
```

```haskell
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.

2.2 Erreurs classiques

2.2.1 Erreur classique #1

```haskell
module Main where

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.
```

2.2.2 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

2.2.3 Erreur classique #2

```haskell
module Main where

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 ++ "!"
```

2.2.4 Erreur classique #2 (fix)

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

```haskell
module Main where

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

3 Concepts avec exemples

3.0.1 Concepts

- *pureté* (par défaut)
- *evaluation paraisseuse* (par défaut)
- *ADT & typage polymorphique*

3.0.2 *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

3.0.3 *Pureté*: Function vs Procedure/Subroutines (exemple)

```
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.

3.0.4 *Pureté*: Gain, paralellisation gratuite

```
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

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

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

3.0.5 *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.

3.0.6 *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 ( x. y.(+ x x)))` il n'est pas nécessaire d'évaluer `y`, dans notre cas `(g b)`

3.0.7 *Évaluation parraisseuse*: Exemple 1

```
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)`.

3.0.8 *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')]
```

3.0.9 *ADT & Typage polymorphique*

Algebraic Data Types.

```
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`

3.0.10 *ADT & Typage polymorphique*: Inférence de type

À partir de :

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

le compilateur peut déduire:

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

3.1 Composabilité

3.1.1 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`

3.1.2 Exemples

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

4 Catégories de bugs évités avec Haskell

4.0.1 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.

4.0.2 Null Pointer Exception: Erreur classique (1)

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

4.0.3 Null Pointer Exception: Erreur classique (2)

```
int foo( x ) {
  ...
  var y = do_shit_1(x);
  ...
  return do_shit_20(x)
```

```
}
...
var val = foo(26/2334 - Math.sqrt(2));

. . .
```

```
888888b.    .d88888b.  888     888 888b     d888 888 888 888 888 888 888
888  "88b  d88P" "Y88b 888     888 8888b   d8888 888 888 888 888 888 888
888  .88P  888     888 888     888 88888b.d88888 888 888 888 888 888 888
8888888K.  888     888 888     888 888Y88888P888 888 888 888 888 888 888
888  "Y88b 888     888 888     888 888 Y888P 888 888 888 888 888 888 888
888    888 888     888 888     888 888  Y8P  888 Y8P Y8P Y8P Y8P Y8P Y8P
888   d88P Y88b. .d88P Y88b. .d88P 888   "   888  "   "   "   "   "
8888888P"   "Y88888P"   "Y88888P"  888       888 888 888 888 888 888 888
```

Null Pointer Exception

4.0.4 Null Pointer Exception: Data type `Maybe`

```
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.

4.0.5 Null Pointer Excepton: Etat

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

4.0.6 Erreur due à une typo

```
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).

4.0.7 Echange de parameters

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

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

4.0.8 Changement intempestif d'un Etat Global

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

`foo` ne peut pas changer `GlobalState`

5 Organisation du Code

5.0.1 Grands Concepts

Procedure vs Functions:

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

5.0.2 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`

5.0.3 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...

5.0.4 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
```

5.0.5 Exemple dans un code réel (2)

```haskell
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)
```

5.1 Règles pragmatiques

5.1.1 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

5.1.2 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`

5.1.3 Services / Lib

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

6 Conclusion

6.0.1 Pourquoi Haskell?

- avantage compétitif: qualité x productivité hors norme
- changera 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

6.0.2 Avantage compétitif

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

7 Appendix

7.0.1 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).

7.0.2 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`