PLP

Interactive Programming

15 November 2025

Table des matières

1 Interactive Programming

La programmation interactive en Haskell permet de créer des applications qui réagissent aux actions de l'utilisateur en temps réel.

1.1 Purity

Haskell est un langage de programmation purement fonctionnel, ce qui signifie que les fonctions n'ont pas d'effets secondaires. Cela facilite la gestion de l'état dans les applications interactives, car chaque état peut être représenté comme une valeur immuable.

L'introduction de programmation interactive en Haskell (par exemple grâce à IO) impose une distinction claire entre les fonctions pures et les fonctions qui interagissent avec le monde extérieur. C'est dans ce contexte que l'on parle souvent de fonctions pures et de fonctions impures. Les fonctions impures sont celles qui effectuent des opérations d'entrée/sortie (I/O) et peuvent avoir des effets secondaires.

Ces fonctions sont donc impactées par l'état du monde extérieur, ce qui les rend moins prévisibles que les fonctions pures. Cependant, Haskell utilise un système de types pour gérer ces effets secondaires de manière contrôlée, en encapsulant les opérations I/O dans le type `IO`.

1.2 Side effects

De manière générale, plusieurs actions peuvent avoir des effects secondaires:

- Afficher du texte à l'écran
- Générer un nombre aléatoire
- Communiquer au travers du réseau
- Query une base de données
- etc…

Intuitivement, ces actions modifient l'état du monde extérieur, et donc le résultat de ces actions peut varier en fonction de cet état.

2 Input and Output

Le système d'entrée/sortie (I/O) en Haskell est conçu pour gérer les interactions avec le monde extérieur tout en préservant la pureté des fonctions. Les opérations I/O sont encapsulées dans le type `IO`, ce qui permet de séparer clairement les fonctions pures des fonctions qui effectuent des opérations d'entrée/sortie.

2.1 Type `IO`

Le type `IO a` représente une action qui, lorsqu'elle est exécutée, produit une valeur de type `a` tout en pouvant avoir des effets secondaires. Par exemple, une action de type `IO String` pourrait lire une ligne de texte depuis l'entrée standard et retourner cette ligne sous forme de chaîne de caractères.

Voici un exemple simple d'une action I/O en Haskell qui lit une ligne de texte depuis l'entrée standard et l'affiche à l'écran:

```haskell
main :: IO ()
main = do
    putStrLn "Entrez une ligne de texte:"
    input <- getLine
    putStrLn ("Vous avez entré: " ++ input)
```

> **i Info**
>
> Dans cet exemple, `putStrLn` est une fonction qui affiche une chaîne de caractères à l'écran, et `getLine` est une fonction qui lit une ligne de texte depuis l'entrée standard. L'utilisation de `do` permet de séquencer les actions I/O de manière lisible.

2.2 Composing I/O Actions

L'opérateur de composition `>>>` permet de chaîner plusieurs actions I/O ensemble. Par exemple:

```haskell
putAB = putChar 'A' >>> putChar 'B'
```

Cette action affichera les caractères “A” et “B” à l'écran lorsqu'elle sera exécutée.

Nous pouvons appliquer un pattern similaire pour afficher une séquence de caractères (string):

```haskell
putStr :: String -> IO ()
putStr []       = return ()
putStr (c : cs) = putChar c >>> putStr cs
```

> **i Info**
>
> Il existe aussi `putStrLn`, qui ajoute automatiquement un saut de ligne à la fin de la chaîne affichée.

2.3 Binding I/O Results

L'opérateur de binding `>>=` permet de capturer le résultat d'une action I/O et de l'utiliser dans une autre action. Par exemple:

```haskell
getAndPrint :: IO ()
getAndPrint = getLine >>= \input -> putStrLn ("Vous avez entré: " ++ input)
```

Dans cet exemple, le résultat de `getLine` est capturé dans la variable `input`, qui est ensuite utilisée dans l'action `putStrLn`.

Cela permet d'écrire du code plus concis et lisible en évitant l'utilisation explicite de `do` notation.

Malheuresement cet opérateur bind ne peut pas être utilisé avec des fonctions pures. Par exemple, le code suivant ne fonctionnera pas:

```
getUpper = getChar >>= \c -> return (toUpper c)

error: Couldn't match expected type 'IO Char'
                  with actual type 'Char'
```

Pour que cela fonctionne, nous devons encapsuler le résultat de `toUpper c` dans une action I/O en utilisant `return` :

```
getUpper :: IO Char
getUpper = getChar >>= \c -> return (toUpper c)
```

> **i Info**
>
> Une fois qu'un calcul entre dans le contexte I/O, il ne peut plus en sortir.

2.4 Basic I/O Functions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
(>>)   :: IO a -> IO b -> IO b

getChar :: IO Char
getLine :: IO String

putChar :: Char -> IO ()
putStr  :: String -> IO ()
putStrLn :: String -> IO ()

print :: Show a => a -> IO ()
```

3 Do Blocks

La notation `do` en Haskell est une syntaxe spéciale qui facilite l'écriture de séquences d'actions I/O. Elle permet de structurer le code de manière plus lisible en utilisant une syntaxe impérative.

Cela permet d'éviter l'utilisation explicite des opérateurs `>>=` et `>>`, rendant le code plus facile à comprendre.

Exemple:

```haskell
getLine :: IO String
getLine = do c <- getChar
    if c == '\n' then return ""
    else do line <- getLine
        return (c:line)
```

- Les actions sont éxécutées dans l'ordre où elles apparaissent et l'indentation est significative.
- Similaire aux générateurs, la syntaxe `<-` permet de capturer le résultat d'une action I/O dans une variable.

3.1 Let

La syntaxe `let` peut également être utilisée dans les blocs `do` pour définir des variables locales sans effectuer d'action I/O. Par exemple:

```haskell
main :: IO ()
main = do
    putStrLn "Calculons la somme de deux nombres."
    putStr "Entrez le premier nombre: "
    input1 <- getLine
    putStr "Entrez le deuxième nombre: "
    input2 <- getLine
    let num1 = read input1 :: Int
        num2 = read input2 :: Int
        sum = num1 + num2
    putStrLn ("La somme est: " ++ show sum)
```

> **i Info**
>
> Le bloc `do` est un sucre syntaxique qui rend le code I/O plus lisible et maintenable. Dans les décors, le compilateur traduit automatiquement la notation `do` en utilisant les opérateurs `>>=` et `>>`.

4 Control flow actions

4.1 Sequence

La fonction standard `sequence_` permet de transformer une liste d'actions I/O en une action I/O qui produit une liste de résultats. Par exemple:

```
sequence_ :: [IO a] -> IO ()
sequence_ []       = return ()
sequence_ (x : xs) = x >> sequence_ xs
```

Cette fonction exécute chaque action I/O dans la liste dans l'ordre, en ignorant les résultats.

4.2 MapM

La fonction standard `mapM_` permet d'appliquer une fonction qui retourne une action I/O à chaque élément d'une liste, et de combiner les résultats en une seule action I/O. Par exemple:

```
mapM_ :: (a -> IO b) -> [a] -> IO ()
mapM_ f []       = return ()
mapM_ f (x : xs) = f x >> mapM_ f xs
```

Cette fonction est utile pour effectuer des opérations I/O sur chaque élément d'une liste de manière concise.

4.3 ForM

La fonction standard `forM_` est similaire à `mapM_`, mais avec les arguments inversés. Elle permet d'appliquer une fonction qui retourne une action I/O à chaque élément d'une liste, en utilisant la liste comme premier argument. Par exemple:

```
forM_ :: [a] -> (a -> IO b) -> IO ()
forM_ []       = return ()
forM_ (x : xs) f = f x >> forM_ xs f
```

Cette fonction est souvent utilisée pour itérer sur une liste tout en effectuant des opérations I/O sur chaque élément.

4.4 When

La fonction standard `when` permet d'exécuter une action I/O conditionnellement, en fonction d'une valeur booléenne. Par exemple:

```
when :: Bool -> IO () -> IO ()
when True action  = action
when False action = return ()
```

Cette fonction est utile pour exécuter des actions I/O uniquement lorsque certaines conditions sont remplies.

4.5 Unless

La fonction standard `unless` est l'inverse de `when`. Elle permet d'exécuter une action I/O uniquement lorsque la condition est fausse. Par exemple:

```
unless :: Bool -> IO () -> IO ()
unless False action = action
unless True action  = return ()
```

Cette fonction est utile pour exécuter des actions I/O lorsque certaines conditions ne sont pas remplies.

5 Modules

En Haskell, les programmes sont une collecteion de modules. Chaque module est un fichier qui contient des définitions de fonctions, de types de données et d'autres entités. Les modules permettent d'organiser le code de manière logique et de réutiliser des fonctionnalités à travers différents programmes.

Pour déclarer un module, on utilise la syntaxe suivante au début du fichier:

```
module MyModule where
-- Définitions de fonctions et de types de données
```

> **i Info**
>
> Les définitions peuvent apparaître dans n'importe quel ordre à l'intérieur d'un module, car Haskell utilise un système de résolution des dépendances pour déterminer l'ordre d'évaluation.

5.1 Importing Modules

Pour utiliser les fonctionnalités d'un module dans un autre module, on utilise la directive `import`. Par exemple:

```
import Data.Char

isNatural :: Char -> Bool
isNatural str = all isDigit str
```

Cela importe toutes les fonctions et types de données définis dans le module `Data.Char`, ce qui permet d'utiliser la fonction `isDigit` pour vérifier si un caractère est un chiffre.

5.2 Selected Imports

Il est possible d'importer uniquement certaines fonctions ou types de données d'un module en utilisant la syntaxe suivante:

```
import Data.Char (isDigit, toUpper)
```

Cela importe uniquement les fonctions `isDigit` et `toUpper` du module `Data.Char`, ce qui peut aider à éviter les conflits de noms et à réduire la taille du namespace.

5.3 Hiding Imports

Il est également possible d'importer un module tout en excluant certaines fonctions ou types de données en utilisant la syntaxe suivante:

```
import Data.List hiding (nub)
```

Cela importe toutes les fonctions et types de données du module `Data.List`, sauf la fonction `nub`.

5.4 Qualified Imports

Pour éviter les conflits de noms entre différents modules, on peut importer un module de manière qualifiée en utilisant la syntaxe suivante:

```
import qualified Data.Map as Map

MyMap = Map.fromList [(1, "one"), (2, "two")]
```

Cela permet d'accéder aux fonctions et types de données du module `Data.Map` en utilisant le préfixe `Map.`, ce qui évite les conflits de noms avec d'autres modules.

5.5 Exporting Modules

Par défaut, toutes les fonctions et types de données définis dans un module sont exportés et peuvent être utilisés par d'autres modules. Cependant, il est possible de contrôler ce qui est exporté en utilisant la syntaxe suivante:

```
module MyModule (myFunction, MyType) where
myFunction x = x
```

6 Programs

Un programme Haskell est constitué d'un ou plusieurs modules, avec un module principal qui contient la fonction `main`. La fonction `main` est le point d'entrée du programme et doit avoir le type `IO ()`.

Voici un exemple simple d'un programme Haskell qui utilise la programmation interactive:

```haskell
-- hello.hs

main :: IO ()
main = putStrLn "Hello, World!"
```

Pour compiler et exécuter ce programme, on peut utiliser les commandes suivantes dans le terminal:

```
ghc hello.hs
./hello
```

Cela va créer 3 fichiers:

1. `hello.hi` : le fichier d'interface
2. `hello.o` : le fichier objet
3. `hello` : le fichier exécutable

Pour renomer le fichier exécutable, on peut utiliser l'option `-o` lors de la compilation:

```
ghc -o mon_programme hello.hs
./mon_programme
```

6.1 getArgs

La fonction `getArgs` du module `System.Environment` permet de récupérer les arguments passés au programme lors de son exécution. Elle retourne une liste de chaînes de caractères représentant les arguments.

```haskell
import System.Environment (getArgs)

main :: IO ()
main = do
  args <- getArgs
  putStrLn ("Arguments: " ++ show args)
```

> ⚠ **Warning**
>
> Il est important de tester les arguments reçus pour éviter les erreurs d'exécution.

6.2 getProgName

La fonction `getProgName` du module `System.Environment` permet de récupérer le nom du programme en cours d'exécution. Elle retourne une chaîne de caractères représentant le nom du programme.

```haskell
import System.Environment (getProgName)

main :: IO ()
main = do
  progName <- getProgName
  putStrLn ("Nom du programme: " ++ progName)
```

7 Files

Un fichier est une ressource de stockage persistant qui permet de sauvegarder des données sur le disque dur. En Haskell, les fichiers sont manipulés à l'aide du module `System.IO`, qui fournit des fonctions pour lire et écrire des fichiers.

Il existe deux types de fichiers:

- Fichiers texte: contiennent des données sous forme de texte lisible par l'homme
- Fichiers binaires: contiennent des données sous forme binaire, non lisible par l'homme

7.1 Dealing with files

Pour manipuler des fichiers en Haskell, on utilise les fonctions du module `System.IO`.

Voici un exemple simple de lecture et d'écriture dans un fichier texte:

```haskell
import System.IO

main :: IO ()
main = do
    -- Écriture dans un fichier
    writeFile "example.txt" "Hello, World!\nThis is a test file."
    -- Lecture depuis un fichier
    contents <- readFile "example.txt"
    putStrLn "Contenu du fichier:"
    putStrLn contents
```

Il est possibles de gérer les lignes d'un fichier en utilisant les fonctions `lines` et `unlines`:

```haskell
import System.IO

main :: IO ()
main = do
    -- Écriture dans un fichier
    let linesToWrite = ["Line 1", "Line 2", "Line 3"]
    writeFile "example.txt" (unlines linesToWrite)
    -- Lecture depuis un fichier
    contents <- readFile "example.txt"
    let fileLines = lines contents
    putStrLn "Lignes du fichier:"
    mapM_ putStrLn fileLines
```