Un ode à la programmation *tacite*

Xavier Van de Woestyne - **LambdaNantes** - xvw.lol - **Tarides**

Nuit des Meetups **2024**

Lambda Nantes

Notre logo ridicule

- Langages fonctionnels, applicatifs, cools et un peu les preuves
- Déjà 4 *Meetups* (OCaml, Haskell et Scala)
- Moralement impliqué dans l'organisation de **ScalaIO 2024** (à Nantes !)
- On recherche toujours des *orateurs/oratrices*
- Un **Workshop** en préparation !!!

*La programmation fonctionnelle est devenu un **trait courant** dans les langages mainstreams.*

Pourtant !

*La programmation fonctionnelle est devenu un **trait courant** dans les langages mainstreams.*

Pourtant !

- Elle serait **trop compliquée**
- Peu adaptée **aux besoins de l'industrie**

*La programmation fonctionnelle est devenu un **trait courant** dans les langages mainstreams.*

Pourtant !

- Elle serait **trop compliquée**
- Peu adaptée **aux besoins de l'industrie**

Pourquoi ?

La programmation fonctionnelle est devenue mainstream donc les afficionados ont dû trouver d'autres sujets de conversations en meetup:

Pourquoi ?

La programmation fonctionnelle est devenue mainstream donc les afficionados ont dû trouver d'autres sujets de conversations en meetup:

- Les usages torturés des **systèmes de types**
- les abstractions *discutablement tirée* de la **théorie des catégories**
- **l'abstraction d'effets**.

- Les usages torturés des **systèmes de types**
- les abstractions *discutablement tirée* de la **théorie des catégories**
- **l'abstraction d'effets**.

Une brochette de sujets intéressants (*mais parfois difficiles à mettre en place industriellement et pouvant être intimidant*).

- Les usages torturés des **systèmes de types**
- les abstractions *discutablement tirée* de la **théorie des catégories**
- **l'abstraction d'effets**.

Une brochette de sujets intéressants (*mais parfois difficiles à mettre en place industriellement et pouvant être intimidant*).

Dans cette présentation, **on va essayer de revenir aux sources** et parler de **composition de fonctions** !

Plan

- Qu'est-ce que la **programmation tacite** (*une monstruosité arrivée via APL*)
- Un petit rappel sur les **types algébriques**
- Lesa types paramétrés
  - La polarité (co et contravariance)
- Quand la grammaire ne suffit plus
- Sur la composition de fonction *générique*

Tout commence avec ce très joli opérateur :

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

Tout commence avec ce très joli opérateur :

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

- `(f . g) x = f (g x)`

Tout commence avec ce très joli opérateur :

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

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

C’est joli !

C'est joli !

C'est aussi la porte ouverte à toute les fenêtre! ***Ca permet de faire de la programmation "dite" tacite****, ou encore* ***point-free****.*

C'est joli !

C'est aussi la porte ouverte à toute les fenêtre! ***Ca permet de faire de la programmation "dite" tacite****, ou encore* ***point-free****.*

En déclaration **pleins d'opérateurs** on peut facilement arriver à ce genre de choses :

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

En déclaration **pleins d'opérateurs** on peut facilement arriver à ce genre de choses :

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

Qui correspond à :

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

En déclaration **pleins d'opérateurs** on peut facilement arriver à ce genre de choses :

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

Qui correspond à :

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

lol

C'est tellement éclaté que :

- Il existe un **bot** sur l'IRC Haskell `\pl` qui convertit du code "normal" en **point-free**
- le site https://pointfree.io qui fait pareil

L'exemple précédent est volontairement exagéré mais :

On trouve souvent ce genre de code dans code-bases Haskell

```
selectM x y = x >>= \case Left  a -> ($a) <$> y
                         Right b -> pure b
```

L'exemple précédent est volontairement exagéré mais :

On trouve souvent ce genre de code dans code-bases Haskell

```
selectM x y = x >>= \case Left  a -> ($a) <$> y
                         Right b -> pure b
```

Ce qui reste **cognitivement difficile à lire**.

L'exemple précédent est volontairement exagéré mais :

On trouve souvent ce genre de code dans code-bases Haskell

```
selectM x y = x >>= \case Left  a -> ($a) <$> y
                         Right b -> pure b
```

Ce qui reste **cognitivement difficile à lire**. C'est une forme **d'obfuscation**.

L'exemple précédent est volontairement exagéré mais :

On trouve souvent ce genre de code dans code-bases Haskell

```
selectM x y = x >>= \case Left  a -> ($a) <$> y
                         Right b -> pure b
```

Ce qui reste **cognitivement difficile à lire**. C'est une forme **d'obfuscation**.

Couplé avec de l'implicites, des modificateurs fixation (`f x y = x 'f' y`) et une bibliothèque standard qui crée des répétitions (`liftM`, `map`, `fmap`, etc.) /shrug

Même s'il existe des raisons valables de s'en servir

```
map (f . g . h) x
```

plutôt que

```
x |> map f |> map g |> map h
```

Même s'il existe des raisons valables de s'en servir

```
map (f . g . h) x
```

plutôt que

```
x |> map f |> map g |> map h
```

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

- En général, quand on **manipule des fonctions régulières**, il vaut mieux l'éviter.

Alors... pourquoi “une ode à cette forme de programmation” ?

*Je prétent **qu'elle peut être utile** et on va le voir en faisant un petit détour par les types algébriques.*

Types algébriques

Un moyen de construire des types plus complexes sur base de types plus spécialisés (d'où le "algébrique")

Des ADTs

- Des produits : `(a, b)`
- Des sommes : `Left a | Right b`

Des types fonctionels :

- `a -> b`

Très utiles pour modeliser un domaine métier ou des structures de données

Fixons le problème à 1000000 de dollars :

```haskell
-- Plus de NullPointerException
data Maybe a =
  | Just a
  | Nothing
```

Très utiles pour modeliser un domaine métier ou des structures de données

Fixons le problème à 1000000 de dollars :

```haskell
-- Plus de NullPointerException
data Maybe a =
  | Just a
  | Nothing
```

Mais potentiellement dur à utiliser

```haskell
incr x = x + 1
prod x = x * 10

v = case Just 10 of
      Nothing -> Nothing
      Just x -> case Just (incr x) of
         Just x -> Just (prod x)
```

Pour palier à ce problème, on propose une fonction

```haskell
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b) =
mapMaybe f x = case x of
  Nothing -> Nothing
  Just v   -> Just (f x)
```

Pour palier à ce problème, on propose une fonction

```
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b) =
mapMaybe f x = case x of
  Nothing -> Nothing
  Just v   -> Just (f x)
```

Qui permet de transformer une fonction `a -> b` en fonction qui agit sur des `Maybe a` pour produire des `Maybe b`.

Pour palier à ce problème, on propose une fonction

```
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b) =
mapMaybe f x = case x of
  Nothing -> Nothing
  Just v   -> Just (f x)
```

Qui permet de transformer une fonction `a -> b` en fonction qui agit sur des `Maybe a` pour produire des `Maybe b`.

Qui nous permet de dériver des combinateurs :

```
replaceInMaybe :: b -> Maybe a -> Maybe b =
replaceInMaybe replacement x =
  mapMaybe (\ _previousValue -> replacement) x
```

Et cette fonction peut se généraliser sur :

des `T a` arbitraires

```
mapList :: (a -> b) -> (List a -> List b)
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b)
etc.
```

Et cette fonction peut se généraliser sur :

des `T a` arbitraires

```
mapList :: (a -> b) -> (List a -> List b)
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b)
etc.
```

- On peut donc le généraliser au moyen de génération de code (TypeClasses ou Modules par exemple), **on appelle ça un Foncteur**. Associé à des “lois” on peut spéculer son comportement **génériquement**

*Cet 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 gramaticales, on utilisait des opérateurs >>=, <$>, <*> ou encore <*? (imposant de la programmation **tacite**)*

Est-ce que ça marche pour tous les `T a` ?

Est-ce que ça marche pour tous les `T a` ?

```
type Predicate a = a -> bool
```

Est-ce que ça marche pour tous les `T a` ?

```
type Predicate a = a -> bool
```

Comment implémenter une fonction `map` pour ce type, **parfaitement légitime** ?

Est-ce que ça marche pour tous les `T a` ?

```
type Predicate a = a -> bool
```

Comment implémenter une fonction `map` pour ce type, **parfaitement légitime** ?

On ne peut pas.

Parce que dans les exemples précédents, `a` était en **position covariante**, donc à droite de la **dernière flèche**. (On pourrait imaginer que `Maybe a` est en fait `() -> Maybe a`)

Dans notre exemple, le a est **avant la dernière flèche**, on dit qu'il est **en position contravariante**.

Dans notre exemple, le a est **avant la dernière flèche**, on dit qu'il est **en position contravariante**.

Et il en existe pleins d'autres qui sont aussi très utiles :

- `type Predicate a = a -> bool`
- `type PrettyPrinter a = a -> string`
- `type Equality a = a -> a -> bool`
- `type Ordering a = a -> a -> int`

Existe-il une fonction qui permet de rendre l'usage de fonction régulière `a -> b` utile sur des `T a` où `a` est contravariant ?

Existe-il une fonction qui permet de rendre l'usage de fonction régulière `a -> b` utile sur des `T a` où `a` est contravariant ?

Oui ! Au moyen de **foncteurs contravriants** (en fait, un **Foncteur** est un raccourcis terminologique pour parler de **foncteur covariant**).

Existe-il une fonction qui permet de rendre l'usage de fonction régulière `a -> b` utile sur des `T a` où `a` est contravariant ?

Oui ! Au moyen de **foncteurs contravriants** (en fait, un **Foncteur** est un raccourcis terminologique pour parler de **foncteur covariant**).

Et sa fonction "standard" est `contramap` :

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

Existe-il une fonction qui permet de rendre l'usage de fonction régulière `a -> b` utile sur des `T a` où `a` est contravariant ?

Oui ! Au moyen de **foncteurs contravriants** (en fait, un **Foncteur** est un raccourcis terminologique pour parler de **foncteur covariant**).

Et sa fonction "standard" est `contramap` :

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

outch, voila qui semble étrange ! Si j'ai un `T a` et une fonction qui va de `b` vers `a`, comment puis-je avoir un `T b` à la fin ?

Voyons avec `Predicate`

```
contramap f predicate = (\x -> predicate (f x))
```

Voyons avec `Predicate`

```
contramap f predicate = (\x -> predicate (f x))
```

- on applique en fait notre fonction avant de la passer au prédicat.
- On peut la lire comme “using” (plutot que “mapping”)

Voyons avec Predicate

```
contramap f predicate = (\x -> predicate (f x))
```

- on applique en fait notre fonction avant de la passer au prédicat.
- On peut la lire comme “using” (plutot que “mapping”)

```
int_to_string :: Int -> String
predicate_of_string :: Predicate String

-- On applique int_to_string avant et bien que l'on aie
-- une fonction de Int vers String
-- On peut bel et bien transformer notre Predicate String en
-- Predicate Int.
```

- `map` applique **après** la transformation (parce que a est **covariant**)
- `contramap` applique **avant** la transformation (parce que a est *contravariant*)

- `map` applique **après** la transformation (parce que `a` est **covariant**)
- `contramap` applique **avant** la transformation (parce que `a` est *contravariant*)

Et pour, par exemple `T a = a -> a` ?

- `map` applique **après** la transformation (parce que `a` est **covariant**)
- `contramap` applique **avant** la transformation (parce que `a` est *contravariant*)

Et pour, par exemple `T a = a -> a` ?

`a` est **invariant** (mais bon... hein)

Ok, cool . . . et ?

Ok, cool . . . et ?

- Grace aux lois **on peut spéculer des invariants** (par exemple, des foncteurs covariants préservent **toujours leurs structure**)
- Ces fonctions minimales (`map` et `contramap`) permettent de dériver des combinateurs utiles

Ok, cool ... et ?

- Grace aux lois **on peut spéculer des invariants** (par exemple, des foncteurs covariants préservent **toujours leurs structure**)
- Ces fonctions minimales (`map` et `contramap`) permettent de dériver des combinateurs utiles

- La relation avec ces **lois** permet de raisonner sur des types `T a`, sans connaitre le `T`.

Ok, cool ... et ?

- Grace aux lois **on peut spéculer des invariants** (par exemple, des foncteurs covariants préservent **toujours leurs structure**)
- Ces fonctions minimales (`map` et `contramap`) permettent de dériver des combinateurs utiles

- La relation avec ces **lois** permet de raisonner sur des types `T a`, sans connaitre le `T`.

En d'autres mots, si je comprend comment utiliser `map` pour `Maybe`, **je sais, *de-facto* comment utiliser `map` pour des `List` (ou des `Result`)** !

Ok, cool . . . et ?

- Grace aux lois **on peut spéculer des invariants** (par exemple, des foncteurs covariants préservent **toujours leurs structure**)
- Ces fonctions minimales (`map` et `contramap`) permettent de dériver des combinateurs utiles

- La relation avec ces **lois** permet de raisonner sur des types `T a`, sans connaitre le `T`.

En d'autres mots, si je comprend comment utiliser `map` pour `Maybe`, **je sais, *de-facto* comment utiliser `map` pour des `List` (ou des `Result`)** !

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`.

Il existe des composition triviale de foncteurs, par exemple si un type est paramétré par `a` et `b`, étant tous les deux **covariants** (ie: `type Prod a b = (a, b)`), on parle de **bifoncteur**.

Allons plus loin

Allons plus loin

qu'en est-il du type `a -> b`.

Allons plus loin

qu'en est-il du type `a -> b`.

- On est **contravariant** en `a`

Allons plus loin

qu'en est-il du type `a -> b`.

- On est **contravariant** en `a`
- On est **covariant** en `b`.

Allons plus loin

qu'en est-il du type `a -> b`.

- On est **contravariant** en `a`
- On est **covariant** en `b`.

On peut donc proposer une fonction qui va agir sur a et b :

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

Allons plus loin

qu'en est-il du type `a -> b`.

- On est **contravariant** en `a`
- On est **covariant** en `b`.

On peut donc proposer une fonction qui va agir sur a et b :

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

Qui lui aussi nous permet de dériver deux fonctions déjà vues !

```
map f v = dimap (\x -> x) f v
contramap f v = dimap f (\x -> x) v
```

Il est aussi possible de généraliser l'opérateur .

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

Il est aussi possible de généraliser l'opérateur `.`

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

Que l'on peut implémenter pour `a -> b` :

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

Il est aussi possible de généraliser l'opérateur `.`

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

Que l'on peut implémenter pour `a -> b` :

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

Il est possible d'imaginer énormément de fonctions aditionnelles qui agissent sur des produits et des sommes par exemple...

Cool, ***on est revenu au point de départ*** *on peut programmer avec des points, des* `snd`*, des* `fst` *et arriver sur du code illisible. . .*

Rappelons-nous que :

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

Rappelons-nous que :

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

C'est pareil pour des `T a b` :

Rappelons-nous que :

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

C'est pareil pour des `T a b` :

- `type Maybable a b = a -> Maybe b`
- `type Validable a g = a -> Validation b`

Rappelons-nous que :

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

C'est pareil pour des `T a b` :

- `type Maybable a b = a -> Maybe b`
- `type Validable a g = a -> Validation b`

- **Et plus génériquement** : `type P a b = a -> T b` pour **tout T a** où `a` est covariant!

Rappelons-nous que :

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

C'est pareil pour des `T a b` :

- `type Maybable a b = a -> Maybe b`
- `type Validable a g = a -> Validation b`

- **Et plus génériquement** : `type P a b = a -> T b` pour **tout T a** où `a` est covariant!

Pour la frime, un `a -> T b`, on appelle ça une **Flèche de Kleisli**, ou encore une **Reader Monade**.

Alors oui...

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

Alors oui...

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

Est probablement moins lisible que :

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

Je suis **entièrement d'accord** !

Je suis **entièrement d'accord** ! **Mais**

Je suis **entièrement d'accord** ! **Mais**

- J'ai dit en introduction **que pour des fonctions régulière, c'est overkill**

Je suis **entièrement d'accord** ! **Mais**

- J'ai dit en introduction **que pour des fonctions régulière, c'est overkill**
- Voyons un autre type :

```
data Task a b = {

   dependencies :: Set File
   action :: a -> IO b

}
```

Je suis **entièrement d'accord** ! **Mais**

- J'ai dit en introduction **que pour des fonctions régulière, c'est overkill**
- Voyons un autre type :

```
data Task a b = {

   dependencies :: Set File
   action :: a -> IO b

}


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

Et par exemple

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

etc.
```

Et par exemple

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

etc.
```

Nos tâches **capturent leurs dépendances** et on peut, par exemple, ne reconstruire **que les fichiers qui doivent être reconstruits** assurant la **minimalité**.

Et par exemple

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

etc.
```

Nos tâches **capturent leurs dépendances** et on peut, par exemple, ne reconstruire **que les fichiers qui doivent être reconstruits** assurant la **minimalité**.

On a toute les briques pour construire un tout petit build-system**

Et à quoi ressemblerait notre code précédent avec notre Task ?

Et à quoi ressemblerait notre code précédent avec notre Task ?

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

Et à quoi ressemblerait notre code précédent avec notre Task ?

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

identique,

Et à quoi ressemblerait notre code précédent avec notre Task ?

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

identique, comprendre l'implémentation à base de fonction **suffit à comprendre comment utiliser celui avec des Tasks** !

Pour la forme :

- `T a b` avec `dimap` s'appelle un **Profoncteur**
- `T a b` avec `compose`, `.`, s'appelle une **Catégorie** (un **Semigroupoid** en vrai mais bon)
- Et un `T a b` avec `compose` et `dimap` (et une fonction aditionnelle `fst`), s'appelle une **Arrow** !

Pour la forme :

- `T a b` avec `dimap` s'appelle un **Profoncteur**
- `T a b` avec `compose`, `.`, s'appelle une **Catégorie** (un **Semigroupoid** en vrai mais bon)
- Et un `T a b` avec `compose` et `dimap` (et une fonction aditionnelle `fst`), s'appelle une **Arrow** !

Réussir à fournir de la syntaxe pour ce genre de construction est **très difficile** et l'approche par abstraction permet de **généraliser des comportements**!

Pour la forme :

- `T a b` avec `dimap` s'appelle un **Profoncteur**
- `T a b` avec `compose`, `.`, s'appelle une **Catégorie** (un **Semigroupoid** en vrai mais bon)
- Et un `T a b` avec `compose` et `dimap` (et une fonction aditionnelle `fst`), s'appelle une **Arrow** !

Réussir à fournir de la syntaxe pour ce genre de construction est **très difficile** et l'approche par abstraction permet de **généraliser des comportements**!
Ici, on a généralisé sur **des trucs qui ressemblent à des fonctions**.

- Ce modèle de Build System est utilisé par le générateur de site Statique YOCaml et est aussi appelé **un build système Applicatif**

Les constructions par “encodages” sont légions ailleurs

(Construction par encodage en opposition à la construction par la grammaire) :

Les constructions par “encodages” sont légions ailleurs

(Construction par encodage en opposition à la construction par la grammaire) :

L’encodage des lambdas sans lambdas

```
interface Lam<A, B> {
  B apply(A x)
}
```

Les constructions par “encodages” sont légions ailleurs

(Construction par encodage en opposition à la construction par la grammaire) :

L’encodage des lambdas sans lambdas

```
interface Lam<A, B> {
  B apply(A x)
}
```

Régler le Callback Hell de JavaScript

- Passage par callback
- Promesses et utilisation de `.then` pour ne plus imposer le *nesting* : **encodage**
- Arrivée de `Async/Await` : **gramnaire**

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)

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)

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** !

- par pitié, venez nombreux à **LambdaNantes**, on s'amuse bien et on a un compte Twitter (**@lambdaNantes**), partiellement actif ! (on a aussi un groupe signal pour organiser des pots)

Merci beaucoup !

Questions, remarques ? Désolé pour les slides faits à l'arrache.