zenika

De JavaScript à Haskell

Le tournant de la rigueur

Loïc TRUCHOT - Mars 2022

SOMMAIRE

ON FAIT EN FONCTION

Ce que je savais déjà faire en JavaScript

JavaScript est un semi-LISP

inspiré par Scheme

- Scheme est un LISP
  - 1er langage Functional Programming (FP)
  - en LISP, tout est fonction
  - permet les “lambda”
  - fonction la plus importante: la liste
  - bcp de fonctions pour manipuler listes:
    - map, filter, reduce, sort, groupBy...
    - Higher Order Function (HOF)

```
const persons = [
  { name: 'loic', age: 37 },
  { name: 'sonemany', age: 34 },
  { name: 'swann', age: 2 },
];

const adultAgesOnly = persons
  .map((person) => person.age)
  .filter((age) => age > 18);

const adultAgesTotal = adultAgesOnly.reduce((total, age) => total + age, 0);

const average = adultAgesTotal / adultAgesOnly.length;

console.log(average); // 35.5
```

Si tout est fonction...

Rien n'est autre chose

- Pas d'opérateur: `+` est fonction `(+ 30 12)`
- Pas de mot-clé: `if` est fonction, pas de `for`
- Pas d'instruction, que des expressions
- Tout est "citoyen de première classe"
  - Tout peut-être "variabilisé"
    - Tout peut-être passé/retourné
      - donc callbacks
      - donc currying

```
const add = (a, b) => a + b;
const divide = (a, b) => a / b;
const gt = a => b => b > a;
const prop = (k) => (o) => o[k];

const adultAgesOnly = persons
  .map(prop('age'))
  .filter(gt(18));
```

```
const adultAgesTotal = adultAgesOnly.reduce(add, 0);

const average = divide(adultAgesTotal, adultAgesOnly.length);
console.log(average); // 35.5
```

Aucune classe

Pas de this, pas de méthodes

```
const length = arr => arr.length;
const map = f => arr => arr.map(f);
const filter = f => arr => arr.filter(f);
const reduce = (f, init) => arr => arr.reduce(f, init);
const last = arr => arr[arr.length - 1];
const scanl = (f, init) => arr => arr.reduce((acc, cur) => [...acc, f(last(acc), cur)], [init]);
const tail = arr => arr.slice(1);
const sum = reduce(add, 0);
const scannedSum = scanl(add, 0);

const extractAges = map(prop('age'));
const takeAdultAges = filter(gt(18));

const adultAgesOnly = takeAdultAges(extractAges(persons));
const adultAgesTotal = tail((scannedSum(adultAgesOnly)));

const average = divide(last(adultAgesTotal), length(adultAgesTotal));
console.log(average); // 35.5 -> oh no, side effect is forbidden !!!
```

- Pas de getters, pas de setters
- Pas d'effets de bords
- Immutabilité généralisée
- LISP: Lots of Irritating and Silly Parentheses

Composition et combinateurs

Maths et lambda calculs à la rescousse

- Science pour combinaison des fonctions: Lambda Calculus
- combinateurs classiques déjà nommés (Church, Curry, Smullyan…)
  - compose, pipe, ap, map, lift…
  - et associé à des structures de données… @see fantasy-land
- Science qui permet de vérifier équivalence de signatures, et les “réduire”

```
(a => a) == (b => b)

[1, 2, 3].map((a => a)) == (a => a)([1, 2, 3])

drop(1) == tail;

compose(tail, scanl) == scan;
```

```
const compose = (f, g) => a => f(g(a));
const liftA2 = f => (g, h) => a => f(g(a))(h(a));
const divideBy = a => b => divide(b, a);
const getAverage = liftA2(divideBy)(length, last);

const extractAdultAges = compose(takeAdultAges, extractAges)
const getScannedAdultAges = compose(
 tail,
 compose(scannedSum, extractAdultAges),
);
const getAdultsAgesAverage = compose(
 getAverage,
 getScannedAdultAges,
);

const average = getAdultsAgesAverage(persons);

console.log(average); // 35.5 -> side effect still forbidden
```

CURRY ROSE

Même chose mais en mieux: lambda calculus typé

Lambda Calculus

Pourquoi tant de parenthèses ?

- Alonzo Church, année 30
- Première formalisation d'un système fonctionnel basé sur la recursion
- Un programme est une fonction qui appelle des fonctions jusqu'à résolution
- Équivalent machine de Turing
- Auto-currying généralisé, application partielle généralisée

Le saviez-vous ?
`Y combinator` ça vient de là ! (https://news.ycombinator.com/, aka Hacker News)

```
I Idiot          identity     λa.a              a => a
K Kestrel        const        λab.a             a => b => a
B Bluebird       compose      λabc.a(bc)        a => b => c => a(b(c))
P Phoenix        lift         λabcd.a(bd)(cd)   a => b => c => d => a(b(d))(c(d))
```

- Finalement: =>, () , même λ représente du "bruit" dans un langage fonctionnel authentique

```javascript
// javascript
const identity = x => x;
identity(42) // 2
```

```haskell
-- haskell
identity x = x
identity 2 -- 2
id 2 -- 2
```

```javascript
//javascript
const add = a => b => a + b;
const mul = a => b => a * b;
const compose = f => g => a => f(g(a));
const doubleAndInc = compose(add(1), mul(2));
doubleAndInc(4); // 9
```

```haskell
-- haskell
compose f g x = f (g x)
(.) f g x = f (g x)
doubleAndInc = compose (1 +) (2 *)
doubleAndInc = (1 +) . (2 *)
doubleAndInc 4 -- 9
```

Haskell 101

fonctions et stdlib

- La fin du bruit, au prix de nouvelles conventions
- curry gratuit pour tout le monde, yum yum
- <function_name> arg1 arg2 = <returned value>
- pas de “signe” réservé
- dépasser LISP avec les infixs
  - `(+) 1 2` == `1 + 2`
  - infix non terminé = application partielle
  - `(1 +) 2` == 3
  - `inc x = (1 +) x`
  - `inc 2` == 3
  - `inc = (1 +)` était suffisant en fait
- dépasser la notion de méthode avec les infixs
  - JS: `[1, 2, 3].includes(2) // true`
  - haskell 1: `elem 2 [1,2,3]  -- True`
  - haskell 2: `` 2 `elem` [1,2,3]  -- True ``

Haskell 102

Types, main, variables

Fortement et statiquement typé

- Si type oublié, inférence de type
- “a”, “b”, “c” pour “tout et n’imp”, générique
- Classiques: Bool, Int, String (ou [Char]), Float…
- Num a => a pour les typeclasses (Int et Float sont Num)

```
module Main where

doubleAndInc :: Int -> Int
doubleAndInc = (1 +) . (2 *)

main = do
        let result = doubleAndInc 4
        print result -- 9
```

Scaffolding minimal

- Le programme haskell à un point de montage:
  - la fonction main dans le module Main.hs
    - “do” notation pour dénoter une suite d’instruction. Une suite de QUOI ??? chut.
    - “let” pour déclarer une variable dans ce scope
    - ici, main fait un effet de bord ne retourne rien. Un effet de QUOI ??? chut.
- Un fichier = un module
- stdlib:
  - environ 200 functions (ex: print, elem, ., +, $…)
    https://www.haskell.org/onlinereport/prelude-index.html
  - quelques “expressions” utilitaires qui cachent des fonctions (let, if-then-else, case of…)

IO MAN, C'EST DE LA PURE

Ce qu'implique la fin des effets de bord

« Rien ne se perd, rien ne se crée : tout se transforme ».
Lavoisier (1743-1794)

Une suite de QUOI ???

Des effets de QUOI???

Si les effets de bords étaient impossibles

- Pas de hello-world
- Pas de programme tout court
- Pas d'ordinateur tout court
- le monde réel est-il un méga effet de bord ?
- Et notre console.log() ?

```
module Main where

doubleAndInc :: Int -> Int
doubleAndInc = (1 +) . (2 *)

jsAdd :: (Int, Int) -> Int
jsAdd (a, b) = a + b

main :: IO ()
main = do
        let result = doubleAndInc $ jsAdd (1, 3)
        print result  -- ???
```

Effets de bord encadrés

- Les effets de bords ne peuvent pas avoir lieu si le type ne le précise pas
- Ce type est IO, et prend l'argument de ce qui est retourné
  - IO Int, IO [Char]...
- Si un programme ne retourne rien, il peut retourner un tuple vide à défaut: ()
- Un tuple est une liste de taille fixe, dont chaque membre est typé de façon statique

Bien

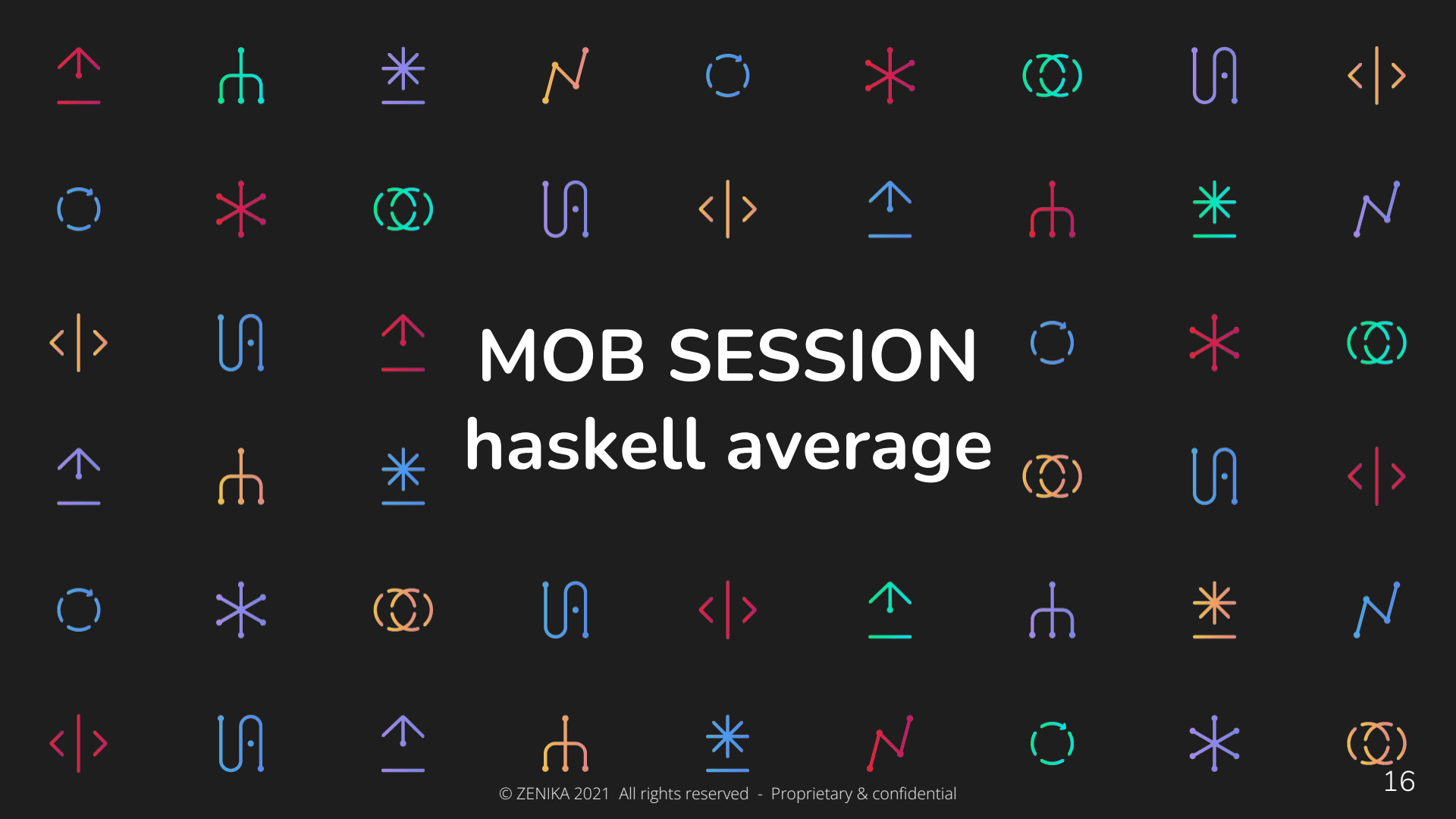
La fin des effets de bords

- Computer **Science**
- Des fonctions sûres et sans surprises
- Une refacto très aisée
- Autocompletion de compositions
  - `putStrLn . show == print`
- Documentation optimum
- Hoogle et le partage de signature
- Memoization
- Lazy evaluation
- Un type pour les side effects: IO

Pas bien

La fin des effets de bords

- **Computer** Science
- Ticket d'entrée trop cher
- Produit trop cher
- Dev trop cher
- Marché trop niche
- Perfs < C, Rust, Java, jeux vidéo…
- IO hell ? Chassez le naturel il revient au galop



UNE BELLE PROMESSE

La fin de la complexité accidentelle

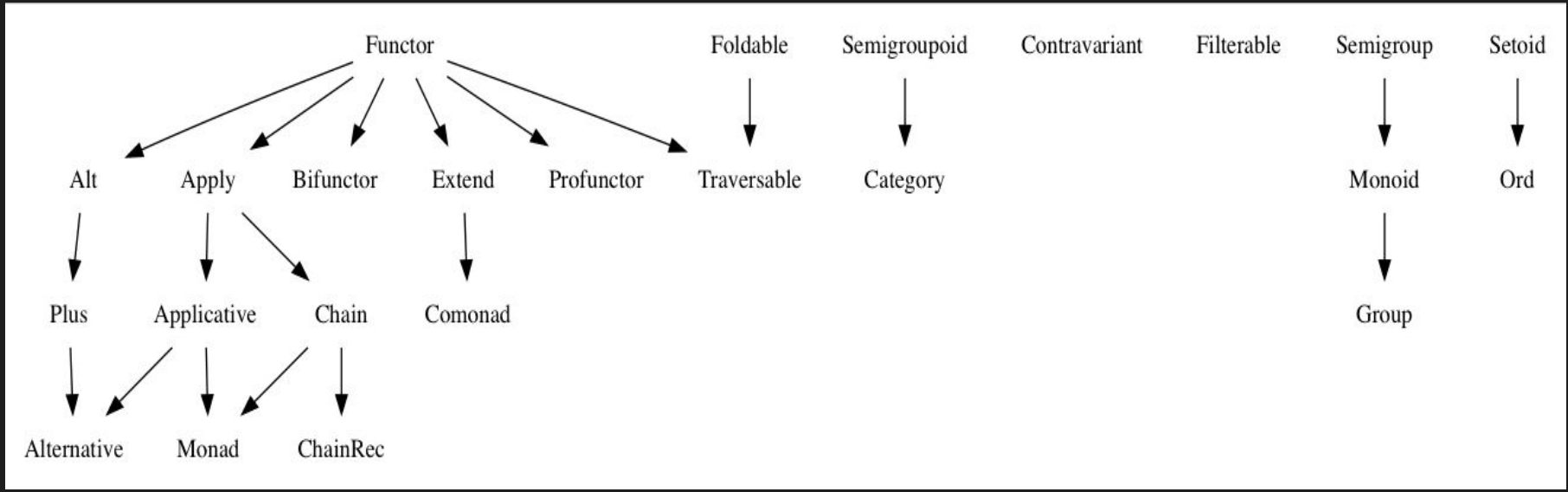
Typeclasses 101

- En JS, nous avons la Promesse
- Rien en commun avec l'Observable
- Ni avec la liste (Array)
- Ni avec le tuple
- Ni avec Optional (Maybe)
- Ni avec Result (Either)
- Pourtant, ces structures partages
  - map
  - flatMap
  - lift
  - ap
  - of
  - un tas d'helpers fonctionnels
- Seuls les mathématiciens qui résonnent en catégorie le savent…
  - + les haskell devs

```
import { promises as fsp} from "fs";

fsp.readFile("./data/persons.json")
  .then(JSON.parse)
  .then(getAdultsAgesAverage)
  .then(console.log)
```

Bienvenue à fantasy land



https://github.com/fantasyland/fantasy-land

Typeclass 102

À peu près conforme à fantasy-land

```
                        Bounded a
                        Enum a
                        Eq a
          (Fractional a) => Floating a
                 (Num a) => Fractional a
                        Functor f
        (Real a, Enum a) => Integral a
                        Monad m
          (Eq a, Show a) => Num a
                  (Eq a) => Ord a
                        Read a
          (Num a, Ord a) => Real a
(RealFrac a, Floating a) => RealFloat a
   (Real a, Fractional a) => RealFrac a
                        Show a
```

- Class ou pas class ? Vraie classes ?

  “It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.”
  Alan Perlis' Epigrams on Programming (1982).

- Chaque typeclass apporte son lot du methodes
- Les maths aussi ça change: système qui évolue encore

MOB SESSION
async average

CRYPTO-FASCIME

La barrière des symboles et noms cryptiques

TITRE

Sous-titre

Corps

MERCI