

# Programmation fonctionnelle et principe de réalité: gérer les effets

Xavier Van de Woestyne • @vdwxv • <https://xvw.github.io> • Margo Bank

“Les langages de programmation fonctionnelle ne sont **pas** utilisables pour des programmes du monde réel”

**Beaucoup de programmeurs**

“Les langages de programmation fonctionnelle ne sont **pas** utilisables pour des programmes du monde réel”

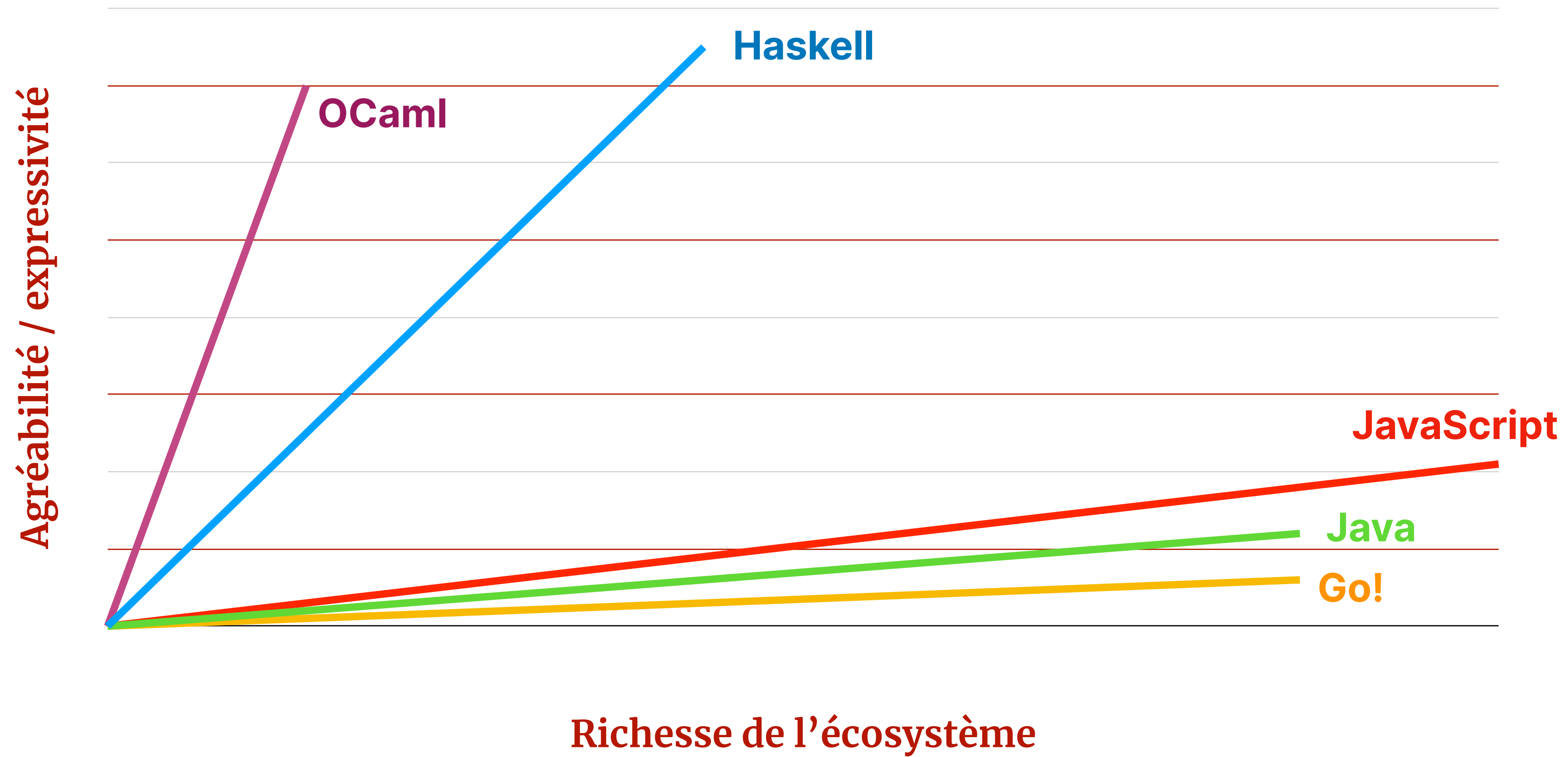
\*pure

Le fameux monde réel

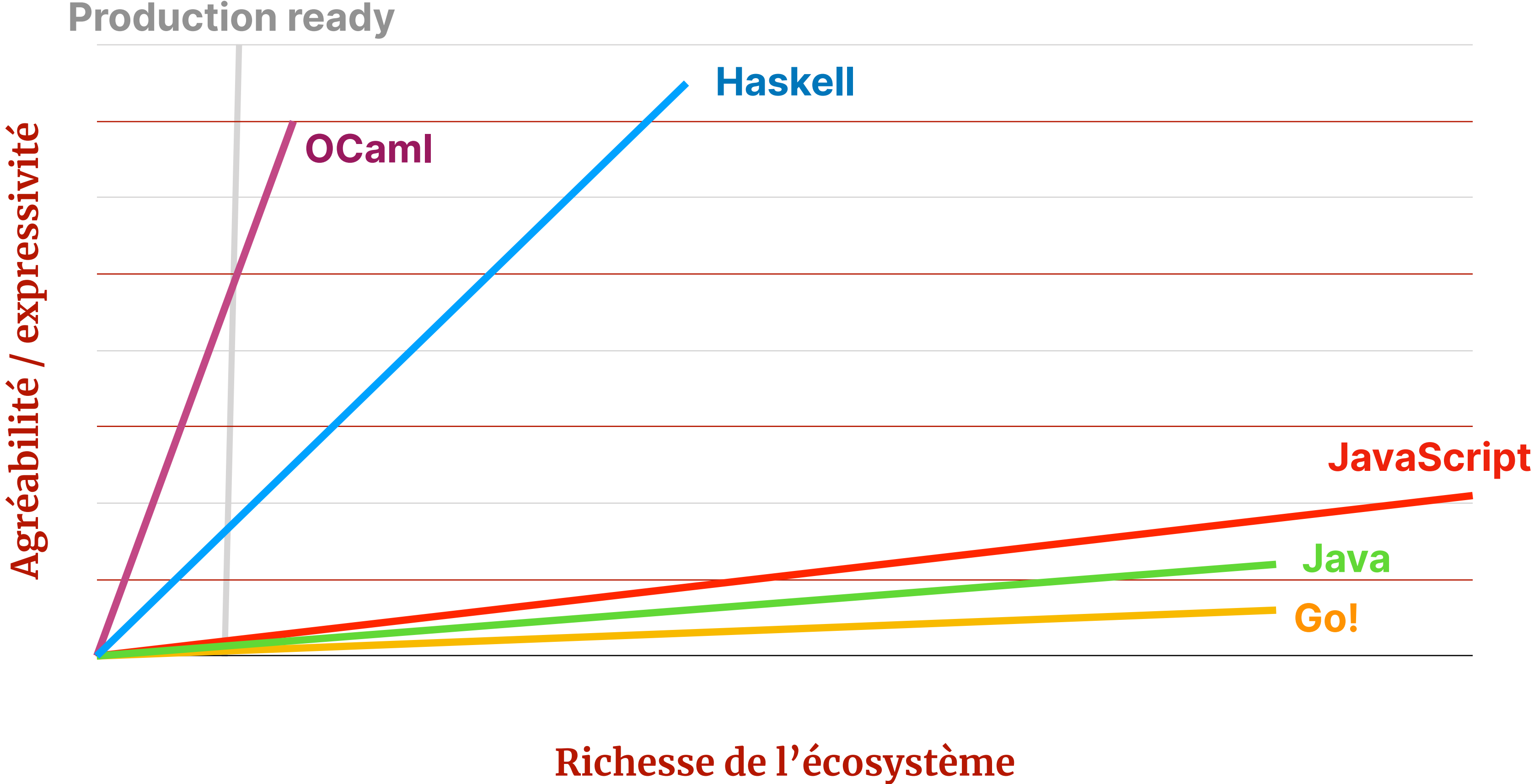
Beaucoup de programmeurs

... du monde réel, qui n'ont généralement jamais utilisé de langage de programmation fonctionnelle pure...

# Une vision très (très) personnelle



# Une vision très (très) personnelle



**Cependant, même si la recherche ne “serait pas”  
*Production-ready...***

- Intégration des **lambda's** dans la majeure partie des langages
- Ajout de **système de types** (graduel ou non)
- Ajout de manières de mimer les **types algébriques** (familles scellées)
- C#, Java, Kotlin, Scala, TypeScript, Rust, Go!

## Objectifs de la présentation

- Comprendre ce qu'est un langage de programmation fonctionnelle
- Différencier les effets et les calculs
- Comprendre comment, dans un langage pur, exprimer des effets
- Les avantages de la gestion explicite des effets
- Présenter quelques exemples

“**Haskell**, an advanced, purely functional programming language”

<https://haskell.org>



“Haskell, an advanced, purely functional programming language”

<https://haskell.org>

Assez facile à expliquer !

# Programmer avec des fonctions

- Fonctions comme des valeurs de premier ordre
- Que l'on peut passer en argument (d'autres fonctions)
- Que l'on peut renvoyer

## Style impératif

```
for (const x of [1, 2, 3]) {  
  console.log(x)  
}
```

## Style fonctionnel

```
[1, 2, 3].forEach(console.log)
```

# Programmer avec des fonctions

- Fonctions comme des valeurs de premier ordre
- Que l'on peut passer en argument (d'autres fonctions)
- Que l'on peut renvoyer

## Style impératif

```
for (const x of [1, 2, 3]) {  
  console.log(x)  
}
```

Hérite de la **machine de Turing**

## Style fonctionnel

```
[1, 2, 3].forEach(console.log)
```

Hérite du  **$\lambda$ -calcul**

“Haskell, an advanced, purely functional programming language”

Assez facile à expliquer !

<https://haskell.org>

Assez facile à expliquer !

## Programmer avec des fonctions... pures

- Fonctions au sens “**mathématique**” du terme
- Pas de **déclaration**, ou **d’instruction**, que des expressions
- Pas de **mutation**
- Renvoie toujours le **même résultat** pour une **même entrée**

“Haskell, an advanced, purely functional programming language”

Assez facile à expliquer !

Assez facile à expliquer !

<https://haskell.org>

Mais comment faire des programmes utiles ?

“Haskell, an advanced, purely functional programming language”

Assez facile à expliquer !

Assez facile à expliquer !

<https://haskell.org>

Mais comment faire des programmes utiles ?

Plus dur à expliquer ...

## Malheureusement, un programme n'est pas constitué que de “fonctions pures”\*

Parfois, on voudrait communiquer avec le monde, modifier un environnement.  
On voudrait “**effectuer des effets**”.



Sauf :

- Un programme vide
- Un programme qui ne s'exécute pas

**Malheureusement, un programme n'est pas  
constitué que de “fonctions pures”\***

Parfois, on voudrait communiquer avec le monde, modifier un environnement.  
On voudrait “**effectuer des effets**”.

**Mutations**

**Aléatoire**

**Gestionnaire de  
Ressources**

**Non déterminisme**

**I/O**

**Exceptions**

**Points de contrôles**

**Lecture d'un  
environnement**

**Logging**

## Les effets sont nécessaires... mais peuvent être ennuyants...

- Ils peuvent rendre le code difficile à **tester**
- Ils peuvent faire mentir (dans beaucoup de langages) la signature de type

```
print : (String) → Void
```

Ce que l'on voudrait : 

```
print : (String) → Void & IO
```

## Les effets sont nécessaires... mais peuvent être ennuyants...

- Ils peuvent rendre le code difficile à **tester**
- Ils peuvent faire mentir (dans beaucoup de langages) la signature de type

Ce que l'on voudrait :

```
print : (String) → Void  
print : (String) → Void & IO
```

C'est ce que l'on appelle  
un **effet de bord**

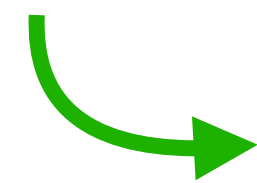


## **Mais alors, comment fait Haskell?**

- Qui permet de faire des vrais logiciels
- A priori sans effet de bords ?

## Mais alors, comment fait Haskell?

- Qui permet de faire des vrais logiciels
- A priori sans effet de bords ?



**Jusqu'à ce que l'on demande la tête (ou la queue) d'une liste vide ...**

## Un premier effet : l'exception

On voudrait représenter des calculs qui peuvent **échouer**

`head(list)` : (**List**<**A**>) → **A** = ...

`tail(list)` : (**List**<**A**>) → **List**<**A**> = ...

Ces fonctions échouent si on les appliques à **une liste vide**

# Un premier effet : l'exception

On voudrait représenter des calculs qui peuvent **échouer**

```
type Result<A> :  
  Ok: A  
  | Error: String
```

**La forme :** `f : return_type & effect`

**Devient :** `f : effect(return_type)`

```
head(list) : (List<A>) → Result<A> =  
  match list →  
  | [] : Error("La liste est vide")  
  | x ++ _ : Ok(x)
```

```
tail(list) : (List<A>) → Result<List<A>> =  
  match list →  
  | [] : Error("La liste est vide")  
  | _ ++ rest : Ok(rest)
```



# Un premier effet : l'exception

On voudrait représenter des calculs qui peuvent **échouer**

```
type Result<A> :  
  Ok: A  
  | Error: String
```

On définit un *Wrapper*  
qui est un **ADT**

**La forme :** `f : return_type & effect`

**Devient :** `f : effect(return_type)`

On exploite le  
**polymorphisme paramétrique**

On déconstruit la liste

```
head(list) : (List<A>) → Result<A> =  
  match list →  
  | [] : Error("La liste est vide")  
  | x ++ _ : Ok(x)
```

```
tail(list) : (List<A>) → Result<List<A>> =  
  match list →  
  | [] : Error("La liste est vide")  
  | _ ++ rest : Ok(rest)
```

## Ce n'est pas très agréable à utiliser

On doit chaque fois déconstruire le résultat

```
match tail([1, 2, 3, 4]) →  
  | Error(err) : Error(err)  
  | Ok(x) :  
    match tail(x) →  
      | Error(err) : Error(err)  
      | Ok(y) : match head(y) →  
        | Error(err) : Error(err)  
        | Ok(z) : Ok(z + 10)
```

# Améliorons ça avec des combinateurs

```
( >=> ) result f  
: (Result<A>, (A → Result<B>)) → Result<B>
```

=

```
match result →  
| Error(err) : Error(err)  
| Ok(x) : f(x)
```

```
( >|= ) result f  
: (Result<A>, (A → B)) → Result<B>
```

=

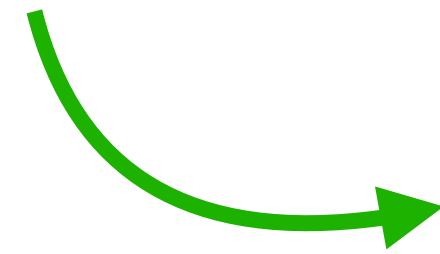
```
match result →  
| Error(err) : Error(err)  
| Ok(x) : Ok(f(x))
```

# Améliorons ça avec des combinateurs

```
( >=> ) result f  
: (Result<A>, (A → Result<B>)) → Result<B>
```

=

```
match result →  
| Error(err) : Error(err)  
| Ok(x) : f(x)
```

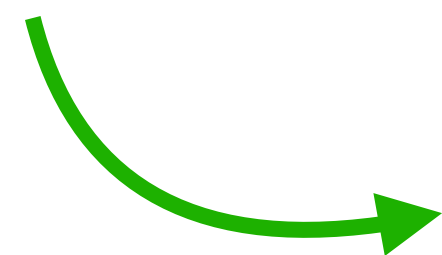


Déballe la valeur d'un **Result<A>** et l'applique à  
Une fonction qui va de **A** vers **Result<B>**  
(on appelle cette fonction **Bind** ou **FlatMap**)

```
( >|= ) result f  
: (Result<A>, (A → B)) → Result<B>
```

=

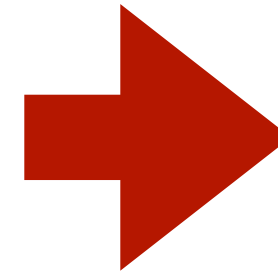
```
match result →  
| Error(err) : Error(err)  
| Ok(x) : Ok(f(x))
```



Déballe la valeur d'un **Result<A>** et l'applique à  
Une fonction qui va de **A** vers **B** et emballe le  
résultat de l'application dans un **Result<B>**  
(on appelle cette fonction **Map**)

## Améliorons ça avec des combinateurs

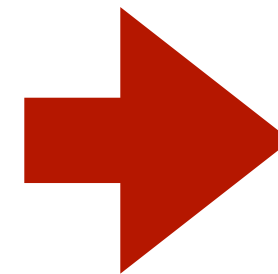
```
match tail([1, 2, 3, 4]) →  
| Error(err) : Error(err)  
| Ok(x) :  
  match tail(x) →  
  | Error(err) : Error(err)  
  | Ok(y) : match head(y) →  
    | Error(err) : Error(err)  
    | Ok(z) : Ok(z + 10)
```



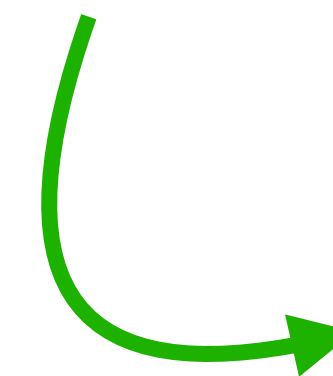
```
tail([1, 2, 3, 4])  
  >>= tail  
  >>= tail  
  >>= head  
  >>= {x → ( x + 10 )}
```

# Améliorons ça avec des combinateurs

```
match tail([1, 2, 3, 4]) →  
| Error(err) : Error(err)  
| Ok(x) :  
  match tail(x) →  
  | Error(err) : Error(err)  
  | Ok(y) : match head(y) →  
    | Error(err) : Error(err)  
    | Ok(z) : Ok(z + 10)
```



```
tail([1, 2, 3, 4])      Ok([2, 3, 4])  
>= tail                Ok([3, 4])  
>= tail                Ok([4])  
>= head                Ok(4)  
>= {x → ( x + 10 )}    Ok(14)
```



Une fonction **Lambda** !

# Un exemple de vrai programme

**Main () =**

```
program : Result<Int> =  
  tail([1, 2, 3, 4])  
  >=> tail  
  >=> tail  
  >=> head  
  >F {x → ( x + 10 )}
```

```
match program →  
  | Ok (result):  
    log("Correctement terminé avec: $result")  
  | Error (error):  
    log("Erreur: $error")
```

# Un exemple de vrai programme

**Main () =**

```
program : Result<Int> =  
  tail([1, 2, 3, 4])  
  >>= tail  
  >>= tail  
  >>= head  
  >|= {x → ( x + 10 )}
```

On déclare notre programme

On interprète sa forme normale

```
match program →  
  | Ok (result):  
    log("Correctement terminé avec: $result")  
  | Error (error):  
    log("Erreur: $error")
```



## Haskell, un $\lambda$ -Calcul computationnel

- Le  **$\lambda$ -calcul** peut être étendu pour capturer des sémantiques plus riches
- Haskell repose sur un  **$\lambda$ -calcul computationnel** (avec des types, **System-F**)

## L'idée derrière $\lambda$ -Calcul computationnel

- Distingue clairement **les valeurs** (résultats de calculs) et **les calculs** (produisant des valeurs)
- Un **calcul** produisant une **valeur de type A** est un **type de forme T<A>**

Introduit par **E. Moggi** vers 1990

## Haskell, un $\lambda$ -Calcul computationnel

- Le  **$\lambda$ -calcul** peut être étendu pour capturer des sémantiques plus riches
- Haskell repose sur un  **$\lambda$ -calcul computationnel** (avec des types, **System-F**)

## L'idée derrière $\lambda$ -Calcul computationnel

- Distingue clairement **les valeurs** (résultats de calculs) et **les calculs** (produisant des valeurs)
- Un **calcul** produisant une **valeur de type A** est un **type de forme T<A>**

T est un constructeur de type,  
par exemple **Result<A>**

## En complément de $T\langle A \rangle$

$T\langle A \rangle$  doit être adjoint de deux opérations de bases pour donner une sémantique aux langages à effets :

- **Pure** :  $(A) \rightarrow T\langle A \rangle$

Produit une valeur sans effet, **un calcul trivial**

- **( $\gg$ )** :  $(T\langle A \rangle, (A \rightarrow T\langle B \rangle)) \rightarrow T\langle B \rangle$

Effectue le calcul **A**, lie sa valeur à **x** puis effectue le calcul **B** et renvoie le résultat

## En complément de $T\langle A \rangle$

$T\langle A \rangle$  doit être adjoint de deux opérations de bases pour donner une sémantique aux langages à effets :

- **Pure** :  $(A) \rightarrow T\langle A \rangle$

Produit une valeur sans effet, **un calcul trivial**

Pour **Result**, ce serait : **pure**  $x = \mathbf{Ok}(x)$

- **( $\gg=$ )** :  $(T\langle A \rangle, (A \rightarrow T\langle B \rangle)) \rightarrow T\langle B \rangle$

Effectue le calcul **A**, lie sa valeur à **x** puis effectue le calcul **B** et renvoie le résultat

```
(  $\gg=$  ) result f
: (Result<A>, (A  $\rightarrow$  Result<B>))  $\rightarrow$  Result<B>
=
match result  $\rightarrow$ 
| Error(err) : Error(err)
| Ok(x) : f(x)
```

**Ce triplet  $(T\langle A \rangle, \text{pure}, \gg=)$  est un Triplet de Kleisli**

Encoder un “nouvel effet”, consiste à implémenter le protocole suivant, tout en respectant Certaines lois.

**protocol**  $T\langle A \rangle \rightarrow$

$\text{pure} : (A) \rightarrow T\langle A \rangle$

$(\gg=) : (T\langle A \rangle, (A \rightarrow T\langle B \rangle)) \rightarrow T\langle B \rangle$

Le **trick** consiste à **transformer tout effet en valeur**

Que l'on appelle, par abus de langage  
bénin, une **Monade**



**Ce triplet  $(T\langle A \rangle, \text{pure}, \gg=)$  est un Triplet de Kleisli**

Encoder un “nouvel effet”, consiste à implémenter le protocole suivant, tout en respectant  
Certaines lois.

**protocol**  $T\langle A \rangle \rightarrow$

$\text{pure} : (A) \rightarrow T\langle A \rangle$

$(\gg=) : (T\langle A \rangle, (A \rightarrow T\langle B \rangle)) \rightarrow T\langle B \rangle$

Le **trick** consiste à **transformer tout effet en valeur**

Que l'on appelle, par abus de langage  
bénin, une **Monade**



**Ce triplet  $(T\langle A \rangle, \text{pure}, \gg=)$  est un Triplet de Kleisli**

Encoder un “nouvel effet”, consiste à implémenter le protocole suivant, tout en respectant  
Certaines lois.

**protocol**  $T\langle A \rangle \rightarrow$

$\text{pure} : (A) \rightarrow T\langle A \rangle$

$(\gg=) : (T\langle A \rangle, (A \rightarrow T\langle B \rangle)) \rightarrow T\langle B \rangle$

Le **trick** consiste à **transformer tout effet en valeur**



Ok pour Result, mais comment ça fonctionne  
avec des effets plus complexe, par exemple, l'IO ?

**Main () =**

```
program : Result<Int> =  
  tail([1, 2, 3, 4])  
  >= tail  
  >= tail  
  >= head  
  >= {x → ( x + 10 )}
```

La déclaration du programme est pure

Mais son interprétation ne l'est potentiellement pas


```
match program →  
  | Ok (result):  
    log("Correctement terminé avec: $result")  
  | Error (error):  
    log("Erreur: $error")
```



## Un “Hello World” en Haskell

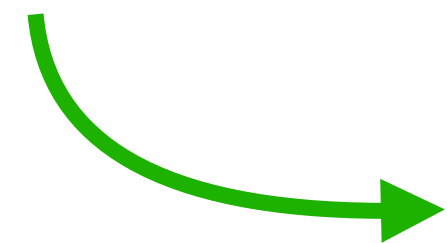
```
main :: IO ()  
main =  
    putStrLn "Hello World"
```

## Un “Hello World” en Haskell

`main :: IO ()`  Est une Monade `IO<Void>`


`main =`

`putStrLn "Hello World"`



A le type `putStrLn :: String → IO ()`

## Un “Hello World” en Haskell

`main :: IO ()`  Est une Monade `IO<Void>`

`main =`  
 `putStrLn “Hello World”`

 A le type `putStrLn :: String → IO ()`

 En Haskell, le “main” est en fait une **valeur, de type IO()**

En Haskell, on décrit un programme dont l'unité  
la plus basse est l'IO, et c'est le Runtime de Haskell  
Qui en fournit l'interpréteur.

## Rapide résumé

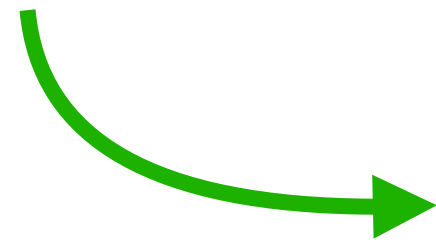
- Les monades offrent une manière de représenter les effets sous forme de valeur
- Donc les effets représentés par ces valeurs doivent être interprétés
- En complément, il est possible de représenter des flots de calcul
- (  $\gg=$  ) est un peu l'équivalent du retour à la ligne dans un langage impératif

## Imaginons un exemple impératif

```
println("What is your name?")  
name = readLine()  
println("Hello $name")
```

# Imaginons un exemple impératif

```
println("What is your name?")  
name = readLine()  
println("Hello $name")
```



**Construction de l'ensemble des opérations dans un ADT**

```
type IO =  
  | Read : String  
  | Print : String
```

```
println("What is your name?")  
val name = readLine()  
println("Hello $name")
```

Construction de l'ensemble des opérations dans un ADT

```
type IO =  
  | Read : String  
  | Print : String
```

Construction d'un interpréter

```
run task =  
  match task →  
  | Print(message): println(message)  
  | Read(question):  
    println(question)  
    answer = readLine()  
    // Ignoring the result ATM  
  Void
```





## Construction d'un interpréteur

```
run task =  
  match task →  
  | Print(message): println(message)  
  | Read(question):  
    println(question)  
    answer = readLine()  
    // Ignoring the result ATM  
Void
```



## Construction d'un programme

```
program = [  
  Read("What is your name?")  
  , Print("Hello!")  
]
```

```
forEach(program, run) // On exécute le programme
```

## Construction d'un interpréteur

```
run task =  
  match task →  
  | Print(message): println(message)  
  | Read(question):  
    println(question)  
    answer = readLine()  
    // Ignoring the result ATM  
Void
```

Mais nous n'avons pas accès au nom...

## Construction d'un programme

```
program = [  
  Read("What is your name?")  
  , Print("Hello!")  
]
```

```
forEach(program, run) // On exécute le programme
```

## Construction d'un interpréteur

```
run task =  
  match task →  
  | Print(message): println(message)  
  | Read(question):  
    println(question)  
    answer = readLine()  
    // Ignoring the result ATM  
Void
```

## Construction d'un programme

```
program = [  
  Read("What is your name?")  
  , Print("Hello!")  
]
```

```
forEach(program, run) // On exécute le programme
```

Et quand bien même. Comment composer des Monades différentes ?

Mais nous n'avons pas accès au nom...

## A ce stade, nous avons trois possibilités

- Utiliser des **Transformations de monades**
- Utiliser des **Monades Libres**
- Utiliser des **Effets algébriques** et **des gestionnaire d'effets**

## A ce stade, nous avons trois possibilités

● ~~Utiliser des Transformations de monades~~

- Enormément de Boilerplate
- Gestion manuelle du séquençage d'effets

● Utiliser des Monades Libres

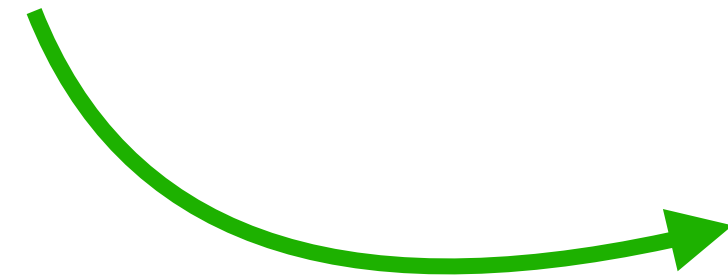
- Très dur à expliquer en si peu de temps
- Emprunte mémoire non négligeable

● Utiliser des Effets algébriques et des gestionnaire d'effets

- Implique une modification du langage

# **Les effets algébriques et leurs gestionnaires**

# Les effets algébriques et leurs gestionnaires



**Dernière partie, je le promet !**

**Et c'est beaucoup plus simple  
que tout ce qu'on a vu !!!**

# **Les effets algébriques et leurs gestionnaires**

**Dernière partie, je le promet !**

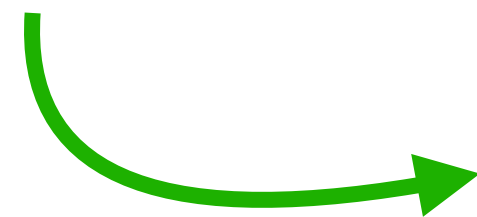


## De la propagation a la provenance d'effets

- Le  **$\lambda$ -calcul** computationnel (et les monades) expriment la **propagation d'effets** génériques indépendamment du type d'effet considéré
- Peut-on prendre en compte, génériquement, les opérations qui “produisent” des effets ?
  - **Erreur** : raise
  - **IO** : print, read
  - **Etats mutables** : set/get

## De la propagation a la provenance d'effets

- Le  **$\lambda$ -calcul** computationnel (et les monades) expriment la **propagation d'effets** génériques indépendamment du type d'effet considéré
- Peut-on prendre en compte, génériquement, les opérations qui “produisent” des effets ?
  - **Erreur** : raise
  - **IO** : print, read
  - **Etats mutables** : set/get



Oui, en 2003: Power et Plotkin proposent une vision algébrique  
De la “performance d'effets”.

**Concrètement, les effets algébriques cristallisent  
l'idée derrière la séparation de la description du programme  
et de l'exécution de l'effet décrit dans le programme**

Soit le fait de décrire un programme qui peut lancer un ou plusieurs effets (un peu à la manière d'une exception)

Concrètement, les effets algébriques cristallisent l'idée derrière la séparation de la description du programme et de l'exécution de l'effet décrit dans le programme

Soit fournir un gestionnaire pour une fonction qui "lance" un effet

# A quoi cela ressemble

On offre la possibilité de décrire un effet, par exemple :

```
effect io =  
  | Print : String → unit  
  | Read  : String
```

Et on ajoute à un genre de **λ-calcul** computationnel cette opération:

```
perform(Effet ...args)
```

Par exemple :

```
perform(Print("Hello World"))  
Perform(Read)
```

Impact sur le système de type

```
une_fonction_qui_perform : (Input) → output & (effet1, effet2...)
```

# A quoi cela ressemble

On offre la possibilité de décrire un effet, par exemple :

```
effect io =
```

```
  | Print : String → unit
```

```
  | Read  : String
```



Qu'est-ce que ça a d'algébrique ?

Et on ajoute à un genre de **λ-calcul** computationnel cette opération:

```
perform(Effet ...args)
```

Par exemple :

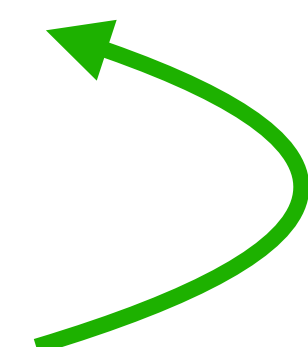
```
perform(Print("Hello World"))
```

```
Perform(Read)
```

La syntaxe change en fonction des langages

Impact sur le système de type

```
une_fonction_qui_perform : (Input) → output & (effet1, effet2...)
```



# Reprenons notre exemple IO

```
effect io =
```

```
  | Print : String → unit
```

```
  | Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

```
handle program →
```

```
  | Print (message), continuation :
```

```
    println(message)
```

```
    continuation ()
```

```
  | Read, continuation :
```

```
    line = readLine ()
```

```
    continuation line
```

# Reprenons notre exemple IO

```
effect io =
```

```
  | Print : String → unit
```

```
  | Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

```
handle program →
```

```
  | Print (message), continuation :
```

```
    println(message)
```

```
    continuation ()
```

```
  | Read, continuation :
```

```
    line = readLine ()
```

```
    continuation line
```



On **perform** l'effet **Print**



# Reprenons notre exemple IO

```
effect io =
```

```
  | Print : String → unit
```

```
  | Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

```
handle program →
```

```
  | Print (message), continuation : On capture l'effet et la continuation
```

```
    println(message) On affiche le message
```

```
    continuation ()
```

```
  | Read, continuation :
```

```
    line = readLine ()
```

```
    continuation line
```

# Reprenons notre exemple IO

```
effect io =
```

```
| Print : String → unit
```

```
| Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

On reprend l'exécution du programme



```
handle program →
```

```
| Print (message), continuation :
```

```
  println(message)
```

```
  continuation ()
```

```
| Read, continuation :
```

```
  line = readLine ()
```

```
  continuation line
```

# Reprenons notre exemple IO

```
effect io =
```

```
  | Print : String → unit
```

```
  | Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

```
handle program →
```

```
  | Print (message), continuation :
```

```
    println(message)
```

```
    continuation ()
```

```
  | Read, continuation :
```

```
    line = readLine ()
```

```
    continuation line
```

On **perform** l'effet **Read**



# Reprenons notre exemple IO

```
effect io =
```

```
  | Print : String → unit
```

```
  | Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

```
handle program →
```

```
  | Print (message), continuation :
```

```
    println(message)
```

```
    continuation ()
```

```
  | Read, continuation :
```

```
    line = readLine () On demande de lire l'entrée standard
```

```
    continuation line
```

# Reprenons notre exemple IO

```
effect io =
```

```
  | Print : String → unit
```

```
  | Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

```
handle program →
```

```
  | Print (message), continuation :
```

```
    println(message)
```

```
    continuation ()
```

```
  | Read, continuation :
```

```
    line = readLine ()
```

```
    continuation line
```



On reprend l'exécution du programme

# Reprenons notre exemple IO

```
effect io =
```

```
| Print : String → unit
```

```
| Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

```
handle program →
```

```
| Print (message), continuation :
```

```
  println(message)
```

```
  continuation ()
```

```
| Read, continuation :
```

```
  line = readLine ()
```

```
  continuation line
```



On **perform** l'effet **Print**

# Reprenons notre exemple IO

```
effect io =
```

```
| Print : String → unit
```

```
| Read  : String
```

```
program : void & (io) =
```

```
  perform (Print "What's your name")
```

```
  name = perform (Read)
```

```
  perform (Print name)
```

On reprend l'exécution du programme



```
handle program →
```

```
| Print (message), continuation :
```

```
  println(message)
```

```
  continuation ()
```

```
| Read, continuation :
```

```
  line = readLine ()
```

```
  continuation line
```

# Reprenons notre exemple IO

```
effect io =  
  | Print : String → unit  
  | Read  : String  
  
program : void & (io) =  
  perform (Print "What's your name")  
  name = perform (Read)  
  perform (Print name)  
  Fin  
handle program →  
  | Print (message), continuation :  
    println(message)  
    continuation ()  
  | Read, continuation :  
    line = readLine ()  
    continuation line
```

- Le programme est une **valeur**, donc complètement pur !
- Le handler exécute les effets de bords
- Perform "envoie l'effet" et ses paramètres ainsi que la continuation qui correspond à la suite du programme.
- Il peut exister des Handler par défaut et des effets définis dans la bibliothèque standard

D'où la nécessité d'être capable de compiler correctement les continuations (ce pourquoi **Kotlin** est plutôt efficace)



## Des exemple amusants

**handle** program →

```
| Print (message), continuation :  
    println(message)  
    continuation ()  
  
| Read, continuation :  
    line = readLine ()  
    continuation "Xavier"
```

**handle** program →

```
| Print (message), continuation :  
    continuation ()  
    println(message)  
  
| Read, continuation :  
    line = readLine ()  
    continuation line
```

## Bénéfices indéniables

- Programmation pure !
- Contrôle très fin sur le flot du programme
- Optimisation sur le séquençage des effets via les équations algébriques
- Très simple à tester car on peut écrire son propre handler
- Propose une manière systématique de séparer l'algorithme/programme de la plomberie
- Ça permet de faire des trucs très rigolos... (inverser la continuation et l'action)

## Compléments

- Il est très facile d'exprimer de la concurrence
- Adjoint à un runtime multi-coeur, on peut exprimer de la concurrence multi-coeur
- Se compose mieux que des monades grâce à la forme : (Input)  $\rightarrow$  Output & (...effets)

## Choses ballots

- Très dur à implémenter et à typer

# Quelques langages qui intègrent des effets algébriques

- **Haskell** via **Polysemy** ou **Fused Effects**
- **Links** avec des session types (expérimental)
- **Koka** (expérimental)
- **Unison** (en développement)
- **Eff** (expérimental et en développement)
- **Frank** (en développement)
- **OCaml + multicore** (expérimental et en développement)
- **Idris** avec des monades et des types dépendants
- **Fstar** avec des monades et des types dépendants

En complément, on trouve beaucoup de bibliothèques qui gèrent les effets, de manière monadique (Cat, ZIO, ScalaZ, Arrow.Fx etc.)

## Aller plus loin !

- Expérimenter plus d'effets via les **Applicatives, Comonades, Arrows**
- Essayer le plus possible de séparer les parties impures et pures de son programme
- Se plonger plus en profondeur dans les systèmes de types.

**Merci !**

@vdwxv - <https://xvw.github.io>