Programmation Fonctionnelle

Applicatives

Michael Lampis

2023-2024

Functor – Rappel

- On a vu la classe (paramétrique) Functor

```
Prelude> :info Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  {-# MINIMAL fmap #-}
```

Exemples :

- `Maybe`, `[]`, `BinTree`
- Idée : si `f` est un functor et `a` est un type, `f a` est un type qui représente une collection (un container) d'éléments de type `a`.
- Notamment, on a le droit d'utiliser `fmap` pour appliquer une fonction sur chaque élément de la collection.

```
Prelude> fmap (+2) (Just 3)
Just 5
Prelude> fmap (+2) [1,2,3,4]
[3,4,5,6]
```

Applicatives

- La fonction `fmap` nous donne une façon pratique d'accéder aux contenus d'un functor, sans prendre en compte la structure du functor (donc, sans faire de pattern matching).
- Désavantage : `fmap` ne peut être utilisé qu'avec des fonctions d'une seule variable.
- Cas d'usage : on aimerait pouvoir faire quelque chose comme la ligne suivante

```
fmap2 (+) (Just 2) (Just 3)
.. -> Just 5
fmap2 (+) [2] [3]
.. -> [5]
```

- Les **applicatives** sont une sous-classe des functors qui nous offrent cette fonctionnalité :
- Une applicative est un container qui nous permet d'appliquer des fonctions de plusieurs variables sur ses contenus.

Applicatives

- On peut demander au compilateur des précisions :

```
Prelude> :info Applicative
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  {-# MINIMAL pure, (<*>) #-}
```

- Idée principale : la fonction `<*>` prend
  - Une collection de fonctions (`f (a->b)`)
  - Une collection de données (`f a`)
- Et applique chaque fonction de la collection sur chaque élément pour obtenir une collection de résultats.
- Donc, ça sert à quoi, et quelle est la différence avec `fmap` ?

fmap vs $< * >$

- Reprenons notre exemple :

```
fmap2 (+) (Just 2) (Just 3)
```

- Pourquoi est-ce que ça ne marche pas avec la fonctionnalité de functor ?
  - Quel est le type de `fmap (+) (Just 2)` ?
  - ... `Maybe (Int -> Int)`
  - Donc, `(fmap (+) (Just 2)) (Just 5)` ne compile pas
  - Or, on a un functor qui contient une fonction (`fmap (+) (Just 2)`) et un functor qui contient une valeur (`Just 3`)
  - Donc, il nous faut une fonction qui prend ces deux ingrédients et produit un functor qui contient le résultat.
- Rappel :

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

L'opération $<*>$

On écrit donc

```
Prelude> fmap (+) (Just 2) <*> (Just 3)
Just 5
```

- L'expression `fmap (+) (Just 2)` a type `Maybe (Int->Int)`
- L'expression `Just 3` a type `Maybe Int`
- `(<*>) :: Applicative f => f (a -> b) -> f a -> f b`
- Donc, ce qu'on a écrit est bien typé (étant donné que `Maybe` est une instance de `Applicative`)
- De la même façon on peut écrire

```
Prelude> fmap (+) [2] <*> [3]
[5]
```

Applicatives – Décryptage

```
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  {-# MINIMAL pure, (<*>) #-}
```

- Un type `f` d'un seul paramètre
  - `f` a kind `* -> *`
- ..est une instance d'applicative si `f` est un Functor est on a les fonctions
  - `pure` : prend un élément de type `a` et retourne une collection triviale qui ne contient que cet élément
  - `<*>` : prend une collection de fonctions `a->b` et une collection de `a` et applique chaque fonction sur chaque élément pour obtenir une collection de `b`

Maybe I'm Applicative?

- On a fait la supposition que `Maybe` est une instance de `Applicative`. Comment ça marche ?
- Dans la bibliothèque de Haskell on a :

```
instance Applicative Maybe where
  pure                    = Just
  (Just f) <*> (Just x)   = Just (f x)
  _ <*> _                 = Nothing
```

Explication :

- `pure :: a -> Maybe a` construit un container trivial qui ne contient que l'élément donné.
- `<*>` retourne `Nothing` si un des deux arguments et `Nothing`. Sinon, le premier argument contient une fonction qui est appliqué sur le contenu du deuxième argument.

Applications de Maybe Applicative

```
Prelude> fmap (++) (Just "a") <*> (Just "b")
Just "ab"
Prelude> fmap (++) (Just "a") <*> Nothing
Nothing
```

Puisque la fonction `fmap` est très souvent utilisée, on a le synonyme `<$>` défini comme `f <$> x = fmap f x`. Donc on peut écrire

```
Prelude>  (++) <$> (Just "a") <*> (Just "b")
Just "ab"
Prelude>  (*) <$> (Just 3) <*> (Just 5)
Just 15
```

Intérêt : ce n'est pas loin d'écrire simplement `(*) 3 5`, mais nous permet de garder les `Maybe`
On peut aussi profiter de `pure` pour écrire

```
Prelude> pure (*) <*> (Just 3) <*> (Just 5)
Just 15
```

Listes Applicatives

- Un autre functor qu'on a vu est le type de listes `[]`.
- Ce type est aussi une instance de `Applicative` avec la définition suivante

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [ f x | f<-fs, x<-xs]
```

- `pure` retourne la liste triviale qui ne contient que l'élément donné
- `<*>` prend une liste de fonctions et une liste d'éléments et applique chaque fonction sur chaque élément pour produire une liste de résultats.

Listes – Exemples

```
Prelude> [(7+),(2*),(3^)] <*> [1,2,3]
[8,9,10,2,4,6,3,9,27]
Prelude> (+) <$> [1,2,3] <*> [4,5,6]
[5,6,7,6,7,8,7,8,9]
Prelude> (++) <$> ["a","b"] <*> ["c","d","e"]
["ac","ad","ae","bc","bd","be"]
Prelude> (:) <$> "ab" <*> ["c","d","e"]
["ac","ad","ae","bc","bd","be"]
Prelude> (.) <$> [(2+),(3+)] <*> [(*4),(*5)] <*> [1,2,3]
[6,10,14,7,12,17,7,11,15,8,13,18]
```

Les Lois des Applicatives

- Comme les Functors, toutes les instances de la classe `Applicative` doivent satisfaire quelques conditions (lois)
  - Ces conditions ne sont pas vérifiables par le compilateur. C'est la responsabilité du programmeur de les vérifier.

Lois des Applicatives :

1. `pure id <*> x = x`
2. `pure f <*> pure x = pure (f x)`
3. `u <*> pure y = pure ($ y) <*> u`
4. `(.) <*> u <*> v <*> w = u <*> (v <*> w)`

- 1 dit que la fonction `id` ne change rien (logique). 2 dit que si on a une seule fonction `f` et une seule valeur `x`, on applique `f` sur `x`
- Pour 3, rappelez–vous que `($ y) f` équivaut à `f y`. `pure` ne devrait pas affecter cette équivalence.
- Pour 4, on se rappelle que `(.) f g x` équivaut à `f (g x)`.

SequenceA

- L'idée des applicatives est de nous permettre d'utiliser des fonctions qui combinent les éléments contenus dans un functor
  - ...alors qu'un functor basique ne nous permet que d'appliquer une fonction sur chaque élément séparément.
- Regardons une application avec la fonction `SequenceA`

```
sequenceA :: (Applicative f) => [f a] -> f [a]
-- En fait sequenceA :: (Applicative f, Traversable t) =>
                                        t (f a) -> f (t a)
-- mais on parlera de Traversable plus tard...
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
-- Exemple
Prelude> sequenceA [Just 3, Just 4, Just 5]
Just [3,4,5]
-- car (:) <$> (Just 3) <*> Just [4,5] --> Just [3,4,5]
-- et  (:) <$> (Just 4) <*> Just [5]   --> Just [4,5]
-- et  (:) <$> (Just 5) <*> Just []    --> Just [5]
```

SequenceA – Exemples

- Donc, ça fait quoi `SequenceA` ?
- Transforme une liste de containers en un container d'une liste.

Essayez de prédire ce qui est retourné par les expressions suivantes :

```
Prelude> sequenceA [Just 3, Nothing , Just 5]
```

```
Prelude> sequenceA ["ab"]
```

```
Prelude> sequenceA ["ab","cd",""]
```

```
Prelude> sequenceA ["ab","cd","ef"]
```

SequenceA – Exemples

- Donc, ça fait quoi `SequenceA` ?
- Transforme une liste de containers en un container d'une liste.

Essayez de prédire ce qui est retourné par les expressions suivantes :

```
Prelude> sequenceA [Just 3, Nothing , Just 5]
Nothing

Prelude> sequenceA ["ab"]


Prelude> sequenceA ["ab","cd",""]


Prelude> sequenceA ["ab","cd","ef"]
```

SequenceA – Exemples

- Donc, ça fait quoi `SequenceA` ?
- Transforme une liste de containers en un container d'une liste.

Essayez de prédire ce qui est retourné par les expressions suivantes :

```
Prelude> sequenceA [Just 3, Nothing , Just 5]
Nothing

Prelude> sequenceA ["ab"]
["a","b"]

Prelude> sequenceA ["ab","cd",""]


Prelude> sequenceA ["ab","cd","ef"]
```

SequenceA – Exemples

- Donc, ça fait quoi `SequenceA` ?
- Transforme une liste de containers en un container d'une liste.

Essayez de prédire ce qui est retourné par les expressions suivantes :

```
Prelude> sequenceA [Just 3, Nothing , Just 5]
Nothing

Prelude> sequenceA ["ab"]
["a","b"]

Prelude> sequenceA ["ab","cd",""]
[]

Prelude> sequenceA ["ab","cd","ef"]
```

SequenceA – Exemples

- Donc, ça fait quoi `SequenceA` ?
- Transforme une liste de containers en un container d'une liste.

Essayez de prédire ce qui est retourné par les expressions suivantes :

```
Prelude> sequenceA [Just 3, Nothing , Just 5]
Nothing

Prelude> sequenceA ["ab"]
["a","b"]

Prelude> sequenceA ["ab","cd",""]
[]

Prelude> sequenceA ["ab","cd","ef"]
["ace","acf","ade","adf","bce","bcf","bde","bdf"]
```

Références

À regarder aussi :

- LYAH chapitre 11 `http://learnyouahaskell.com/functors-applicative-functors-and-monoids`
  - Functors redux et Applicatives Functors
- Haskell Wikibook `https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor`
- Haskell Wikibook `https://en.wikibooks.org/wiki/Haskell/Applicative_functors`