1 GHC

2 DEBUGGERS

3 CABAL

4 NOMMAGE DES IDENTIFIANTS

Contraintes

En Haskell, la casse du premier caractère d'un identifiant a une importance.

Type d'identifiant	Initiale haut de casse	Initiale bas de casse
Variable		■
Fonction		■
Type	■	
Constructeur	■	

Conventions

5 TYPES DE DONNÉES

Types élémentaires

Haskell fournit un grand nombre de types élémentaires, dont les plus importants sont résumés dans ce tableau :

Type	Description	Bits	Bounded	Floating	Fractional	Integral	Num	Real
`Double`	Virgule flottante, double précision		■	■			■	■
`Float`	Virgule flottante, simple précision		■	■			■	■
`Int`	Entier signé à précision fixe, intervalle minimum $[-2^{29}; 2^{29} - 1]$							
`Int8`	Entier signé de 8 bits							
`Int16`	Entier signé de 16 bits							
`Int32`	Entier signé de 32 bits							
`Int64`	Entier signé de 64 bits							
`Integer`	Arbitrary precision signed integer							
`Rational` ou `Ratio a`	Nombre rationnel de précision arbitraire							
`Word8`	Entier non signé de 8 bits							
`Word16`	Entier non signé de 16 bits							
`Word32`	Entier non signé de 32 bits							
`Word64`	Entier non signé de 64 bits							

Types numériques essentiels, d'après O'Sullivan et al. 2008

Types composites

Haskell connaît deux types composites : les **tableaux** et les **n-uplets**. Le type tableau se note `[a]`, un n-uplet se note `(a, b)`, `(a, b, c)`, *etc.*. Un n-uplet a au moins deux éléments, à l'exception de `()`, le n-uplet vide qui indique qu'une fonction ne renvoie pas de valeur.

Les tableaux sont typés : un tableau ne peut contenir des éléments que d'un type unique, pleinement paramétrisé. Par exemple, un tableau de type `[(Integer, Char)]` (tableau de couples `Integer`, `Char`) ne peut pas contenir un autre type de n-uplet, par exemple `(Integer, String)`.

6 DÉFINITION DE TYPES

Types personnalisés

On crée un type avec le mot-clé `data`. Au plus simple, un type réunit plusieurs variables membres :

```
ghci> data MonType = MonType String Integer
ghci> let a = MonType "abc" 123
```

(La première occurence de MonType est le nom du type, la seconde est le nom du constructeur)

■ **Accesseurs explicites (*record syntax*)** Un accesseur explicite définit automatiquement une fonction qui prend un objet du nouveau type en paramètre et renvoie la valeur

```
data Book = Book {
    bookISBN  :: Int,          -- bookISBN :: Book -> Int
    bookAuthors  :: [String], -- bookAuthors :: Book -> [String]
    bookTitle  :: String       -- bookTitle :: Book -> String
    }
```

■ **Types algébriques** Un type algébrique présente une alternative, en offrant plusieurs constructeurs. Le plus répandu est `Maybe` :

```
data Maybe a = Nothing | Just a
```

Tous les types définis avec `data` sont algébriques, même s'ils n'offrent qu'un seul constructeur (O'Sullivan et al. 2008).

■ **Types récursifs** Le type liste natif est défini récursivement. On peut le réimplémenter comme suit :

```
data List a = Empty | Cons a (MyList a)
```

Synonymes de types

`type` crée un synonyme d'un type existant. Les synonymes et le type auquel ils renvoient sont interchangeables.

```
type ObjectId = Int16
```

Les synonymes créés avec `type` peuvent servir :

- À clarifier le sens des champs dans les types personnalisés sans accesseurs (`type ISBN = Int` pour un type Book, par exemple).
- Comme notation abrégée pour des types complexes fréquemment utilisés.

```
type Authors = [String]
type Title = String
type ISBN = Int
type Year = Int
data Book2 = Authors Title Year ISBN
```

Synonymes « forts »

- **Maybe**
- **Either**

7 CLASSES DE TYPE (*TYPE CLASSES*)

Les classes de type ne sont pas des classes au sens que ce terme possède en POO. Elles sont plus proches de ce qu'on nomme des interfaces : elles décrivent des fonctions pour lesquelles un type qui appartient à la classe fournit une implémentation.

```
class EvalToBool a where
    toBool  :: a -> Bool

instance EvalToBool Integer where
    toBool x = x /= 0
```

8 FONCTIONS ET VARIABLES

Les variables sont immutables, ce qui, combiné avec les principe de transparence référentielle (→11), d'évaluation paresseuse (→12) et d'application partielle (→17), permet de voir qu'il n'existe aucune différence stricte entre une fonction et une variable.

```
a = 3.14
c = a ** 2
b x = x * a
```

Signature de type

■ **Pour une variable,** la signature de type est très simple, et peut en général être omise, l'inférence automatique suffisant en général.

```
a  :: Int
a = 3
```

■ **Pour une fonction,** la signature a la forme `f  :: a -> b` , ce qui signifie que la fonction prend un paramètre de type `a` et renvoie une valeur de type `b`.

```
f  :: [Integer] -> Integer
```

f prend un tableau d'entiers et renvoie un entier.

```
g  :: Integer -> Integer -> Integer
```

g prend deux entiers et renvoie un entier (plus précisément, g prend un entier et renvoie une fonction qui prend un entier et renvoie un entier (→17)).

```
h  :: [a] -> Integer
```

h prend un tableau d'un type quelconque a (→15) et renvoie un entier

```
i  :: [a] -> a
```

i prend un tableau d'un type quelconque a et renvoie un a

```
j  :: (Eq a) => [a] -> a
```

j prend un tableau d'un type quelconque a instance de la classe de type (→7) `Eq` et renvoie une valeur de type `a`

Portée des variables et variables locales

Fonctions préfixes et infixes

Fonctions pures et impures

9 FILTRAGE PAR MOTIF ET GARDES

Gardes

10 STRUCTURES CONDITIONNELLES

`if ... then ... else`

Renvoie une valeur, par conséquent, le type de l'expression de `then` et de `else` doit être le même.

On peut souvent éviter d'utiliser un if en le remplaçant par un garde (→9.1)

`case`

11 TRANSPARENCE RÉFÉRENTIELLE

La transparence référentielle est une propriété des expressions pures qui fait que toute expression peut être remplacée par son résultat sans modifier le comportement du programme.

12 ÉVALUATION PARESSEUSE

```
let a = [1..] -- a est la liste de l'ensemble des entiers positifs
let b = map ((^^) 2) a
```

L'évaluation paresseuse a un prix, qui est une plus grande consommation de mémoire : au lieu d'évaluer `2 + 2`, Haskell stocke un *thunk*, c'est à dire en gros un calcul différé. Mais sur les gros traitements récursifs, l'accumulation de thunks peut entrainer rapidement un débordement de mémoire. La commande seq force l'évaluation et permet d'éviter un débordement de mémoire.

L'évaluation paresseuse obéit à des règles strictes

Il est possible de déterminer avec précision *si* une expression va être évaluée, et si oui *quand*. C'est parce qu'il est garanti qu'une expression dont le résultat n'est pas utilisé ne sera pas évaluée qu'on peut, par exemple, programmer des opérateurs logiques court-circuitants directement en Haskell, ou manipuler des suites infinies.

seq

13 **LAMBDAS**

14 OPÉRATIONS SUR DES LISTES

15 POLYMORPHISME

Polymorphisme paramétrique

N'importe quelle fonction (→8) ou type (→6) peut accepter des paramètres d'un type non défini. Sa signature remplace dans ce cas le nom d'un type par un paramètre de type, qui commence par une minuscule (→4).

■ **Types polymorphiques** Le type `Maybe`, qui représente une valeur possible, est un exemple de type polymorphique. Il a deux constructeurs : `Nothing` et `Just a`. `Nothing` ne prend pas de paramètre, et représente l'absence de valeur. `Just a` prend un paramètre du type quelconque `a`.

```
ghci> :type Just 3
Just 3  :: Num a => Maybe a
ghci> :type Just "Une chaîne"
Just "Une chaîne"  :: Maybe [Char]
ghci> :type Nothing
Nothing  :: Maybe a
```

■ **Fonctions polymorphiques** Une fonction peut accepter, ou renvoyer, des types non-définis.

```
third  :: [a] -> Maybe a
third (_ :_ :x :_) = Just x
third _ = Nothing
```

« Théorèmes gratuits »

Comme une fonction polymorphique n'a pas accès au type réel de son paramètre, on peut déduire (au sens strict) ce qu'elle peut faire à sa seule signature.

La fonction `head  :: [a] -> a` n'a pas accès au type `a`, et par conséquent ne peut ni construire un nouvel `a`, ni modifier un des `a` du tableau qu'elle reçoit : elle doit en renvoyer un tel quel. On peut donc déduire que `head b `elem` b`.

La fonction `fst  :: (a, b) -> a` ne peut *rien* faire d'autre que renvoyer le premier élément de la paire qui lui est passée, et ignorer le second.

Wadler (1989) explicite le soubassement logico-mathématique de ce principe et montre des applications à des cas beaucoup plus complexes que ces quelque exemples.

16 **RÉCURSIVITÉ**

Récursivité en queue

Folds

17 APPLICATION PARTIELLE ET *CURRYING*

Une fonction, quel que soit le nombre de paramètres avec lequel elle a été déclarée, ne prend qu'un seul paramètre et renvoie une autre fonction. Le type de `+`, par exemple, est : `Num a => Num a -> Num a -> Num a`, ce qui signifie que `+` prend un premier paramètre d'un type de type `Num`

18 MODULES

Haskell dispose d'un mécanisme d'importation de modules.

Écrire un module

Un module a le même nom que le fichier .hs qui le contient, et ce nom commence par une majuscule (→4). La déclaration de module a la syntaxe suivante :

```
-- MyModule.hs
module Mod
    (
      x,
      y,
      z
    ) where
-- code
```

Cette déclaration exporte les identifiants x, y et z du code qui la suit. On exporterait la totalité des noms en enlevant la parenthèse, et aucun en la laissant vide.

Importation de modules

```
-- Commande                          Importé
import Mod                        -- x, y, z, Mod.x, Mod.y, Mod.z
import Mod ()                     -- (Nothing!)
import Mod (x,y)                  -- x, y, Mod.x, Mod.y
import qualified Mod              -- Mod.x, Mod.y, Mod.z
import qualified Mod (x,y)        -- Mod.x, Mod.y
import Mod hiding (x,y)           -- z, Mod.z
import qualified Mod hiding (x,y) -- Mod.z
import Mod as Foo                 -- x, y, z, Foo.x, Foo.y, Foo.z
import Mod as Foo (x,y)           -- x, y, Foo.x, Foo.y
import qualified Mod as Foo       -- Foo.x, Foo.y, Foo.z
import qualified Mod as Foo (x,y) -- Foo.x, Foo.y
```

D'après Hudak et al. 2000

19 COMPOSITION

20 MONOÏDES

21 MONADES

Le wiki Haskell décrit les monades comme suit :

> Monads in Haskell can be thought of as composable computation descriptions. The essence of monad is thus separation of composition timeline from the composed computation's execution timeline, as well as the ability of computation to implicitly carry extra data, as pertaining to the computation itself, in addition to its one (hence the name) output, that it will produce when run (or queried, or called upon). This lends monads to supplementing pure calculations with features like I/O, common environment or state, etc.

Références

Hudak, Paul, John Peterson et Joseph Fasel (2000). *A Gentle Introduction to Haskell. Version 98*.
URL : https://www.haskell.org/tutorial/index.html.

O'Sullivan, Bryan, Don Stewart et John Goerzen (2008). *Real World Haskell*. O'Reilly.
URL : http://book.realworldhaskell.org/.

Wadler, Philip (1989). *Theorems for Free !*
URL : http://www.cs.sfu.ca/CourseCentral/831/burton/Notes/July14/free.pdf.