

## PLP

## Monads and more

13 December 2025

## Table des matières

<b>1</b>	<b>Functors</b>	<b>1</b>
1.1	Functors class	2
1.2	Opérateur <code>&lt;\$</code>	2
1.3	Implement Functor for Data Types	2
<b>2</b>	<b>Applicative</b>	<b>2</b>
2.1	<code>pure</code> and <code>&lt;*&gt;</code>	3
2.2	Functor Applicative	3
2.2.1	Example with Maybe	3
2.3	Laws of Applicative	3
<b>3</b>	<b>Monads</b>	<b>3</b>
3.1	<code>return</code> and <code>&gt;=&gt;</code>	4
3.2	Monad class	4
3.3	do Notation	4
3.4	Laws of Monads	4
<b>4</b>	<b>Lazy Evaluation</b>	<b>4</b>
4.1	Evaluating expressions	5
4.2	Reduction strategies	5
4.2.1	Innermost reduction	5
4.2.2	Outermost reduction	5
4.3	Number of reductions	6
4.4	Sharing of expressions	6
4.5	Infinite lists	7
4.6	Strict evaluation	7

# 1 Functors

Le langage Haskell fournit une fonction `fmap` qui permet d'appliquer une fonction ordinaire à une valeur encapsulée dans un contexte (un type de données paramétré). Par exemple, si nous avons une liste de nombres et que nous voulons ajouter 1 à chaque élément, nous pouvons utiliser `fmap` pour appliquer la fonction `(+1)` à chaque élément de la liste.

Cela nous permet donc d'être agnostique au contexte dans lequel se trouve la valeur. Que ce soit une liste, un `Maybe`, ou tout autre type de données qui implémente l'interface des foncteurs, nous pouvons utiliser `fmap` pour appliquer une fonction à la valeur encapsulée.

## ⚠ Warning

Sur une paire, `fmap` n'applique la fonction que sur le second élément.

## 1.1 Functors class

La classe des foncteurs est définie comme suit en Haskell :

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
```

Un foncteur doit respecter deux lois importantes :

1. **Identité** : `fmap id = id`
2. **Composition** : `fmap f (fmap g x) == fmap (f . g) x`

## 1.2 Opérateur <\$

L'opérateur `<$` permet de remplacer toutes les valeurs dans un contexte par une valeur spécifique. Par exemple, si nous avons une liste `[1, 2, 3]` et que nous voulons remplacer tous les éléments par 0, nous pouvons utiliser `0 <$ [1, 2, 3]`, ce qui donnera `[0, 0, 0]`.

```
> "abc" <$ Just 123
Just "abc"
> "abc" <$ Nothing
Nothing
```

## 1.3 Implement Functor for Data Types

Pour implémenter la classe des foncteurs pour un type de données personnalisé, nous devons définir comment `fmap` agit sur ce type. Par exemple, pour une liste personnalisée, nous pourrions définir `fmap` comme suit :

```
data Option a = Some a | None

instance Functor Option where
  fmap f None = None
  fmap f (Some x) = Some (f x)

> fmap (+1) None
None
> fmap (+1) (Some 42)
Some 43
```

## 2 Applicative

Nous souhaiterions, dans un foncteur, pouvoir appliquer un nombre illimité d'arguments. C'est le rôle des applicatifs. Pour cela, nous avons besoin de deux fonctions principales : `pure` et `<*>`.

### 2.1 `pure` and `<*>`

La fonction `pure` permet d'encapsuler une valeur dans un contexte applicatif, tandis que l'opérateur `<*>` permet d'appliquer une fonction encapsulée dans un contexte à une valeur également encapsulée dans un contexte.

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

### 2.2 Functor Applicative

Les classes de foncteurs qui supportent les opérations `pure` et `<*>` sont appelées des applicatifs. Elles ont comme définition:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

#### 2.2.1 Example with Maybe

```
instance Applicative Maybe where
  pure :: a -> Maybe a
  pure = Just

  (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  Nothing <*> _ = Nothing
  (Just g) <*> mx = fmap g mx

> pure (+1) <*> Just 1
Just 2
> pure (+) <*> Just 1 <*> Just 2
Just 3
> pure (+) <*> Nothing <*> Just 2
Nothing
```

### 2.3 Laws of Applicative

Les applicatifs doivent respecter les lois suivantes :

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

### 3 Monads

Les monades sont une extension des applicatifs qui permettent de chaîner des opérations dépendantes les unes des autres. Elles introduisent deux fonctions principales : `return` (équivalent à `pure`) et `>>=` (bind).

#### 3.1 `return` and `>>=`

La fonction `return` encapsule une valeur dans une monade, tandis que l'opérateur `>>=` permet de chaîner des opérations qui retournent des valeurs encapsulées dans une monade.

#### 3.2 Monad class

La classe des monades est définie comme suit en Haskell :

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

#### 3.3 do Notation

Haskell fournit une syntaxe spéciale appelée « do notation » pour faciliter la lecture et l'écriture de chaînes d'opérations monadiques. Dans notre contexte, nous avons vu cette annotation avec les `IO` actions. Cependant, les `IO` sont des opérations monadiques comme les autres. Par exemple, pour la monade `Maybe`, nous pourrions écrire :

```
safeDivide :: Double -> Double -> Maybe Double
safeDivide _ 0 = Nothing
safeDivide x y = Just (x / y)

result :: Maybe Double
result = do
  a <- safeDivide 10 2
  b <- safeDivide a 0
  return b
```

Cela retrouverait `Nothing` car la deuxième division est invalide. Cela permet d'écrire des chaînes d'opérations monadiques de manière plus lisible, tout en s'assurant que les erreurs sont correctement propagées.

#### 3.4 Laws of Monads

Les monades doivent respecter les lois suivantes :

1. **Left identity** : `return a >>= f == f a`
2. **Right identity** : `m >>= return == m`
3. **Associativity** : `(m >>= f) >>= g == m >>= (\x -> f x >>= g)`

## 4 Lazy Evaluation

Haskell utilise l'évaluation paresseuse (lazy evaluation) par défaut, ce qui signifie que les expressions ne sont évaluées que lorsqu'elles sont nécessaires. Cela permet de travailler avec des structures de données infinies et d'améliorer les performances en évitant des calculs inutiles.

### 4.1 Evaluating expressions

L'évaluation paresseuse permet notamment 3 choses intéressantes :

- Evite de devoir faire des évaluations inutiles.
- Permet au programme d'être plus modulaire.
- Permet de travailler avec des structures de données infinies.

Le principe est simple: les expressions sont évaluées ou réduites en appliquant successivement des définitions de fonctions jusqu'à obtenir une valeur finale. Cependant, cette évaluation n'est effectuée que lorsque la valeur est réellement nécessaire pour le programme.

Exemple:

```
square n = n * n

square (3 + 4)
= square 7
= 7 * 7
= 49
```

On pourrait aussi écrire:

```
square (3 + 4)
= (3 + 4) * (3 + 4)
= 7 * (3 + 4)
= 7 * 7
= 49
```

### 4.2 Reduction strategies

Il existe plusieurs stratégies de réduction pour l'évaluation paresseuse, notamment :

- **Innermost reduction** : Réduit les expressions les plus internes en premier.
- **Outermost reduction** : Réduit les expressions les plus externes en premier.

#### 4.2.1 Innermost reduction

L'innermost reduction consiste à réduire les expressions les plus internes en premier. Dans certains cas, cela peut conduire à des évaluations inutiles ou à des boucles infinies.

```
loop = tail loop

fst = (1, loop)
= fst(1, tail loop)
= fst(1, tail (tail loop))
= ...
```

Dans cet exemple, l'innermost reduction conduit à une boucle infinie car `loop` est évalué de manière répétée.

#### ⚠ Warning

Cette stratégie ne se termine pas dans cet exemple, illustrant ainsi un inconvénient potentiel de l'innermost reduction.

#### 4.2.2 Outermost reduction

L'outermost reduction, en revanche, réduit les expressions les plus externes en premier.

```
fst = (1, loop)
= 1
```

Ici, l'outermost reduction permet d'obtenir la valeur `1` sans évaluer `loop`, évitant ainsi une boucle infinie.

### i Info

Cette stratégie permet d'éviter les évaluations inutiles et de travailler avec des structures de données infinies. N'importe quelle expression qui puisse avoir une séquence de réduction finie alors l'outermost reduction la trouvera.

## 4.3 Number of reductions

Le nombre de réductions nécessaires pour évaluer une expression peut varier en fonction de la stratégie de réduction utilisée. En général, l'outermost reduction peut nécessiter moins de réductions que l'innermost reduction, surtout dans le cas de structures de données infinies ou de calculs conditionnels.

### Innermost reduction:

```
square (3 + 4)
= square 7
= 7 * 7
= 49
```

3 opérations de réduction.

### Outermost reduction:

```
square (3 + 4)
= (3 + 4) * (3 + 4)
= 7 * (3 + 4)
= 7 * 7
= 49
```

4 opérations de réduction.

La où l'outermost reduction excelle, c'est dans le traitement des structures de données infinies. Cependant, elle nécessite souvent plus d'étape dans le cas où nous avons plusieurs paramètres à évaluer.

## 4.4 Sharing of expressions

Pour optimiser l'évaluation paresseuse, Haskell utilise une technique appelée « sharing » (partage) des expressions. Cela signifie que lorsqu'une expression est évaluée, son résultat est stocké et réutilisé chaque fois que cette expression est référencée à nouveau. Cela évite les évaluations redondantes et améliore les performances.

```
square (3 + 4)
= (. * .) (3 + 4)
  |____|_____^
= (. * .) 7
  |____|__^
= 49
```

Fig. 1. – Capture des slides du cours – Sharing of expressions in lazy evaluation

On appelle cette technique de réduction **lazy evaluation with sharing expressions** (nous utilisons une outermost reduction avec partage d'expressions). Grâce à cette technique, Haskell peut évaluer efficacement des expressions complexes tout en minimisant le nombre de réductions nécessaires.

## 4.5 Infinite lists

Grâce à la nouvelle stratégie d'évaluation paresseuse avec partage d'expressions, Haskell peut travailler avec des listes infinies de manière efficace. Par exemple, nous pouvons définir une liste infinie de 1 comme suit :

```
ones :: [Int]
ones = 1 : ones
```

En utilisant cette liste avec de l'**innermost reduction**, nous aurions:

```
head ones = head (1 : ones)
           = head (1 : 1 : ones)
           = head (1 : 1 : 1 : ones)
           = ...
```

Cela ne se termine jamais. Cependant, avec l'**outermost reduction with sharing expressions**, nous avons:

```
head ones = head (1 : ones)
           = 1
```

Ici, Haskell évalue uniquement la partie nécessaire de la liste infinie pour obtenir le premier élément, évitant ainsi une évaluation infinie.

## 4.6 Strict evaluation

En Haskell, la fonction `seq` permet de forcer l'évaluation stricte d'une expression. Cela signifie que l'expression est évaluée immédiatement, plutôt que d'être laissée en attente pour une évaluation paresseuse ultérieure.

La fonction `seq` prend deux arguments et retourne le deuxième. Cependant elle n'ignore pas le premier argument, il est réduit à une valeur avant de retourner le second argument.

Haskell propose un autre opérateur `$!` qui est une version stricte de l'application de fonction. Il force l'évaluation de son argument avant d'appliquer la fonction.

```
(!$) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Example

```
const :: a -> b -> a
const x _ = x

> const 42 undefined
42
> const 42 $ undefined
42
> const 42 $! undefined
*** Exception: undefined
```