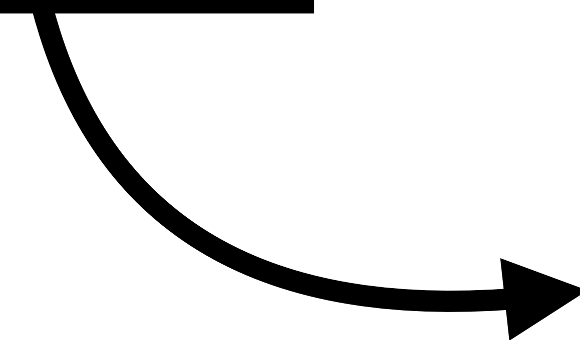
Une brève introduction aux

effets algébriques pour écrire des "***vrais logiciels***"

Xavier Van de Woestyne

Une brève introduction aux

effets algébriques pour écrire des "*vrais logiciels*"



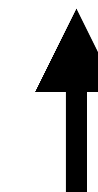
Dans le sens **biaisé** du terme

Xavier Van de Woestyne

Une brève introduction aux

effets algébriques pour écrire des "*vrais logiciels*"

industriel



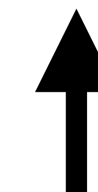
Dans le sens **biaisé** du terme

Xavier Van de Woestyne

Une brève introduction aux

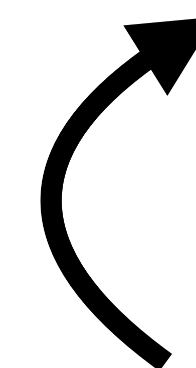
effets algébriques pour écrire des "*vrais logiciels*"

industriel



Dans le sens **biaisé** du terme

- Margo**Bank**
- xvw.github.io
- twitter/vdwxv
- xvw@merveille.town



Xavier Van de Woestyne

“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...

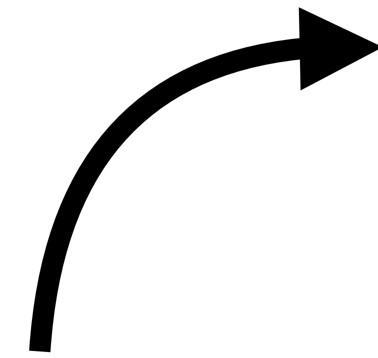
Cependant, les langages fonctionnels ne “seraient pas” *Production-ready…* mais

- 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

- Présenter une définition des effets et effets de bord
- Observer les effets en Haskell
- Comprendre l'intérêt de leur contrôle
- Présenter les effets algébriques et les gestionnaires
- Observer un cas d'usage
- Parler de la gestion d'effets dans les langages *mainstream*

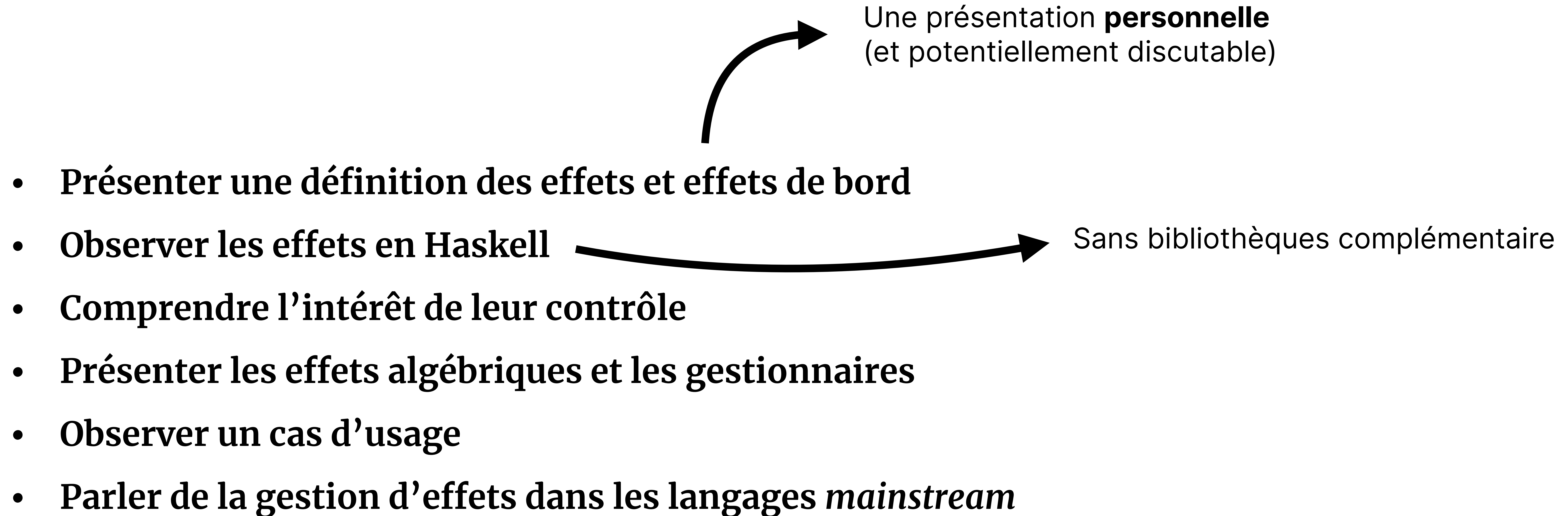
Objectifs de la présentation



Une présentation **personnelle**
(et potentiellement discutable)

- **Présenter une définition des effets et effets de bord**
- **Observer les effets en Haskell**
- **Comprendre l’intérêt de leur contrôle**
- **Présenter les effets algébriques et les gestionnaires**
- **Observer un cas d’usage**
- **Parler de la gestion d’effets dans les langages *mainstream***

Objectifs de la présentation



- **Présenter une définition des effets et effets de bord**
- **Observer les effets en Haskell**
- **Comprendre l'intérêt de leur contrôle**
- **Présenter les effets algébriques et les gestionnaires**
- **Observer un cas d'usage**
- **Parler de la gestion d'effets dans les langages *mainstream***

Objectifs de la présentation

Une présentation **personnelle**
(et potentiellement discutable)

- **Présenter une définition des effets et effets de bord**
- **Observer les effets en Haskell**
- **Comprendre l'intérêt de leur contrôle**
- **Présenter les effets algébriques et les gestionnaires**
- **Observer un cas d'usage**
- **Parler de la gestion d'effets dans les langages *mainstream***

Sans bibliothèques complémentaire

Dijkstra déteste !
ref: Twitter madness

Objectifs de la présentation

Une présentation **personnelle**
(et potentiellement discutable)

- **Présenter une définition des effets et effets de bord**
- **Observer les effets en Haskell**
- **Comprendre l'intérêt de leur contrôle**
- **Présenter les effets algébriques et les gestionnaires**
- **Observer un cas d'usage**
- **Parler de la gestion d'effets dans les langages *mainstream***

Sans bibliothèques complémentaire

Dijkstra déteste !
ref: Twitter madness

Avec le langage **Koka**

Effets et effets de bords

Fonction pure

- Totale
- Déterministe
- Sans effets (uniquement capable de calculer des choses sans dépendance)

Fonction impure

- Toutes les fonctions qui ne sont pas pures

Effets et effets de bords

Très utile quand on **exécute un programme**

Fonction pure

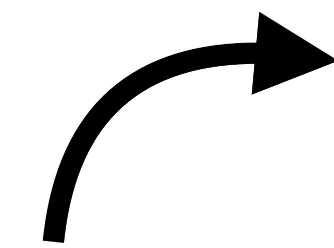
- Totale
- Déterministe
- Sans effets (uniquement capable de calculer des choses sans dépendance)

Beaucoup plus facile à **tester** et à **optimiser !**

Fonction impure

- Toutes les fonctions qui ne sont pas pures

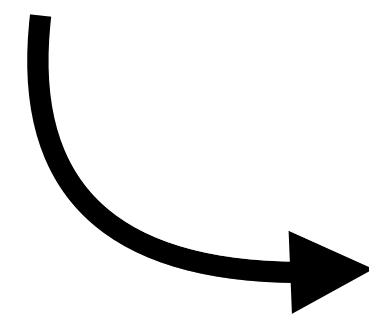
Effets et effets de bords



Très utile quand on **exécute un programme**

Function pure

- Totale
- Déterministe
- Sans effets (uniquement capable de calculer des choses sans dépendance)



Beaucoup plus facile à **tester** et à **optimiser !**

Un effet est donc, quelque chose de de "**non computationnel**".

Function impure

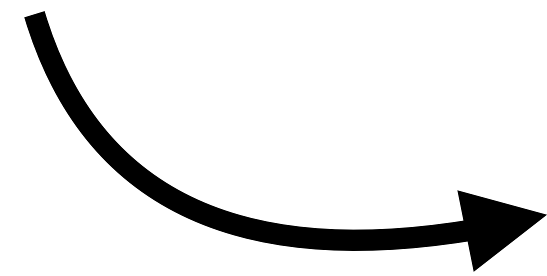
- Toutes les fonctions qui ne sont pas pures

Effets et effets de bords

La seule manière “éprouvée” de tester unitaire ment une **fonction impure** est d’utiliser de l’**injection de dépendances.**

Effets et effets de bords

La seule manière “éprouvée” de tester unitaire ment une **fonction impure** est d’utiliser de l’**injection de dépendances.**



Les **mocks** n’étant, au final, que de l’**injection de dépendance à postériori**

Effets et effets de bords

Autrement dit !

Un effet *peut être perçu comme* une **action** qui a besoin d'être **exécuté par une autorité centrale** qui devra **gérer cet effet.**

Par exemple:

- Lire et écrire sur la sortie standard
- Interagir avec une base de données
- Gérer de l'aléatoire
- Le lancement d'une exception

Autrement dit !

Un effet *peut être perçu comme* une **action** qui a besoin d'être **exécuté par une autorité centrale** qui devra **gérer cet effet.**

Par exemple:

- Lire et écrire sur la sortie standard → **STDin** et **STDout** comme autorité
- Interagir avec une base de données → La **base de donnée**
- Gérer de l'aléatoire → Des éléments relatifs à l'**OS**
- Le lancement d'une exception → Le block **try/catch**

Et un effet de bord ?

Même s'il existe beaucoup de définition (et parfois des raccourcis comme **effet = effet de bord**).
Je préfère résumer un effet de bord, dans le contexte d'un langage statiquement typé comme :

"un effet de bord est un effet qui n'est pas reflété dans la signature de type de la fonction qui l'exécute."

Généralement :

- On a : $\Gamma \vdash e : \tau$
- On voudrait : $\Gamma \vdash e : \tau \ \& \ \text{effects}$

Par exemple:

- `val print_endline : string -> unit`
- `val print_endline : string -> unit & console`

Type are docs !

Avec des effets de bords, c'est beaucoup moins vrai !

```
List.map : (‘a -> ‘b) -> ‘a list -> ‘b list
```

Type are docs !

Avec des effets de bords, c’est beaucoup moins vrai !

```
List.map : (‘a -> ‘b) -> ‘a list -> ‘b list
on_countries : country list -> unit
```

Qu’est ce que cette fonction fait ?

Type are docs !

Avec des effets de bords, c’est beaucoup moins vrai !

```
List.map : (‘a -> ‘b) -> ‘a list -> ‘b list
on_countries : country list -> unit
let on_countries countries =
   List.map (fun country ->
      launch_a_funcking_rocket_to country
   ) countries
```

En vrai la signature ne ment pas tant que ça…

Type are docs !

Ça aurait pu être encore pire

```
List.map : (‘a -> ‘b) -> ‘a list -> ‘b list
on_countries : country list -> unit

let on_countries countries =
   List.map (fun country ->
      print_endline “-” ^ country
   ) countries
```

Cette fois, la signature ment totalement !

```
~ xvw ./run_program Belgique France USA
    - Belgique
    - France
    - USA
```

**Mais à quoi bon gérer ses effets ? Il suffit d'avoir
De la discipline non?**

Mais à quoi bon gérer ses effets ? Il suffit d'avoir De la discipline non?

- Pour avoir des signatures exhaustives !
- Ça rend le code plus facile à raisonner
- Pour les rendre plus facilement testables!
- Dans le cas des effets algébriques, cela permet des optimisations non négligeables !

Sur la testabilité

Un exemple de très mauvais code !

```
import java.util.Scanner

fun successor() {
  val reader = Scanner(System.`in`)
  print("Enter a number: ")
  val input = reader.nextInt()
  val result = input + 1
  println("The successor of [$input] is [$result] ")
}

fun main() {
  successor()
}
```

Sur la testabilité

C'est un exemple !

Un exemple de très mauvais code !

```
import java.util.Scanner

fun successor() {
  val reader = Scanner(System.`in`)
  print("Enter a number: ")
  val input = reader.nextInt()
  val result = input + 1
  println("The successor of [$input] is [$result] ")
}

fun main() {
  successor()
}
```

Une dépendance !

Un effet !

Un autre effet !

Encore un autre effet !

La partie **pure** de notre programme

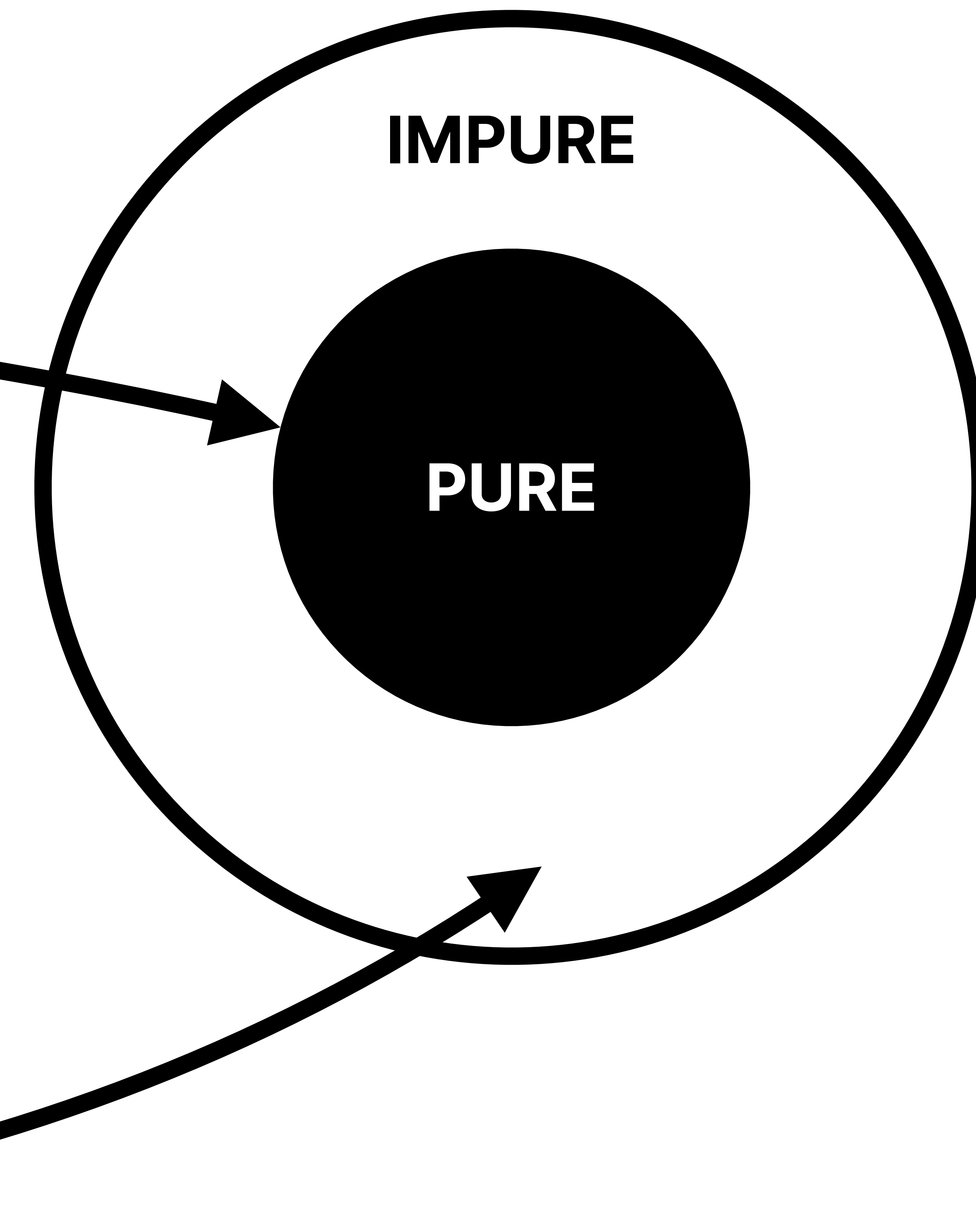
Sur la testabilité

Ce que l'on voudrait !

```kotlin
import java.util.Scanner

fun successor(x: Int) : Int {
  val result = x + 1
  return result
}

fun main() {
  val reader = Scanner(System.`in`)
  print("Enter a number: ")
  val input = reader.nextInt()
  val result = successor(input)
  println("The successor of [$input] is [$result] ")
}
```



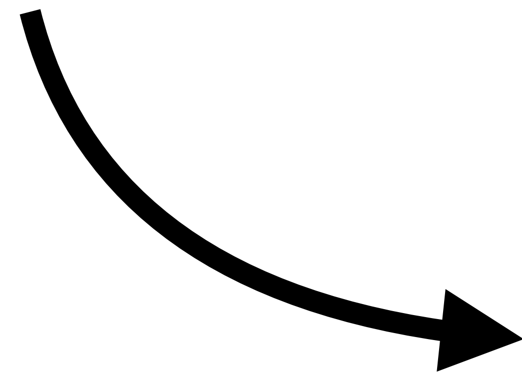
- La fonction **successor** est facile à tester
- On exécute les effets dans la fonction **main**

Sur la testabilité

C’est en grande partie ce à quoi essaient de répondre bon nombre d’architectures (ie: **l’architecture hexagonale**).

Sur la testabilité

C’est en grande partie ce à quoi essaient de répondre bon nombre d’architectures (ie: **l’architecture hexagonale**).



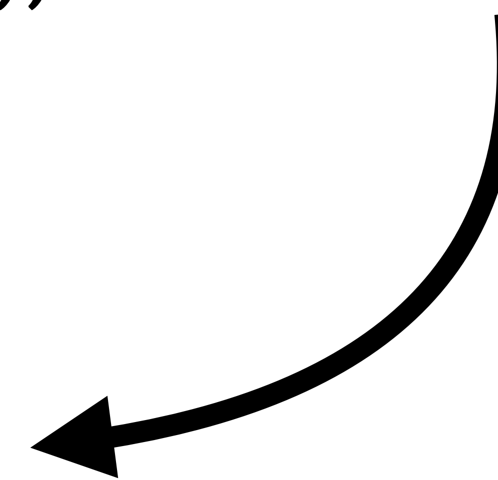
Les effets algébriques offrent une manière systématique de séparer la partie pure et impure du programme.

Sur le cas de Haskell

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

Sur le cas de Haskell

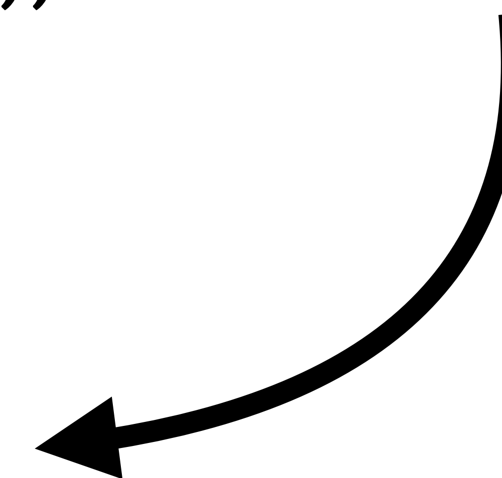
“**Haskell**, an advanced, **purely functional** programming language”



En d’autres mots… Haskell ne permet **que de manipuler des fonctions pures…**

Sur le cas de Haskell

"**Haskell**, an advanced, **purely functional** programming language"



En d'autres mots… Haskell ne permet **que de manipuler des fonctions pures…**

Donc Haskell est … inutile ?

Sur le cas de Haskell

"**Haskell**, an advanced, **purely functional** programming language"

En d'autres mots… Haskell ne permet **que de manipuler des fonctions pures…**

Donc Haskell est … inutile ?

Lol non (enfin je ne crois pas)

Sur le cas de Haskell

"**Haskell**, an advanced, **purely func**
programming language"

En d'autres mots... Haskell ne permet **que de manipuler des fonctions pures...**

Donc Haskell est ... inutile ?

Lol non

Premier Trick (de Haskell) : transformer les effets en valeur !

Par exemple **Exception** en **Either a b** le constructeur Either caractérise l'effet :

```
fonction_pouvant_echouer : () -> Either Error a
```

Sur le cas de Haskell

"**Haskell**, an advanced, **purely func**
programming language"

En d'autres mots… Haskell ne permet **que de manipuler des fonctions pures…**

Donc Haskell est … inutile ?

Lol non

Premier Trick (de Haskell) : transformer les effets en valeur !

Par exemple **Exception** en **Either a b** le constructeur Either caractérise l'effet :

```
fonction_pouvant_echouer : () -> Either Error a
```

Ok, mais alors… comment ça marche pour… IO ?

Sur le cas de Haskell

```
main :: IO ()
main =
  putStrLn “Hello World”
```

Sur le cas de Haskell

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

Sur le cas de Haskell

Un programme Haskell est en fait une description de programme **dont l'autorité centrale est son runtime**

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

IO est spécifiquement **l'atome d'un calcul à effet**

$\Gamma \vdash \mathbf{e} : \tau$ **& effects** devient $\Gamma \vdash \mathbf{e} : \mathbf{Effect}(\tau)$

Sur le cas de Haskell

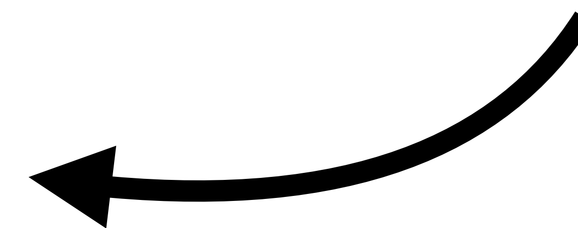
Nous somme sauf !

```
on_countries : () -> IO ()
```

Sur le cas de Haskell

Nous somme saufs !

```
on_countries : () -> IO ()
```



Il y'a **IO**… notre fonction peut **potentiellement lancer une roquette !**

Sur le cas de Haskell

- Haskell utilise la forme : **Effet t** pour exprimer **t & effet**
- Comme **IO** est l'atome (dans IO c'est openbar) il n'indique "plus que la présence d'effet"
- Cependant, ça permet de manipuler des fonctions pures !

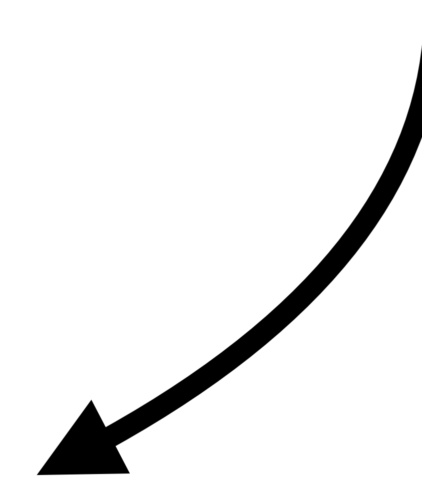
Les effets algébriques permettent, en complément du marquage, d'être plus précis sur les effets qui sont mis en oeuvre dans une fonction.

Les effets algébriques

- Définir un ensemble d’effets, comme des simple constructeurs (ou des fonctions)
- Pour exécuter une fonction qui propage des effets, il faut fournir un interpréteur pour les effets propagés par la fonction.

Les effets algébriques

- Définir un ensemble d’effets, comme des simple constructeurs (ou des fonctions)
- Pour exécuter une fonction qui propage des effets, il faut fournir un interpréteur pour les effets propagés par la fonction.



L’interpréteur est l’autorité centrale et peut référer à d’autres interpréteurs.

Koka

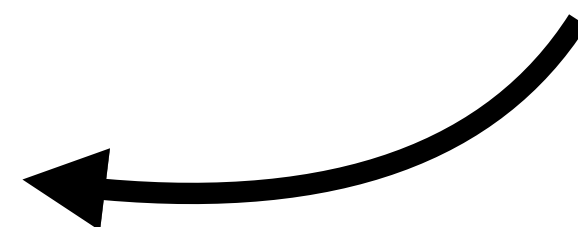
- Un petit langage qui supporte les effets algébriques
- Qui peut compiler vers JavaScript (et dont la syntaxe ressemble à JS)
- Qui est très intéressant pour s'initier aux effets algébriques (de mon point de vue)

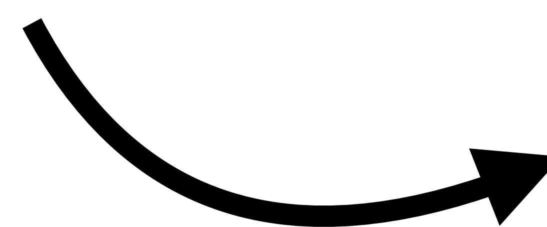
Koka

Les fonctions reflètent l'effet qu'elles exécutent par exemple :

```
fun hello(name) {
  println("Hello " + name + "!")
}
```

Aura le type : `(name: string) -> console ()`





Effet de la fonction

Type de retour de la fonction

Un cas d’usage

```kotlin
fun sayHello() {
  println("What is your name?")
  val name = readLine()!!
  println("Hello $name")
}

fun main() {
  sayHello()
}
```

Transformation en Koka

```
effect mumble {
  fun grumble(message: string) : ()
}
```

Transformation en Koka

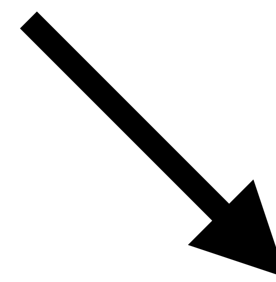
```
effect mumble {
  fun grumble(message: string) : ()
}

fun mumbling() : mumble () {
  grumble("Hello World!")
}
```

Transformation en Koka

```
effect mumble {
  fun grumble(message: string) : ()
}
```

```
fun mumbling() : mumble () {
  grumble("Hello World!")
}
```



```
error: there are unhandled effects for the main expression
  inferred effect: test/mumble
```

Transformation en Koka

```
effect mumble {
  fun grumble(message: string) : ()
}

fun mumbling() : mumble () {
  grumble("Hello World!")
}
```

```
error: there ar
  inferred effe
```

```
val mumble_handler = handler {
  grumble(message) -> println(message)
}
```

```
mumble_handler{ mumbling() }
```

On ne peut **qu'exécuter des fonction pour lesquels il existe un handler dans le scope.** Ici println possède un handler par défaut.

Transformation en Koka

```
effect mumble {
  fun grumble(message: string) : ()
}
```

```
fun mumbling() : mumble () {
  grumble("Hello World!")
  grumble("Good bye World!")
}

val mumble_handler = handler {
  grumble(message) -> println(message)
}

fun main() {
   mumble_handler{ mumbling() }
}
```

Transformation en Koka

```
effect mumble {
  fun grumble(message: string) : ()
}
```

```
fun mumbling() : mumble () {
  grumble("Hello World!")
  grumble("Good bye World!")
}

val mumble_handler = handler {
  grumble(message) -> println(message)
}

fun main() {
   mumble_handler{ mumbling() }
}
```

N'affiche que "Hello World!"

Transformation en Koka

Les effets algébriques laissent au gestionnaire d’effet la décisions de continuer le calcul où non.

Transformation en Koka

Les effets algébriques laissent au gestionnaire d'effet la décisions de continuer le calcul où non.

```
val mumble_handler = handler {
  grumble(message) -> {
     println(message)
     resume()
  }
}
```

Transformation en Koka

Les effets algébriques laissent au gestionnaire d’effet la décisions de continuer le calcul où non.

```
val mumble_handler = handler {
  grumble(message) -> {
     println(message)
     resume()
  }
}
```

Transformation en Koka

Kotlin

```
fun sayHello() {
  println("What is your name?")
  val name = readLine()!!
  println("Hello $name")
}

fun main() {
  sayHello()
}
```

Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Transformation en Koka

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}

fun program() : interaction () {
  val name = ask("What's your name? ")
  show("Hello " + name)
}

val hello_handler = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    println(message)
    resume(())
  }
}
```

Bénéfices

- Séparation systématique entre l'algorithme et la plomberie nécéssaire à l'algorithme
- On permet au développeur de définir ses effets et il se focalise sur l'émission des effets
- Un consommateur peut construire son propre gestionnaire (et potentiellement changer le flot de contrôle)

Bénéfices

- Séparation systématique entre l'algorithme et la plomberie nécéssaire à l'algorithme
- On permet au développeur de définir ses effets et il se focalise sur l'émission des effets
- Un consommateur peut construire son propre gestionnaire (et potentiellement changer le flot de contrôle)

```
val hello_handler_reversed = handler {
  ask(message) -> {
    val name = question(message)
    resume(name)
  }
  show(message) -> {
    resume(())
    println(message)
  }
}

fun program() {
  show("Hello World")
  val x = ask("What is your name?")
  show("Hello " + x)
}
```

```
What is your name? <input>
Hello <input>
Hello World
```

Bénéfices

- Séparation systématique entre l'algorithme et la plomberie nécéssaire à l'algorithme
- On permet au développeur de définir ses effets et il se focalise sur l'émission des effets
- Un consommateur peut construire son propre gestionnaire (et potentiellement changer le flot de contrôle)
- Tests unitaires simples (il suffit de fournir un autre gestionnaire)

```
val test_handler = handler {
  ask(_) -> {
    val accumulator = get()
    set(accumulator + ";Xavier")
    resume("Xavier")
  }
  show(message) -> {
    val accumulator = get()
    set(accumulator + ";" + message)
    resume(())
  }
}
(() -> <interaction, state<string>> ()) -> state<string> ()
```

```
fun test() {
  val result = state_handler("start"){
    test_handler{
       program()
    }
  }
  assert(
    "String should be equals",
    result == "start;Xavier;Hello Xavier;end")
         // Au final, voici à quoi devrait ressembler notre
         // résultat accumulé
}
```

Bénéfices

- Séparation systématique entre l'algorithme et la plomberie nécéssaire à l'algorithme
- On permet au développeur de définir ses effets et il se focalise sur l'émission des effets
- Un consommateur peut construire son propre gestionnaire (et potentiellement changer le flot de contrôle)
- Tests unitaires simples (il suffit de fournir un autre gestionnaire)
- Possibilité de combiner des effets (et de fournir plusieurs gestionnaires) sans altérer le programme initial

Bénéfices

```
effect user_database {
  fun create_user(username: string) : ()
  fun update_user(
     old_username: string,
     new_username: string
  ) : ()
  fun drop_user(username: string) : ()
}

fun program() {
  create_user("xavier")
  update_user("xavier", "xvw")
  drop_user("xvw")
}
```

```
val logger_user_handler = handler {
  create_user(username) -> {
    println("LOG: create_user [" + username + "]")
    create_user(username)
    resume(())
  }
  update_user(old, new) -> {
    println("LOG: update_user [" + old + "] by ["+ new +"]")
    update_user(old, new)
    resume(())
  }
  drop_user(username) -> {
    println("LOG: drop_user [" + username + "]")
    drop_user(username)
    resume(())
  }
}
```

```
Type : () -> <user_database, logger> ()
```

Bénéfices

- Séparation systématique entre l'algorithme et la plomberie nécéssaire à l'algorithme
- On permet au développeur de définir ses effets et il se focalise sur l'émission des effets
- Un consommateur peut construire son propre gestionnaire (et potentiellement changer le flot de contrôle)
- Tests unitaires simples (il suffit de fournir un autre gestionnaire)
- Possibilité de combiner des effets (et de fournir plusieurs gestionnaires) sans altérer le programme initial
- Peut servir de base à l'implémentation de construction complexes (while/async/exception)

Quels langages Mainstreams proposent des effets algébriques ?

Quels langages Mainstreams proposent des effets algébriques ?

Aucuns

Je suis vraiment désolé, mes slides sont vraiment nuls…