UNE ODE À LA PROGRAMMATION TACITE

Xavier Van de Woestyne — xvw.lol — @xvw@merveilles.town

UNE ODE À LA PROGRAMMATION TACITE

Xavier Van de Woestyne — xvw.lol — @xvw@merveilles.town

FORK IT!
COMMUNITY - PLACE

Building Functional Systems

Our solutions help developers **build robust, secure, high-performance** applications whilst **maintaining crucial reliability**.

UNE ODE À LA PROGRAMMATION TACITE

Xavier Van de Woestyne — xvw.lol — @xvw@merveilles.town

Building Functional Systems

Our solutions help developers **build robust, secure, high-performance** applications whilst **maintaining crucial reliability**.

UNE ODE À LA PROGRAMMATION TACITE

Xavier Van de Woestyne — xvw.lol — @xvw@merveilles.town

Tarides

FORK IT!
COMMUNITY - PLACE

C'est notre logo ridicule !

Building Functional Systems

Our solutions help developers **build robust, secure, high-performance** applications whilst **maintaining crucial reliability**.

LET'S GO !
Tacite, que l'on appelle aussi point-free

UNE ODE À LA PROGRAMMATION TACITE

Xavier Van de Woestyne — xvw.lol — @xvw@merveilles.town

Tarides

FORK IT!
COMMUNITY - PLACE

C'est notre logo ridicule !

ATTENTION

En essayant de suivre la charte FORK IT! COMMUNITY PLACE

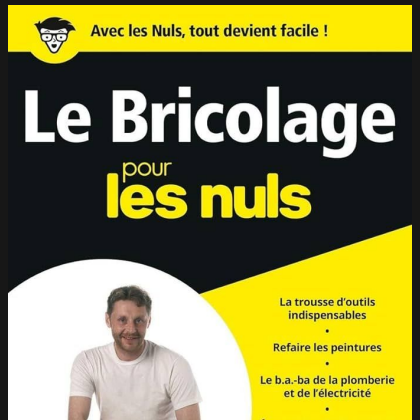
ATTENTION


En essayant de suivre la
charte
FORK IT!
COMMUNITY PLACE
ATTENTION
maladroitement

En essayant de suivre la charte FORK IT!

ATTENTION

maladroitement

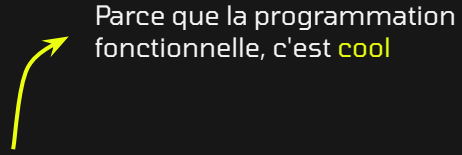


désolé.

PLAN & OBJECTIFS

- Une introduction *rapide* à la programmation tacite
- Comprendre pourquoi c'est mal !
- S'initier aux raisonnements équationnels et génériques
  - Généraliser des comportements
  - Comprendre la polarité dans les génériques
- Comprendre pourquoi (le tacite) ça peut être bien

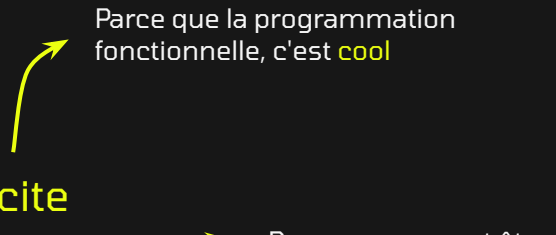
PLAN & OBJECTIFS



- Une introduction *rapide* à la programmation tacite
- Comprendre pourquoi c'est mal !
- S'initier aux raisonnements équationnels et génériques
  - Généraliser des comportements
  - Comprendre la polarité dans les génériques
- Comprendre pourquoi (le tacite) ça peut être bien

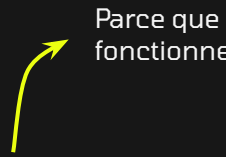
PLAN & OBJECTIFS

- Une introduction *rapide* à la programmation tacite
- Comprendre pourquoi c'est mal !
- S'initier aux raisonnements équationnels et génériques
  - Généraliser des comportements
  - Comprendre la polarité dans les génériques
- Comprendre pourquoi (le tacite) ça peut être bien



PLAN & OBJECTIFS

- Une introduction *rapide* à la programmation tacite
- Comprendre pourquoi c'est mal !
- S'initier aux raisonnements équationnels et génériques
  - Généraliser des comportements
  - Comprendre la polarité dans les génériques
- Comprendre pourquoi (le tacite) ça peut être bien



PLAN & OBJECTIFS

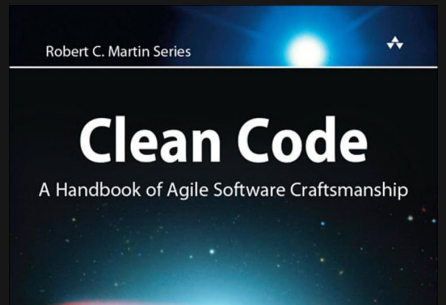
Parce que la programmation fonctionnelle, c'est cool

- Une introduction *rapide* à la programmation tacite
- Comprendre pourquoi c'est mal !
- S'initier aux raisonnements équationnels et génériques
  - Généraliser des comportements
  - Comprendre la polarité dans les génériques
- Comprendre pourquoi (le tacite) ça peut être bien

Parce que ça peut être vraiment laid

Pour comprendre pourquoi ça peut être cool

Parce que tout est contextuel et que le dogmatisme, c'est mal



PLAN & OBJECTIFS

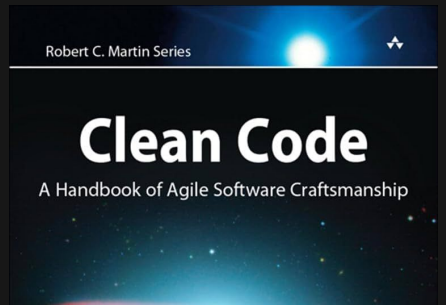
Parce que la programmation fonctionnelle, c'est cool

- Une introduction *rapide* à la programmation tacite
- Comprendre pourquoi c'est mal !
- S'initier aux raisonnements équationnels et génériques
  - Généraliser des comportements
  - Comprendre la polarité dans les génériques
- Comprendre pourquoi (le tacite) ça peut être bien

Parce que ça peut être vraiment laid

Pour comprendre pourquoi ça peut être cool

Parce que tout est contextuel et que le dogmatisme, c'est mal



La compréhension précise de la syntaxe du code n'est pas une obligation pour comprendre la présentation !

Je reste ce soir a Rouen et j'aime bien la bière, et discuter :)

PLAN & OBJECTIFS

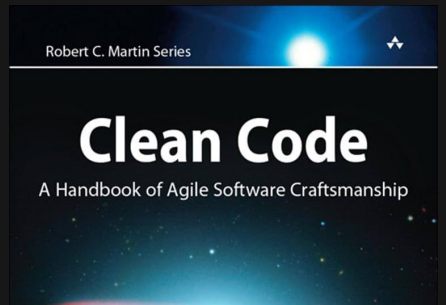
Parce que la programmation fonctionnelle, c'est cool

- Une introduction *rapide* à la programmation tacite
- Comprendre pourquoi c'est mal !
- S'initier aux raisonnements équationnels et génériques
  - Généraliser des comportements
  - Comprendre la polarité dans les génériques
- Comprendre pourquoi (le tacite) ça peut être bien

Parce que ça peut être vraiment laid

Pour comprendre pourquoi ça peut être cool

Parce que tout est contextuel et que le dogmatisme, c'est mal



La compréhension précise de la syntaxe du code n'est pas une obligation pour comprendre la présentation !

Sur la Programmation Fonctionnelle

un trait de programmation
inspiré du λ-calcul

Sur la Programmation Fonctionnelle

présent dans une grande partie
des langages populaires

un trait de programmation
inspiré du λ-calcul

Sur la Programmation Fonctionnelle

présent dans une grande partie
des langages populaires

très clivante
(pour d'obscures (et mauvaises) raisons)

un trait de programmation
inspiré du λ-calcul

Sur la Programmation Fonctionnelle

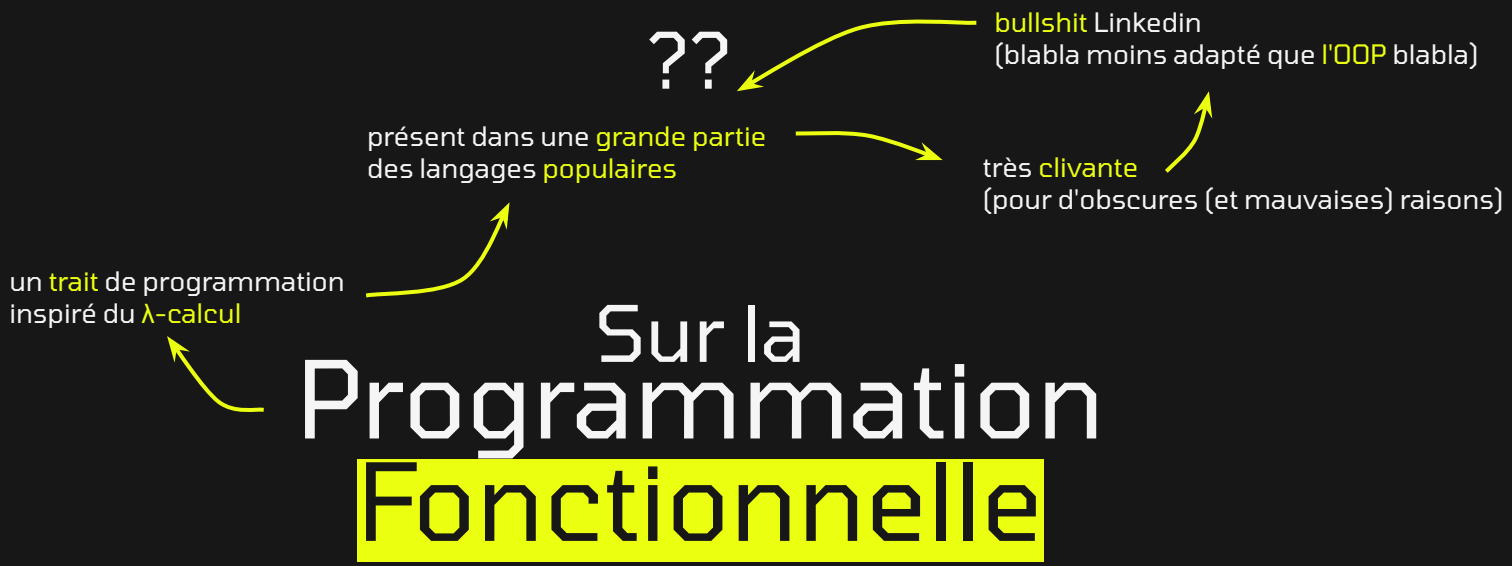
bullshit Linkedin
(blabla moins adapté que l'OOP blabla)

présent dans une grande partie
des langages populaires

très clivante
(pour d'obscures (et mauvaises) raisons)

un trait de programmation
inspiré du λ-calcul

Sur la Programmation Fonctionnelle


??
bullshit Linkedin
(blabla moins adapté que l'OOP blabla)
présent dans une grande partie
des langages populaires
très clivante
(pour d'obscures (et mauvaises) raisons)
un trait de programmation
inspiré du λ-calcul
Sur la
Programmation
Fonctionnelle

??

bullshit Linkedin
(blabla moins adapté que l'OOP blabla)

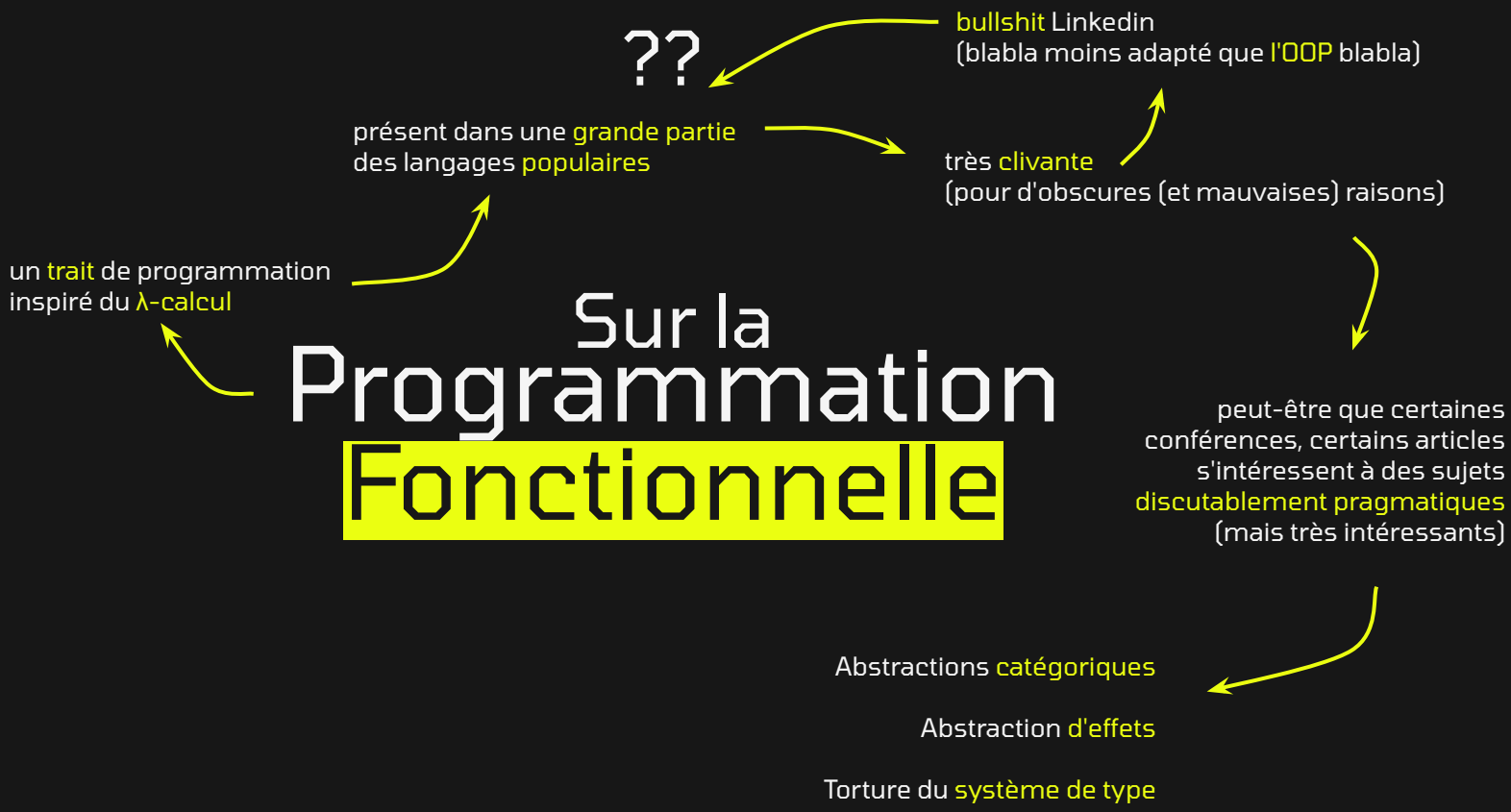
présent dans une grande partie
des langages populaires

très clivante
(pour d'obscures (et mauvaises) raisons)

un trait de programmation
inspiré du λ-calcul

Sur la Programmation Fonctionnelle

peut-être que certaines
conférences, certains articles
s'intéressent à des sujets
discutablement pragmatiques
(mais très intéressants)


??
bullshit Linkedin
(blabla moins adapté que l'OOP blabla)
présent dans une grande partie
des langages populaires
très clivante
(pour d'obscures (et mauvaises) raisons)
un trait de programmation
inspiré du λ-calcul
Sur la
Programmation
Fonctionnelle
peut-être que certaines
conférences, certains articles
s'intéressent à des sujets
discutablement pragmatiques
(mais très intéressants)
Abstractions catégoriques
Abstraction d'effets
Torture du système de type

Sur la Programmation Fonctionnelle

??

bullshit Linkedin
(blabla moins adapté que l'OOP blabla)

présent dans une grande partie
des langages populaires

très clivante
(pour d'obscures (et mauvaises) raisons)

un trait de programmation
inspiré du λ-calcul

peut-être que certaines
conférences, certains articles
s'intéressent à des sujets
discutablement pragmatiques
(mais très intéressants)

Abstractions catégoriques

Abstraction d'effets

Torture du système de type

Retour aux sources !
Parlons de
fonctions et de **composition de fonctions**

Sur la Programmation Fonctionnelle

??

bullshit Linkedin
(blabla moins adapté que l'OOP blabla)

très clivante
(pour d'obscures (et mauvaises) raisons)

présent dans une grande partie
des langages populaires

un trait de programmation
inspiré du λ-calcul

peut-être que certaines
conférences, certains articles
s'intéressent à des sujets
discutablement pragmatiques
(mais très intéressants)

Abstractions catégoriques

Abstraction d'effets

Torture du système de type

Retour aux sources !
Parlons de
fonctions et de **composition de fonctions**

d'une manière discutablement
pédagogique. Pour le fun

Sur la Programmation Fonctionnelle

??

bullshit Linkedin
(blabla moins adapté que l'OOP blabla)

très clivante
(pour d'obscures (et mauvaises) raisons)

présent dans une grande partie
des langages populaires

un trait de programmation
inspiré du λ-calcul

en Haskell

d'une manière discutablement
pédagogique. Pour le fun

Retour aux sources !
Parlons de
fonctions et de **composition de fonctions**

Abstractions catégoriques

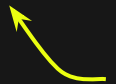
Abstraction d'effets

Torture du système de type

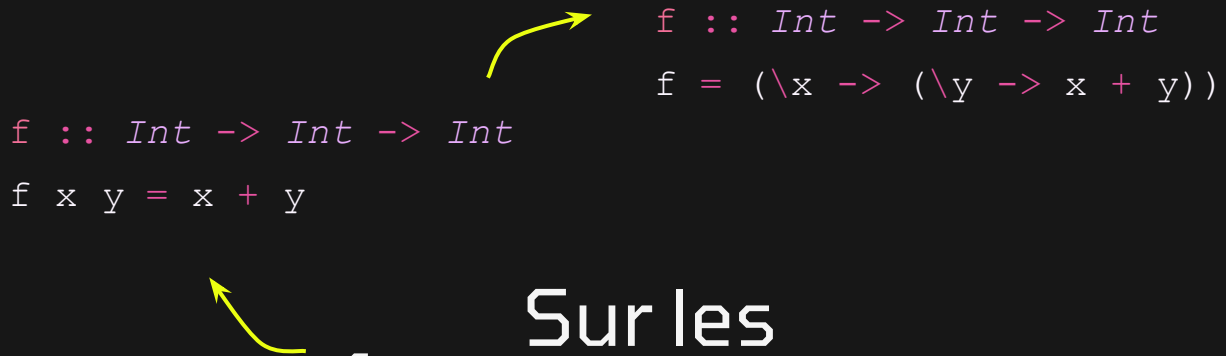
peut-être que certaines
conférences, certains articles
s'intéressent à des sujets
discutablement pragmatiques
(mais très intéressants)

Sur les fonctions en programmation Fonctionnelle

```haskell
f :: Int -> Int -> Int
f x y = x + y
```



Sur les fonctions en programmation Fonctionnelle



Sur les fonctions en programmation Fonctionnelle

```haskell
f :: Int -> Int -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

Sur les fonctions en programmation Fonctionnelle

```haskell
f :: Int -> Int -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

Sur les fonctions en programmation Fonctionnelle

ce qui explique la signature de type et l'absence de virgules

```haskell
f :: (Int, Int) -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

Sur les fonctions en programmation Fonctionnelle

ce qui explique la signature de type et l'absence de virgules

```haskell
f :: (Int, Int) -> Int
f x y = x + y
```

ce procédé s'appelle la curryfication

```haskell
f :: Int -> Int -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

Sur les fonctions en programmation Fonctionnelle

ce qui explique la signature de type et l'absence de virgules

```haskell
f :: (Int, Int) -> Int
f x y = x + y
```

ce procédé s'appelle la curryfication

En référence au logicien Haskell Curry

```haskell
f :: Int -> Int -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

Sur les fonctions en programmation Fonctionnelle

ce qui explique la signature de type et l'absence de virgules

```haskell
f :: (Int, Int) -> Int
f x y = x + y
```

ce procédé s'appelle la curryfication

En référence au logicien Haskell Curry

cheh
qui n'en est **pas** l'inventeur
(Moses Schönfinkel)

Sur les fonctions en programmation Fonctionnelle

```
f :: Int -> Int -> Int
f x y = x + y
```

ce qui offre un feature amusante

```
incr :: Int -> Int
incr x = f 1 x
```

```
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

ce qui explique la signature de type et l'absence de virgules

```
f :: (Int, Int) -> Int
f x y = x + y
```

ce procédé s'appelle la curryfication

En référence au logicien Haskell Curry

cheh
qui n'en est **pas** l'inventeur
(Moses Schönfinkel)

Sur les fonctions en programmation Fonctionnelle

```
f :: Int -> Int -> Int
f x y = x + y
```

```
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

ce qui explique la signature de type et l'absence de virgules

```
f :: (Int, Int) -> Int
f x y = x + y
```

ce procédé s'appelle la curryfication

En référence au logicien Haskell Curry

cheh
qui n'en est **pas** l'inventeur
(Moses Schönfinkel)

ce qui offre un feature amusante

```
incr :: Int -> Int
incr x = f 1 x
```

f 1 retourne une fonction attendant **y**

Sur les fonctions en programmation Fonctionnelle

```haskell
f :: Int -> Int -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

ce qui explique la signature de type et l'absence de virgules

```haskell
f :: (Int, Int) -> Int
f x y = x + y
```

ce procédé s'appelle la curryfication

En référence au logicien Haskell Curry

cheh
qui n'en est **pas** l'inventeur
(Moses Schönfinkel)

ce qui offre un feature amusante

```haskell
incr :: Int -> Int
incr x = f 1 x
```

f 1 retourne une fonction attendant **y**

```haskell
incr :: Int -> Int
incr = f 1
```

Sur les fonctions en programmation Fonctionnelle

```haskell
f :: Int -> Int -> Int
f x y = x + y
```

```haskell
f :: Int -> Int -> Int
f = (\x -> (\y -> x + y))
```

le \ dénote une lambda
donc **f x y** est une fonction **f** qui attend un argument **x** et renvoie une fonction qui attend un argument **y**

ce qui explique la signature de type et l'absence de virgules

```haskell
f :: (Int, Int) -> Int
f x y = x + y
```

ce qui offre un feature amusante

```haskell
incr :: Int -> Int
incr x = f 1 x
```

f 1 retourne une fonction attendant **y**

```haskell
incr :: Int -> Int
incr = (+) 1
```

ce procédé s'appelle la curryfication

En référence au logicien Haskell Curry

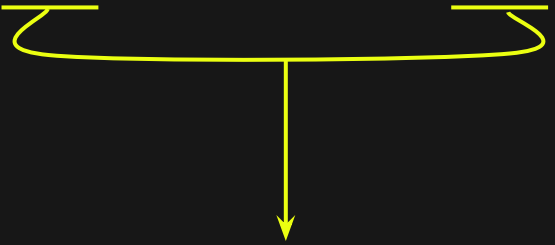
cheh
qui n'en est **pas** l'inventeur
(Moses Schönfinkel)

Un autre exemple

```
sum list = foldl (+) list
```

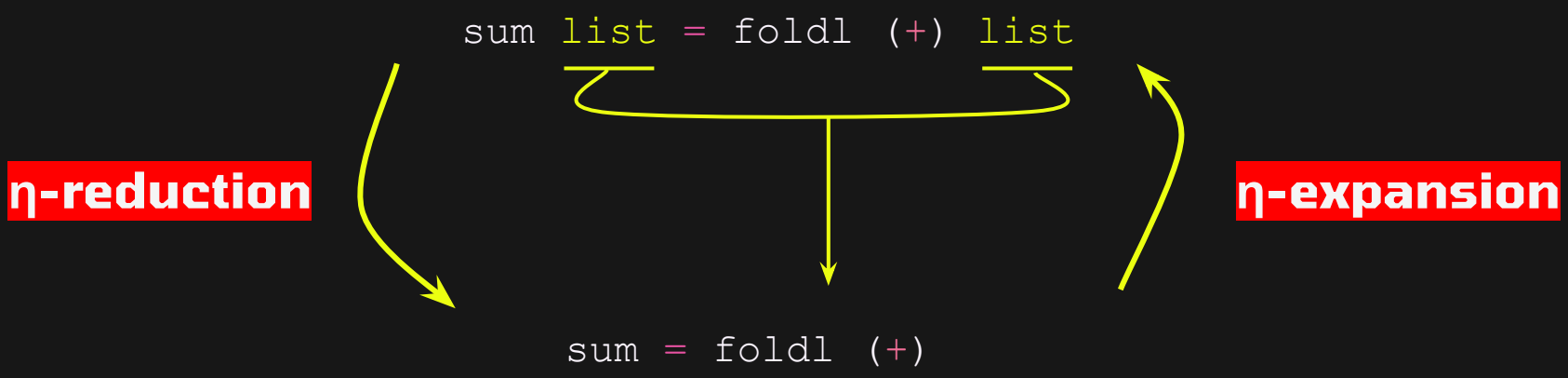
Un autre exemple

```
sum list = foldl (+) list
```

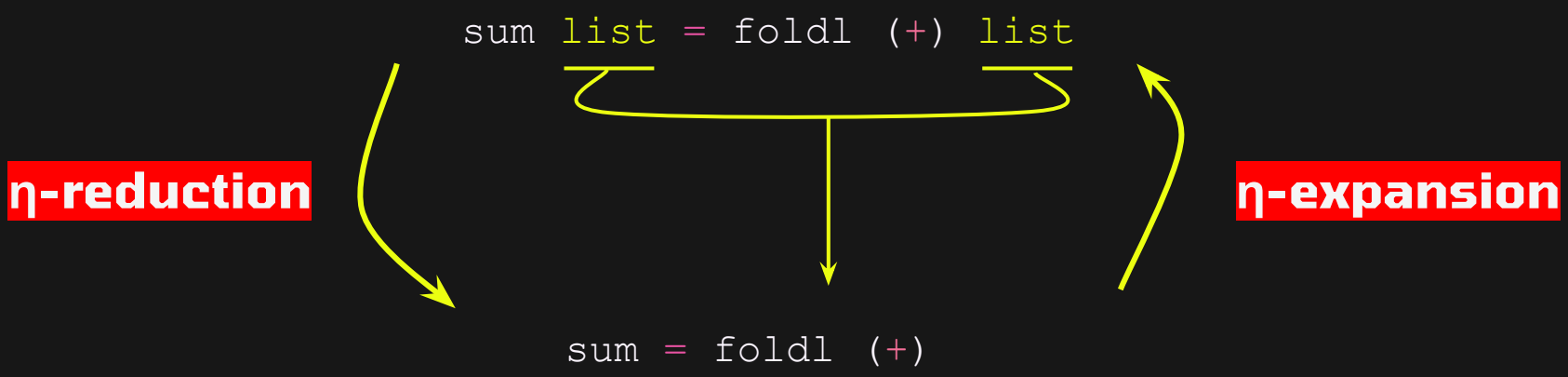


```
sum = foldl (+)
```

Un autre exemple



Un autre exemple



Il ne nous manque plus qu'un seul ingrédient pour découvrir la **programmation tacite**

Un tout petit opérateur qui semble anodin

Un tout petit opérateur qui semble anodin


```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

Un tout petit opérateur qui semble anodin

héhéhé

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

un opérateur qui reproduit la composition mathématique de fonctions

Un tout petit opérateur qui semble anodin

héhéhé

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

un opérateur qui reproduit la composition mathématique de fonctions

```
(f . g) x         =  f (g x)
(f . g . h . i) x =  f (g (h (i x)))
```

l'opérateur est associatif

Un tout petit opérateur qui semble anodin

héhéhé

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

un opérateur qui reproduit la composition mathématique de fonctions

```
(f . g) x          =  f (g x)
(f . g . h . i) x  =  f (g (h (i x)))
```

l'opérateur est associatif

et c'est le rôle de la programmation tacite (ou point-free) :

Remplacer toutes les applications de fonction par de la composition de fonction !

Un tout petit opérateur qui semble anodin

héhéhé

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

un opérateur qui reproduit la composition mathématique de fonctions

```
(f . g) x          =  f (g x)
(f . g . h . i) x  =  f (g (h (i x)))
```

l'opérateur est associatif

et c'est le rôle de la programmation tacite (ou point-free) :

Remplacer toutes les applications de fonction par de la composition de fonction !

pourquoi point-free, s'il y a plus de point ?

Un tout petit opérateur qui semble anodin

héhéhé

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

un opérateur qui reproduit la composition mathématique de fonctions

```
(f . g) x         =  f (g x)
(f . g . h . i) x =  f (g (h (i x)))
```

l'opérateur est associatif

et c'est le rôle de la programmation tacite (ou point-free) :

Remplacer toutes les applications de fonction par de la composition de fonction !

pourquoi point-free, s'il y a plus de point ?

naïvement, les points dénotent les arguments, supprimé par éta-réduction

Ça peut produire du code très lisible

```
map (f . g . h . i) [1, 2, 3]
```

versus

```
map (\x -> f (g (h (i x)))) [1, 2, 3]
```

Ça peut produire du code plus efficace

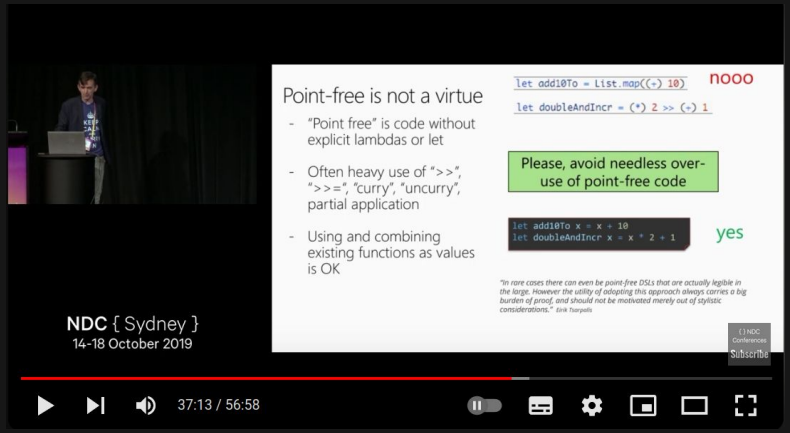
```
map (f . g . h . i) [1, 2, 3]
```

versus

```
[1, 2, 3] |> map f |> map g |> map h |> map i
```

et toujours plus lisible

POURTANT



POURTANT

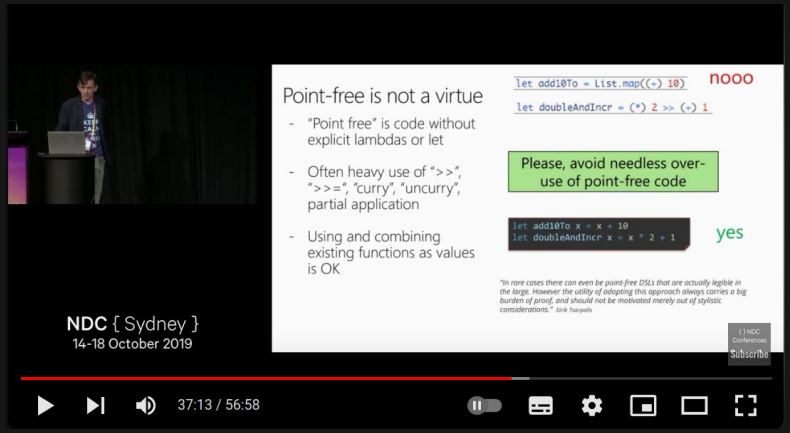
Don Syme, créateur de F#,
dans la section "Code que je n'aime pas"

Pourquoi ?

On trouve **beaucoup d'articles** de type
Consider Point Free Style harmful

J'ai été de mauvaise foi

- toutes mes fonctions n'avaient qu'un seul argument
- les exemples étaient triviaux



POURTANT

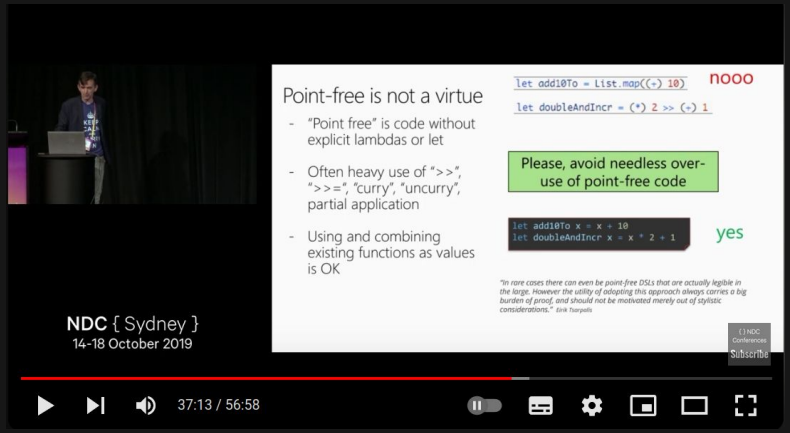
Don Syme, créateur de F#,
dans la section "Code que je n'aime pas"

Pourquoi ?

On trouve **beaucoup d'articles** de type
Consider Point Free Style harmful

J'ai été de mauvaise foi

- toutes mes fonctions n'avaient qu'un seul argument
- les exemples étaient triviaux



Dans certains contextes il faut beaucoup plus de ***helpers***

$, flip, snd, fst, curry, uncurry, left, right, id etc.

POURTANT

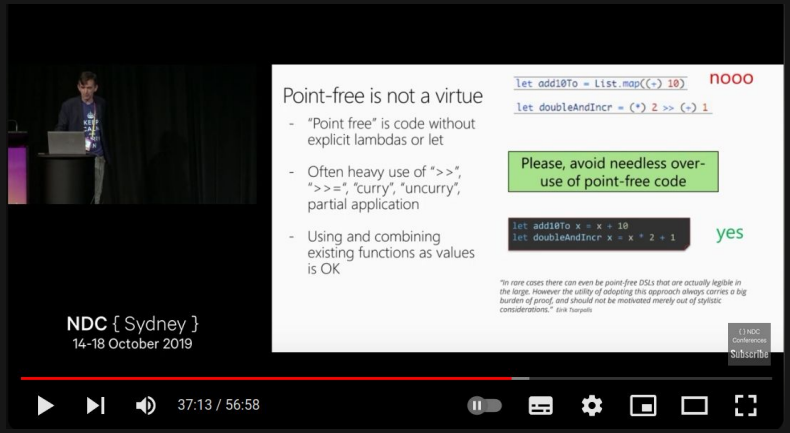
Don Syme, créateur de F#, dans la section "Code que je n'aime pas"

Pourquoi ?

On trouve **beaucoup d'articles** de type Consider Point Free Style harmful

J'ai été de mauvaise foi

- toutes mes fonctions n'avaient qu'un seul argument
- les exemples étaient triviaux



Dans certains contextes il faut beaucoup plus de ***helpers***

$, flip, snd, fst, curry, uncurry, left, right, id etc.

ce qui peut **rapidement** amener à du code parfaitement illisible

c'est une forme d'obfuscation très élaborée !

POURTANT

Pourquoi ?

Don Syme, créateur de F#, dans la section "Code que je n'aime pas"

On trouve **beaucoup d'articles** de type Consider Point Free Style harmful

libera.chat
#haskell

libera.chat
#haskell

`/pl`

un **bot** qui convertit une
fonction en fonction point-free

libera.chat
#haskell

/pl

un **bot** qui convertit une
fonction en fonction point-free

on a une fonction toute mignonne

```
example f g (a,b) =
    (f a, g b)
```

libera.chat
#haskell

/pl

un **bot** qui convertit une
fonction en fonction point-free

on a une fonction toute mignonne

```
example f g (a,b) =
    (f a, g b)
```

```
example =
    flip flip snd . (ap .)
    . flip flip fst . ((.) .)
    . flip . (((.) . (,)) .)
```

libera.chat
#haskell

/pl

un **bot** qui convertit une fonction en fonction point-free

on a une fonction toute mignonne

```
example f g (a,b) =
    (f a, g b)
```

```
example =
    flip flip snd . (ap .)
    . flip flip fst . ((.) .)
    . flip . (((.) . (,)) .)
```

impressionnant mais :

- illisible
- incompréhensible
- moche
- si une machine peut le faire, c'est que ce n'est pas si smart que ça.

toujours **à proscrire ?**

Quand s'en servir ?

Quand s'en servir ?

dans un exemple où **la lisibilité n'est pas entravée**
(idéalement si ça améliore l'expression)

```
map (f . g . h . i) [1, 2, 3]
```

Quand s'en servir ?

dans un exemple où **la lisibilité n'est pas entravée**
(idéalement si ça améliore l'expression)

```
map (f . g . h . i) [1, 2, 3]
```

ce qui est ... très subjectif

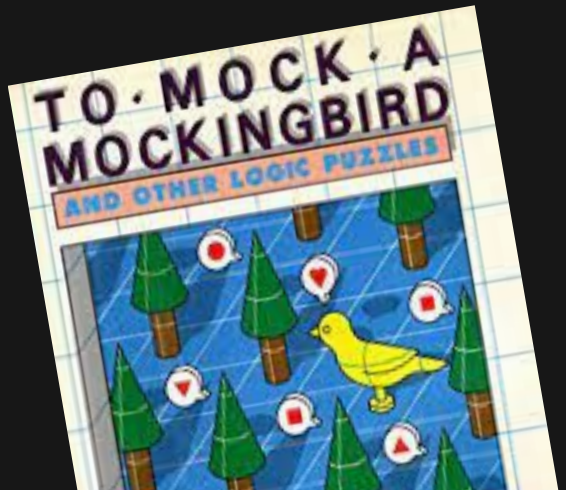
Quand s'en servir ?

dans un exemple où **la lisibilité n'est pas entravée**
(idéalement si ça améliore l'expression)

```
map (f . g . h . i) [1, 2, 3]
```

ce qui est ... très subjectif

pour s'initier à la **logique combinatoire**



- La programmation tacite est rarement légitime, utilisée comme telle

- La programmation tacite est rarement légitime, utilisée comme telle
- elle rend le code très compliqué à lire

- La programmation tacite est rarement légitime, utilisée comme telle
- elle rend le code très compliqué à lire
- elle donne souvent l'impression que l'on est intelligent quand on l'écrit

- La programmation tacite est rarement légitime, utilisée comme telle
- elle rend le code très compliqué à lire
- elle donne souvent l'impression que l'on est intelligent quand on l'écrit
- mais une machine peut automatiser la transformation de code régulier en code point-free
  (donc probablement pas si intelligent que ça. . .)

- La programmation tacite est rarement légitime, utilisée comme telle
- elle rend le code très compliqué à lire
- elle donne souvent l'impression que l'on est intelligent quand on l'écrit

un peu léger pour parler d'ODE

- mais une machine peut automatiser la transformation de code régulier en code point-free
  (donc probablement pas si intelligent que ça. . .)

Petit détour par les abstractions

liée au Higher Kinded Polymorphism

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

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

```haskell
data Result a =
    Ok a
  | Error String
```

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

```
data Result a =
    Ok a
  | Error String
```

```
data List a =
    Cons (a, List a)
  | Nil
```

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

```
data Result a =
    Ok a
  | Error String
```

```
data List a =
    Cons (a, List a)
  | Nil
```

```
map  f (Just x) = Just (f x)
map _ Nothing  = Nothing
```

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

```haskell
map  f (Just x) = Just (f x)
map _ Nothing  = Nothing
```

```haskell
data Result a =
    Ok a
  | Error String
```

```haskell
map f (Ok x) = Ok(f x)
map _ err    = err
```

```haskell
data List a =
    Cons (a, List a)
  | Nil
```

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

```haskell
map  f (Just x) = Just (f x)
map _ Nothing  = Nothing
```

```haskell
data Result a =
    Ok a
  | Error String
```

```haskell
map f (Ok x) = Ok(f x)
map _ err    = err
```

```haskell
data List a =
    Cons (a, List a)
  | Nil
```

```haskell
map f (Cons (x, xs)) = Cons (f x, map xs)
map _ Nil            = Nil
```


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

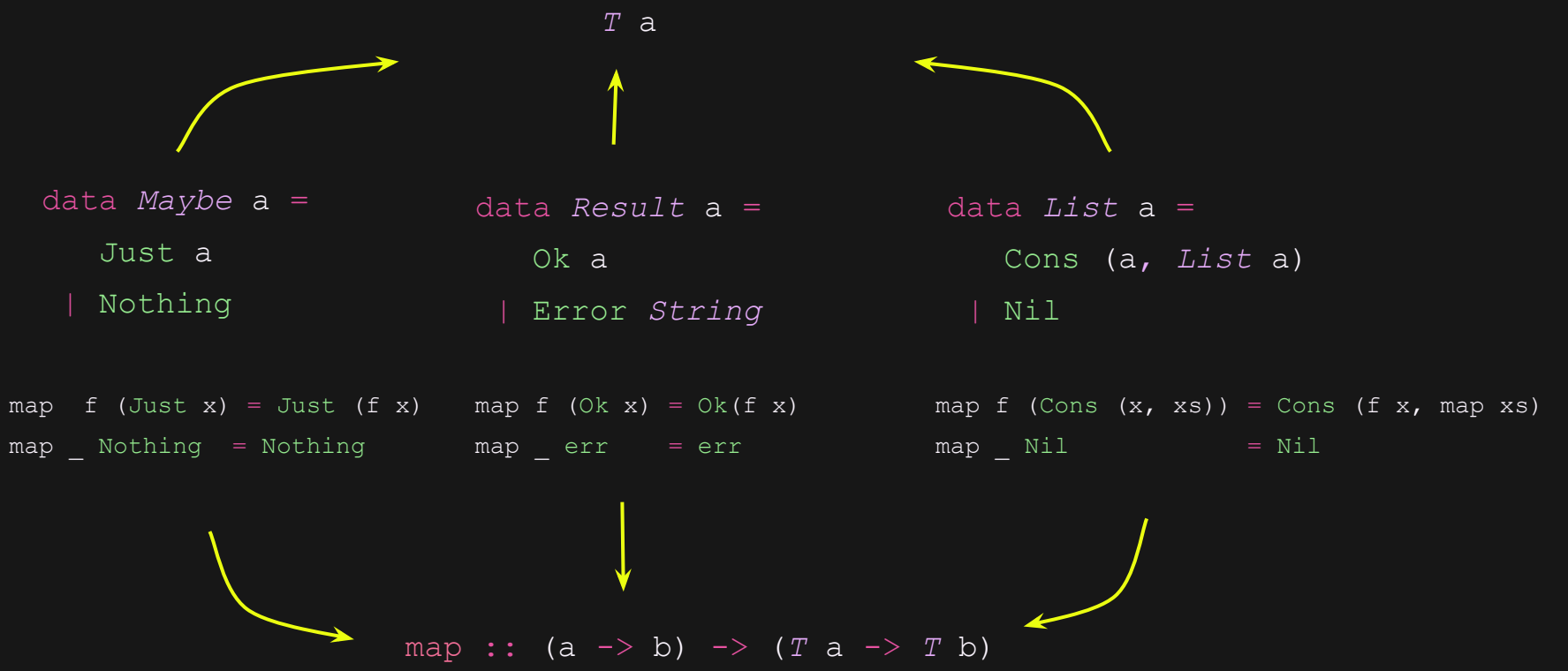
```
map  f (Just x) = Just (f x)
map _ Nothing   = Nothing
```

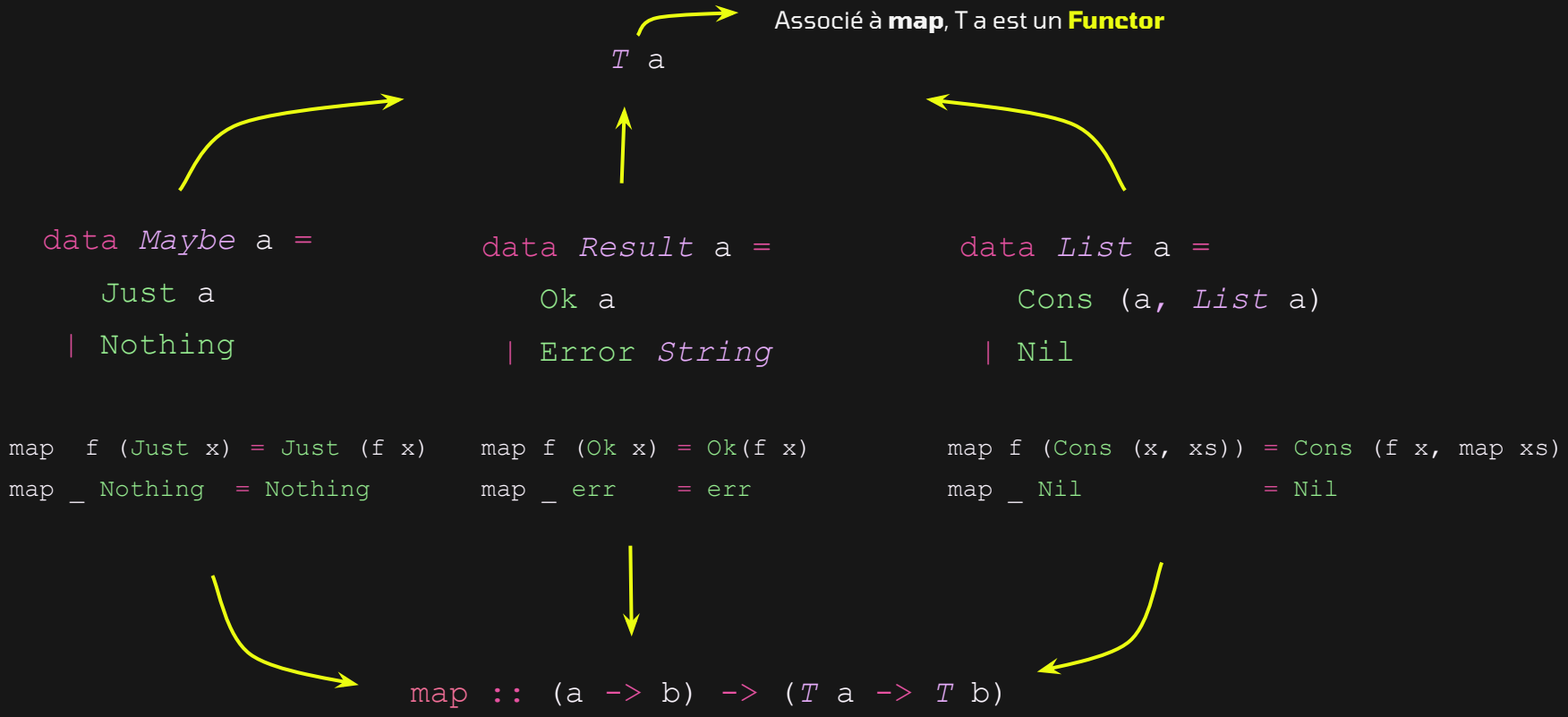
```
data Result a =
    Ok a
  | Error String
```

```
map f (Ok x) = Ok(f x)
map _ err    = err
```

```
data List a =
    Cons (a, List a)
  | Nil
```

```
map f (Cons (x, xs)) = Cons (f x, map xs)
map _ Nil            = Nil
```


T a
data Maybe a =
Just a
| Nothing
data Result a =
Ok a
| Error String
data List a =
Cons (a, List a)
| Nil
map f (Just x) = Just (f x)
map _ Nothing = Nothing
map f (Ok x) = Ok(f x)
map _ err = err
map f (Cons (x, xs)) = Cons (f x, map xs)
map _ Nil = Nil
map :: (a -> b) -> (T a -> T b)


Associé à **map**, T a est un **Functor**
T a
data Maybe a =
Just a
| Nothing
data Result a =
Ok a
| Error String
data List a =
Cons (a, List a)
| Nil
map f (Just x) = Just (f x)
map _ Nothing = Nothing
map f (Ok x) = Ok(f x)
map _ err = err
map f (Cons (x, xs)) = Cons (f x, map xs)
map _ Nil = Nil
map :: (a -> b) -> (T a -> T b)

Associé à **map**, T a est un **Functor**

T a

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

```
map  f (Just x) = Just (f x)
map _ Nothing   = Nothing
```

```
data Result a =
    Ok a
  | Error String
```

```
map f (Ok x) = Ok(f x)
map _ err    = err
```

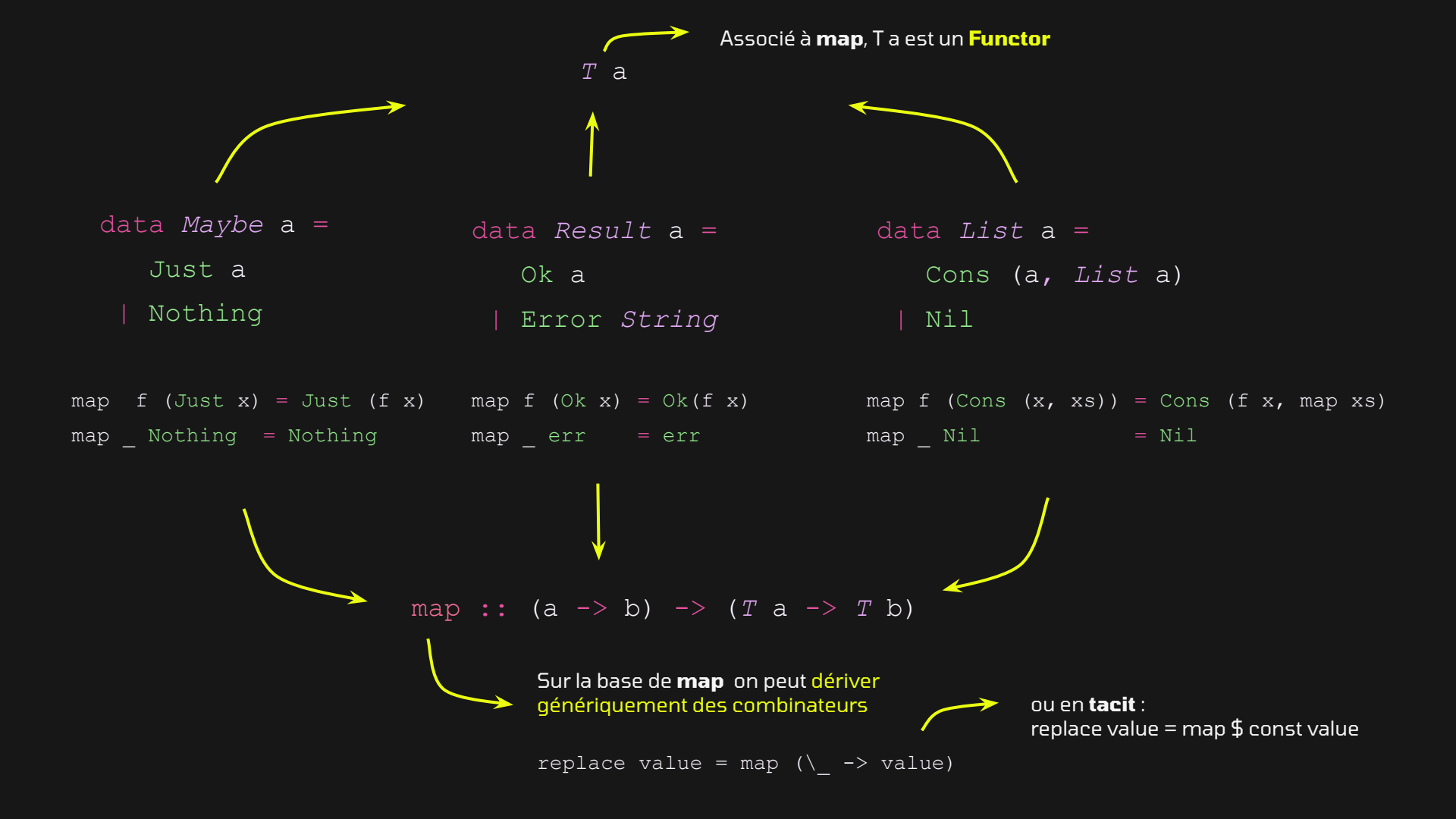
```
data List a =
    Cons (a, List a)
  | Nil
```

```
map f (Cons (x, xs)) = Cons (f x, map xs)
map _ Nil            = Nil
```

```
map :: (a -> b) -> (T a -> T b)
```

Sur la base de **map** on peut dériver génériquement des combinateurs

```
replace value = map (\_ -> value)
```


T a
Associé à **map**, T a est un **Functor**
data Maybe a =
Just a
| Nothing
data Result a =
Ok a
| Error String
data List a =
Cons (a, List a)
| Nil
map f (Just x) = Just (f x)
map _ Nothing = Nothing
map f (Ok x) = Ok(f x)
map _ err = err
map f (Cons (x, xs)) = Cons (f x, map xs)
map _ Nil = Nil
map :: (a -> b) -> (T a -> T b)
Sur la base de **map** on peut dériver génériquement des combinateurs
ou en **tacit** :
replace value = map $ const value
replace value = map (_ -> value)

Cette contrainte se matérialise de cette forme

Si pour un type T paramétré par a

Cette contrainte se matérialise de cette forme

Si pour un type T paramétré par a

j'ai une fonction map telle que :
map :: (a -> b) -> (T a -> T b)
en respectant certaines lois

Cette contrainte se matérialise de cette forme

Si pour un type T paramétré par a

j'ai une fonction map telle que :
map :: (a -> b) -> (T a -> T b)
en respectant certaines lois

Alors T a est un foncteur

Cette contrainte se matérialise de cette forme

Si pour un type T paramétré par a

j'ai une fonction map telle que :
map :: (a -> b) -> (T a -> T b)
en respectant certaines lois

Alors T a est un foncteur

et je peux produire gratuitement, par exemple,
une fonction telle que :
replace:: b -> (T a -> T b)

Cette contrainte se matérialise de cette forme

Si pour un type T paramétré par a

j'ai une fonction map telle que :
map :: (a -> b) -> (T a -> T b)
en respectant certaines lois

par le biais de contraintes
(traits, classe de types, modules
ou encore copié/collé)

Alors T a est un foncteur

et je peux produire gratuitement, par exemple,
une fonction telle que :
replace:: b -> (T a -> T b)

C'est un motif tellement récurrent que beaucoup de langages l'ont intégré (avec d'autres joyeuseries comme les Applicatives et les Monades) dans la grammaire de leur langage (Haskell, Fsharp, OCaml et Scala).

Avants ces avancées grammaticales, on utilisait des opérateurs
>>=, <$>, <*> ou encore <*?
(imposant de la programmation tacite)

C'est un motif tellement récurrent que beaucoup de langages l'ont intégré (avec d'autres joyeuseries comme les Applicatives et les Monades) dans la grammaire de leur langage (Haskell, Fsharp, OCaml et Scala).

Avants ces avancées grammaticales, on utilisait des opérateurs
>>=, <$>, <*> ou encore <*?
(imposant de la programmation tacite)

ce qui introduit un premier **cas d'usage sérieux** à la programmation tacite !

Elle permet d'incuber des évolutions dans le langage
et d'expérimenter leur expérience utilisateur

Cette contrainte se matérialise de cette forme

Si pour un type T paramétré par a

j'ai une fonction map telle que :
map :: (a -> b) -> (T a -> T b)
en respectant certaines lois

par le biais de contraintes
(traits, classe de types, modules
ou encore copié/collé)

Peut-on toujours trouver une
map pour
n'importe quel T a ?

Alors T a est un foncteur

et je peux produire gratuitement, par exemple,
une fonction telle que :
replace:: b -> (T a -> T b)

Comment trouver map pour ce type:

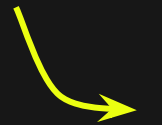
```
type Predicate a = a ->Bool
```

Comment trouver map pour ce type:

```
type Predicate a = a ->Bool
```

a est en position contravariante

Comment trouver map pour ce type:
type Predicate a = a ->Bool



a est en position contravariante

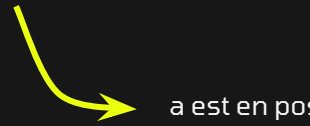
```haskell
data Maybe a =
   Just a
 | Nothing
```

```haskell
data Result a =
   Ok a
 | Error String
```

```haskell
data List a =
   Cons (a, List a)
 | Nil
```

a est en position covariante

Comment trouver map pour ce type:
type Predicate a = a ->Bool



a est en position contravariante

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

```
data Result a =
   Ok a
 | Error String
```

```
data List a =
   Cons (a, List a)
 | Nil
```

Cette contrainte se matérialise de cette forme

Si pour un type T paramétré par a covariant

j'ai une fonction map telle que :
map :: (a -> b) -> (T a -> T b)
en respectant certaines lois

par le biais de contraintes
(traits, classe de types, modules
ou encore copié/collé)

on peut toujours trouver une
map pour
n'importe quel T a covariant

Alors T a est un foncteur

et je peux produire gratuitement, par exemple,
une fonction telle que :
replace:: b -> (T a -> T b)

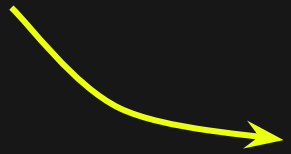
Il existe aussi des T a contravariants intéressants

Il existe aussi des T a contravariants intéressants

- type Predicate a = a -> Bool
- type PrettyPrinter a = a -> String
- type Equatable a = a -> a -> Bool
- type Orderable a = a -> a -> Int

Il existe aussi des T a contravariants intéressants

- type Predicate a = a -> Bool
- type PrettyPrinter a = a -> String
- type Equatable a = a -> a -> Bool
- type Orderable a = a -> a -> Int

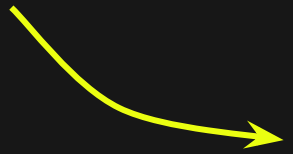


Et si on peut implémenter

```
contramap:: (b -> a) -> (T a -> T b)
```

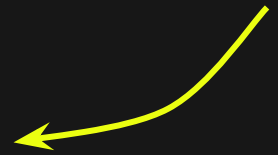
Il existe aussi des T a contravariants intéressants

- type Predicate a = a -> Bool
- type PrettyPrinter a = a -> String
- type Equatable a = a -> a -> Bool
- type Orderable a = a -> a -> Int



Et si on peut implémenter

```
contramap:: (b -> a) -> (T a -> T b)
```



On a un foncteur contravariant

modulo quelques lois, comme chaque fois

Il existe aussi des T a contravariants intéressants

- type Predicate a = a -> Bool
- type PrettyPrinter a = a -> String
- type Equatable a = a -> a -> Bool
- type Orderable a = a -> a -> Int

Contramap peut sembler peu intuitif

```
contramap:: (b -> a) -> (T a -> T b)
map       :: (a -> b) -> (T a -> T b)
```

Et si on peut implémenter

```
contramap:: (b -> a) -> (T a -> T b)
```

On a un foncteur contravariant

modulo quelques lois, comme chaque fois

Il existe aussi des T a contravariants intéressants

- type Predicate a = a -> Bool
- type PrettyPrinter a = a -> String
- type Equatable a = a -> a -> Bool
- type Orderable a = a -> a -> Int

Contramap peut sembler peu intuitif

```
contramap:: (b -> a) -> (T a -> T b)
map       :: (a -> b) -> (T a -> T b)
```

nit: remplacer contramap par using

Et si on peut implémenter

```
contramap:: (b -> a) -> (T a -> T b)
```

On a un foncteur contravariant

modulo quelques lois, comme chaque fois

```
contramap:: (b -> a) -> (T a -> T b)
map       :: (a -> b) -> (T a -> T b)
```

```
contramap:: (b -> a) -> (T a -> T b)
map       :: (a -> b) -> (T a -> T b)
```

comme a **est covariant** on
applique la fonction après.

on produit une valeur

comme a **est contravariant** on applique la fonction avant.

on provisionne une valeur

```
contramap:: (b -> a) -> (T a -> T b)
map       :: (a -> b) -> (T a -> T b)
```

comme a **est covariant** on applique la fonction après.

on produit une valeur

ET POUR a -> a ?

ET POUR a -> a ?

invariant et on s'en
fiche un peu

ET À QUOI ÇA SERT DE SAVOIR ÇA ?

On peut dériver des combinateurs via des
fonctions minimales

ET À QUOI ÇA SERT DE SAVOIR ÇA ?

On peut dériver des combinateurs via des fonctions minimales

ET À QUOI ÇA SERT DE SAVOIR ÇA ?

on peut spéculer sur des invariants et donc, du comportement

On peut dériver des combinateurs via des fonctions minimales

ET À QUOI ÇA SERT DE SAVOIR ÇA ?

on peut spéculer sur des invariants et donc, du comportement

Si je comprend les fonctions minimales, je peux **de facto** raisonner sur toute l'API

La paramétricité et peu d'abstraction nous donnent des théorèmes gratuits et des intuitions fortes sur comment utiliser des valeurs de type T a, peu importe le T

ON PEUT
GENERALISER
NOTRE RÉFLEXION

ON PEUT
GENERALISER
NOTRE RÉFLEXION

pour un T a b

ON PEUT GENERALISER NOTRE RÉFLEXION

pour un T a b

si a et b sont covariants on parle de bifoncteur

ON PEUT GENERALISER NOTRE RÉFLEXION

pour un T a b

si a et b sont covariants on parle de bifoncteur

```
bimap::
    (a -> c) ->
    (b -> d) ->
    (T a b -> T c d)
```

ON PEUT
GENERALISER
NOTRE RÉFLEXION

pour un T a b

ON PEUT GENERALISER NOTRE RÉFLEXION

pour un T a b

si a est contravariant et b est covariant on parle de profoncteur

ON PEUT GENERALISER NOTRE RÉFLEXION

pour un T a b

si a est contravariant et b est covariant on parle de profoncteur

```
dimap::
    (c -> a) ->
    (b -> d) ->
    (T a b -> T c d)
```

Nous permettant
d'implémenter gratuitement

```
map       f = dimap id f
contramap f = dimap f id
```

ON PEUT GENERALISER NOTRE REFLEXION

```
dimap::
    (c -> a) ->
    (b -> d) ->
    (T a b -> T c d)
```

pour un T a b

si a est contravariant et b est covariant on parle de profoncteur

UN CANDIDAT ?

Le type des fonctions `a -> b`

UN CANDIDAT ?

Le type des fonctions a -> b

UN CANDIDAT ?

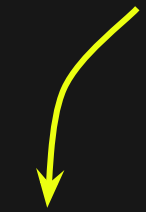
On pourrait se demander si
on peut généraliser la composition ?

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
(.) :: T b c -> T a b -> T a c
```

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```



```
(.) :: T b c -> T a b -> T a c
```

Super, on est revenu au point de départ... mais génériquement !

T a en fonction de la variance de a nous permet de raisonner arbitrairement sur T sans le connaitre !

T a en fonction de la variance de a nous permet de raisonner arbitrairement sur T sans le connaitre !

C'est pareil pour une grande classe de T a b :

- type Maybale a b = a -> Maybe b
- type Failable a b = a -> Failable b

T a en fonction de la variance de a nous permet de raisonner arbitrairement sur T sans le connaitre !

C'est pareil pour une grande classe de T a b :

- type Maybale a b = a -> Maybe b
- type Failable a b = a -> Failable b

Plus génériquement : type P a b = a -> T b où est T covariant

T a en fonction de la variance de a nous permet de raisonner arbitrairement sur T sans le connaitre !

C'est pareil pour une grande classe de T a b :

- type Maybale a b = a -> Maybe b
- type Failable a b = a -> Failable b

Plus génériquement : type P a b = a -> T b où est T covariant

Pour la frime on appelle ce type une Flêche de Kleisli, ou une Reader Monad

Alors, oui...

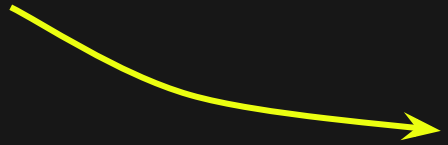
```
let
    fullContent      = readFile "post.md"
    (meta, content)  = extractMetaData
    contentHtml      = MarkdownToHtml content
    finalContent     = injectInTemplate "aPost.tpl.html"
in
writeFile "post.html" finalContent
```

est plus lisible que :

Alors, oui...

```
let
    fullContent      = readFile "post.md"
    (meta, content)  = extractMetaData
    contentHtml      = MarkdownToHtml content
    finalContent     = injectInTemplate "aPost.tpl.html"
in
writeFile "post.html" finalContent
```

est plus lisible que :

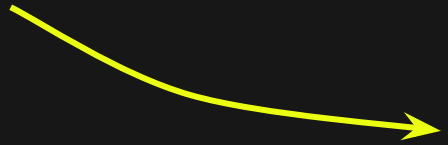


```
readFile "post.md"
. extractMetaData
. snd MarkdownToHtml
. injectInTemplate "aPost.tpl.html"
. writeFile "post.html"
```

Alors, oui...

```
let
    fullContent      = readFile "post.md"
    (meta, content)  = extractMetaData
    contentHtml      = MarkdownToHtml content
    finalContent     = injectInTemplate "aPost.tpl.html"
in
writeFile "post.html" finalContent
```

est plus lisible que :



```
readFile "post.md"
. extractMetaData
. snd MarkdownToHtml
. injectInTemplate "aPost.tpl.html"
. writeFile "post.html"
```

```
data Task a b = {
   dependencies :: Set File
   action :: a -> IO b
}
```

```haskell
data Task a b = {
   dependencies :: Set File
   action :: a -> IO b
}
```

```haskell
(.) (Task d1 a1) (Task d2 a2) = Task {
   dependencies = d1 U d2
   action = a1 <=< a2 -- on exécute a1 puis a2
}
```

```haskell
data Task a b = {
   dependencies :: Set File
   action :: a -> IO b
}
```

```haskell
(.) (Task d1 a1) (Task d2 a2) = Task {
   dependencies = d1 U d2
   action = a1 <=< a2 -- on exécute a1 puis a2
}
```

Avec par exemple :

```haskell
readFile :: Filepath -> Task () String
readFile file = Task {
   dependencies = Set.singleton file
   action = \() -> IO.readFile file
}
```

```haskell
data Task a b = {
   dependencies :: Set File
   action :: a -> IO b
}
```

```haskell
(.) (Task d1 a1) (Task d2 a2) = Task {
   dependencies = d1 U d2
   action = a1 <=< a2 -- on exécute a1 puis a2
}
```

Avec par exemple :

```haskell
readFile :: Filepath -> Task () String
readFile file = Task {
   dependencies = Set.singleton file
   action = \() -> IO.readFile file
}
```

Le code ne change pas !

```haskell
readFile "post.md"
. extractMetaData
. snd MarkdownToHtml
. injectInTemplate "aPost.tpl.html"
. writeFile "post.html"
```

```haskell
data Task a b = {
   dependencies :: Set File
   action :: a -> IO b
}
```

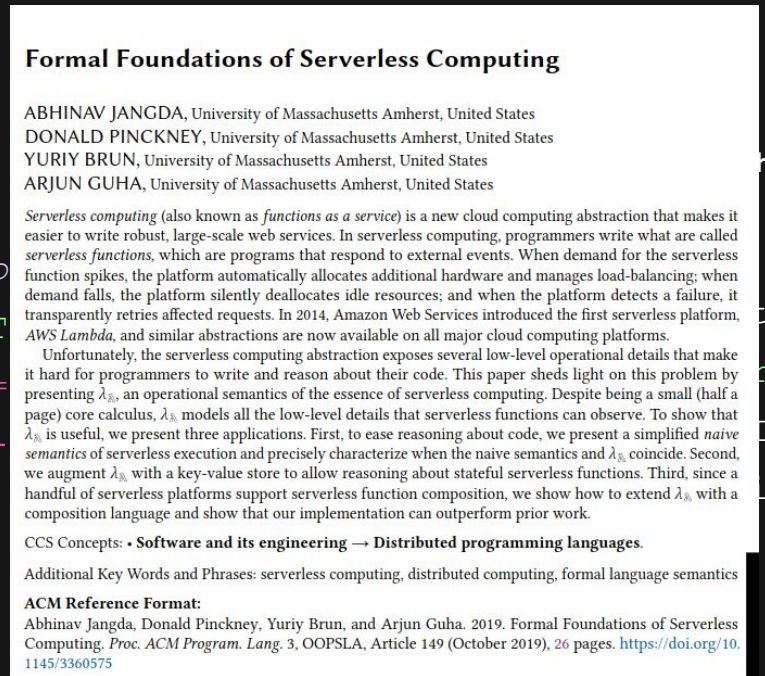
```haskell
(.) (Task d1 a1) (Task d2 a2) = Task {
   dependencies = d1 U d2
   action = a1 <=< a2 -- on exécute a1 puis a2
}
```

Avec par exemple :

```haskell
readFile :: Filep
readFile file = T
   dependencies =
   action = \() -
}
```

hange pas !

```haskell
"post.md"
tMetaData
rkdownToHtml
nTemplate "aPost.tpl.html"
le "post.html"
```



Formal Foundations of Serverless Computing

ABHINAV JANGDA, University of Massachusetts Amherst, United States
DONALD PINCKNEY, University of Massachusetts Amherst, United States
YURIY BRUN, University of Massachusetts Amherst, United States
ARJUN GUHA, University of Massachusetts Amherst, United States

Serverless computing (also known as *functions as a service*) is a new cloud computing abstraction that makes it easier to write robust, large-scale web services. In serverless computing, programmers write what are called *serverless functions*, which are programs that respond to external events. When demand for the serverless function spikes, the platform automatically allocates additional hardware and manages load-balancing; when demand falls, the platform silently deallocates idle resources; and when the platform detects a failure, it transparently retries affected requests. In 2014, Amazon Web Services introduced the first serverless platform, *AWS Lambda*, and similar abstractions are now available on all major cloud computing platforms.

Unfortunately, the serverless computing abstraction exposes several low-level operational details that make it hard for programmers to write and reason about their code. This paper sheds light on this problem by presenting $\lambda_{\text{乂}}$, an operational semantics of the essence of serverless computing. Despite being a small (half a page) core calculus, $\lambda_{\text{乂}}$ models all the low-level details that serverless functions can observe. To show that $\lambda_{\text{乂}}$ is useful, we present three applications. First, to ease reasoning about code, we present a simplified *naive semantics* of serverless execution and precisely characterize when the naive semantics and $\lambda_{\text{乂}}$ coincide. Second, we augment $\lambda_{\text{乂}}$ with a key-value store to allow reasoning about stateful serverless functions. Third, since a handful of serverless platforms support serverless function composition, we show how to extend $\lambda_{\text{乂}}$ with a composition language and show that our implementation can outperform prior work.

CCS Concepts: • **Software and its engineering → Distributed programming languages**.

Additional Key Words and Phrases: serverless computing, distributed computing, formal language semantics

ACM Reference Format:
Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 149 (October 2019), 26 pages. https://doi.org/10.1145/3360575

Pour la forme

T a b avec dimap s'appelle un Profoncteur

T a b avec compose s'appelle un Semigroupoid (si on lui ajoute un élément neutre, ça devient une catégorie)

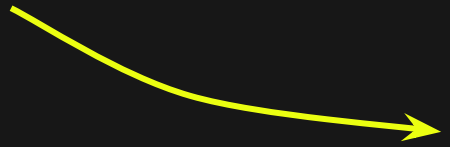
T a b avec compose, dimap, et une fonction additionnelle fst s'appelle une Arrow

Pour la forme

T a b avec dimap s'appelle un Profoncteur

T a b avec compose s'appelle un Semigroupoid (si on lui ajoute un élément neutre, ça devient une catégorie)

T a b avec compose, dimap, et une fonction additionnelle fst s'appelle une Arrow



Ces encodages permettent de s'abstraire de la grammaire (et on l'a vu ailleurs: Js, Java etc.)

Pour conclure

Même si ça peut tendre à du code plus dur à lire, je vous invite à encoder les Task sans utiliser de point-free (tout en ayant les même garanties)

Le point-free/tacite peut donc avoir parfois des avantages (en attendant une évolution de la grammaire ou pour palier à des choses que la grammaire ne peut pas atteindre)

Raisonner par des abstractions permet de généraliser des comportements et de capturer des intuitions !

Pour conclure

Même si ça peut tendre à du code plus dur à lire, je vous invite à encoder les Task sans utiliser de point-free (tout en ayant les même garanties)

Le point-free/tacite peut donc avoir parfois des avantages (en attendant une évolution de la grammaire ou pour palier à des choses que la grammaire ne peut pas atteindre)

Raisonner par des abstractions permet de généraliser des comportements et de capturer des intuitions !

Merci !