PLP

Monads and more

18 November 2025

Table des matières

1 Functors 1
1.1 Functors class 2
1.2 Operateur `<$` 2
1.3 Implement Functor for Data Types 2
2 Applicative 2
2.1 `pure` and `<*>` 3
2.2 Functor Applicative 3
2.2.1 Example with Maybe 3
2.3 Laws of Applicative 3
3 Monads 3
3.1 `return` and `>>=` 4
3.2 Monad class 4
3.3 do Notation 4
3.4 Laws of Monads 4

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 :

```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 Operateur `<$`

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]`.

```haskell
> "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 :

```haskell
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 :

```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 :

```haskell
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)`