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 »

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

Haskell n'a pas de notion de variable au sens qu'a ce terme en programmation procédurale. Il est possible d'assigner une expression ou une valeur à un nom, avec la syntaxe `nom = expression`, mais `nom` est immuable, et est donc plus proche d'une constante (c'est une variable au sens mathématique du terme).

En combinant ceci avec les principe de transparence référentielle 14, d'évaluation paresseuse 15 et d'application partielle 11, on voit facilement qu'il n'existe aucune différence stricte entre une fonction et une variable, donc qu'il n'existe pas de variables. Par exemple :

```
a = 3 * 2
times3 x = 3 * x
b = times3 2
c = 6
```

Ici, `times3` est une fonction, `a`, `b` et `c` des variables. Dans la mesure où la valeur d'aucune n'est évaluée tant qu'elle n'est pas utilisée, la variable `a` a strictement la même valeur que `b`, qui n'est pas 6, mais le calcul différé (le *thunk*) `3 * 2`.

> ⚠ Cette identité n'est vraie que des fonctions pures. Les fonctions impures, comme par exemple `getLine`, peuvent évidemment renvoyer un résultat différent à chaque invocation. Voir **??**: **??.**

La suite de cette fiche ne s'intéresse donc qu'aux fonctions, puisque les « variables » n'en sont qu'un cas particulier.

SIGNATURE DE TYPE

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`.

Les fonctions d'ordre supérieur utilisent les parenthèses pour indiquer qu'elles prennent une autre fonction en paramètre. Par exemple, le type `map :: (a -> b) -> [a] -> [b]` se lit : `map` prend comme premier paramètre une fonction quelconque `x :: a -> b`.

Une variable ou une fonction sans paramètres a pour type `nom :: Type`.

FONCTIONS PRÉFIXES ET INFIXES

Une fonction est dite préfixe si son nom est placé avant ses arguments, et infixe si son nom est placé entre ses arguments. `map` est une fonction préfixe, `+` est infixe. La distinction est syntaxique, et se fait au niveau des caractères qui constituent le nom de la fonction.

■ **Une fonction infixe** a un nom composé uniquement de symboles non alphanumériques : `+`, `*` ou `>>=` sont infixes.

On peut utiliser une fonction infixe comme préfixe en entourant son nom de parenthèses : `(+) 1 1`.

■ **Une fonction préfixe** a un nom composé de caractères alphanumériques. `map`, `elem` ou `foldr` sont préfixes.

On peut utiliser une fonction préfixe comme infixe en entourant son nom de *backticks* : `1 `elem` [1..10]`.

9 DÉFINITION DE FONCTIONS

Une fonction se définit de la façon suivante :

```
add :: a -> b  -- Signature de type, généralement optionnel.
add x = expr x
```

Une fonction infixe se définit en entourant son nom de parenthèses, comme pour l'utiliser en préfixe :

```
(+*) a b = a + b + a * b
```

PARAMÈTRES

■ **Déconstruction de types composites.** « Déconstruire » un argument d'une fonction permet d'obtenir directement les arguments du constructeur. Par exemple, la fonction suivante déconstruit un constructeur de paire (tuple de deux éléments) pour en renvoyer le premier :

```
toggle :: (a, b) -> a
toggle (x, y) = (y, x)
```

Un paramètre non utilisé peut être remplacé par un `_` :

```
duplFirst :: (a, b) -> (a, a)
duplFirst (x, _) = (x, x)
```

On n'a pas besoin du second membre de la paire : on la décompose donc en évitant de nommer cet élément.

De la même façon, si le paramètre est un `Maybe`, on peut récupérer directement sa valeur en déconstruisant `Just` :

```
double :: Maybe Int -> Int
double (Just x) = x * 2
```

■ **Motifs nommés.** On peut avoir besoin de déconstruire un paramètre selon un motif en conservant le paramètre entier. Les motifs nommés permettent d'éviter des suites déconstruction-reconstruction redondantes.

La fonction `suffixes` (O'Sullivan et al. 2008, p. 103) renvoie tous les suffixes d'une liste. Elle peut s'écrire :

```
suffixes :: [a] -> [[a]]
suffixes xs@(_:xs') = xs : suffixes xs'
suffixes _ = []
```

FONCTIONS LOCALES

On peut définir des fonctions dont la visibilité est limitée à une fonction. C'est utile pour définir des constantes, ou fournir des fonctions utilitaires qui n'ont pas besoin d'être disponibles au niveau du module. Haskell propose deux syntaxes : `let`, qui place les variables locales *avant* le code de la fonction, et `where`, qui les positionne *après*.

```
circLet :: Fractional a => a -> a
circLet radius = let pi   = 3.14
                     diam = 2 * radius
                 in pi * diam
```

```
circWhere :: Fractional a => a -> a
circWhere radius = pi * diam
    where pi   = 3.141592653589793
          diam = 2 * radius
```

- Le choix de l'une ou de l'autre syntaxe est une question de lisibilité.
- On peut les imbriquer : une fonction définie dans une clause `let`/`where` peut à son tour définir des fonctions locales, etc.
- La visiblité des fonctions locales est limitée à la définition englobante.

FIXITÉ (PRÉCÉDENCE ET ASSOCIATIVITÉ)

L'associativité et la précédence sont collectivement nommées « fixité ». La fixité d'une fonction infixe (et de n'importe quelle fonction préfixe dans sa forme infixe, comme `` `elem` ``) est fixée par une déclaration `infixl` (associatif à gauche), `infixr` (associatif à droite) ou `infix` (non-associatif), suivie de l'ordre de précédence compris entre 0 et 9 et du nom de la fonction :

```
(+*) :: Num a => a -> a -> a
infixl 9 +*
(+*) a b = a + b + a * b
```

Il est possible de définir la fixité d'une fonction locale, directement dans la clause `let` ou `where` où elle est définie.

10 FILTRAGE PAR MOTIF ET GARDES

Le filtrage par motifs et l'emploi de gardes permettent de proposer différentes implémentations d'une même fonction selon les paramètres qui y sont passés, de façon similaire à l'emploi de cas en notation mathématique :

$$f(x) = \begin{cases} f(x-1) + x & \text{si } x > 0 \\ 1 & \text{sinon} \end{cases}$$

Le filtrage par motifs permet de choisir une implémentation selon le type et dans une certaine mesure la valeur des paramètres, les gardes selon une expression arbitraire.

Le filtrage par motif et les gardes permettent de définir plusieurs cas qui se recouvrent. Par exemple, une fonction peut fournir une implémentation pour n'importe quelle liste, et une autre pour n'importe quelle liste *non vide*. Haskell utilise toujours la première implémentation qui s'applique aux paramètres, dans l'ordre de déclaration : il faut donc déclarer les moins générales en premier.

FILTRAGE PAR MOTIFS

Le filtrage par motifs permet de filtrer selon un constructeur ou selon une valeur arbitraire.

■ **Par constructeur.** Le filtrage par constructeurs permet de sélectionner quel constructeur d'un type algébrique 6.1.2 correspond à quelle implémentation.

```
maybeIntToStr :: Maybe Int -> String
maybeIntToStr (Just a) = show a
maybeIntToStr Nothing  = "NaN"
```

```
mySum :: (Num a) => [a] -> a
mySum (x:xs) = x + mySum xs
mySum []     = 0 -- [] est le constructeur de liste vide,
                 -- pas une valeur littérale !
```

L'implémentation de `mySum` n'est pas optimale : il vaudrait mieux utiliser `foldr` ??.

■ **Par valeur littérale.**

■ **Paramètres ignorés.** Certaines implémentations d'une fonction peuvent ne pas faire usage de tous les paramètres. On ignore un paramètre dans la définition avec le symbole `_` :

```
timesX :: (Num a) => a -> [a] -> [a]
timesX a (x:xs) = a * x:timesX a xs
timesX _ []     = []
```

Listing 1 : La dernière ligne définit le comportement de `timesX` face à une liste vide. Dans ce cas, le multiplicateur n'est pas utilisé.

GARDES

Un garde est une expression de type `Bool`. Si l'expression s'évalue à `True`, l'implémentation qui suit est utilisée. Les gardes permettent d'éviter les constructions conditionnelles avec `if` ⁇. Leur syntaxe est :

```
func args | garde = impl
```

Par exemple, une fonction qui détermine si un nombre est pair, qui s'implémenterait naïvement sous la forme `isEven x = if x `mod` 2 == 0 then True else False` peut s'écrire plus lisiblement :

```
isEven x | x `mod` 2 == 0 = True
isEven _ = False
```

ou encore :

```
isEven x | x `mod` 2 == 0 = True
         | otherwise = False
```

`otherwise` est une constante définie dans le Prélude. Sa valeur est simplement `True`.

La syntaxe `f a | garde1, garde2, etc = impl` est une nouveauté de Haskell 2010.

11 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`

12 **LAMBDAS**

13 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 10.2

CASE

14 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.

15 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

16 OPÉRATIONS SUR DES LISTES

17 POLYMORPHISME

POLYMORPHISME PARAMÉTRIQUE

N'importe quelle fonction ❽ ou type ❻ 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 ❹.

■ **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.

POLYMORPHISME *AD HOC*

18 **RÉCURSIVITÉ**

RÉCURSIVITÉ EN QUEUE

FOLDS

19 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 ❹. 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

20 COMPOSITION

21 MONOÏDES

22 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.