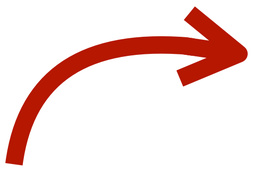
Programmation fonctionnelle

et principe de réalité : gérer les effets

Xavier Van de Woestyne - Memo.bank - https://xvw.github.io - @vdwxv - xvw@merveille.town

Programmation fonctionnelle

et principe de réalité : gérer les effets



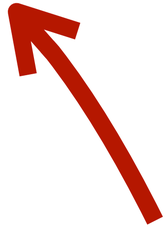
J'aime beaucoup la programmation fonctionnelle statiquement typée… (comme OCaml)

Xavier Van de Woestyne - Memo.bank - https://xvw.github.io - @vdwxv - xvw@merveille.town

Programmation fonctionnelle

et principe de réalité : gérer les effets

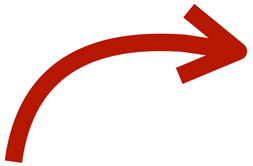
On crée une banque from-scratch !
(Avec entre autre Kotlin, Elixir)



Xavier Van de Woestyne - Memo.bank - https://xvw.github.io - @vdwxv - xvw@merveille.town

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

Beaucoup de programmeurs



Je pense que c'est faux

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

Beaucoup de programmeurs

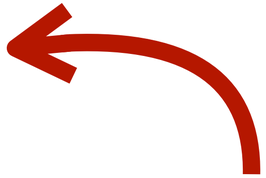
Objectifs

- Comprendre la notion “d’effet” et “d’effet de bord”
- Comprendre pourquoi vouloir contrôler ces effets
- Présenter une manière de faire avec un langage adapté
- Faire l’apologie de la programmation fonctionnelle

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

https://www.haskell.org/

Ça veut dire qu'il est avancé !



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

https://www.haskell.org/

Approche fonctionnelle

Inspirée du λ-calcul

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

VS

Approche impérative

Inspirée des machines de Turing

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

On n'y manipule que des fonctions et des expressions

Ça veut dire qu'il est avancé !

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

https://www.haskell.org/

Approche fonctionnelle

Inspirée du λ-calcul

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

VS

Approche impérative

Inspirée des machines de Turing

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

Où on ne manipule que des fonctions pures… et où il est impossible d'avoir des effets de bords

On n'y manipule que des fonctions et des expressions

Ça veut dire qu'il est avancé !

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

https://www.haskell.org/

Quelques définitions

Fonction pure

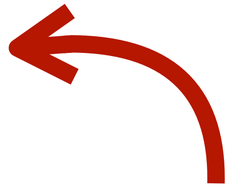
- Totale
- Déterministe
- Sans effets (uniquement capable de calculer des choses sans dépendance)

Fonction impure

- Toutes les fonctions qui ne sont pas pures

Quelques définitions

- Relativement difficile à tester (la seule méthode est l'injection de dépendances)
- Relativement difficile à optimiser
- On aimerait ne jamais en avoir…

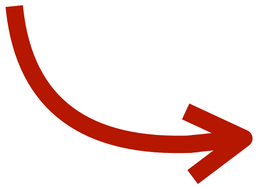


Fonction impure

- Toutes les fonctions qui ne sont pas pures

Fonction pure

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



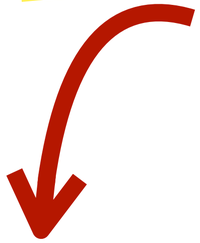
- Très facile à tester
- Très facile à optimiser (β-reduction jusque'a la forme normale)
- On aimerait ne programmer qu'avec des fonctions pures !

Haut les cœurs !

Ne programmons plus qu’avec des fonctions pures !

Haut les cœurs !

Ne programmons plus qu'avec des fonctions pures !



Un programme composé exclusivement de fonctions pures
Est un programme que l'on n'exécute pas.

Haut les cœurs !

Ne programmons plus qu'avec des fonctions pures !

Un programme composé exclusivement de fonctions pures
Est un programme que l'on n'exécute pas.

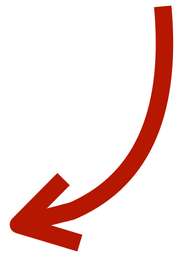
Ce qui parfois, peut être utile :

- une bibliothèque
- un assistant de preuves

Les programmes du monde réel

produisent des **effets**

Les programmes du monde réel produisent des **effets**



Soit… des choses non “calculable”

Les programmes du monde réel produisent des **effets**

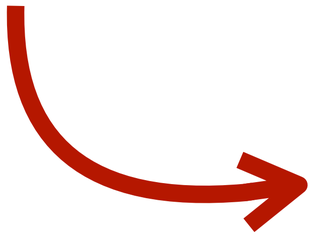
Soit… des choses non “calculable”

- Des mutations
- De l’I/O
- Des boucles infinies
- De la partialité
- Des communications avec le monde extérieur
- etc.

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

Lire/Ecrire sur la sortie standard

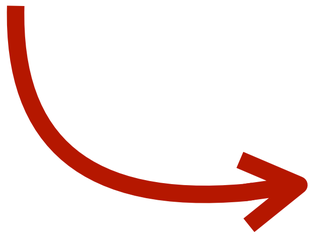
Un effet *peut être perçu comme* une **action** qui a besoin d'être **exécuté par une autorité centrale** qui devra **gérer cet effet.**



STDin/STDout

Communiquer avec une base de données

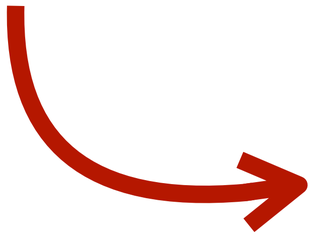
Un effet *peut être perçu comme* une **action** qui a besoin d'être **exécuté par une autorité centrale** qui devra **gérer cet effet.**



Le serveur de la base de données

L'aléatoire

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



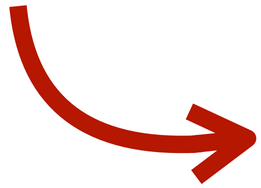
Le système d'exploitation

Et *quid* des effets de bords ?

> “un effet de bord est un effet qui **n’est pas reflété dans la signature de type de la fonction qui l’exécute.**”

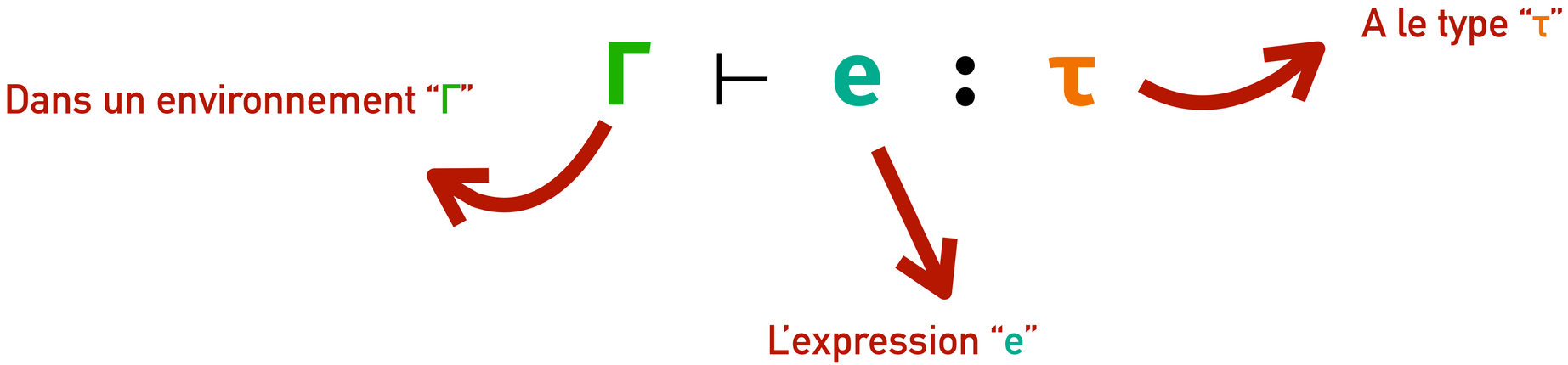
Et *quid* des effets de bords ?

“un effet de bord est un effet qui **n’est pas reflété dans la signature de type de la fonction qui l’exécute.**”

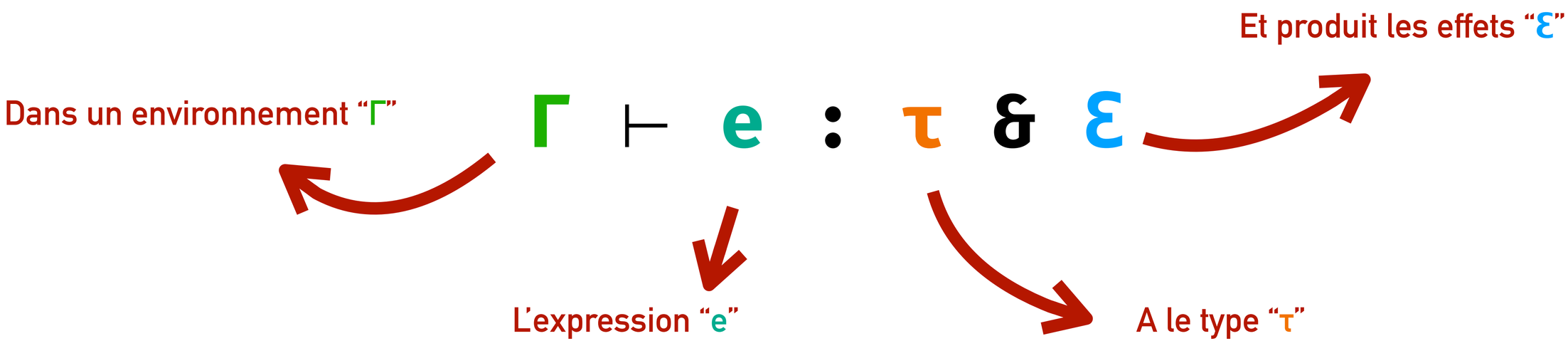


```
> print_endline (* écrit sur la sortie *)
> string → unit <fun>
```

Dans beaucoup de langages statiquement typés on a :



Et ce que l’on voudrait :



$$\Gamma \vdash e : \tau$$

```
> print_endline (* écrit sur la sortie *)
> string → unit <fun>
```

$$\Gamma \vdash e : \tau \ \& \ \varepsilon$$

```
> print_endline (* écrit sur la sortie *)
> string → unit & <console> <fun>


> read_db_and_print
> unit → unit & <console, db> <fun>
```

$$\Gamma \vdash e : \tau$$

```
> print_endline (* écrit sur la sortie *)
> string → unit <fun>
```

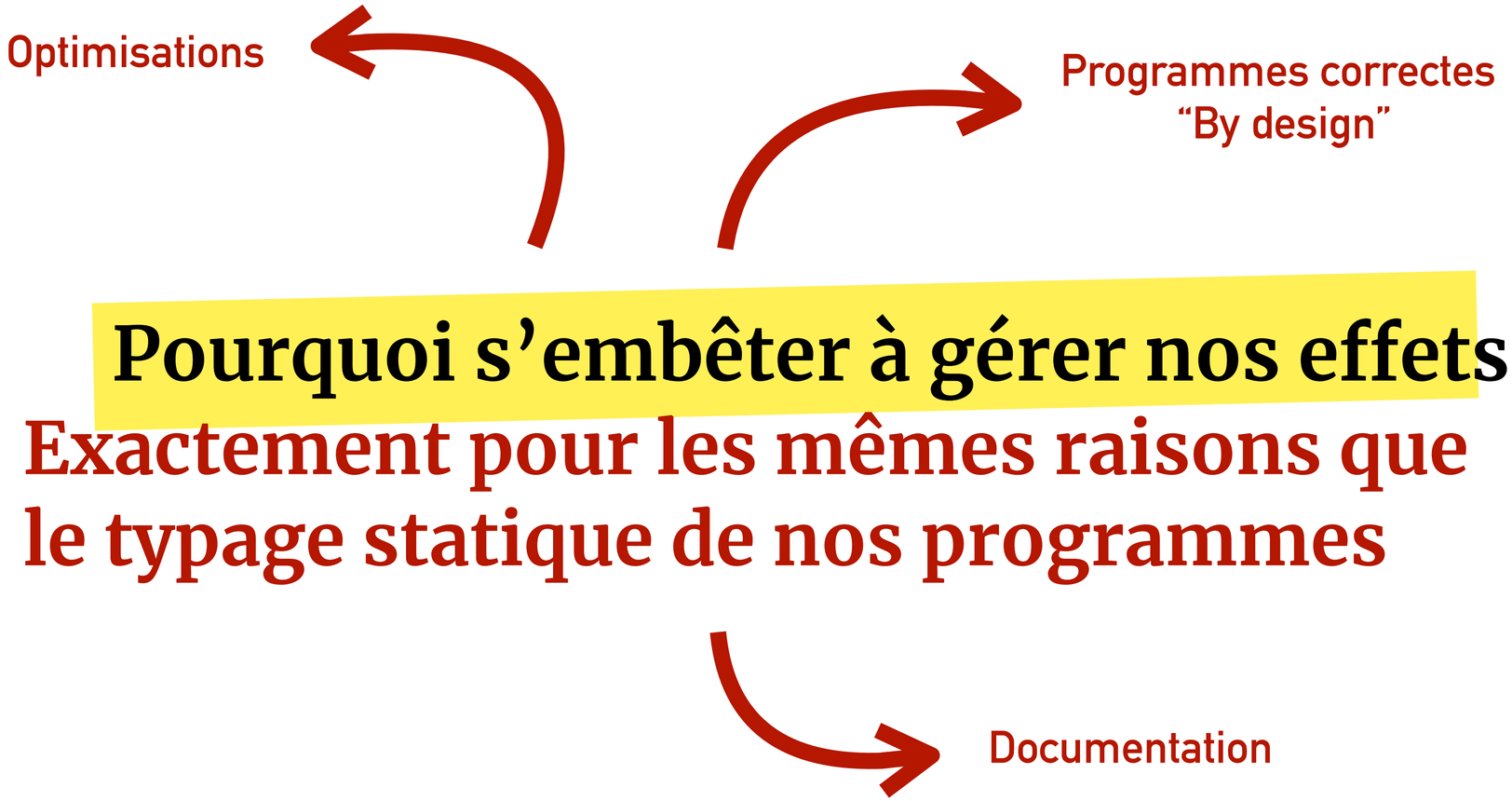
Pourquoi ?
Est-ce juste une hystérie de programmeur fonctionnel ?

$$\Gamma \vdash e : \tau \ \& \ \varepsilon$$

```
> print_endline (* écrit sur la sortie *)
> string → unit & <console> <fun>


> read_db_and_print
> unit → unit & <console, db> <fun>
```

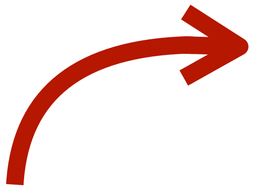
Pourquoi s’embêter à gérer nos effets

Exactement pour les mêmes raisons que le typage statique de nos programmes



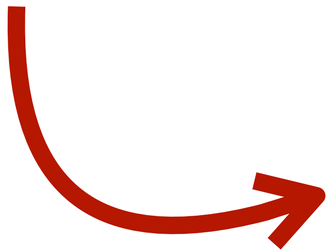
Documentation pour le compilateur = optimisation

Documentation



L'intuition sur ce que fait la fonction semble correcte

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



Par contre… ici

Documentation

L'intuition sur ce que fait la fonction semble correcte

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

on_countries : country list -> unit
```

Par contre… ici

```
let on_countries countries =
   let _ = List.map (fun country ->
      launch_a_funcking_rocket_to country
   ) countries in ()
```

Au final, la signature ne ment pas tant que ça…

Documentation

L'intuition sur ce que fait la fonction semble correcte

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

on_countries : country list -> unit
```

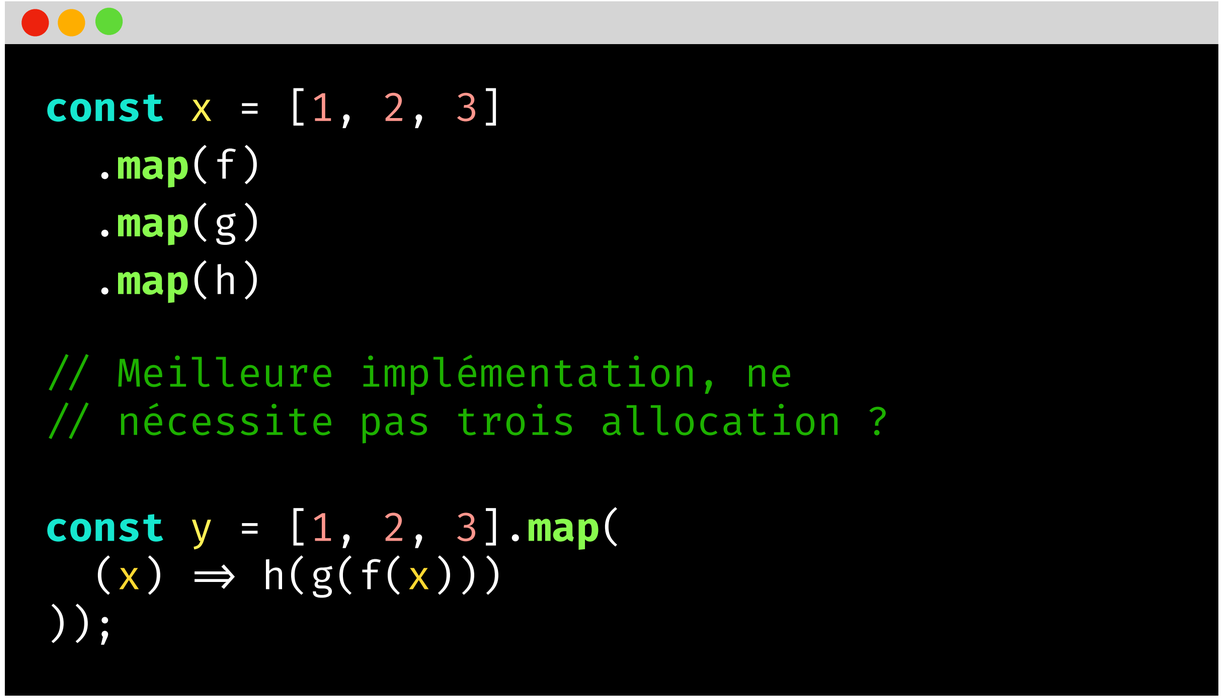
Par contre… ici

```
let on_countries countries =
   let _ = List.map (fun country ->
      print_endline country
   ) countries in ()
```

Ça aurait pût être pire !!!

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

Documentation & optimisation
Un exemple



```
const x = [1, 2, 3]
  .map(f)
  .map(g)
  .map(h)

// Meilleure implémentation, ne
// nécessite pas trois allocation ?

const y = [1, 2, 3].map(
  (x) ⇒ h(g(f(x)))
));
```

Documentation & optimisation
Un exemple

```
const x = [1, 2, 3]
  .map(f)
  .map(g)
  .map(h)

// Meilleure implémentation, ne
// nécessite pas trois allocation ?

const y = [1, 2, 3].map(
  (x) ⇒ h(g(f(x)))
));
```

Est-ce réellement vrai ?

Documentation & optimisation
Un exemple

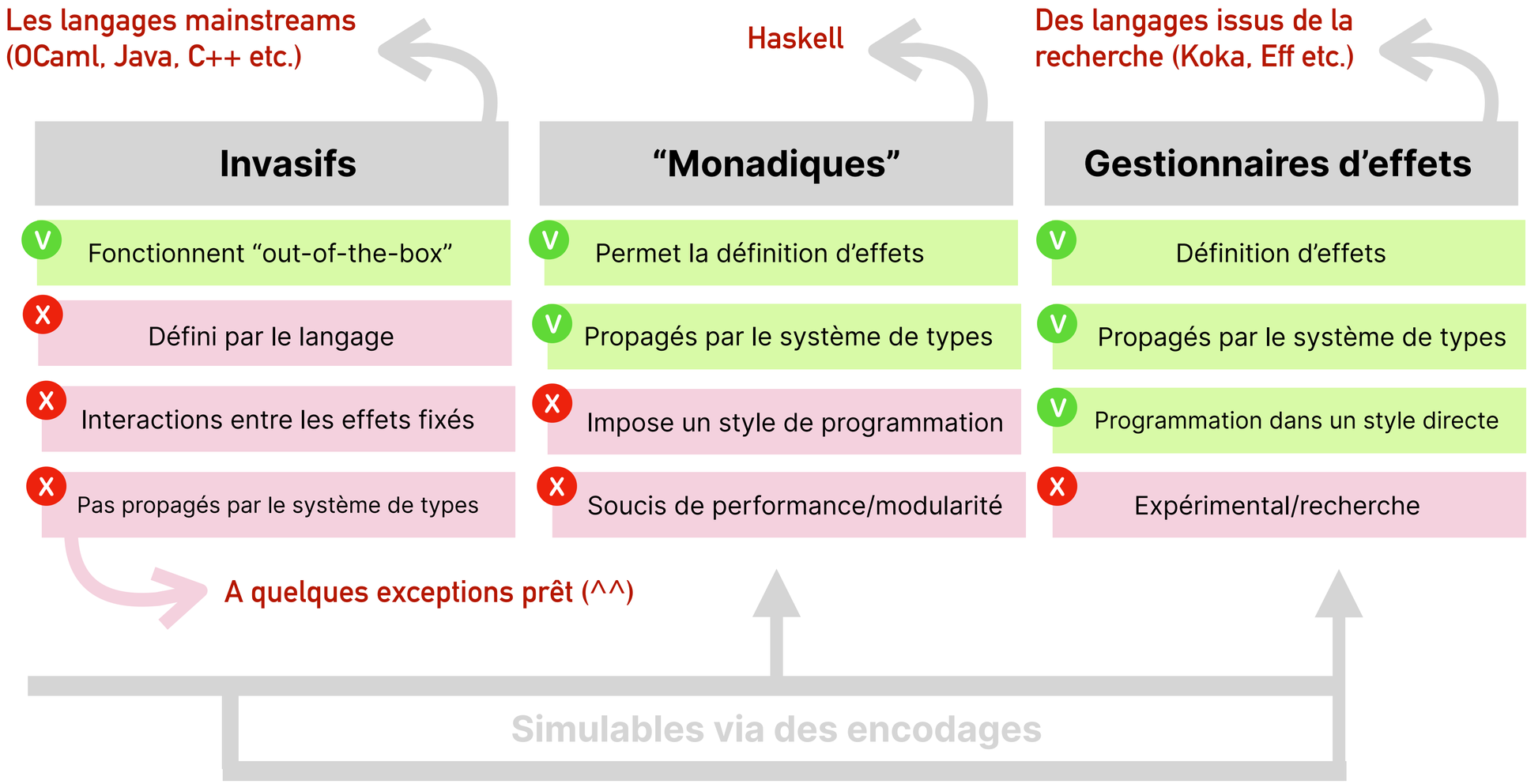
```
const x = [1, 2, 3]
  .map(f)
  .map(g)
  .map(h)

// Meilleure implémentation, ne
// nécessite pas trois allocation ?

const y = [1, 2, 3].map(
  (x) ⇒ h(g(f(x)))
));
```

Est-ce réellement vrai ?

Oui, si f, g et h sont pures
(Donc qu'elles ne produisent aucun effet)

```
const f = (x) ⇒ { console.log("foo"); return x ; }
const g = (x) ⇒ { console.log("bar"); return x ; }
const h = (x) ⇒ { console.log("aie"); return x ; }
```

Les effets dans les langages de programmation



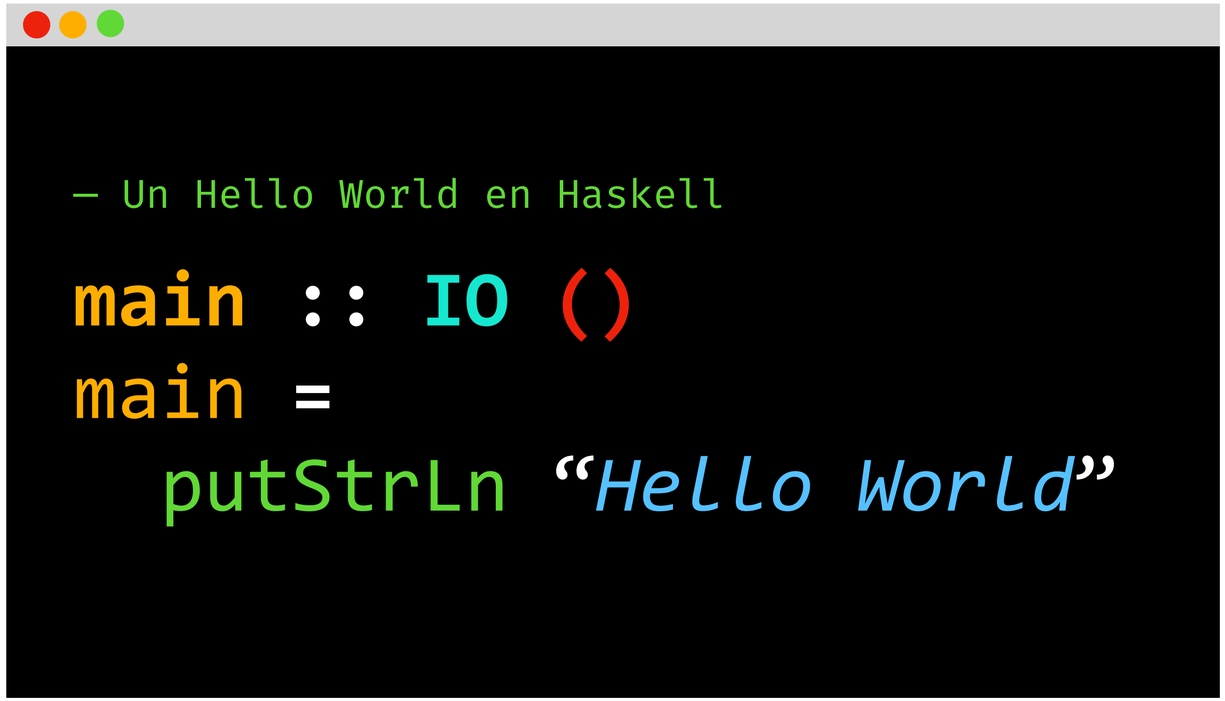
Retour sur le cas Haskell

Retour sur le cas Haskell

“IO ()” est en fait équivalent à :

unit & <io>

Retour sur le cas Haskell



- En Haskell, on peut marquer une fonction comme “exécutant un effet” au moyen du type **IO t**.
- Une expression de type **IO t** ne sera exécutée **que par le Runtime Haskell**.
- En Haskell, **on décrit des programmes** qui seront interprétés par le runtime de Haskell.

Retour sur le cas Haskell

IO dénote la présence d'un effet.

- En Haskell, on peut marquer une fonction comme "exécutant un effet" au moyen du type **IO t**.

- Une expression de type **IO t** ne sera exécutée **que par le Runtime Haskell**.

Runtime qui est testé et éprouvé

- En Haskell, **on décrit des programmes** qui seront interprétés par le runtime de Haskell.

Retour sur le cas Haskell

IO dénote la présence d'un effet.

- En Haskell, on peut marquer une fonction comme "exécutant un effet" au moyen du type **IO t**.

- Une expression de type **IO t** ne sera exécutée **que par le Runtime Haskell**.

Runtime qui est testé et éprouvé

- En Haskell, **on décrit des programmes** qui seront interprétés par le runtime de Haskell.

Ok, on a IO et les autres ?

Comment représenter
La partialité, les erreurs, etc?

En transformant les effets en valeurs

Par exemple :

```haskell
— La partialité

data Maybe a =
  | Just a
  | Nothing
```

```haskell
— Un calcul échouable

data Either a b =
  | Left a
  | Right b
```

Et il existe énormément d'autres encodages, correspondant à pleins de types d'effets différents.

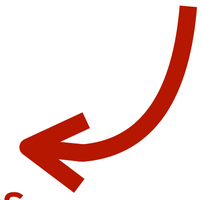
IO étant l'effet le plus primitif, il est celui qui est confiné aux extrémités du programmes : **son interprétation par le runtime.**

Construire un programme revient à **décrire** l'ensemble des effets que l'on veut utiliser au moyen de **types de données adéquats** et de fournir une **transformation pour ce type vers IO** :

```haskell
— Un exemple naïf

main :: IO ()
main =
  case a_failable_function () of
    Left error -> printLn ("an error !:" ++ error)
    Right _success -> printLn "Everything's good"
```

Il se trouve que beaucoup de types qui **caractérisent des effets,** dont IO, respectent l'interfaces et les lois d'une monade, mais c'est de l'ordre du détail.

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



Nous sommes saufs !
Merci les effets explicites

Est-il possible d’aller plus loin ?

- IO est clairement un pas dans la bonne direction
- Mais comment tester correctement une valeur IO ?

Premières volée de question ?

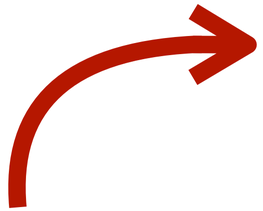
Les effets algébriques et leurs gestionnaires à la rescousse

Avec le langage Koka*

Formellement, quécé

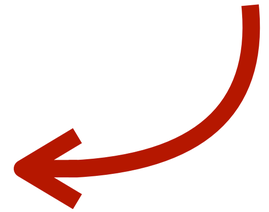
- Définir un effet, comme une collection de simple constructeurs (ou des fonctions)
- Pour exécuter une fonction qui propage des effets, il faut fournir un interpréteur pour les effets propagés par la fonction.

Formellement, quécé

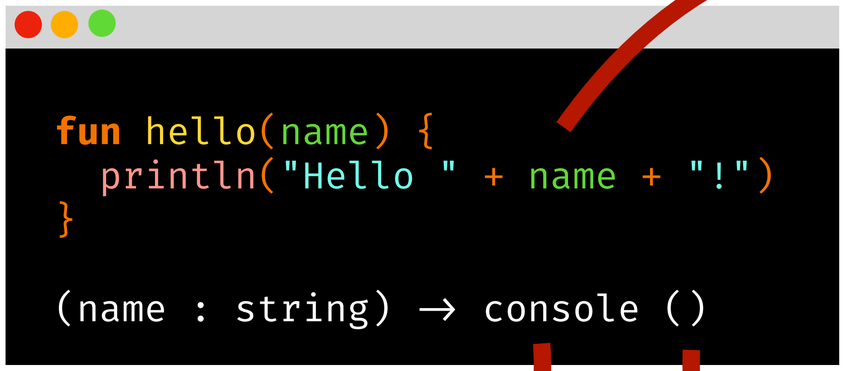


Les constructeurs caractérisent les
Opérations relatives à l'effet.
Par exemple, pour un état mutable : GET et SET.
Pour console : READ, PRINT

- Définir un effet, comme une collection de 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 devient l'autorité centrale dont nous avions parlé



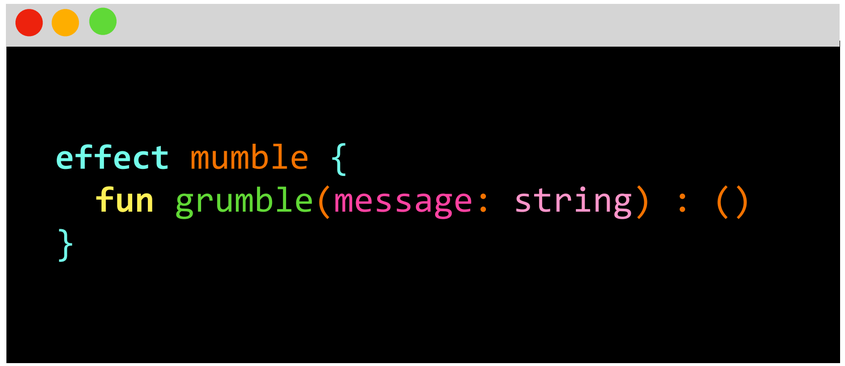
"println" propage l'effet console

Effets exécutés par la fonction

Type de retour de la fonction

Le compilateur vérifie statiquement que chaque **exécution d'effets** possède, dans son **scope** un interpréteur pour l'effet propagé.

Définissons un effet



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

Définissons un effet

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

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

Définissons un effet

```
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
```

Définissons un effet

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

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


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

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

Définition d'un interpréteur pour "grumble"

Exécution via l'interpréter

Définissons un effet

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

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


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

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

Définition d'un interpréteur pour "grumble"

Exécution via l'interpréter

Le programme compile car L'effet **mumble** a été réduit, via un gestionnaire dans un effet (**console**) qui possède un interpréteur.

Définissons un effet

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

fun mumbling() : mumble () {
  grumble("Hello World!”)
  grumble(“Goodbye World!")
}

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

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

N’affiche que “Hello World!”

Définissons un effet

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

fun mumbling() : mumble () {
  grumble("Hello World!”)
  grumble(“Goodbye World!")
}

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

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

N’affiche que “Hello World!”

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

Ils sont analogues à **des exceptions résumables :** throw = la performance d’un effet et catch est un gestionnaire.

Transcription d’un programme Kotlin a Koka

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

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

```koka
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

Exécution du programme

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

Exécution du programme

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

Exécution du programme

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

Exécution du programme

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

Exécution du programme

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask("What is your name?")
  show("Hello " + name)
}

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

Exécution du programme

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

Exécution du programme

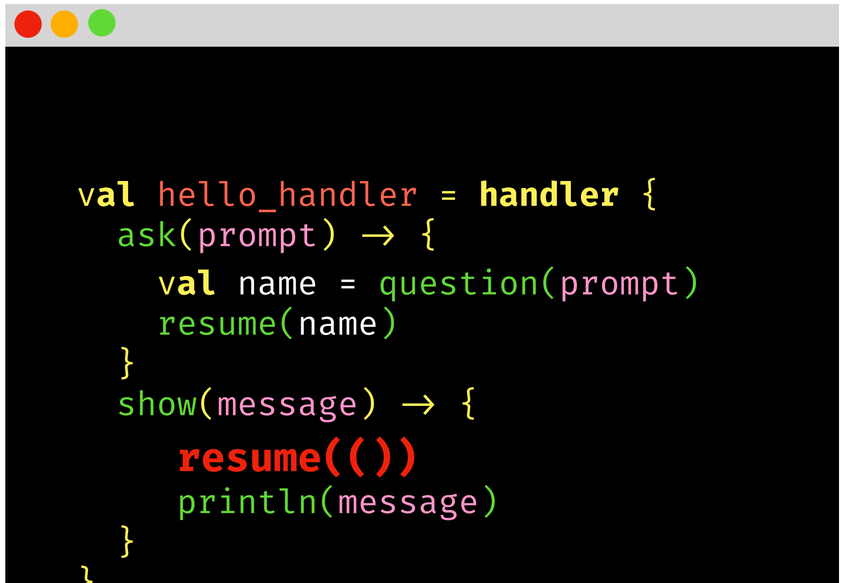
```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

Exécution du programme

```
effect interaction {
  fun show(message: string) : ()
  fun ask(message: string) : string
}
fun program() : interaction () {
  val name = ask(“What is your name?”)
  show(“Hello “ + name)
}

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

- **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)**

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



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

Tester un programme

Il suffit de fournir un interpréteur !

```
val test_handler = handler {
  ask(_) -> {
    val acc = get()
    set(accumulator + “;Xavier”)
    resume(“Xavier”)
  }
  show(message) -> {
     val acc = get()
     set(accumulator + “;” + message)
     resume(())
  }
}

// (()
    -> <interaction, state<string>> ()
    )
    -> state<string> ()
```

Ici on interprète le programme avec un **état mutable**. A la fin de l’interprétation, on a un état qui stocke une chaine de caractères.

```
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
}
```

En résumé ?

Les effets algébriques ?

La définition d’effets correspond à **“définir dénotationnellement”** un effet.

La définition du gestionnaire en donne son sens **opérationnel**.

En résumé ?

Les effets algébriques ?

- Ils permettent de préserver un **style naturel**
- Les effets connus par le compilateur sont **optimisables** (ie: SET + SET)
- L'intérpéteur à du **pouvoir sur le flot d'exécution** (permet d'implémenter par exemple des mécanismes plus spécialisés)
- Ils séparent systématiquement l'expression dénotationnelle d'un programme de son expression opérationnelle
- Ils reflètent les effets propagés dans le système de type (via du row polymorphism pour Koka)
- Ils sont facilement testable (l'interpréter peut aussi être adapté au contexte, Browser par exemple)

C'est une forme d'injection de dépendance avec avec un gestionnaire du flot de contrôle. Ça se marrie plutôt bien avec le core fonctionnel et le shell impératif.

Quels langages Mainstream

Proposent des effets algébriques ?

Quels langages Mainstream

Proposent des effets algébriques ?

Aucuns

Quels langages Mainstream

Proposent des effets algébriques ?

Aucuns

C’est une fonctionnalité expérimentale mais prometteuse.
Peut être qu’un jour, à la manière de la vérification statique des types, la gestion d’effets deviendra mainstream !

Erreurs à éviter : représenter la logique métier sous forme d’effets

FIN

Merci beaucoup !

Xavier Van de Woestyne - Memo.bank - https://xvw.github.io - @vdwxv - xvw@merveille.town