Table des matières

1. Type system

Haskell est un langage ayant les caractérisiques de typages fort:

- Typage fort et statique
- Types et classes de types
- Inférence de types
- Paramétrisation polymorphique

> 📝 – **Info**
> Les programmes Haskell peuvent être considéré comme `type-safe` (sécurité de type). Cela signifie que les erreurs de types sont détectées à la compilation et non à l'exécution, ce qui réduit le nombre d'erreurs potentielles lors de l'exécution du programme.

1.1. Typing expression

Chaque expression dans Haskell a un type, qui est déterminé lors de la compilation. Le type d'une expression peut être explicite (déclaré par le programmeur) ou implicite (inféré par le compilateur).

Dans le cas ou celle-ci est définie explicitement, on utilise la syntaxe suivante:

```
> :type 'c'
'c' :: Char
```

1.1.1. Listes

Dans le cas d'une liste nous aurions la syntace suivante:

```
> :type [1, 2, 3]
[1, 2, 3] :: [Int]
[False, False, True] :: [Bool]
```

1.1.2. Tuples

Pour les tuples, la syntaxe est la suivante:

```
> :type (1, 'a', True)
(1, 'a', True) :: (Int, Char, Bool)
```

1.1.3. Fonctions

Pour les fonctions, la syntaxe est la suivante:

```
even :: Int -> Bool
gcd :: Int -> Int -> Int // gcd prend deux Int et retourne un Int
```

1.1.3.1. Signature de type

La signature de type d'une fonction décrit les types de ses arguments et le type de sa valeur de retour. Par exemple, la signature de type de la fonction `even` est `Int -> Bool`, ce qui signifie qu'elle prend un argument de type `Int` et retourne une valeur de type `Bool`.

Nous écrirons la signature de type avant la définition de la fonction:

```
even :: Int -> Bool
even n = n `mod` 2 == 0
```

> 📝 – **Info**
> En Haskell, il est recommandé de toujours fournir une signature de type explicite pour les fonctions de haut niveau (top-level functions). Cela améliore la lisibilité du code et aide à prévenir les erreurs de type.

2. Polymorphisme

Haskell supporte deux types de polymorphisme: le polymorphisme paramétrique et le polymorphisme ad-hoc.

2.1. Polymorphisme paramétrique

Le polymorphisme paramétrique permet de définir des fonctions et des types qui peuvent fonctionner avec n'importe quel type. Par exemple, la fonction `length` peut être appliquée à une liste de n'importe quel type:

```haskell
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Ici, `a` est un type générique qui peut représenter n'importe quel type. La fonction `length` peut donc être utilisée avec des listes d'entiers, de caractères, de booléens, etc.

2.2. Polymorphisme ad-hoc

Le polymorphisme ad-hoc permet de définir des fonctions qui peuvent avoir des comportements différents en fonction des types de leurs arguments. Cela est réalisé à l'aide de classes de types. Par exemple, la classe de type `Eq` permet de définir l'égalité pour différents types:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Les types qui sont instances de la classe `Eq` doivent implémenter les fonctions `(==)` et `(/=)`.

2.3. Fonction partiel

Une fonction partielle est une fonction qui n'est pas définie pour tous les arguments possibles. En Haskell, cela peut se produire lorsqu'une fonction utilise des motifs (patterns) qui ne couvrent pas tous les cas possibles. Par exemple, la fonction suivante est partielle car elle n'est pas définie pour une liste vide:

```haskell
head :: [a] -> a
head (x:_) = x
```

Si on appelle `head []`, cela entraînera une erreur d'exécution.