Kotlin pour de la FP ?

Une brève introduction à Arrow

Xavier Van de Woestyne · xvw.github.io · xvw@merveilles.town

Qu’est-ce que Kotlin

*“Kotlin est un langage de programmation **orienté objet** et **fonctionnel**, avec un **typage statique** qui permet de compiler pour la machine virtuelle Java, JavaScript, et vers plusieurs plateformes en natif (grâce à LLVM)”*.

De : la sources de toute vérités, **Wikipedia**

Qu’est-ce que Kotlin

*“Kotlin est un langage de programmation **orienté objet** et **fonctionnel**, avec un **typage statique** qui permet de compiler pour la machine virtuelle Java, JavaScript, et vers plusieurs plateformes en natif (grâce à LLVM)”.*

Ça semble être la définition d’un meilleur JAVA

De : la sources de toute vérités, **Wikipedia**

“***Scala*** *intègre les paradigmes de programmation* ***orientée objet*** *et de programmation* ***fonctionnelle****, avec un* ***typage statique***”.

Aussi de : la sources de toute vérités, **Wikipedia**

Qu’est-ce que Kotlin

“*Kotlin est un langage de programmation* ***orienté objet*** *et* ***fonctionnel****, avec un* ***typage statique*** *qui permet de compiler pour la machine virtuelle Java, JavaScript, et vers plusieurs plateformes en natif (grâce à LLVM)*”.

Ça semble être la définition d’un meilleur JAVA

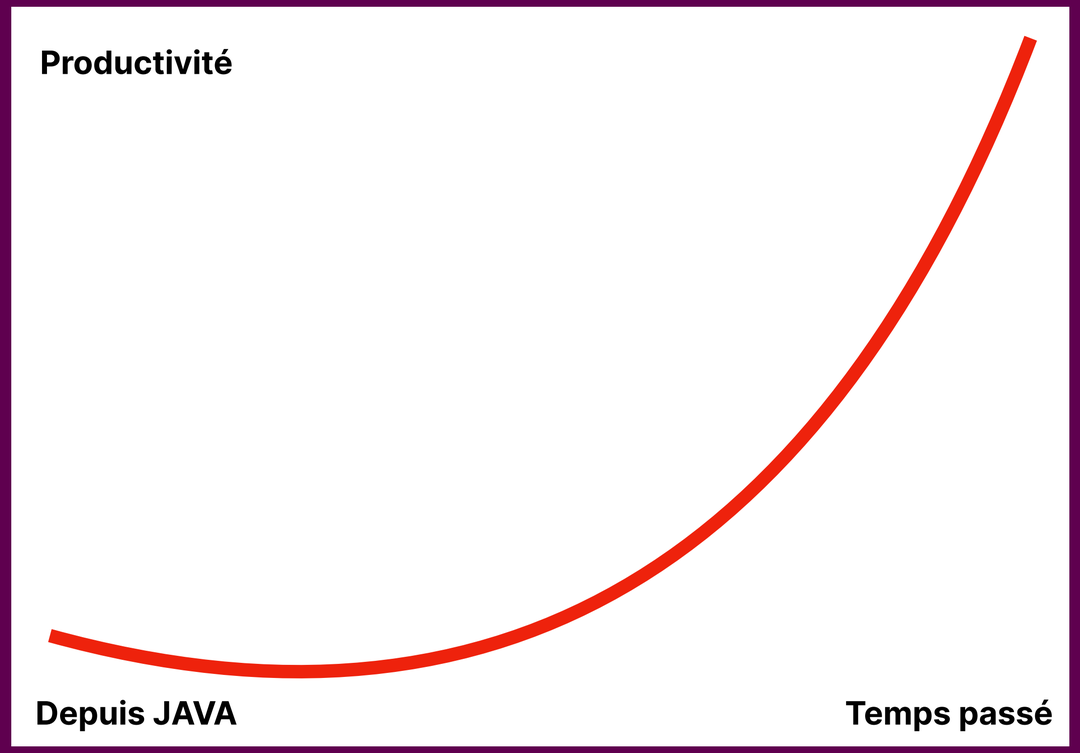
De : la sources de toute vérités, **Wikipedia**

≈Groovy statiquement typé

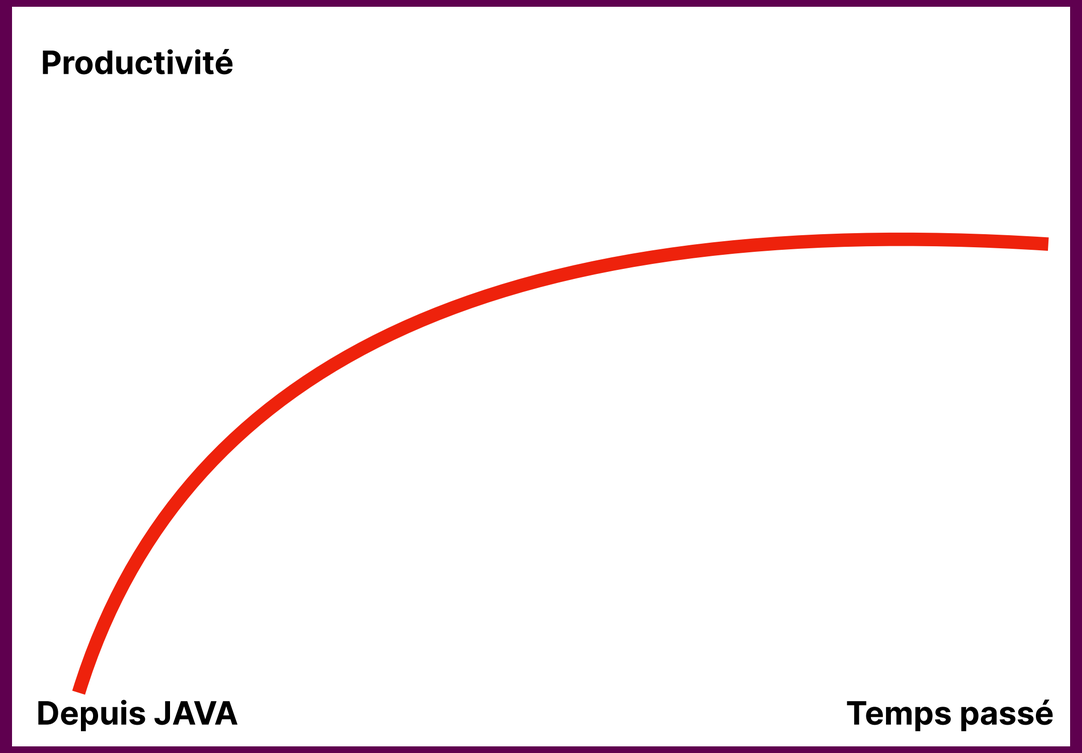
Scala ≈ Kotlin ?

Modulo cible de compilation et … le nom (aha)

Scala



Kotlin



≈Groovy statiquement typé

Scala ≈ Kotlin ?

Modulo cible de compilation et … le nom (aha)

*** courbes au doigt mouillé**

Une autre sources de toute vérités

Moi

Donc choisir **Kotlin** pour démarrer un projet ambitieux est pertinent ! On peut **rapidement scaler** une équipe qui vient de JAVA !

Scala ≈ Kotlin

Qu’est-ce qui justifie une telle différence ?

Kotlin est un langage “fonctionnel”

Scala ≈ Kotlin

Qu’est-ce qui justifie une telle différence ?

Kotlin est un langage “fonctionnel”

Scala ≈ Kotlin

Qu’est-ce qui justifie une telle différence ?

Moi

Un langage fonctionnel est un langage qui permet de manipuler **des fonctions comme des valeurs classiques** du langage. Donc les **prendre en argument** et les **renvoyer**. (Avoir des **Lambda**)

Kotlin est un langage “fonctionnel”

Scala ≈ Kotlin

Qu’est-ce qui justifie une telle différence ?

Moi

Un langage fonctionnel est un langage qui permet de manipuler **des fonctions comme des valeurs classiques** du langage. Donc les **prendre en argument** et les **renvoyer**. (Avoir des **Lambda**)

```
fun main() =
  listOf("Hello", "Lambda Lille").forEach {
      println(it)
  }
```

Kotlin est un langage “fonctionnel”

Scala ≈ Kotlin

Qu’est-ce qui justifie une telle différence ?

Moi

Un langage fonctionnel est un langage qui permet de manipuler **des fonctions comme des valeurs classiques** du langage. Donc les **prendre en argument** et les **renvoyer**. (Avoir des **Lambda**)

C’est une lambda !

```
fun main() =
    listOf("Hello", "Lambda Lille").forEach {
        println(it)
    }
```

Autre personne

Cette définition est NULLE ! Elle **est trop générale** et il devient difficile de trouver un langage qui n’est pas fonctionnel !

Kotlin est un langage “fonctionnel”

Scala ≈ Kotlin

Qu’est-ce qui justifie une telle différence ?

Moi

Un langage fonctionnel est un langage qui permet de manipuler **des fonctions comme des valeurs classiques** du langage. Donc les **prendre en argument** et les **renvoyer**. (Avoir des **Lambda**)

C’est une lambda !

```
fun main() =
    listOf("Hello", "Lambda Lille").forEach {
        println(it)
    }
```

Autre personne

Cette définition est NULLE ! Elle **est trop générale** et il devient difficile de trouver un langage qui n’est pas fonctionnel !

Kotlin est un langage “fonctionnel”

Scala permet de programmer dans un style fonctionnel

Scala ≈ Kotlin

Qu’est-ce qui justifie une telle différence ?

Moi

Un langage fonctionnel est un langage qui permet de manipuler **des fonctions comme des valeurs classiques** du langage. Donc les **prendre en argument** et les **renvoyer**. (Avoir des **Lambda**)

C’est une lambda !

```
fun main() =
    listOf("Hello", "Lambda Lille").forEach {
        println(it)
    }
```

Moi

Donc choisir **Kotlin** pour démarrer un projet ambitieux est pertinent ! On peut **rapidement scaler** une équipe qui vient de JAVA !

Moi

Mais comme il y a des **fonctions d'ordre supérieure** ne Pourrions-nous pas programmer comme en **Haskell** ?

Je veux des ADTs

Je veux des ADTs

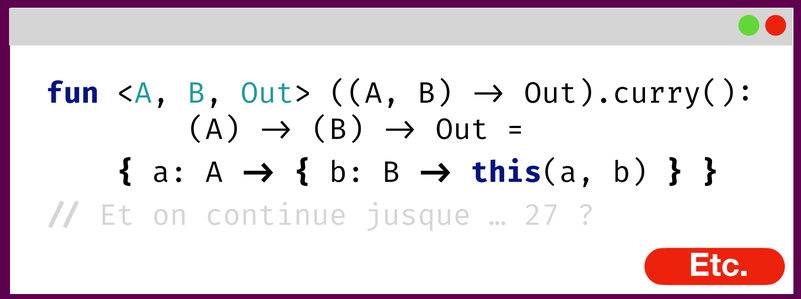
Sealed class

Je veux de l’application partielle

Je veux des ADTs

Sealed class

Je veux de l'application partielle



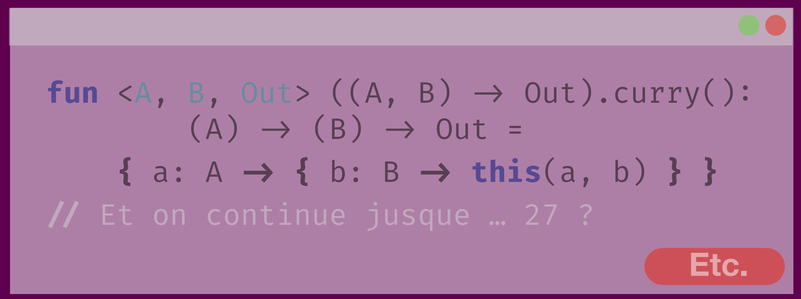
```
fun <A, B, Out> ((A, B) → Out).curry():
        (A) → (B) → Out =
    { a: A → { b: B → this(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux des ADTs

Sealed class

Je veux de l’application partielle



```
fun <A, B, Out> ((A, B) → Out).curry():
        (A) → (B) → Out =
    { a: A → { b: B → this(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux repousser mes effets aux extrémités du programme

Suspensions

Sealed class

Je veux de l’application partielle

```
fun <A, B, Out> ((A, B) → Out).curry():
        (A) → (B) → Out =
    { a: A → { b: B → this(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux généraliser sur des types

Higher Kinded Polymorphism

Suspensions

Sealed class

Je veux de l'application partielle

```
fun <A, B, Out> ((A, B) → Out).curry():
        (A) → (B) → Out =
    { a: A → { b: B → this(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux généraliser sur des types

Higher Kinded Polymorphism

Suspensions

F-bound polymorphism

```
fun <Inner :
    Shape<Inner, *>,
    Param, Out
 > ...
```

Je veux de l'application partielle

```
fun <A, B, Out> ((A, B) → Out).curry():
        (A) → (B) → Out =
    { a: A → { b: B → this(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux généraliser sur des types

Higher Kinded Polymorphism

F-bound polymorphism

```
fun <Inner :
    Shape<Inner, *>,
    Param, Out
 >  ...
```

Defunctionalization

Lightweight higher-kinded polymorphism

Jeremy Yallop and Leo White

University of Cambridge

Abstract. Higher-kinded polymorphism —i.e. abstraction over type *constructors*— is an essential component of many functional programming techniques such as monads, folds, and embedded DSLs. ML-family languages typically support a form of abstraction over type constructors using functors, but the separation between the core language and the

Des … GADTs ?

Je veux de l'application partielle

```
ut).curry():
(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux généraliser sur des types

Je veux repousser mes effets aux

Church Encoding

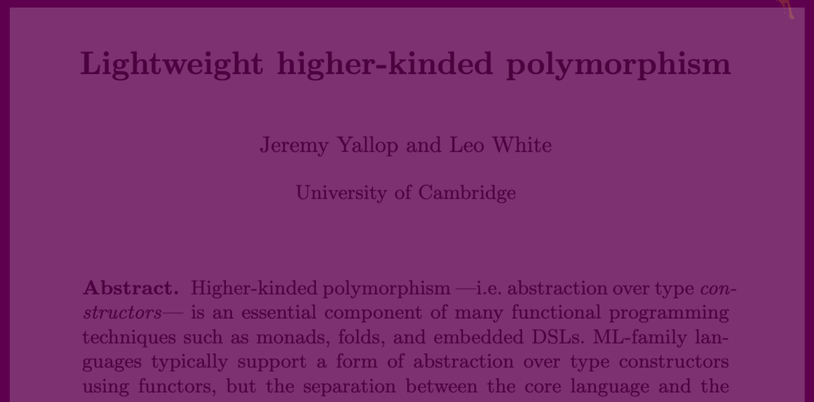
Suspensions

Defunctionalization

F-bound polymorphism

Sealed class

```
fun <Inner :
    Shape<Inner, *>,
    Param, Out
  > ...
```



Lightweight higher-kinded polymorphism

Jeremy Yallop and Leo White

University of Cambridge

Abstract. Higher-kinded polymorphism —i.e. abstraction over type *constructors*— is an essential component of many functional programming techniques such as monads, folds, and embedded DSLs. ML-family languages typically support a form of abstraction over type constructors using functors, but the separation between the core language and the

Des … GADTs ?

Je veux de l'application partielle

```
ut).curry():
(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux généraliser sur des types

Je veux repousser mes effets aux

Church Encoding

+

Le lemme de Yoneda

polymorphism

Suspensions

Defunctionalization

```
fun <Inner :
    Shape<Inner, *>,
    Param, Out
  > ...
```

Lightweight higher-kinded polymorphism

Jeremy Yallop and Leo White

University of Cambridge

Abstract. Higher-kinded polymorphism —i.e. abstraction over type *constructors*— is an essential component of many functional programming techniques such as monads, folds, and embedded DSLs. ML-family languages typically support a form of abstraction over type constructors using functors, but the separation between the core language and the

Des … GADTs ?

Je veux de l’application partielle

Je veux généraliser sur des types

Je veux repousser mes effets aux

Church Encoding

+

Le lemme de Yoneda

Suspensions

Defunctionalization

polymorphsim

Je jure que ce n’est pas une blague :
https://gist.github.com/jbgi/208a1733f15cdcf78eb5

Via Wikipedia

Lightweight higher-kinded polymorphism

Jeremy Yallop and Leo White

University of Cambridge

Abstract. Higher-kinded polymorphism —i.e. abstraction over type *constructors*— is an essential component of many functional programming techniques such as monads, folds, and embedded DSLs. ML-family languages typically support a form of abstraction over type constructors using functors, but the separation between the core language and the

```
fun <Inner :
    Shape<Inner, *>,
    Param, Out
  > ...
```

Des … GADTs ?

Je veux de l'application partielle

```
fun <A, B, Out> ((A, B) → Out).curry():
    (B) → Out =
        { a: A → { b: B → this(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Continuation Passing Style partout !

Suspensions

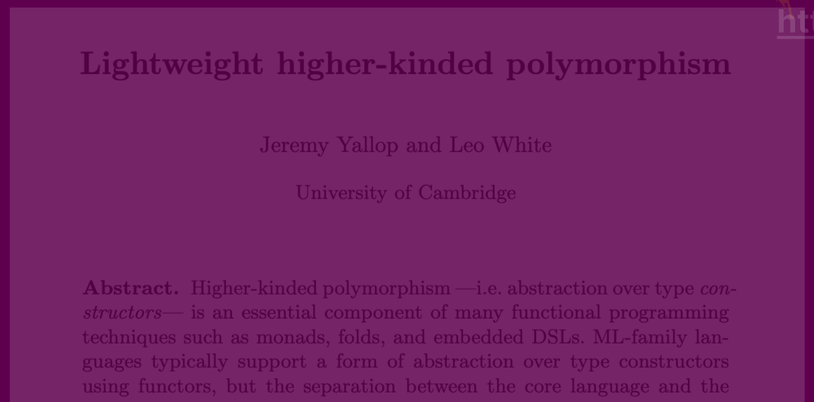
Le lemme de Yoneda

Defunctionalization

Je jure que ce n'est pas une blague :
https://gist.github.com/jbgi/208a1733f15cdcf78eb5

Via Wikipedia

```
fun <Inner :
    Shape<Inner, *>,
    Param, Out
> ...
```



Lightweight higher-kinded polymorphism

Jeremy Yallop and Leo White

University of Cambridge

Abstract. Higher-kinded polymorphism —i.e. abstraction over type *constructors*— is an essential component of many functional programming techniques such as monads, folds, and embedded DSLs. ML-family languages typically support a form of abstraction over type constructors using functors, but the separation between the core language and the

Je veux de l'application partielle

Des … GADTs ?

```
fun <A, B, Out> ((A, B) → Out).curry():
    (B) → Out =
        { a: A → { b: B → this(a, b) } }
// Et on continue jusque … 27 ?
```

Etc.

Je veux généraliser sur des types

Je veux repousser mes effets aux

Passing style

Polymorphism

+

Le lemme de Yoneda

F-bound polymorphsim

Sealed class

Suspensions

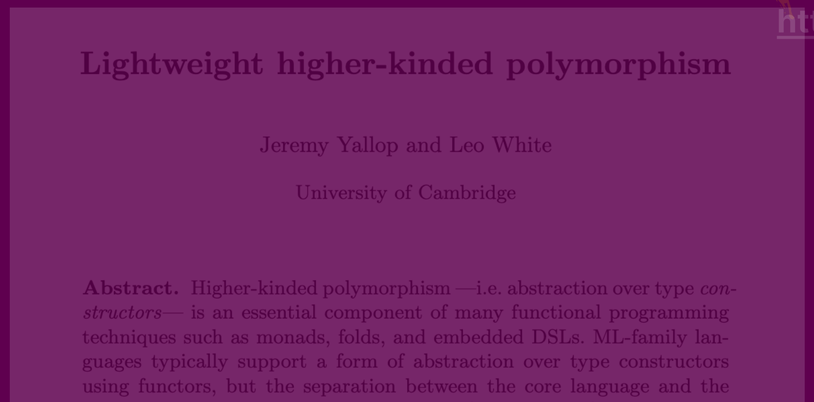
Defunctionalization

Et donc, Kotlin

Je jure que ce n'est pas une blague :
https://gist.github.com/jbgi/208a1733f15cdcf78eb5

Via Wikipedia

```
fun <Inner :
    Shape<Inner, *>,
    Param, Out
> ...
```



Lightweight higher-kinded polymorphism

Jeremy Yallop and Leo White

University of Cambridge

Abstract. Higher-kinded polymorphism —i.e. abstraction over type *constructors*— is an essential component of many functional programming techniques such as monads, folds, and embedded DSLs. ML-family languages typically support a form of abstraction over type constructors using functors, but the separation between the core language and the

Et donc, Kotlin

Un excellent langage de **OOP** de **l'école Scandinave***

*** Didier Plaindoux**

Et donc, Kotlin

Un excellent langage de **OOP** de **l'école Scandinave***
Où les combinateurs fonctionnels permettent d'écrire
du code OOP élégamment

*** Didier Plaindoux**

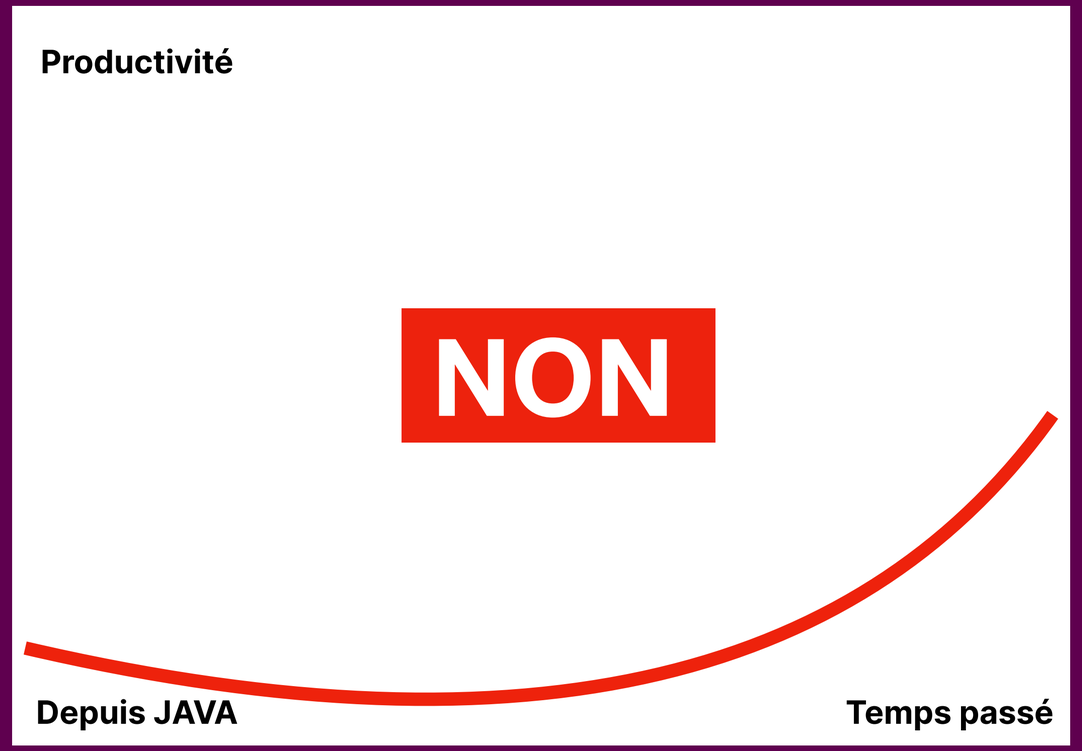
Est-ce que l’on devrait importer des idiomes “purement fonctionnels” en Kotlin?

Et donc, Kotlin

Un excellent langage de **OOP** de **l’école Scandinave***
Où les combinateurs fonctionnels permettent d’écrire du code OOP élégamment

*** Didier Plaindoux**

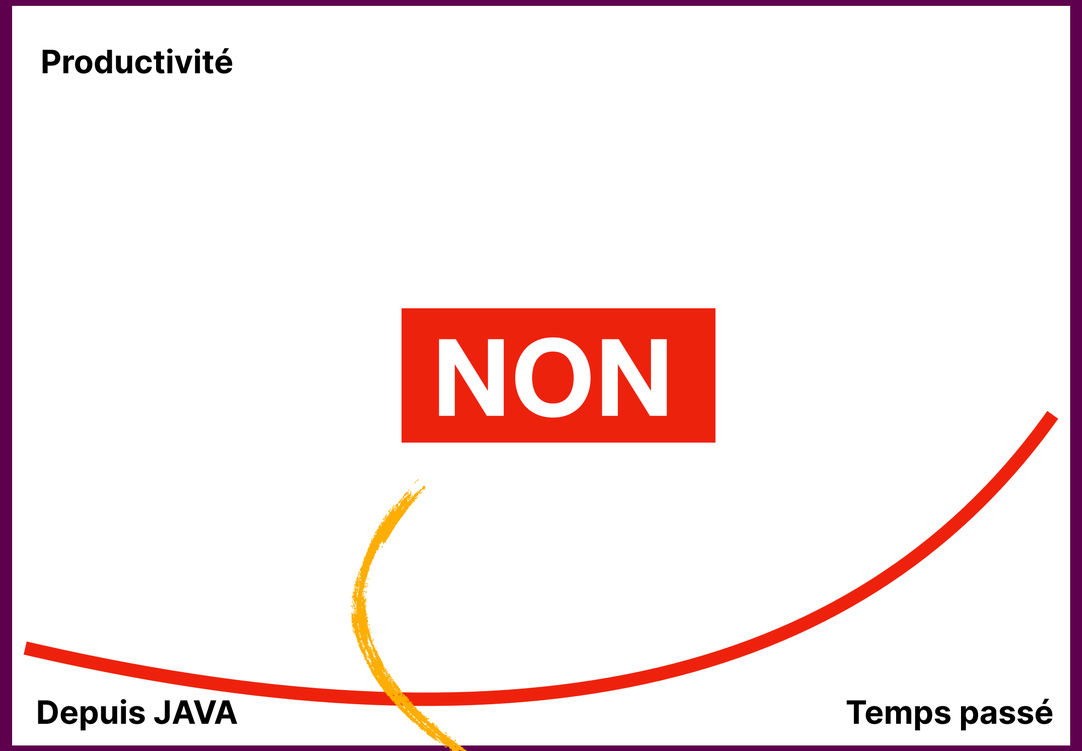
Kotlin purement fonctionnellement



*** courbes au doigt mouillé**

Une autre sources de toute vérités

Kotlin purement fonctionnellement



Les encodages nécessaires pour tenter de se rapprocher de l'expressivité de Haskell, OCaml ou Scala nécessite largement plus d'apprentissage que de juste utiliser un de ces 3 langages...

*** courbes au doigt mouillé**

Une autre sources de toute vérités

Mais… est-ce que la cérémonie engendrée par Scala, OCaml ou Haskell (et d’autres) apporte réellement quelque chose dans l’écriture de logiciel ?

Mais… est-ce que la cérémonie engendrée par Scala, OCaml ou Haskell (et d'autres) apporte réellement quelque chose dans l'écriture de logiciel ?

Oui ! C'est pour ça que je pense qu'il vaut mieux exploiter un langage "dédié" (entre autre) à la programmation fonctionnelle pour apprendre la programmation fonctionnelle.

Nan, trop dur

Kotlin pour de la FP ?

Une brève introduction à Arrow

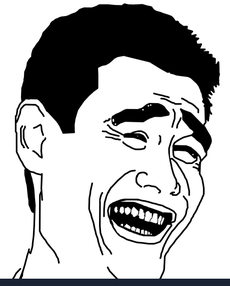
On peut passer à la partie 2
Construction d'un logiciel dans un style fonctionnel

Xavier Van de Woestyne · xvw.github.io · xvw@merveilles.town

Une amusante confusion

Kotlin, pour de la FP?
Une brève introduction à Arrow
par Xavier Van de Woestyne
@xvw@merveilles.town

En apprendre plus sur la programmation fonctionnelle avec Arrow, c'est possible ! Dans cette présentation, je vous propose de découvrir comment s'en servir avec un cas très pratique !



Une mention à ce compte Twitter?
Voila qui est étrange !

N’étant plus sur Twitter, je n’ai hélas pas pu lever l’ambiguïté :(

Ma soumission à LambdaLille

La communication Twitter


ARROW_KT ?
JE PARLAIS DE MONADES DANS
LA BICATÉGORIE DES PROFONCTEURS....
imgflip.com

Blague à part, je n’ai aucune idée de ce que ça veut dire

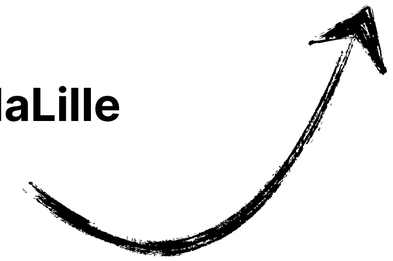
Mise en contexte

Il y a ~deux ans

`grim` sur **LambdaLille**

`didier_p` sur **LambdaLille**

`xvw (moi)` sur **LambdaLille**



Créer une bibliothèque pour arrêter de copier/coller sans arrêt des abstractions récurrentes de projets en projets

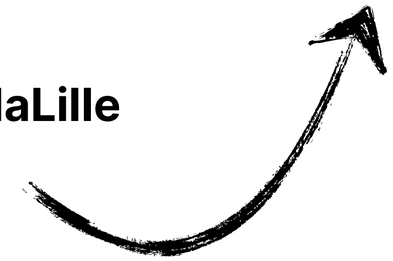
Mise en contexte

Il y a ~deux ans

`grim` sur **LambdaLille**

`didier_p` sur **LambdaLille**

`xvw (moi)` sur **LambdaLille**



Créer une bibliothèque pour arrêter de copier/coller sans arrêt des abstractions récurrentes de projets en projets

Preface: https://github.com/xvw/preface

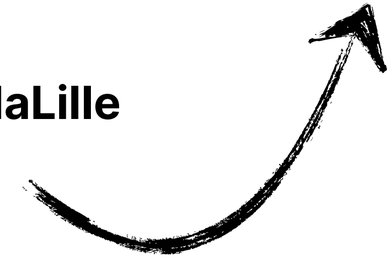
Mise en contexte

Il y a ~deux ans

`grim` sur **LambdaLille**

`didier_p` sur **LambdaLille**

`xvw (moi)` sur **LambdaLille**



Créer une bibliothèque pour arrêter de copier/coller sans arrêt des abstractions récurrentes de projets en projets

Preface: https://github.com/xvw/preface

Juste avant la libération du code

Expérimentation de l’ergonomie de la bibliothèque

Mise en contexte

Il y a ~deux ans

`grim` sur **LambdaLille**

`didier_p` sur **LambdaLille**

`xvw (moi)` sur **LambdaLille**

Créer une bibliothèque pour arrêter de copier/coller sans arrêt des abstractions récurrentes de projets en projets

Preface: https://github.com/xvw/preface

Juste avant la libération du code

Expérimentation de l'ergonomie de la bibliothèque

Wordpress:
https://github.com/xhtmlboi/wordpress
Un générateur de blog statique (utilisable)

Et cette expérience présentait un usage des Arrows

Comme elles sont relativement rarement présentées, et qu’elles n’ont jamais été réellement évoquées à LambdaLille, pourquoi ne pas en parler !

Même si le meilleur générateur de blog statique est probablement `Make`

Nous allons réimplémenter un sous-ensemble de WordPress !

Un générateur de site statique minimaliste avec Make

```
.PHONY: clear

%.html: pages/%.html header.html footer.html
  cat header.html $(<) footer.html > _build/$(@)

clear:
  rm -rf _build/

my_site: index.html about.html contact.html
```

Un générateur de site statique minimaliste avec Make

```
.PHONY: clear

%.html: pages/%.html header.html footer.html
	cat header.html $(<) footer.html > _build/$(@)

clear:
	rm -rf _build/

my_site: index.html about.html contact.html
```

Un générateur de site statique minimaliste avec Make

```
.PHONY: clear

%.html: pages/%.html header.html footer.html
	cat header.html $(<) footer.html > _build/$(@)

clear:
	rm -rf _build/

my_site: index.html about.html contact.html
```

Un générateur de site statique minimaliste avec Make

```
.PHONY: clear

%.html: pages/%.html header.html footer.html
	cat header.html $(<) footer.html > _build/$(@)

clear:
	rm -rf _build/

my_site: index.html about.html contact.html
```

Un générateur de site statique minimaliste avec Make

Très loin d'être parfait pour plusieurs points

```
.PHONY: clear

%.html: pages/%.html header.html footer.html
  cat header.html $(<) footer.html > _build/$(@)

clear:
  rm -rf _build/

my_site: index.html about.html contact.html
```

Un générateur de site statique minimaliste avec Make

Très loin d'être parfait pour plusieurs points

```
.PHONY: clear

%.html: pages/%.html header.html footer.html
	cat header.html $(<) footer.html > _build/$(@)

clear:
	rm -rf _build/

my_site: index.html about.html contact.html
```

Les dépendances ne sont pas traquées **dynamiquement**

Un générateur de site statique minimaliste avec Make

Très loin d'être parfait pour plusieurs points

```
.PHONY: clear

%.html: pages/%.html header.html footer.html
  cat header.html $(<) footer.html > _build/$(@)

clear:
  rm -rf _build/

my_site: index.html about.html contact.html
```

Qui veut écrire du bash, dans des scénarios plus complexes ?
Pas nous ! On veut faire du OCaml !

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect
end
```

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect
end
```

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect
end
```

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect
end
```

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect
end
```

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect
end
```

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect

  include Preface.Make.Freer_monad.Over
      (struct type 'a t = 'a effect end)
end
```

Premièrement, on définit tous les effets qui seront nécessaires
(Ça permettra de ne plus s'en soucier après)

```
module Effect = struct
  type 'a effect =
    | File_exists : filename → bool effect
    | Get_mtime : filename → int Preface.Try.t effect
    | Read_file : filename → string Preface.Try.t effect
    | Write_file : (filename * string) → unit Preface.Try.t effect
    | Log : string → unit effect

  include Preface.Make.Freer_monad.Over
      (struct type 'a t = 'a effect end)

  let file_exists name = perform (File_exists name)
  let get_mtime name = perform (Get_mtime name)
   (** Etc.*)
end
```

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
      if mtime > other_mtime then
        Effect.log "Header a été modifié"
      else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

On peut maintenir décrire des programme “à interpréter plus tard”
Par exemple

```
(* val my_program : int → unit Effect.t *)

let my_program other_mtime =
  let open Effect.Monad in
  let* file_exists = Effect.file_exists "header.html" in
  if file_exists then
    let* potential_mtime = Effect.get_mtime "header.html" in
    match potential_mtime with
    | Error _ → Effect.log "une erreur est survenue"
    | Ok mtime →
       if mtime > other_mtime then
         Effect.log "Header a été modifié"
       else Effect.log "Header n'a pas été modifié"
  else Effect.log "Le fichier header n'existe pas"
```

my_program est juste **une description de programme**

On donne un **sens dénotationnel** à un programme et on lui attribuera un **sens opérationnel** à postériori (**Handling des effets**).

- Ça facilite l’écriture de tests unitaires
- Ça permet de ne pas “plus s’en soucier” pour cette présentation

Maintenant, on voudrait décrire une liste de dépendances

```
%.html: pages/%.html header.html footer.html
	cat header.html $(<) footer.html > _build/$(@)
```

Soit:

- une liste de **filename**
- Si un fichier revient deux fois, il ne doit être présent **qu'une fois**

Maintenant, on voudrait décrire une liste de dépendances

```
%.html: pages/%.html header.html footer.html
	cat header.html $(<) footer.html > _build/$(@)
```

Soit:

- une liste de **filename**
- Si un fichier revient deux fois, il ne doit être présent **qu'une fois**

Un travail idéal pour un **Set**

Maintenant, on voudrait décrire une liste de dépendances

```
%.html: pages/%.html header.html footer.html
	cat header.html $(<) footer.html > _build/$(@)
```

Soit:

- une liste de **filename**
- Si un fichier revient deux fois, il ne doit être présent **qu'une fois**

Un travail idéal pour un **Set**

```
module Deps = struct
  include Set.Make (String)
end
```

Améliorons l'API de notre liste de dépendances

```
val Deps.need_update : Deps.t → filename → bool Try.t Effect.t
```

- Si le fichier **n'existe pas** : renvoie **`Ok (true)`**
- Si le fichier **existe** mais qu'il est a un **mtime > à tout ceux des dépendances** : renvoie **`Ok (false)`**
- Si le fichier **existe** mais qu'il est a un **mtime <= à tout ceux des dépendances** : renvoie **`Ok (true)`**
- Si une erreur survient (un fichier dans les dépendances n'existe pas) : renvoie **`Error (l'exception qui décrit l'erreur)`**

Maintenant, on ne peut créer un fichier que si c’est nécéssaire

```
let write_file_if_needed target deps task =
  let open Effect.Monad in
  let* maybe_need_update = Deps.need_update deps target in
  match maybe_need_update with
  | Error _ → Effect.log "Oh flute, une erreur pour récupérer les deps"
  | Ok false → Effect.log "Pas besoin de mise à jour"
  | Ok true →
    (task >>= Effect.write_file target >>= function
     | Error _ → Effect.log "Oh flute, une erreur pour écrire le fichier"
     | Ok () → Effect.log "fichier écrit")
```

S’il n y a pas eu d’erreur et qu’il faut créer
Le fichier… on l’écrit !

Nous avons maintenant les même capacité que la règle Make

```
let try_this = function
  | Ok x → Effect.return x
  | Error _ → Effect.log ("Flute, encore une erreur")

let create file deps task =
  let deps = Deps.of_list deps in
  write_file_if_needed file deps task
```

```
create "test.html" [“page.html”] (
   let open Effect.Monad in
   let* file_content = Effect.read_file "page.html" in
   try_this file_content
)
```

```
test.html : page.html
  cat page.html > test.html
```

Nous avons maintenant les même capacité que la règle Make

```
let try_this = function
  | Ok x → Effect.return x
  | Error _ → Effect.log ("Flute, encore une erreur")

let create file deps task =
  let deps = Deps.of_list deps in
  write_file_if_needed file deps task
```

Obligation de spécifier les dépendances :'(

```
create "test.html" [“page.html”] (
   let open Effect.Monad in
   let* file_content = Effect.read_file "page.html" in
   try_this file_content
)
```

```
test.html : page.html
  cat page.html > test.html
```

Encapsulons tout ça dans un type !

Capturer les dépendance à la définition de la règle

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

Capturer les dépendance à la définition de la règle

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

Capturer les dépendance à la définition de la règle

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }

let read_file filename = {
    dependencies = Deps.singleton filename
  ; task = (fun () →
    let open Effect.Monad in
    Effect.read_file filename >>= function
    | Error exn → Effect.throw (exn)
    | Ok content → return content)
  }
```

Capturer les dépendance à la définition de la règle

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }

let read_file filename = {
    dependencies = Deps.singleton filename
  ; task = (fun () →
    let open Effect.Monad in
    Effect.read_file filename >>= function
    | Error exn → Effect.throw (exn)
    | Ok content → return content)
  }
```

Capturer les dépendance à la définition de la règle

```
let create_file target rule =
  write_file_if_needed target rule.dependencies (rule.task ())
```

Maintenant, plus besoin de spécifier manuellement les dépendances

```
let rule = create_file "test.html" (
   read_file "page.html"
)
```

Et c’est exactement le même exemple que précédemment.

Capturer les dépendance à la définition de la règle

```
let create_file target rule =
  write_file_if_needed target rule.dependencies (rule.task ())
```

Maintenant, plus besoin de spécifier manuellement les dépendances

```
let rule = create_file "test.html" (
   read_file "page.html"
)
```

Actuellement… on ne peut exécuter qu’une seule tâche…
C’est un peu balot. **On voudrait pouvoir composer nos règles**

Et c’est exactement le même exemple que précédemment.

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

La composition des dépendances est assez triviale,
C'est simplement l'union des deux Sets.

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

La composition des dépendances est assez triviale,
C'est simplement l'union des deux Sets.

```
let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 }
```

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

```
let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 }
```

Maintenant, comment composer les Tasks?

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

```
let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 }
```

Soit (‘b→ ‘c Effect.t) → (‘a → ‘b Effect.t) → (‘a → ‘c Effect.t)

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

```
let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 }
```

Soit `(‘b→ ‘c Effect.t) → (‘a → ‘b Effect.t) → (‘a → ‘c Effect.t)`

Oh, mais c’est exactement le même type que la **composition de Kleisli !**

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }


let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
  ;  task = Effect.(left.task ⟸ right.task)
  }
```

Composition de règles

Peut on promouvoir une fonction arbitraire en règle ?

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }


let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 ;  task = Effect.(left.task ⟸ right.task)
 }
```

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }

let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 ;  task = Effect.(left.task ⟸ right.task)
 }
```

Peut on promouvoir une fonction arbitraire en règle ?

A priori une fonction ne génère pas de dépendances. Donc oui:

```
let arrow f = {
    dependencies = Deps.empty
 ;  task = fun x → Effect.return (f x)
 }
```

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }

let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 ;  task = Effect.(left.task ⟸ right.task)
 }
```

Nous avons presque tous les ingrédients pour construire une Arrow

Peut on promouvoir une fonction arbitraire en règle ?

A priori une fonction ne génère pas de dépendances. Donc oui:

```
let arrow f = {
    dependencies = Deps.empty
 ;  task = Effect.return (f x)
 }
```

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }

let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 ;  task = Effect.(left.task ⟸ right.task)
 }
```

Peut on promouvoir une fonction arbitraire en règle ?

A priori une fonction ne génère pas de dépendances. Donc oui:

```
let arrow f = {
    dependencies = Deps.empty
 ;  task = Effect.return (f x)
 }
```

Ce qui nous permettra de construire de nouvelles règles et de les composer !

Nous avons presque tous les ingrédients pour construire une Arrow

Composition de règles

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

Les familiers avec les Arrows auront reconnu la **Kleisli Arrow**.

```
let compose left right = {
    dependencies = Deps.union
        left.dependencies right.dependencies
 ;  task = Effect.(left.task <=< right.task)
 }
```

Peut on promouvoir une fonction arbitraire en règle ?

A priori une fonction ne génère pas de dépendances. Donc oui:

```
let arrow f = {
    dependencies = Deps.empty
 ;  task = Effect.return (f x)
 }
```

Ce qui nous permettra de construire de nouvelles règles et de les composer !

Nous avons presque tous les ingrédients pour construire une Arrow

Arrow

- Permet de manipuler “**des genres de fonctions**” (par exemple, notre règle)
- La Arrow la plus triviale est (‘a → ‘b)
- Oblige à programmer en ***pointfree*** pouvant rendre le code illisible
- Expose une **collection d’opérateurs** (*pointfree oblige*)
- N’est en fait pas grand chose de plus que **Category** et un **Strong Profunctor**
- Compose relativement bien avec les Applicatives

Arrow

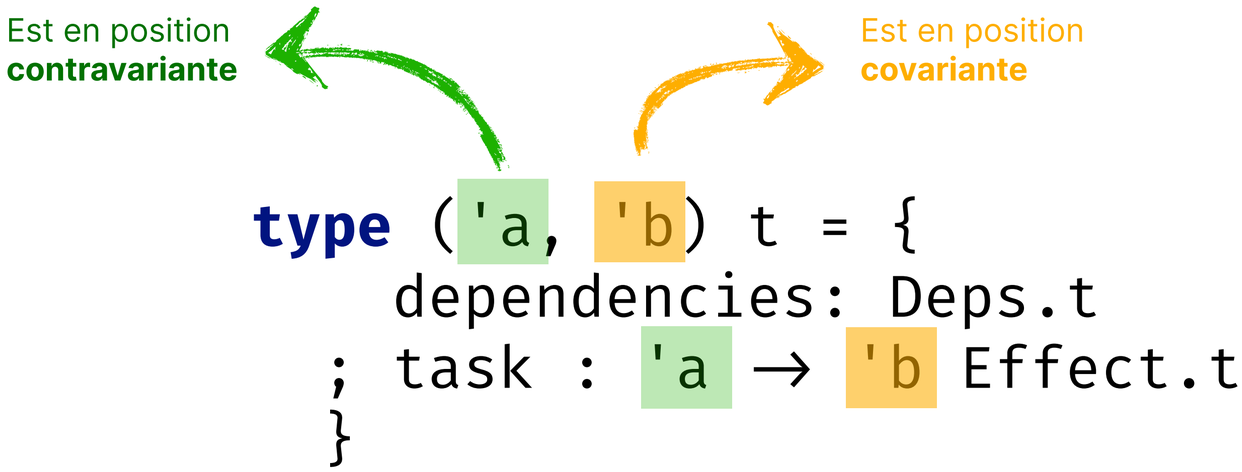
- Permet de manipuler “**des genres de fonctions**” (par exemple, notre règle)
- La Arrow la plus triviale est (‘a → ‘b)
- Oblige à programmer en ***pointfree*** pouvant rendre le code illisible
- Expose une **collection d’opérateurs** (*pointfree oblige*)
- N’est en fait pas grand chose de plus que **Category** et un **Strong Profunctor**
- Compose relativement bien avec les Applicatives

Dans les faits, ce n’est rien de plus que de la composition de “fonctions”

Mes *pain-points*

- Comprendre les schémas explicatifs/diagrammes
- Un peu comme **Contravariant**, peu intuitif si on est pas préparé (comme moi!)

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```


Est en position
contravariante
Est en position
covariante
type ('a, 'b) t = {
dependencies: Deps.t
; task : 'a → 'b Effect.t
}

Est en position
contravariante

Est en position
covariante

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

La variance est définie par la position par la position par rapport à la flèche de la fonction

Est en position
contravariante

Est en position
covariante

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

La variance est définie par la position par la position par rapport à la flèche de la fonction

Et oui, une Arrow est un profoncteur

Est en position
contravariante

Est en position
covariante

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

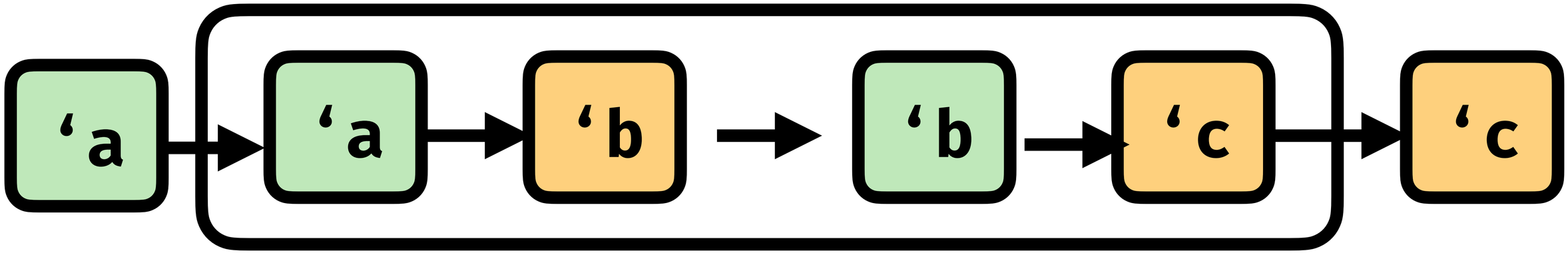
La variance est définie par la position par la position par rapport à la flèche de la fonction

Et oui, une Arrow est un profoncteur

Arrow expose pleins de combinateurs
pour **composer** des arrows entre elles

L’aspect profoncteur (Foncteur contravariant + Foncteur covariant)
peu rendre l’interprétation des fonctions compliqué. (IMHO)

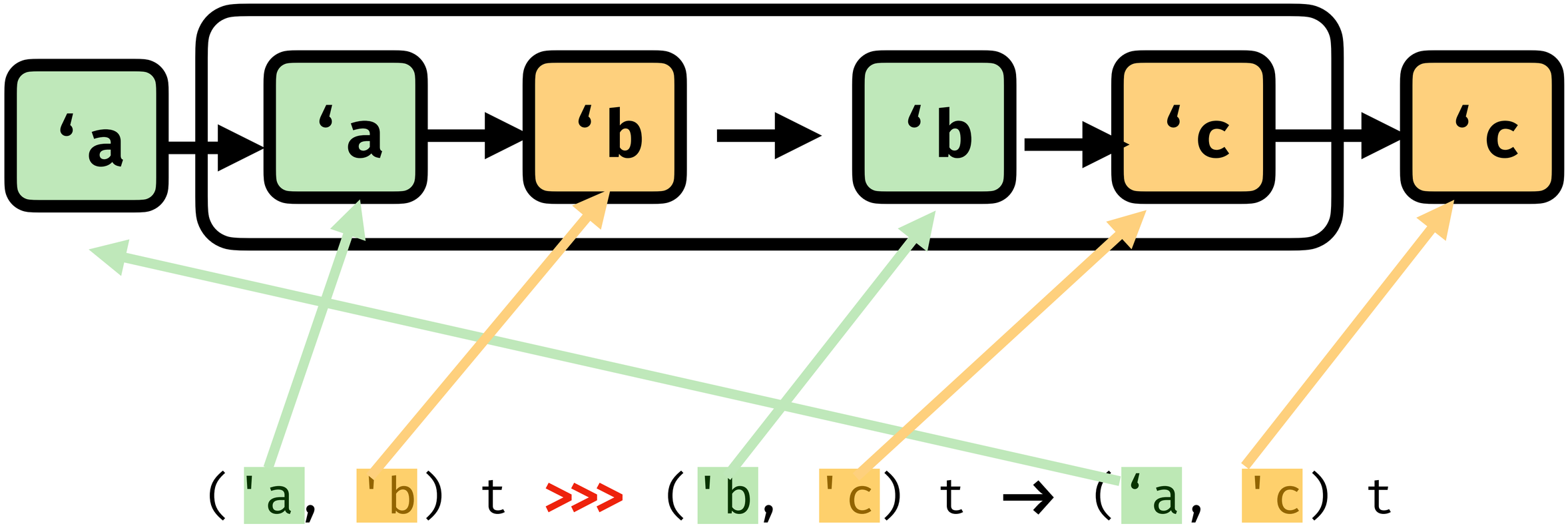
Composition (>>>)




```
('a, 'b) t >>> ('b, 'c) t → (‘a, 'c) t
```

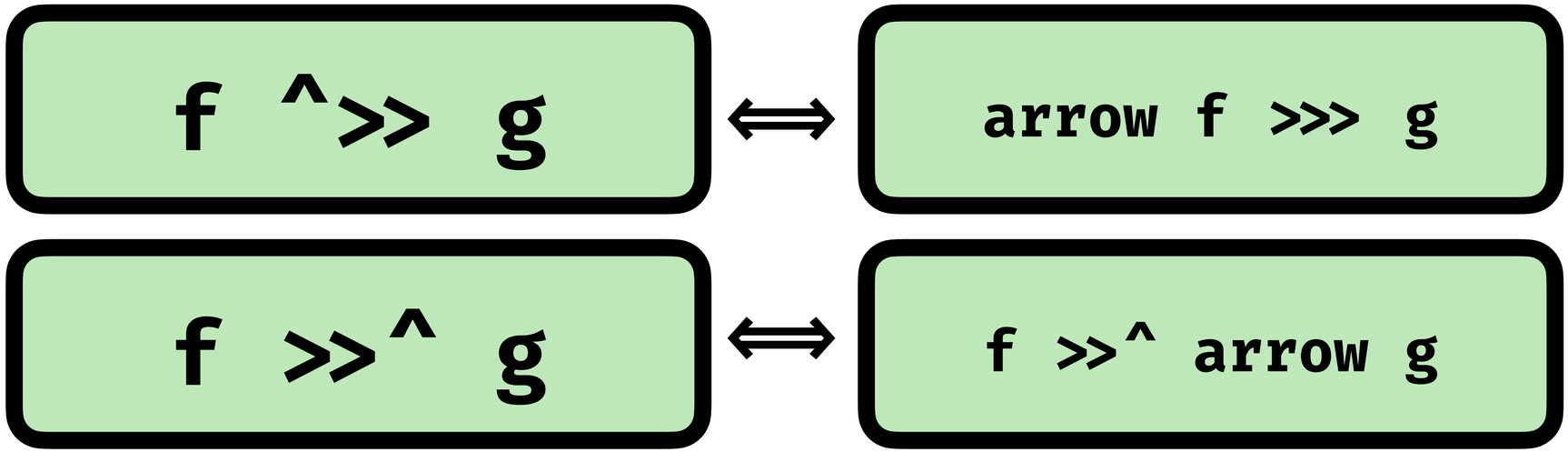
L’aspect profoncteur (Foncteur contravariant + Foncteur covariant)
peu rendre l’interprétation des fonctions compliqué. (IMHO)

Composition (>>>)



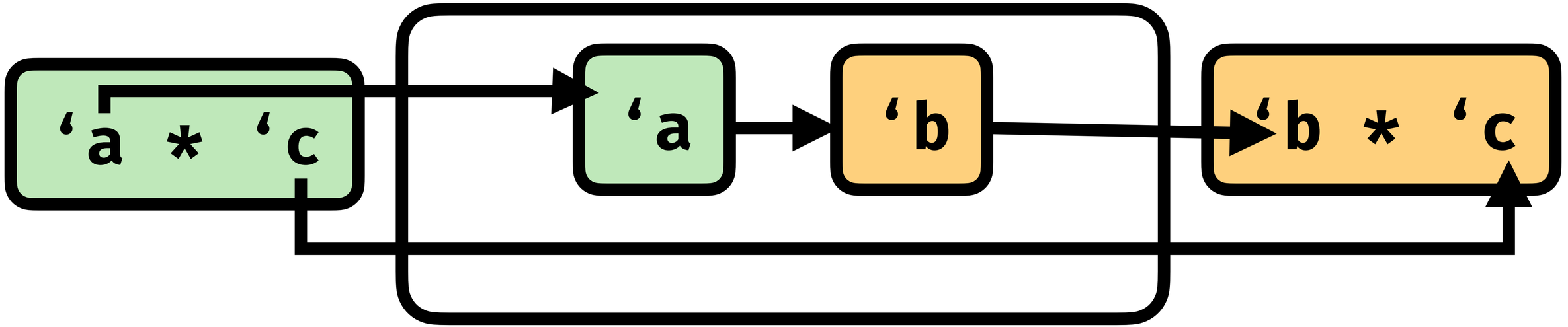
L’aspect profoncteur (Foncteur contravariant + Foncteur covariant)
peu rendre l’interprétation des fonctions compliqué. (IMHO)

Composition (pré/post application)



L’aspect profoncteur (Foncteur contravariant + Foncteur covariant)
peu rendre l’interprétation des fonctions compliqué. (IMHO)

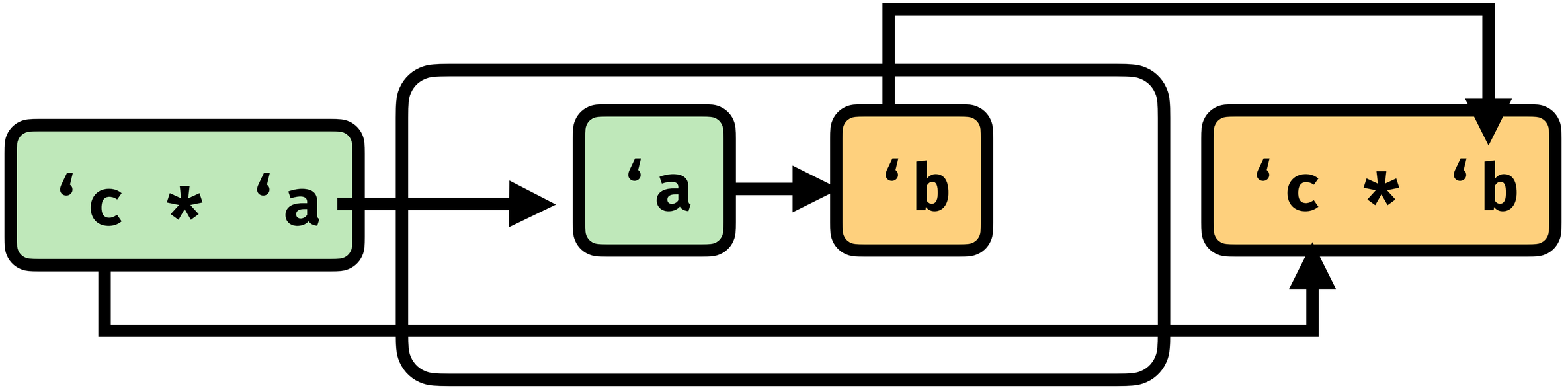
fst




```
fst: (‘a, 'b) t → (‘a * ‘c, ‘b * ‘c) t
```

L’aspect profoncteur (Foncteur contravariant + Foncteur covariant)
peu rendre l’interprétation des fonctions compliqué. (IMHO)

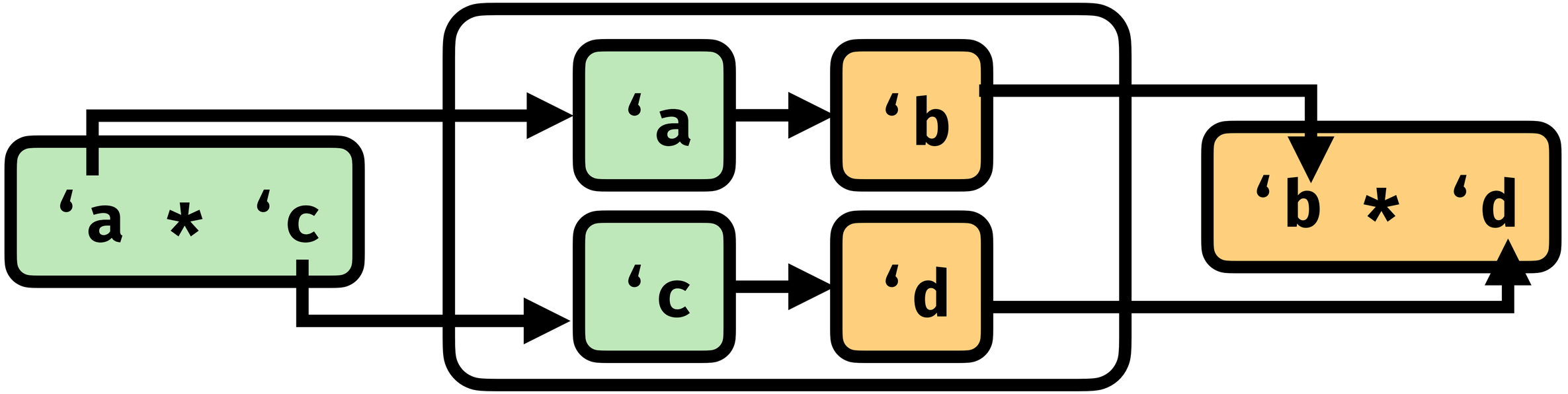
snd




```
snd: (‘a, 'b) t → (‘c * ‘a, ‘c * ‘a) t
```

L’aspect profoncteur (Foncteur contravariant + Foncteur covariant)
peu rendre l’interprétation des fonctions compliqué. (IMHO)

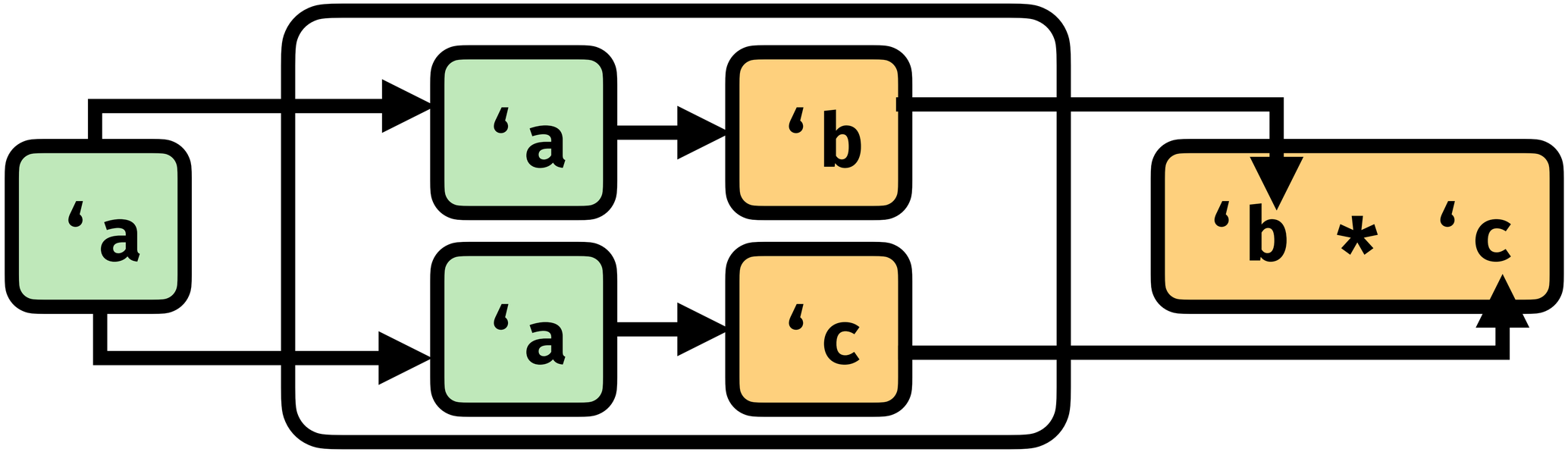
split (***)




```
('a, 'b) t *** ('c, ‘d) t → (‘a * ‘c, ‘b * d) t
```

L’aspect profoncteur (Foncteur contravariant + Foncteur covariant)
peu rendre l’interprétation des fonctions compliqué. (IMHO)

fan-out (&&&)




```
('a, 'b) t &&& ('a, ‘c) t → (‘a, ‘b * c) t
```

```
type ('a, 'b) t = {
    dependencies: Deps.t
  ; task : 'a → 'b Effect.t
  }
```

read_file est une fonction qui prend un chemin en argument
Et renvoie une **arrow (unit, string)**.

```
let read_file filename = {
    dependencies = Deps.singleton filename
  ; task = (fun () →
    let open Effect.Monad in
    Effect.read_file filename >>= function
    | Error exn → Effect.throw (exn)
    | Ok content → return content)
  }
```

En utilisant les combinateurs de Arrows, il est possible de construire des Arrows plus riches (qui font plus)

Créons une Arrow qui concatène deux fichiers

Une approche “tentante” serait d’utiliser **&&&**

```
let concat_files file_a file_b =
  ((read_file file_a) &&& (read_file file_b))
  >>> arrow (fun (a, b) → a ^ b)

concat_files :
   filename → filename → (unit → string) t
```

Créons une Arrow qui concatène deux fichiers

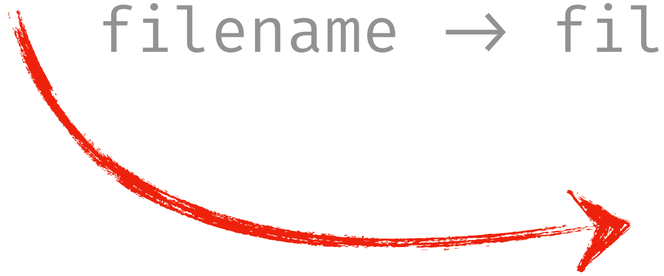
Une approche “tentante” serait d’utiliser **&&&**

```
let concat_files file_a file_b =
  ((read_file file_a) &&& (read_file file_b))
  >>^(fun (a, b) → a ^ b)

concat_files :
   filename → filename → (unit → string) t
```

Créons une Arrow qui concatène deux fichiers

Une approche “tentante” serait d’utiliser **&&&**

```
let concat_files file_a file_b =
  ((read_file file_a) &&& (read_file file_b))
  >>^(fun (a, b) → a ^ b)

concat_files :
  filename → filename → (unit → string) t
```



Ça fonctionne, mais quel est le soucis ?

Créons une Arrow qui concatène deux fichiers

Une approche “tentante” serait d’utiliser **&&&**

```
let concat_files file_a file_b =
  ((read_file file_a) &&& (read_file file_b))
  >>^(fun (a, b) → a ^ b)

concat_files :
   filename → filename → (unit → string) t
```

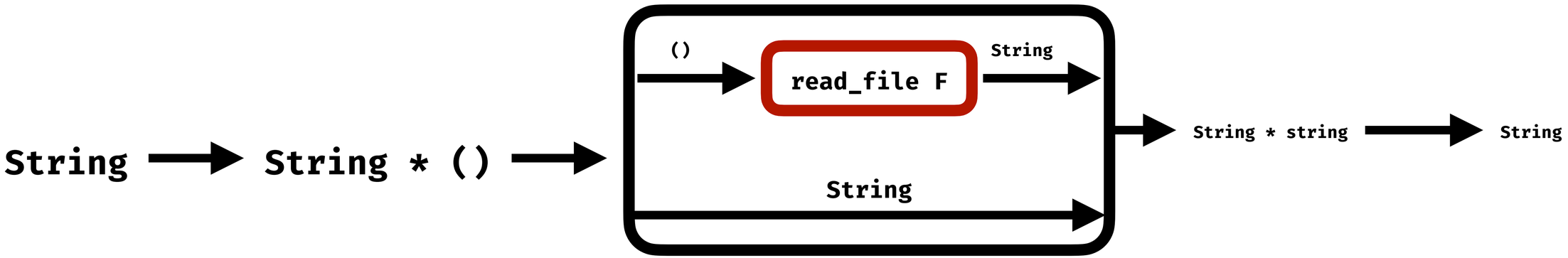
Ce code “cloture” la Arrow, et ne la rend
Pas composable.
Il faudrait ajouter une fonction par nombre
De fichiers que l’on voudrait concaténer.

Créons une Arrow qui concatène deux fichiers

Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```

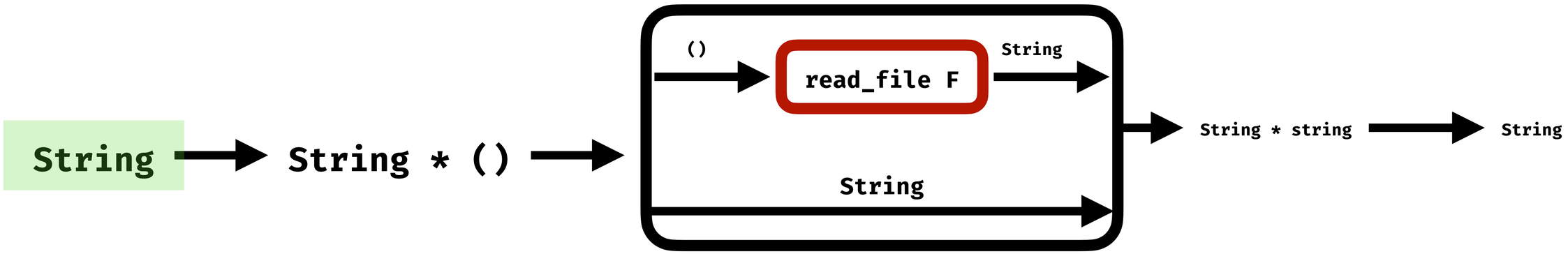


Créons une Arrow qui concatène deux fichiers

Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```

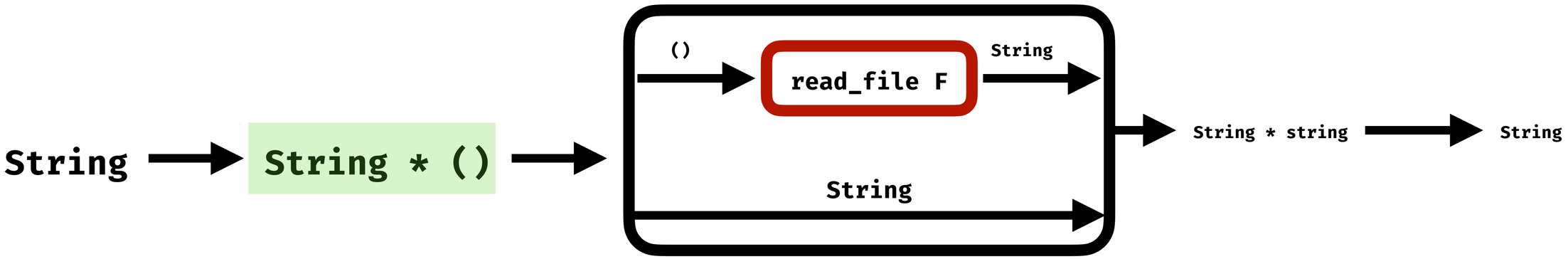


Créons une Arrow qui concatène deux fichiers

Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```

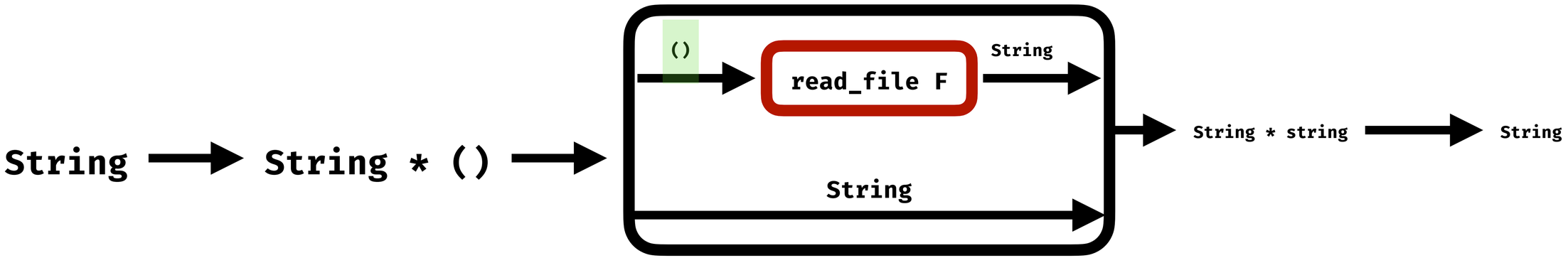


Créons une Arrow qui concatène deux fichiers

Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```

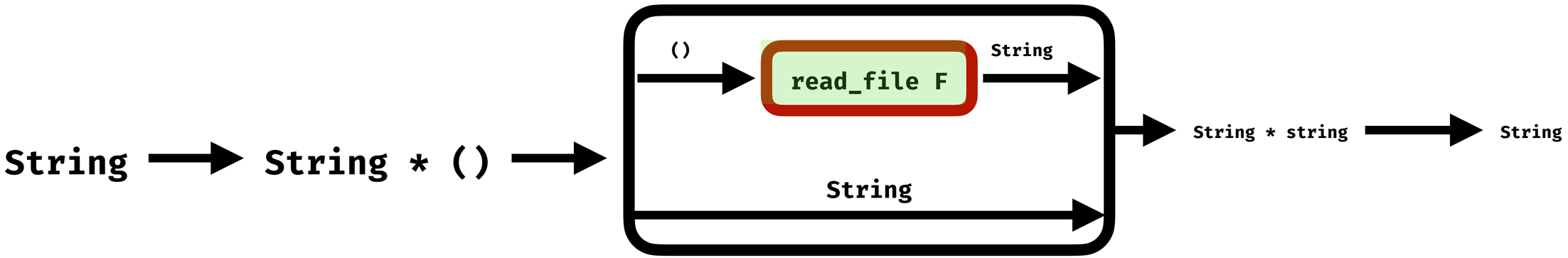


Créons une Arrow qui concatène deux fichiers

Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```

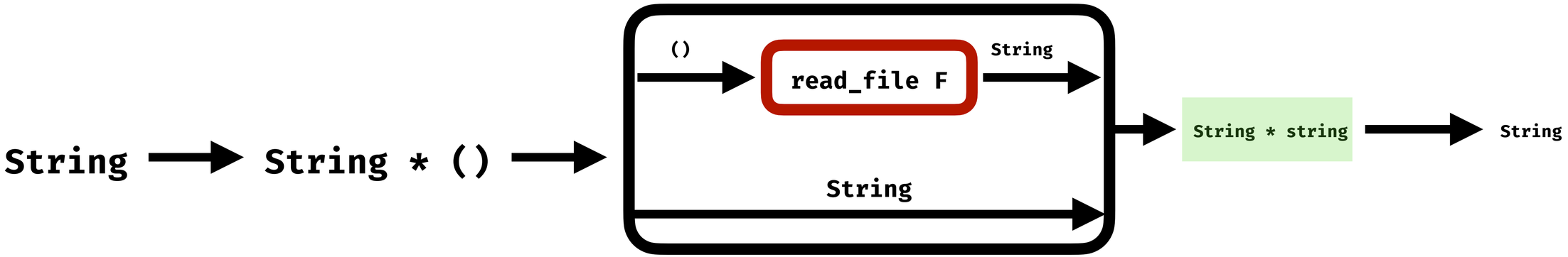


Créons une Arrow qui concatène deux fichiers

Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```



Créons une Arrow qui concatène deux fichiers

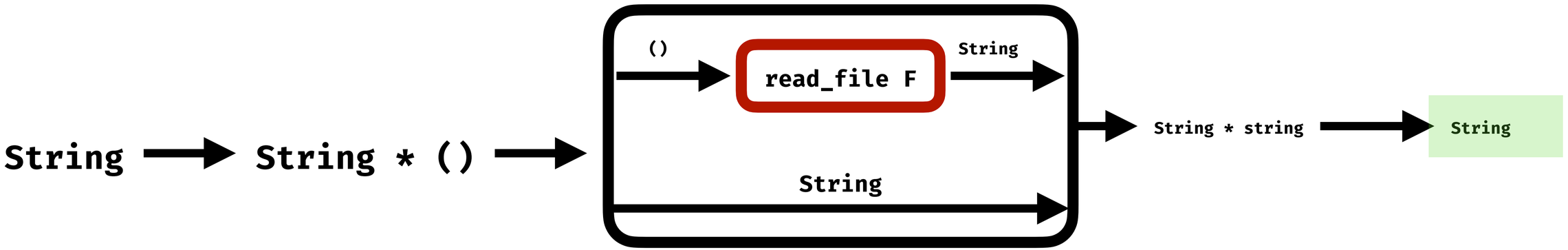
Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```

Et maintenant, pipe compose !

```
let rule =
  create_file "contact.html" (
      let open Arrow in
      read_file "templates/header.html"
      >>> pipe_content "pages/contact.html"
      >>> pipe_content "templates/footer.html"
  )
```



Créons une Arrow qui concatène deux fichiers

Une meilleure approche serait d'utiliser **snd**

```
let pipe_content filename =
  (fun x → (x, ()))
  ^>> snd (read_file filename)
  >>^ (fun (a, b) → a ^ b)
```

```
pipe_content :
   filename → (string→ string) t
```

Et maintenant, pipe compose !

```
let rule =
  create_file "contact.html" (
      let open Arrow in
      read_file "templates/header.html"
      >>> pipe_content "pages/contact.html"
      >>> pipe_content "templates/footer.html"
  )
```

Et si j'inspecte les dépendances de cette règle:

```
[ “templates/header.html”
; “pages/contact.html
; “templates/footer.html” ]
```

Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

  >>> read_file_with_metadata
        (module Metadata.Article) article_target

  >>> snd process_markdown

  >>> apply_as_template
        (module Metadata.Article) “article.html"


  >>> apply_as_template
        (module Metadata.Article) “layout.html"

  >>^ Preface.Pair.snd
```

Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

    >>> read_file_with_metadata
          (module Metadata.Article) article_target

    >>> snd process_markdown

    >>> apply_as_template
          (module Metadata.Article) “article.html"

    >>> apply_as_template
          (module Metadata.Article) “layout.html"

    >>^ Preface.Pair.snd
```

```
let watch path =
  { dependencies = Deps.singleton (Deps.file path)
  ; task = Effect.return
  }

let track_binary_update = watch Sys.argv.(0)
```

Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

  >>> read_file_with_metadata
        (module Metadata.Article) article_target

  >>> snd process_markdown

  >>> apply_as_template
        (module Metadata.Article) “article.html"

  >>> apply_as_template
        (module Metadata.Article) “layout.html"

  >>^ Preface.Pair.snd
```

```
let watch path =
  { dependencies = Deps.singleton (Deps.file path)
  ; task = Effect.return
  }

let track_binary_update = watch Sys.argv.(0)
```

Ajoute le binaire qui génère le blog dans Les dépendances.

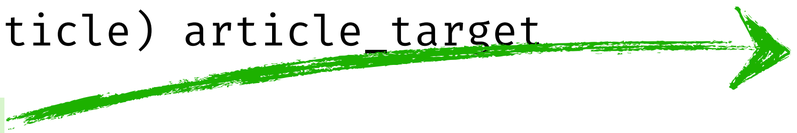
Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

  >>> read_file_with_metadata
        (module Metadata.Article) article_target

  >>> snd process_markdown

  >>> apply_as_template
        (module Metadata.Article) “article.html"


  >>> apply_as_template
        (module Metadata.Article) “layout.html"


  >>^ Preface.Pair.snd
```

Lit un fichier et son “header” et renvoie une pair “Header/contenu”

Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

  >>> read_file_with_metadata
        (module Metadata.Article) article_target

  >>> snd process_markdown

  >>> apply_as_template
        (module Metadata.Article) “article.html"

  >>> apply_as_template
        (module Metadata.Article) “layout.html"

  >>^ Preface.Pair.snd
```



Transforme le second élément de la pair de Markdown à HTML

Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

  >>> read_file_with_metadata
        (module Metadata.Article) article_target

  >>> snd process_markdown

  >>> apply_as_template
        (module Metadata.Article) “article.html"

  >>> apply_as_template
        (module Metadata.Article) “layout.html"

  >>^ Preface.Pair.snd
```

Injecte le contenu dans une template,
Substitue les métadonnées dans le template
Et renvoie une pair
“metadonnées/nouveau contenu”

Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

  >>> read_file_with_metadata
        (module Metadata.Article) article_target

  >>> snd process_markdown

  >>> apply_as_template
        (module Metadata.Article) “article.html"

  >>> apply_as_template
        (module Metadata.Article) “layout.html"

  >>^ Preface.Pair.snd
```

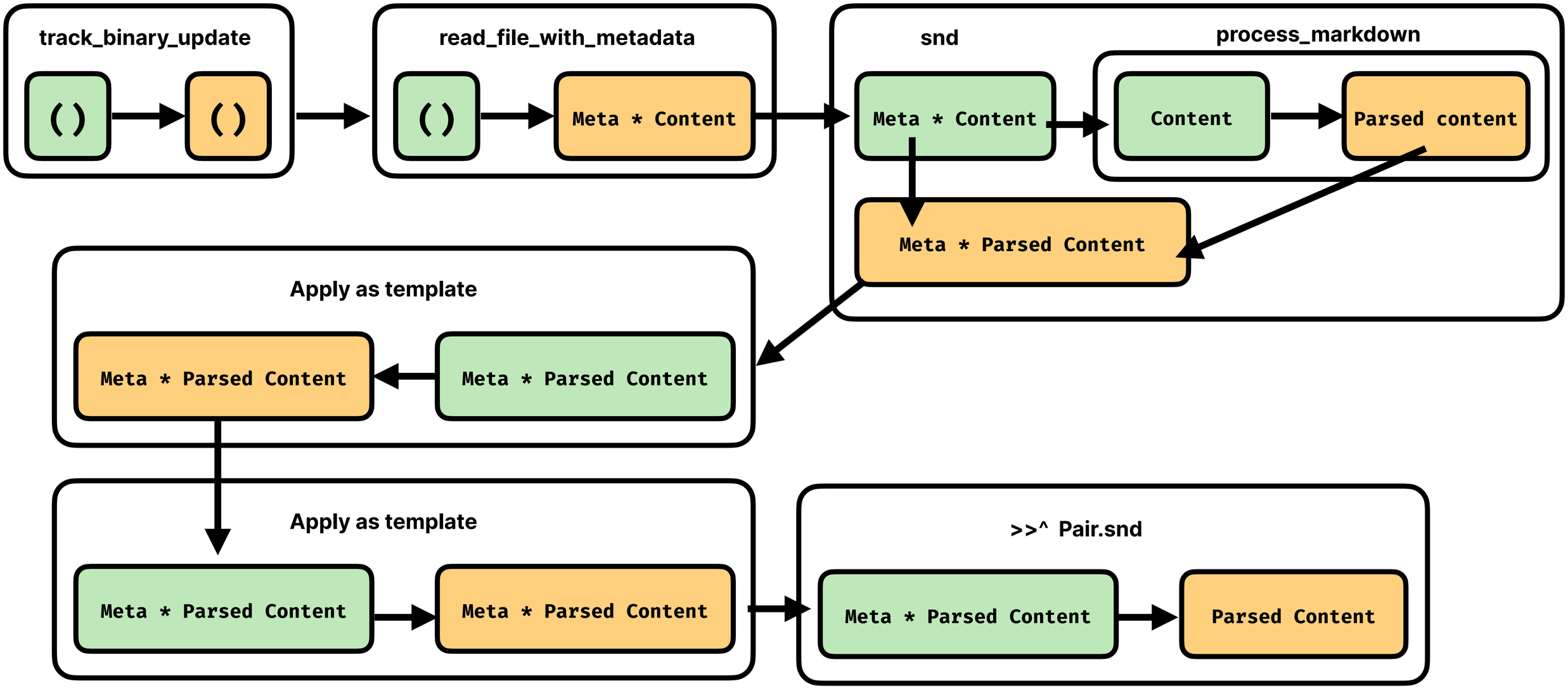
Injecte le contenu dans une template,
Substitue les métadonnées dans le template
Et renvoie une pair
“metadonnées/nouveau contenu”

Arrow plus complexe tirée de Wordpress

```
let article_rule article_target =

  track_binary_update

  >>> read_file_with_metadata
        (module Metadata.Article) article_target

  >>> snd process_markdown

  >>> apply_as_template
        (module Metadata.Article) “article.html"


  >>> apply_as_template
        (module Metadata.Article) “layout.html"

  >>^ Preface.Pair.snd
```

N’écrit que le contenu (injecté dans Article puis dans Layout), *discard* les métadonnées

Arrow plus complexe tirée de Wordpress



C’est la fin :)

Même si c’était une présentation un peu naïve et très concrète, elle a survolés plusieurs concepts assez cool (IMHO) en programmation fonctionnelle.

Abstraction des effets avec Freer

Ce qui rend le projet assez facile à tester. Je vous invite à regarder le Repo de Wordpress

C’est la fin :)

Même si c’était une présentation un peu naïve et très concrète, elle a survolés plusieurs concepts assez cool (IMHO) en programmation fonctionnelle.

Abstraction des effets avec Freer

Ce qui rend le projet assez facile à tester. Je vous invite à regarder le Repo de Wordpress

C’est la fin :)

Même si c’était une présentation un peu naïve et très concrète, elle a survolés plusieurs concepts assez cool (IMHO) en programmation fonctionnelle.

Un use-case concret aux Arrows

Abstraction des effets avec Freer
Ce qui rend le projet assez facile à tester. Je vous invite à regarder le Repo de Wordpress

Kotlin brille dans beaucoup de domaines, mais pas sur qu'il aurait été possible d'implémenter si facilement ce genre de constructions. N'hésitez pas à me prouver le contraire !

C'est la fin :)

Même si c'était une présentation un peu naïve et très concrète, elle a survolés plusieurs concepts assez cool (IMHO) en programmation fonctionnelle.

Un use-case concret aux Arrows

Alors que l’on trouve énormément d’exemples sur Applicative et Monades, les Arrows, qui se trouvent juste entre les deux sont plus rares.

Alors que l’on trouve énormément d’exemples sur Applicative et Monades, les Arrows, qui se trouvent juste entre les deux sont plus rares.

Les combinateurs sont assez désagréable à utiliser mais ils permettent de décrire des DSL “relativement agréable à utiliser”.

Pourtant, elles peuvent être très utiles quand on manipule des “genres de fonctions”.

Alors que l’on trouve énormément d’exemples sur Applicative et Monades, les Arrows, qui se trouvent juste entre les deux sont plus rares.

Les combinateurs sont assez désagréable à utiliser mais ils permettent de décrire des DSL “relativement agréable à utiliser”.

Pourtant, elles peuvent être très utiles quand on manipule des “genres de fonctions”.

Alors que l’on trouve énormément d’exemples sur Applicative et Monades, les Arrows, qui se trouvent juste entre les deux sont plus rares.

Si ça vous a plus, et que vous faites du OCaml, n’hésitez pas à essayer :

- https://github.com/xvw/preface
- https://github.com/xhtmlboi/wordpress
- Et a nous faire des retours <3