Lambda-calcul et programmation fonctionnelle

Tiago de Lima

22 janvier 2025
(version 0.7.4)

Table des matières

1 Notions élémentaires **1**
- 1.1 Modes de programmation . . . 1
- 1.2 Haskell . . . 3
- 1.3 Notions élémentaires . . . 4
- 1.4 Exercices . . . 8

2 Récursivité **9**
- 2.1 Définition de récursivité . . . 9
- 2.2 Création d'une définition récursive . . . 10
- 2.3 Correction . . . 11
- 2.4 Terminaison . . . 12
- 2.5 Récursivité terminale . . . 14
- 2.6 Exercices . . . 16

3 Types **19**
- 3.1 La notion de type . . . 19
- 3.2 Types de base . . . 20
- 3.3 Polymorphisme . . . 24
- 3.4 Fonctionnelles et fonctions d'ordre supérieur . . . 25
- 3.5 Classes de types . . . 29
- 3.6 Inférence de type . . . 33
- 3.7 Exercices . . . 36

4 Types structurés **39**
- 4.1 Modules . . . 39
- 4.2 Définition de types structurés . . . 40
- 4.3 Catégories de types structurés . . . 43
- 4.4 Exemple . . . 47
- 4.5 Exercices . . . 48

5 Schémas de programme **49**
- 5.1 Introduction . . . 49
- 5.2 Schéma réduction . . . 52
- 5.3 Schémas plier . . . 53
- 5.4 Schéma map . . . 55

5.5 Schéma filtrer 56
5.6 Exercices 57

6 Classes de type **59**
6.1 Polymorphisme paramétrique 59
6.2 Classes de types 61
6.3 Instances 61
6.4 Exemple 63
6.5 Héritage 64

7 Monades, entrées et sorties **67**
7.1 Introduction 67
7.2 Foncteurs 67
7.3 Applicatifs 69
7.4 Monades 70
7.5 Entrées et sorties 76
7.6 Exercices 79

8 Lambda-calcul **81**
8.1 Introduction 81
8.2 Syntaxe 81
8.3 Signification des lambda-expressions 82
8.4 Réduction 83
8.5 Stratégies de réduction 86
8.6 Exercices 89

Chapitre 1

Notions élémentaires

1.1 Modes de programmation

Un **mode de programmation** est une façon de construire un programme résolvant un problème donné.

Les principaux modes de programmation sont :

— Impératif
— Orienté objet
— Logique
— Fonctionnel

La plus part de langages de programmation récents sont **multi-paradigme** (multi-mode).

Exemples :

— Python et C++ sont impératifs, orientés objet et permettent des fonctions anonymes.
— Java est orienté objet et permet des fonctions anonymes.
— OCaml est fonctionnel, orienté objet et impératif.

Mode orienté objet

Le **mode orienté objet** est placé à part car il s'adresse à un problème différent.

Ce mode a pour objectif premier de faciliter le développement et maintenance des grands logiciels permettant de gérer séparément de parties disjointes.

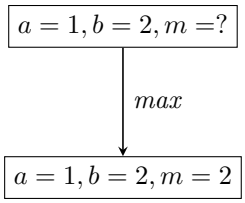
Les autres modes ont l'objectif de définir et organiser les calculs à réaliser par l'ordinateur ou la plateforme d'exécution.

Mode impératif

Dans le **mode impératif**, un programme est vu comme une fonction de transition qui transforme l'état de mémoire de l'ordinateur dans un autre état de mémoire.

Exemple 1 (C)**.**

```c
int max(int a, int b) {
    int m;
    if (a >= b) m = a;
    else m = b;
    return m;
}
```



Mode logique

Dans le **mode logique**, un programme est vu comme un prédicat qui relie l'entrée à la sortie du programme.

Exemple 2 (Prolog)**.**

```
max(A, B, M) :- A >= B, M = A.
max(A, B, M) :- B > A, M = B.
```

$$A \geq B \wedge M = A \rightarrow max(A, B, M) \wedge$$
$$B > A \wedge M = B \rightarrow max(A, B, M)$$

Exécution :

```
?- max(1, 2, M)
M = 2
yes
?-
```

Mode fonctionnel

Dans le **mode fonctionnel**, un programme est vu comme une fonction.

Exemple 3 (Haskell)**.**

```
max :: (Integer, Integer) -> Integer
max (a, b) | a >= b = a
           | otherwise = b
```

$$\max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$
$$(a, b) \mapsto \begin{cases} a, & si\ a \geq b \\ b, & sinon \end{cases}$$

Exécution :

```
ghci> max (1, 2)
2
ghci>
```

1.2 Haskell

Il y a plusieurs raisons pour l'utilisation de Haskell comme langage fonctionnel :

— Gratuit.

— Portable (Linux, Windows, Mac)

— Utilisé dans « la vraie vie » (ABN AMRO, Facebook, Google, Intel, NVIDIA, etc.) [1]

— Communauté active.

— **Purement fonctionnel.**

Pureté

Haskell est purement fonctionnel.

Cela veut dire que toutes les fonctions en Haskell sont des fonctions dans le sens mathématique.

Il n'y a pas d'effet de bord. (Ou presque. En effet, ils seront isolés du reste du programme.)

Même les opérations d'entrée et de sortie (IO) sont des descriptions de ce que le programme doit faire (au lieu de comment le faire).

Il n'y a pas d'instruction, uniquement des expressions.

Exécution d'un programme en Haskell

Trois options :

— Compiler le code source et ensuite exécuter le programme (`ghc`).

— Compiler et exécuter « à la volée » (`runghc`).

— Interpréter le code source de manière interactive (`ghci`).

Soit le fichier source `hello.hs` :

```
1 hello = "Hello world!"
```

Pour l'instant, nous allons utiliser uniquement la troisième option :

1. Regardez par exemple : `https://wiki.haskell.org/Haskell_in_industry` et `https://haskellcosm.com/`

```
$ ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :load hello.hs
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, one module loaded.
*Main> hello
"Hello world!"
*Main> :quit
$
```

1.3 Notions élémentaires

Ensemble

Un **ensemble** est entièrement déterminé par des objets (**éléments**) vérifiant la relation d'appartenance.

L'idée est que l'ensemble regroupe des objets partageant les mêmes propriétés.

Exemples : $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$

Des opérations sont possibles et définies sur les objets dès lors qu'ils appartiennent à un ensemble.

Exemples : $+, -, \times, \div$

Type

En informatique, un **type** désigne un ensemble d'objets associé à des opérations qui sont les seules permises et applicables à ces objets.

Un type t **représente** un ensemble E lorsque :

— pour une partie A assez grande de E, on peut faire correspondre à chaque élément x de A, un objet o de t ; et

— les opérations associées aux objets de t correspondent à des opérations sur E.

On dit que l'objet o représente l'élément x et que o est de type t.

Un **type de base** t est un type qui représente un ensemble E associé, et qui est défini par un codage binaire de certains éléments de E.

Quelques exemples en Haskell :

— `Int` représente $\mathbb{Z}$.

— `Double` représente $\mathbb{R}$.

— `Bool` représente $\{\text{vrai}, \text{faux}\}$.

— `Char` représente $\{\text{a}, \text{b}, \ldots, \text{z}\}$ (en effet, tous les caractères UTF-8).

Fonction

En mathématiques, une **fonction** f fait correspondre, à chaque élément x d'un ensemble A (**domaine**), un élément unique (**image**) dans un autre ensemble B (**codomaine**).

Notation :

$$\begin{aligned} f : A &\to B \\ x &\mapsto f(x) \end{aligned}$$

Exemple 4.

$$\begin{aligned} suiv : \mathbb{Z} &\to \mathbb{Z} \\ x &\mapsto x + 1 \end{aligned}$$

En informatique on dit que l'**application** de f à l'argument x a comme résultat $f(x)$.

L'application de $suiv$ à 1 a comme résultat 2 (car $suiv(1) = 2$).

Définir une fonction

Pour définir une fonction, il faut être capable d'exprimer formellement la correspondance entre chaque élément du domaine et son image dans le codomaine.

Exemple 5.

$$\begin{aligned} 0 &\mapsto 1 \\ 1 &\mapsto 2 \\ 2 &\mapsto 3 \\ &\dots \end{aligned}$$

Exemple 6.

$$suiv(n) = n + 1$$

Notez que nous pouvons déduire que le domaine de cette fonction est l'ensemble de tous les objets auxquels nous pouvons ajouter 1.

En Haskell :

```
ghci> suiv n = n + 1
ghci> suiv 1
2
ghci>
```

Conditionnelle

Tous les langages proposent une construction syntaxique pour réaliser la décomposition par cas appelée **alternative** (ou parfois conditionnelle).

En Haskell :

```
ghci> :{
ghci| max (a, b) | a >= b = a
ghci|            | otherwise = b
ghci| :}
ghci> max (1, 2)
2
ghci> max (2, 1)
2
```

Composition de fonctions

La **composition de fonctions** est un procédé qui consiste à définir une fonction en se servant d'une ou plusieurs autres fonctions.

Exemple 7.

```
ghci> aireCarre r = r * r
ghci> volumeCube r = r * aireCarre r
ghci> volumeCube 10
1000
```

En Haskell :

```
ghci> f x = x + 1
ghci> g x = x * 2
ghci> f (g 1)
3
ghci> (f . g) 1
3
ghci>
```

Fonctions sur n-uplets

Par définition, un 2-uplet est un couple, un 3-uplet est un triplet, un 4-uplet est un quadruplet, etc.

Une **fonction sur n-uplets** est une fonction dont le domaine ou le codomaine est un ensemble de n-uplets (avec $n > 1$).

Il existent de constructions syntaxiques pour accéder aux éléments d'un n-uplet.

```
ghci> fst (3, 4)
3
ghci> snd (3, 4)
4
```

```
ghci> hauteur c = fst c
ghci> rayon c = snd c
ghci> aireDisque r = 3.14159265 * r * r
ghci> volumeCylindre c = aireDisque (rayon c) * hauteur c
ghci> volumeCylindre (3, 4)
150.79644720000002
```

Exercice : Écrivez les fonctions `fst` et `snd` en Haskell.

Fonctions anonymes

En mathématique on peut exprimer une fonction sans la nommer.

Par exemple : « la fonction qui à tout x associe $2x + 1$ ».

En Haskell : `\x -> 2 * x + 1`

Exemple 8.

```
ghci> (\x -> x + 1) 1
2
ghci> (\x -> 2 * x + 1) 2
5
```

Définition temporaire

En mathématique on introduit parfois une variable : « soit d le discriminant de l'équation $ax^2 + bx + c = 0$ ».

En Haskell : `let x = b in e`

Exemple 9.

```
ghci> let x = 2 + 3 * 4 in x * x
196
```

Ou, de manière équivalente :

```
ghci> (\x -> x * x) (2 + 3 * 4)
196
```

Démarche déductive

Le mode fonctionnel permet et encourage une démarche déductive.

En connaissant l'expression du résultat en fonction d'une ou plusieurs valeurs intermédiaires, on peut faire l'hypothèse que ces valeurs intermédiaires peuvent être définies à partir des donnés du problème.

On en déduit que la fonction à trouver est la composée de la fonction connue qui permet d'obtenir le résultat à partir de ces valeurs intermédiaires.

On peut appliquer la même démarche pour définir chacune des autres fonctions jusqu'à arriver à des fonctions prédéfinies.

1.4 Exercices

Donnez la réponse en Haskell pour les exercices ci-dessous.

1. Écrivez une fonction qui, étant donné quatre nombres, retourne *vrai* si tous les nombres sont égaux et *faux* sinon.
2. Écrivez une fonction qui, étant donné quatre nombres, retourne le plus grand des quatre.
3. Écrivez une fonction qui, étant donné quatre nombres, retourne le plus éloigné des quatre. (La distance entre deux nombres est la valeur absolue de leur différence. La distance entre un nombre x et un ensemble de nombres est la somme des distances entre x et chacun des nombres de l'ensemble.)
4. Écrivez une fonction qui, étant donné deux points A et B du plan, calcule la longueur du segment $[AB]$.
5. Écrivez une fonction qui détermine s'il est possible de créer une chaîne avec deux dominos donnés.
6. Écrivez une fonction qui détermine s'il est possible de créer une chaîne avec trois dominos donnés.
7. Écrivez une fonction qui, étant donné une somme s en centimes d'euros et des nombres de pièces a, b, c et d, respectivement de 2, 1, 0,5 et 0,10 euros, retourne *vrai* si l'on peut payer exactement la somme en utilisant une partie ou la totalité des pièces et *faux* sinon.

Chapitre 2

Récursivité

2.1 Définition de récursivité

Les langages de programmation purement fonctionnels (par exemple, Haskell) ne contiennent pas d'instruction de répétition (while et for).

La seule manière d'implémenter une répétition est d'utiliser la récursivité.

Une définition est **récursive** lorsqu'elle se sert du nom qu'elle est en train de définir.

Exemple : Un « descendant » d'un individu est l'un de ses enfants ou un « descendant » de l'un de ses enfants.

Exemple :

$$\begin{aligned} somme &: \mathbb{N} \to \mathbb{N} \\ x &\mapsto \begin{cases} 0, & \text{si } x = 0 \\ somme(x-1) + x, & \text{si } x > 0 \end{cases} \end{aligned}$$

Forme générale d'une définition récursive

$$\begin{aligned} f &: D \to C \\ x &\mapsto \begin{cases} e_0, & \text{si } x \in D_0 \\ F(f(e_1), \ldots, f(e_k)), & \text{si } x \in D_1 \end{cases} \end{aligned}$$

où :

- f est le nom de la fonction.
- e_i est une expression qui dépend uniquement de x.
- $F(f(e_1), \ldots, f(e_k))$ dépend uniquement de x et des $f(e_i)$.
- $D_0 \cup D_1 = D$
- $D_0 \cap D_1 = \emptyset$

NB : Il peut y avoir plusieurs lignes de chaque type.
Exemple : La fonction *somme* est dans cette forme :

$$somme : \mathbb{N} \to \mathbb{N}$$
$$x \mapsto \begin{cases} 0, & \text{si } x = 0 \\ somme(x-1) + x, & \text{si } x > 0 \end{cases}$$

— $f = somme$
— $D = C = \mathbb{N}$
— $e_0 = 0$ et $D_0 = \{0\}$
— $k = 1$, $e_1 = x - 1$, $F(f(e_1)) = f(e_1) + x$ et $D_1 = \mathbb{N}^*$
— $\{0\} \cup \mathbb{N}^* = \mathbb{N}$
— $\{0\} \cap \mathbb{N}^* = \emptyset$

La notion de garde

En langage de programmation, une **garde** est une expression booléenne qui doit être vrai pour que ce qui suit dans le programme soit choisi.

Exemple : La fonction *somme* en Haskell en utilisant des gardes :

```
1 somme :: Integer -> Integer
2 somme x | x <= 0 = 0
3         | x > 0  = somme (x - 1) + x
```

La ligne 3 est appelée **cas de base**.

La ligne 4 est appelée **cas récursif**.

2.2 Création d'une définition récursive

Pour créer une définition récursive :

1. Opérer une décomposition par cas et identifier des sous-problèmes de même nature et de taille inférieure.
2. Supposer ces sous-problèmes résolus par autant d'appels récursifs et résoudre le problème entier en combinant les solutions partielles.

Exemple : Créez la définition de *somme* :

1. Deux cas se présentent :
   — si $x \leq 0$, alors la solution est 0.
   — si $x > 0$, alors le calcul de $0 + 1 + \cdots + (x - 1)$ est un sous-problème de même nature : Il suffit de calculer $somme(x - 1)$.
2. Si l'on suppose que $somme(x - 1)$ est résolu, alors il suffit de faire $somme(x - 1) + x$ pour calculer $somme(x)$.

D'où la définition récursive :

```
1 somme :: Integer -> Integer
2 somme x | x <= 0 = 0
3         | x > 0  = somme (x - 1) + x
```

Exercice : Écrivez la fonction $sommeChiffres$ qui calcule la somme des chiffres décimaux d'un entier naturel n. (Par exemple, pour $n = 124$ le résultat doit être $1 + 2 + 4 = 7$.)

Soit $n = c_0 c_1 \dots c_k$:

1. Trois cas se présentent :
   - si $n < 0$, alors la solution (par défaut) est 0.
   - si $k = 0$, c.-à-d. $0 \leq n < 10$, alors la solution est n.
   - si $k > 0$, c.-à-d. $n \geq 10$, alors $c_0 + c_1 + \dots + c_{(k-1)}$ est un sous-problème de même nature. Il s'agit de calculer la somme des chiffres de $\lfloor n/10 \rfloor$.
2. Si l'on suppose que $sommeChiffres(\lfloor n/10 \rfloor)$ est résolu, alors il suffit de faire $(n \mod 10) + sommeChiffres(\lfloor n/10 \rfloor)$ pour calculer $sommeChiffres(n)$.

Nous avons donc :

```
1 sommeChiffres :: Integer -> Integer
2 sommeChiffres n | n < 0     = 0
3                 | n < 10    = n
4                 | otherwise = (n `mod` 10) + sommeChiffres (n `div` 10)
```

2.3 Correction

Pour montrer qu'un programme récursif fonctionne correctement, nous devons écrire une preuve par induction.

Structure d'une preuve par induction : Soit $P(n)$ une propriété d'un entier naturel n. Si les deux conditions suivantes sont vérifiées, alors $P(n)$ est vrai pour tout $n \in \mathbb{N}$:

- **Cas de base :** $P(0)$ est vrai.
- **Cas récursif :** Pour tout $n \geq 0$, $P(n)$ implique $P(n+1)$.

Exemple : Soit la fonction suivante :

```
1 somme :: Integer -> Integer
2 somme x | x <= 0 = 0
3         | otherwise = somme (x - 1) + x
```

Montrer que la définition de la fonction $somme$ est correcte. C'est-à-dire, montrer que $somme(n) = 0 + 1 + \dots + n$ est vrai pour tout $n \in \mathbb{N}$.

Nous allons montrer par induction :

- **Cas de base :** $somme(0) = 0$ (parce que la ligne 3 sera exécutée).
- **Cas récursif :** Soit $n \geq 0$. Si $somme(n) = 0 + 1 + \dots + n$, alors $somme(n+1) = 0 + 1 + \dots + n + (n+1)$ (parce que la ligne 4 sera exécutée).

Donc, $somme(n) = 0 + 1 + \cdots + n$ pour tout $n \in \mathbb{N}$. QED. [1]

Le raisonnement précédent permet non seulement de prouver la correction d'un programme mais aussi de le construire.

On construit le programme en même temps que ça preuve.

Programmer c'est prouver !

2.4 Terminaison

Il est facile de créer des fonctions qui ne terminent pas :

Exemple : *(Cette fonction ne termine pas !)*

```
pair :: Integer -> Bool
pair n | n == 0 = True
       | n > 0 = pair (n - 2)
       | n < 0 = pair (n + 2)
```

Exemple : *(Cette fonction ne termine pas !)*

```
moyenne :: (Integer, Integer) -> Integer
moyenne (n, m) | n == m = n
               | n > m = moyenne (n - 1, m + 1)
               | n < m = moyenne (n + 1, m - 1)
```

Il serait pratique de disposer d'une fonction (un programme) qui teste la terminaison de nos fonctions.

Pourtant, une telle fonction n'existe pas. Voici la preuve (Gödel, 1930 ; Turing, 1936) :

Supposons que nous avons créé une fonction qui teste si toute fonction termine. Appelons-la `termine`.

Donc, nous pouvons créer une autre fonction. Appellon-la `absurde` :

```
absurde x | termine (absurde x) = absurde x
          | otherwise = True
```

Si `absurde` termine, alors `absurde` ne termine pas.

Si `absurde` ne termine pas, alors `absurde` termine.

Mais ceci est un absurde !

Donc, `termine` n'existe pas. QED.

Heureusement, **dans certains cas particuliers**, nous pouvons prouver qu'une fonction termine.

Soit une fonction récursive f qui reçoit comme arguments $(a_1, \ldots, a_k) \in D_1 \times \cdots \times D_k$. Si f respecte les conditions suivantes, alors elle termine :

1. Elle comporte au moins un **cas de base**.
   - C'est-à-dire, un cas particulier, basé sur les arguments $a_1, \ldots, a_k$, dont le traitement n'utilise pas d'appel récursif.
2. Le cas de base est atteint après un nombre **fini** d'appels récursifs.

1. QED = « *Quod erat demonstrandum* »

— Pour cela, il faut s'assurer que tout appel récursif est toujours plus proche des cas de base que l'appel courant.

Exemple : Soit la fonction suivante :

```
somme :: Integer -> Integer
somme n | n <= 0 = 0
        | n > 0  = somme (n - 1) + n
```

Cette fonction termine pour tout $n \in \mathbb{Z}$ car :

— **Cas de base :** Si la fonction est appelée avec $n \leq 0$, elle retourne 0 et termine.

— Il reste le cas où $n > 0$, qui est traité avec un **appel récursif :**

  — Dans ce cas, la fonction est appelée récursivement avec $n - 1$.

  — $n - 1$ est plus proche du cas de base que n pour tout n qui ne fait pas partie du cas de base, c.-à-d., $n \in \mathbb{Z}$ tel que $n > 0$.

  — Donc, des appels récursifs successifs tomberont nécessairement dans un des cas de base.

— Puisque tous les cas de base terminent, la fonction termine. QED.

Exercice : Montrez que cette fonction termine pour tout $(a, b) \in \mathbb{Z} \times \mathbb{Z}$.

```
pgcd :: Integer -> Integer
pgcd (a, b) | a < 0 || b < 0 = 0
            | a >= 0 && b == 0 = a
            | a >= 0 && b > 0 = pgcd(b, a `mod` b)
```

Cette fonction termine pour tout $(a, b) \in \mathbb{Z} \times \mathbb{Z}$ car :

— Il y a trois **cas de base :**

  — Si appelée avec $a < 0$, elle retourne 0 et termine.

  — Si appelée avec $b < 0$, elle retourne 0 et termine.

  — Si appelée avec $a \geq 0$ et $b = 0$, elle retourne a et termine.

— Il reste le cas où $a \geq 0$ et $b > 0$, qui est traité avec un **appel récursif :**

  — Dans ce cas, la fonction est appelée récursivement avec $(b, a \mod b)$.

  — Nous avons $a \mod b < b$ pour tout $b \in \mathbb{Z}$ tel que $b > 0$.

  — Donc, des appels récursifs successifs tomberont nécessairement dans un des cas de base.

— Puisque tous les cas de base terminent, la fonction termine. QED.

Exercice : Corrigez cette fonction et montrez que la version corrigée termine pour tout $n \in \mathbb{Z}$.

```
pair :: Integer -> Bool
pair n | n == 0 = True
       | n > 0 = pair (n - 2)
       | n < 0 = pair (n + 2)
```

Terminaison

Version corrigée :

```
pair :: Integer -> Bool
pair n | n == 0 = True
       | n == 1 = False
       | n > 0 = pair (n - 2)
       | n < 0 = pair (n + 2)
```

Cette fonction termine pour tout $n \in \mathbb{Z}$ car :

- **Cas de base :**
  - $n = 0$ termine.
  - $n = 1$ termine.
- **Cas récursifs :**
  - Si $n > 1$, alors un appel récursif est fait avec $n - 2$. Ceci est plus proche des cas de base pour tout $n \in \mathbb{Z}$ tel que $n > 1$.
  - Si $n < 0$, alors un appel récursif est fait avec $n + 2$. Ceci est plus proche des cas de base pour tout $n \in \mathbb{Z}$ tel que $n < 0$.

2.5 Récursivité terminale

Souci d'efficacité

Comparée à une boucle (while ou for), la récursivité entraîne des opérations supplémentaires (inhérents à la gestion de la récursivité et non au problème à résoudre) que le compilateur a des difficultés à éliminer.

Cependant, il existe une forme de récursivité pour laquelle le compilateur est capable d'éliminer tels opérations supplémentaires.

Cette forme est reconnaissable lorsqu'il n'y a aucune opération qui suit l'appel récursif.
Une récursivité est **terminale** lorsqu'elle suit la forme générale suivante :

$$f : D \to C$$
$$x \mapsto \begin{cases} e_0, & \text{si } x \in D_0 \\ f(e_1) & \text{si } x \in D_1 \end{cases}$$

où :

- f est le nom de la fonction.
- e_1 est une expression qui dépend uniquement de x.
- $D_0 \cup D_1 = D$
- $D_0 \cap D_1 = \emptyset$

Exemple :

```
1 pair :: Integer -> Bool
2 pair n | n == 0 = True
3        | n == 1 = False
4        | n > 0 = pair (n - 2)
5        | n < 0 = pair (n + 2)
```

Nous avons :

— Les lignes 3 et 4 sont les cas de base.

— Les lignes 5 et 6 sont les cas récursifs.

— $e_1 = n - 2$

— $e'_1 = n + 2$

Exercice : Est-ce une récursivité terminale ?

```
1 pgcd :: (Integer, Integer) -> Integer
2 pgcd (a, b) | a < 0 || b < 0 = 0
3             | a >= 0 && b == 0 = a
4             | a >= 0 && b > 0 = pgcd(b, a `mod` b)
```

Oui car nous avons :

— Les lignes 3 et 4 sont les cas de base.

— La ligne 5 est le cas récursif.

— $e_1 = (b, a \mod b)$

Exercice : Transformez la fonction `fac` ci-dessous en une fonction récursive terminale.

```
1 fac :: Integer -> Integer
2 fac n | n < 0 = 0
3       | n == 0 = 1
4       | n > 0 = fac (n - 1) * n
```

Astuce : vous avez droit de faire plusieurs fonctions.

```
1 fac' :: Integer -> Integer
2 fac' n = facAux (1, n)
3
4 facAux :: (Integer, Integer) -> Integer
5 facAux (a, n) | n < 0 = 0
6               | n == 0 = a
7               | n > 0 = facAux(a * n, n - 1)
```

Notez que `fac'` n'est pas une fonction récursive.

La fonction `facAux` est récursive et terminale car :

— La première et la deuxième lignes sont les cas de base.

— La troisième ligne est le cas récursif.

— $e_1 = (a \times n, n - 1)$.

Une version avec une sous-fonction :

```
fac'' :: Integer -> Integer
fac'' n = facAux (1, n)
  where
    facAux (a, n) | n < 0 = 0
                  | n == 0 = a
                  | n > 0 = facAux(a * n, n - 1)
```

La démonstration de la terminaison est la même.
Avantages :

— Performance (même performance qu'avec un langage impératif).

— Absence de débordement de la pile.

Inconvénients :

— Moins de lisibilité

— Le gain de performance peut être illusoire.

Récursivité mutuelle (croisée)

Exemple :

```
pair :: Integer -> Bool
pair n | n < 0 = pair (abs n)
       | n == 0 = True
       | n > 0 = impair (n - 1)

impair :: Integer -> Bool
impair n | n < 0 = impair (abs n)
         | n == 0 = False
         | n == 1 = True
         | n > 1 = pair (n - 1)
```

2.6 Exercices

Exercice 1. La technique de multiplication russe permet de calculer le produit de deux nombres entiers naturels en utilisant seulement des additions et de multiplications et divisions par 2. Cette technique est basée sur la propriété :

$$xy = \left(\frac{x}{2}\right) 2y \text{ si } x \text{ est pair}$$

1. Écrivez une définition récursive de la multiplication en utilisant uniquement des additions et multiplications et divisions par 2.
2. Écrivez une fonction qui calcule le produit de deux entiers naturels en utilisant cette technique.

Exercice 2. Le mélange américain des cartes consiste à imbriquer deux paquets de cartes par insertion des cartes de l'un dans l'autre pour former le paquet final. L'imbrication obéit à une seule règle : l'ordre de cartes de chaque paquet doit subsister dans le paquet final.

1. Considérez deux paquets de cartes toutes différentes. Écrivez une fonction qui, étant donné le nombre de cartes de chaque paquet, calcule le nombre de paquets finaux différents qu'il est possible d'obtenir avec le mélange américain.
2. Même question mais avec trois paquets.

Exercice 3. Donald Knuth (1976) a introduit un opérateur double flèche pour puissance itérée :

$$a \uparrow\uparrow b = a^{a^{a^{\cdots}}} \qquad (b \text{ fois })$$

puis l'opérateur triple flèche :

$$a \uparrow\uparrow\uparrow b = a \uparrow\uparrow a \uparrow\uparrow a \ldots \qquad (b \text{ fois })$$

puis l'opérateur quadruple flèche, et ainsi de suite.

1. Écrivez une définition récursive de l'opérateur n-flèche ($\uparrow^n$), pour tout $n \geq 0$.
2. Écrivez une fonction pour calculer $a \uparrow^n b$.

Exercice 4.

1. Écrivez une fonction récursive qui calcule le n-ième nombre de la séquence de Fibonacci. Pour rappel, cette séquence est définie comme suit :

$$\begin{aligned} F_n &= 0, \text{ if } n \leq 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

2. Écrivez la démonstration de la terminaison de la fonction de l'exercice précédent.
3. Écrivez une fonction récursive qui calcule le n-ième nombre de la séquence de Tribonacci. Il s'agit d'une version de Fibonacci à trois termes, c.-à-d. :

$$\begin{aligned} T_n &= 0, \text{ if } n \leq 0 \\ T_1 &= 1 \\ T_n &= T_{n-1} + T_{n-2} + T_{n-3} \end{aligned}$$

4. Écrivez une fonction récursive qui calcule le n-ième nombre de la séquence de n-step Fibonnaci. Il s'agit d'une généralisation de Fibonacci à n termes, c.-à-d. :

$$\begin{aligned} N_k^{(n)} &= 0, \text{ if } k \leq 0 \\ N_1^{(n)} &= 1 \\ N_k^{(n)} &= N_{k-1}^{(n)} + N_{k-2}^{(n)} + \cdots + N_{k-n}^{(n)} \end{aligned}$$

5. Écrivez la démonstration de la terminaison de la fonction de l'exercice précédent.
6. Écrivez les définitions des fonctions qui calculent le n-ième terme des séquences de Tetranacci, Pentanacci et Hexanacci.

Exercice 5. La conjecture de Goldbach est une assertion mathématique non démontrée qui s'énonce comme suit : tout nombre entier pair supérieur à 3 peut s'écrire comme la somme de deux nombres premiers.

1. Écrire une fonction récursive qui teste si un entier naturel est premier. On rappelle qu'un entier naturel est premier si ces seuls diviseurs sont 1 et lui-même.
2. Écrire une fonction récursive qui, étant donné un entier n, vérifie la conjecture de Goldbach pour tous les entiers pairs entre 4 et n.

Chapitre 3

Types

3.1 La notion de type

Types

Les types aident à vérifier la correction des programmes et à les traduire en code exécutable.

Les types ont une importance accrue dans les langages fonctionnels car ils servent de fondement à certains langages, notamment ML et ses descendants (dont Haskell).

En programmation fonctionnelle, un objet appartenant à un type est une **constante** entièrement déterminée par sa valeur.

Langages faiblement et fortement typés

Un langage est **fortement typé** quand un seul type est attribué à toute expression bien formée du langage.

— Pour la plupart de ces langages, le type est déterminé et vérifié avant l'exécution du programme (**statiquement**).

Un langage est **faiblement typé** quand une même expression peut avoir plusieurs types.

— Pour la plupart de ces langages, le type est déterminé et vérifié au moment de l'exécution du programme (**dynamiquement**).

Haskell est fortement typé

Haskel est fortement typé. Donc, toute expression a un type.

Nous pouvons demander le type d'une expression avec la commande `:type`.

```
ghci> :type True
True :: Bool
ghci> :type False
False :: Bool
```

Nous pouvons aussi utiliser la commende `:set +t` pour demander que les types des expressions soient toujours affichés :

```
ghci> :set +t
ghci> True && False
False
it :: Bool
ghci> False || True
True
it :: Bool
```

Pour arrêter l'affichage des types, utilisez `:unset +t`.

Langages faiblement et fortement typés

L'approche faiblement typé est plus souple, mais avec cet approche le programme peut mal fonctionner (sans forcement s'interrompre) en raison d'une erreur de type qui n'a pas été détectée plus tôt.

L'approche fortement typé est plus contraignant, mais de tels erreurs sont impossibles.

Les langages fortement typés incluent donc les notions de **constructeur de type** ou **inférence de type** définissant une véritable théorie de types.

3.2 Types de base

Types de base

Quelques types de base en Haskell :

type	description
`Char`	caractères unicode
`Bool`	valeurs logique
`Int`	entiers (taille fixe)
`Integer`	entiers (taille illimitée)
`Float`	virgules flottantes
`Double`	virgules flottantes (double précision)

Listes

Une **liste** est un conteneur de valeurs dont tous les éléments sont du **même type** :

```
ghci> nombres = [1,2,3]
ghci> nombres
[1,2,3]
ghci> bools = [True, False, True]
ghci> bools
[True,False,True]
ghci> chaines = ["quelques", "chaines"]
ghci> chaines
["quelques","chaines"]
ghci> autre = [True,'a']

<interactive>:7:15: error:
```

```
    . Couldn't match expected type 'Bool' with actual type 'Char'
    . In the expression: 'a'
      In the expression: [True, 'a']
      In an equation for 'autre': autre = [True, 'a']
```

« Consing »

Nous pouvons ajouter un élément à une liste avec l'opérateur `(:)`, appelé « cons » :

```
ghci> nombres = [1,2,3]
ghci> nombres
[1,2,3]
ghci> 0 : nombres
[0,1,2,3]
ghci> 0 : [1,2,3]
[0,1,2,3]
ghci> 0 : []
[0]
ghci> 0 : 1 : []
[0,1]
```

Le terme « cons » vient de « constructor » de listes.

En effet, la syntaxe `[1, 2, 3]` n'est rien d'autre que du sucre syntaxique. Toutes les listes en Haskell sont faites avec cons.

```
ghci> 0 : 1 : 2 : [] == [0, 1, 2]
True
ghci> True : False : [] == [True, False]
True
```

Attention à cette erreur très commune :

```
ghci> True : False

<interactive>:19:8: error:
    . Couldn't match expected type '[Bool]' with actual type 'Bool'
    . In the second argument of '(:)', namely 'False'
      In the expression: True : False
      In an equation for 'it': it = True : False
```

Les chaines sont des listes

Les chaines de caractères sont des listes de `Char` :

```
ghci> :type "chaine"
"chaine" :: String
ghci> "chaine" == ['c', 'h', 'a', 'i', 'n', 'e']
True
ghci> "hey" == 'h' : 'e' : 'y' : []
True
```

Listes de listes

Les listes de Haskell peuvent contenir tout type de valeur, pourvu que tous ses éléments soit du même type :

```
ghci> listeDeListe = [[1,2],[3,4],[4,5]]
ghci> listeDeListe
[[1,2],[3,4],[4,5]]
```

Exercice : Lesquelles sont valides ? Réécrivez les listes valides avec cons.

1. `[1,2,3,[]]`
2. `[1,[2,3],4]`
3. `[[1,2,3],[]]`

Exercice : Lesquelles sont valides ? Réécrivez les listes valides avec crochets :

1. `[]:[[1,2,3],[4,5,6]]`
2. `[]:[]`
3. `[]:[]:[]`
4. `[1]:[]:[]`
5. `["hi"]:[1]:[]`

Filtrage par motif

Le **filtrage par motif** (ou *pattern matching*) consiste à utiliser un motif à la place d'un paramètre formel pour designer une valeur d'un type structuré.

En Haskell :

```
1 somme :: Integer -> Integer
2 somme 0 = 0
3 somme n = n + somme (n - 1)
```

Nous pouvons le mélanger avec des gardes :

```
1 somme :: Integer -> Integer
2 somme 0 = 0
3 somme n | n < 0 = n + somme (n + 1)
4         | otherwise = n + somme (n - 1)
```

Filtrage par motif avec de listes

Exemple :

```
1 sommeList :: [Integer] -> Integer
2 sommeList [] = 0
3 sommeList (x:xs) = x + sommeList xs
```

Exemple :

```
1 longueur :: [Integer] -> Integer
2 longueur [] = 0
3 longueur (_:xs) = 1 + longueur xs
```

Exemple :

```
sommeUnSurDeux :: [Integer] -> Integer
sommeUnSurDeux [] = 0
sommeUnSurDeux [_] = 0
sommeUnSurDeux (_ : y : zs) = y + sommeUnSurDeux zs
```

Type produit (tuples)

Soit $A_1, \dots A_n$ de types bien formés. La notation $A_1 \times \cdots \times A_n$ est un type bien formé.

Les éléments de $A_1 \times \cdots \times A_n$ sont des n-uplets $(a_1, \dots, a_n)$ où chaque a_i est du type A_i.

Exemples :

— Le type $\mathbb{N} \times \mathbb{N}$ contient les éléments $(0, 0)$, $(0, 1)$, $(45, 2029)$, etc.

— Le type $\mathbb{Z} \times \mathbb{R} \times \mathbb{Q}$ contient $(-1, \pi, 0)$, $(-234, 1.34, \frac{1}{3})$, etc.

En Haskell :

```
ghci> :type ('a', 'b', 'c')
('a', 'b', 'c') :: (Char, Char, Char)
ghci> :type (True, False)
(True, False) :: (Bool, Bool)
```

Listes vs. tuples

Différences entre listes et tuples :

— Il n'est pas possible d'utiliser cons pour construire un tuple.

— Les éléments d'un tuple n'ont pas besoin d'être d'un même type.

Il est possible de combiner les types :

```
((2,3), True)
((2,3), [2,3])
[(1,2), (3,4), (5,6)]
```

Exercice : Lequelles sont valides ?

1. `1:(2,3)`
2. `(2,4):(2,3)`
3. `(2,4):[]`
4. `[(2,4),(5,5),('a','b')]`
5. `([2,4],[2,2])`

3.3 Polymorphisme

Types polymorphes

Le type d'un tuple est défini par ces éléments :

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

Nous avons déjà rencontré la fonction `fst` :

```
ghci> fst ('a', True)
'a'
ghci> fst (True, 'a')
True
```

Quelle est le type de la fonction `fst` ?

```
ghci> :type fst
fst :: (a, b) -> a
```

Polymorphisme

Une expression est **polymorphe** quand elle peut servir sans modifications dans de contextes différents.

Du grec, « poly » = plusieurs et « morphê » = formes.

Un grand intérêt du polymorphisme est de permettre d'écrire une seule fonction qui prend en argument des valeurs appartenant de plusieurs types, plutôt que d'écrire plusieurs fonctions.

Le polymorphisme permet donc de rendre un programme plus général.

Le type de `fst` est `(a, b) -> a`. Ici, `a` et `b` sont des variables de type.

Une **variable de type** identifie un type quelconque (parmi les types possibles).

Un type est **polymorphe** si son expression comporte une variable de type non instanciée ou instanciée avec un type polymorphe.

Soit le type d'une fonction `f` :

```
f :: a -> a
```

Nous avons que l'argument et le retour de `f` sont du même type.

Soit le type d'une fonction `f`

```
f :: a -> b
```

Nous avons que tant l'argument come le retour de `f` sont de n'importe quel type. Les deux peuvent ou pas être le même.

Exercice : Quelle est donc le type de `snd` ?

```
ghci> :type snd
snd :: (a, b) -> b
```

Polymorphisme

Exercice : Écrivez la fonction identité (n'oubliez pas le type).

```
ghci> myId 1
1
ghci> myId "oui"
"oui"
```

```
myId :: a -> a
myId x = x
```

Exercice : Écrivez la fonction `mySwap` :

```
ghci> mySwap (1, 2)
(2,1)
ghci> mySwap (1, "oui")
("oui",1)
```

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

3.4 Fonctionnelles et fonctions d'ordre supérieur

Type flèche (fonctions)

Soit A_1 et A_2 deux types bien formés. La notation $A_1 \to A_2$ est un type bien formé.

Les éléments de $A_1 \to A_2$ sont les fonctions de domaine A_1 et codomaine A_2.

Exemples :

— Le type $\mathbb{R} \to \mathbb{R}$ contient les fonctions qu'associent un nombre réel à un nombre réel.

— Le type $(\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$ contient les fonctions qu'associent un nombre réel à tout couple de nombre réels.

Exemple :

$$\begin{aligned} somme : \mathbb{N} &\to \mathbb{N} \\ x &\mapsto \begin{cases} 0, & \text{if } x = 0 \\ x + somme(x-1), & \text{sinon} \end{cases} \end{aligned}$$

Fonctionnelles

Une **fonctionnelle** est une fonction dont l'argument est une fonction.

Exemple : Une suite arithmétique est une fonction qu'associe un entier naturel à tout entier naturel. Par exemple :

$$\begin{aligned} suite : \mathbb{N} &\to \mathbb{N} \\ n &\mapsto \begin{cases} 1, & \text{si } n = 0 \\ suite(n-1) + 2, & \text{sinon} \end{cases} \end{aligned}$$

Nous voulons maintenant définir une fonctionnelle qui calcule la raison d'une suite arithmétique donnée.

$$\begin{aligned} raison : (\mathbb{N} \to \mathbb{N}) &\to \mathbb{N} \\ f &\mapsto f(1) - f(0) \end{aligned}$$

La fonction *raison* associe un entier naturel à toute fonction qu'associe un entier naturel à tout entier naturel.

Nous avons, par exemple, $raison(suite) = suite(1) - suite(0) = 3 - 1 = 2$.

Fonctionnelles

En Haskell :

```
suite :: Integer -> Integer
suite n | n <= 0 = 1
        | otherwise = suite (n - 1) + 2

raison :: (Integer -> Integer) -> Integer
raison f = (f 1) - (f 0)
```

```
ghci> suite 0
1
ghci> suite 1
3
ghci> raison suite
2
```

Fonctionnelles

Nous pouvons utiliser raison avec d'autres suites arithmétiques :

```
suite2 :: Integer -> Integer
suite2 n | n <= 0 = 1
         | otherwise = suite2 (n - 1) + 5
```

```
ghci> suite2 0
1
ghci> suite2 1
6
ghci> raison suite2
5
```

Fonctions d'ordre supérieur

Une **fonction d'ordre supérieur** est une fonction dont le résultat est une fonction.

— Une **fonction d'ordre** 1 est une fonction dont le résultat n'est pas une fonction.

— Une **fonction d'ordre** 2 est une fonction dont le résultat est une fonction d'ordre 1.

— Une **fonction d'ordre** n est une fonction dont le résultat est d'ordre $n - 1$.

Fonctions d'ordre supérieur

Exemple : Nous voulons créer une fonction *sign* qu'à tout entier n associe une fonction f_n. La fonction f_n associe l'expression $(-1)^n x$ à tout nombre réel x. (c.à.d., x si n est pair et $-x$ sinon).

$$\begin{aligned} sign : \mathbb{N} &\to (\mathbb{R} \to \mathbb{R}) \\ n &\mapsto f_n \end{aligned}$$

où :

$$\begin{aligned} f_n : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto (-1)^n x \end{aligned}$$

Autrement dit :

$$\begin{aligned} sign : \mathbb{N} &\to (\mathbb{R} \to \mathbb{R}) \\ n &\mapsto \lambda x.((-1)^n x) \end{aligned}$$

Fonctions d'ordre supérieur

En Haskell :

```
sign :: Integer -> (Double -> Double)
sign n = f where
    f x = (-1) ^ n * x
```

ou bien :

```
sign :: Integer -> (Double -> Double)
sign n = \x -> (-1) ^ n * x
```

```
ghci> (sign 5) 3.5
-3.5
```

En Haskell, l'application est associative à gauche. Cela veut cire que : `f x z` est équivalent à `(f x) z`.

Donc, nous pouvons faire aussi :

```
ghci> sign 5 3.5
-3.5
```

Différentes types de définition

Ces définitions sont équivalentes en Haskell :

`suiv n = n + 1` `suiv = \n -> n + 1`

Ainsi que :

`add a b = a + b` `add a = \b -> a + b` `add = \a -> \b -> a + b`

Il y a donc $n + 1$ façons différentes de définir une fonction d'ordre n.

Application partielle

En programmation fonctionnelle il est légal d'appliquer une fonction d'ordre n à moins d'n arguments.

```
ghci> add = \x -> \y -> x + y
ghci> suiv = add 1
ghci> suiv 2
3
```

L'expression « `add 1` » est équivalente à $(\lambda x.\lambda y.(x + y))(1)$.

Cette dernière est équivalente à $\lambda y.(1 + y)$. Celle-ci a été nommé `suiv`.

Curryfication

Il y a deux possibilité pour résoudre un problème dont le nombre de donnée à l'entrée est 2.

```
1 add1 :: (Integer, Integer) -> Integer
2 add1 (a, b) = a + b
3
4 add2 :: Integer -> Integer -> Integer
5 add2 a b = a + b
```

Ces deux définitions ne sont pas équivalente puisque leurs types sont différents.

On dit que la fonction `add2` est la forme **curryfiée** de `add1`. (Le terme vient du nom du mathématicien Haskell Curry.)

La forme curryfiée présente l'intérêt de permettre l'application partielle.

Exercice

Écrivez une fonction `myCurry` qu'à toute fonction non-curryfiée de deux arguments associe sa version curryfiée.

```
1 myCurry :: ((a, b) -> c) -> (a -> b -> c)
2 myCurry f = \x -> \y -> f (x, y)
```

Écrivez une fonction `myUncurry` qu'à toute fonction curryfiée de deux arguments associe sa version non-currifiée.

```
1 myUncurry :: (a -> b -> c) -> ((a, b) -> c)
2 myUncurry f = \(x, y) -> f x y
```

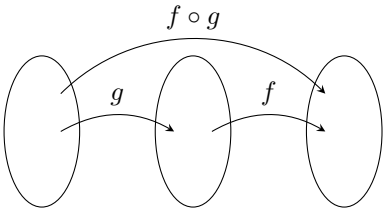
Maintenant nous pouvons faire :

```
ghci> :{
ghci> add1 :: (Integer, Integer) -> Integer
ghci> add1 (a, b) = a + b
ghci> :}
ghci> add2 = myCurry add1
ghci> add1' = myUncurry add2
```

```
ghci> :type add2
add2 :: Integer -> Integer -> Integer
ghci> :type add1'
add1' :: (Integer, Integer) -> Integer
```

Exercice

Écrivez en Haskell la fonctionnelle qui, étant donné deux fonctions f et g, renvoie $f \circ g$, c'est à dire, la composée de g par f.




```
1 compose :: (b -> c) -> (a -> b) -> (a -> c)
2 compose f g = \x -> f (g x)
```

Fonction compose

Maintenant nous pouvons créer des nouvelles fonctions avec `compose` :

```
ghci> add1 x = x + 1
ghci> add2 x = x + 2
ghci> add3 = compose add1 add2
ghci> add3 3
6
```

En Haskell, nous avons un opérateur infixé pour faire la composition de fonctions.

L'expression `f . g` est équivalente à `compose f g`.

```
ghci> add3 = add1 . add2
ghci> add3 3
6
ghci>
```

3.5 Classes de types

Il existe plus d'une manière de représenter les nombres dans l'ordinateur, en particulier, celle utilisée pour les `Int` et celle utilisée pour les `Double`.

L'implémentation, par exemple, de la somme (ainsi des autres opérations) ne peut pas être la même pour ces deux types. D'où la nécessité des fonctions séparées.

Pourtant, en Haskell, la fonction `(+)` fonctionne pour les `Int` et les `Double`, mais pas avec les autres types :

```
ghci> 3 + 4
7
ghci> 4.34 + 3.12
7.46
ghci> 'a' + 'b'

<interactive>:7:1: error:
    . No instance for (Num Char) arising from a use of ‘+’
    . In the expression: 'a' + 'b'
      In an equation for ‘it’: it = 'a' + 'b'
```

Quel est le type de la fonction `(+)`, alors ?

La classe Num

```
ghci> :type (+)
(+) :: Num a => a -> a -> a
```

`Num` est une **classe de types** (typeclass). Cela regroupe tous les types numériques en Haskell.
La partie `Num a =>` du type de `(+)` signifie que `a` est restreinte aux instances de `Num`.

Types numériques

Pour mieux comprendre comment cela fonctionne, faisons quelques testes :

```
ghci> (-7) + 5.12
-1.88
ghci> :t (-7)
(-7) :: Num a => a
ghci> :t 5.12
5.12 :: Fractional p => p
ghci> :t (-7) + 5.12
(-7) + 5.12 :: Fractional a => a
```

Notez que `(-7)` n'est ni `Int` ni `Integer`. Haskell lui attribue le type le plus général possible d'abord.

Ensuite, `5.12` n'est ni `Float` ni `Double`. Haskell lui attribue le type le plus général possible aussi.

Pour faire la somme, les deux nombres doivent être du même type. Donc, Haskell transforme `(-7)` dans un `Fractional`.

Suppose le programme suivant écrit dans un fichier `test.hs` :

```
1 x = 2
```

Ensuite, sur GHCi :

```
ghci> :l test.hs
Main> :t x
x :: Integer
ghci> y = x + 3
ghci> :t y
y :: Integer
```

Maintenant, nous changeons le fichier `test.hs` :

```
x = 2
y = x + 3.1
```

Ensuite, sur GHCi :

```
ghci> :l test.hs
ghci> :t x
x :: Double
ghci> :t y
y :: Double
```

À votre avis, que c'est-t-il passé ?

Soucis avec des types monomorphes

Suppose que nous voulons calculer la moyenne des valeurs dans une liste :

```
ghci> l = [1,2,3]
ghci> sum l / (length l)

<interactive>:7:1: error:
    . No instance for (Fractional Int) arising from a use of ‘/’
    . In the expression: sum l / (length l)
      In an equation for ‘it’: it = sum l / (length l)
```

Le problème vient du type des fonctions `(/)` et `length`, qui ne sont pas compatibles :

```
ghci> :t (/)
(/) :: Fractional a => a -> a -> a
ghci> :t length
length :: Foldable t => t a -> Int
```

Soucis avec des types monomorphes

Pour résoudre ce problème, nous pouvons utiliser la fonction `fromIntegral` :

```
ghci> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

Comme ceci :

```
ghci> sum l / (fromIntegral (length l))
2.0
```

Autres classes de types

Regardons le type de quelques fonctions polymorphes :

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
ghci> :t (<)
(<) :: Ord a => a -> a -> Bool
ghci> :t length
length :: Foldable t => t a -> Int
ghci> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

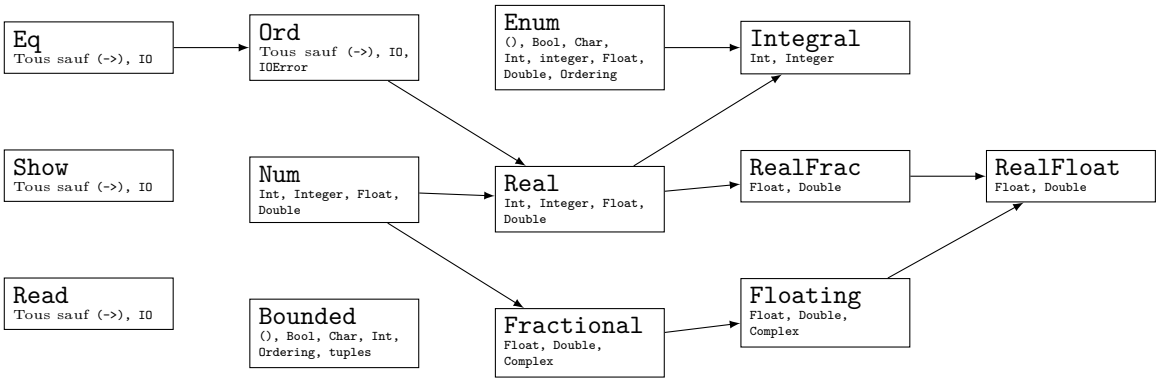
`Eq` est la classe des types qui peuvent être comparés avec l'égalité. (Par exemple, les fonctions n'y font pas partie.)

`Foldable` est la classe types « pliables ». (Nous allons voir ce que cela veut dire plus tard dans la partie Schémas de programmes.)

`Integral` est la classe de types entiers, c.-à-d., `Int` et `Integer`.

Diagramme de classes des types

Voici le diagramme d'héritage des quelques classes pré-définies en Haskell :



Certaines de ces classes nous n'avons pas encore rencontré.

Fonctions de classes de type

Voici quelques fonctions définies par chaque classe de types :

classe	fonctions
Eq	== /=
Show	show
Read	read
Ord	< <= > >= min max
Num	+ - * negate abs signum
Bounded	minBound maxBound
Enum	succ pred
Fractional	/ recip
Integral	div mod quot rem

3.6 Inférence de type

Inférence de type

L'**inférence de type** est un processus implémenté par le contrôleur de types d'un langage fortement typé qui permet de déterminer automatiquement et statiquement le type le plus général de toute expression du langage (sans qu'il soit mentionné explicitement dans le programme).

Nous allons voir l'algorithme W (du système de type HM) qui est utilisé par Haskell et OCaml.

Règles d'inférence de type

Une **règle d'inférence de type** est de la forme générale suivante :

$$\begin{cases} e_1 :: t_1 \\ e_2 :: t_2 \\ \dots \\ e_n :: t_n \end{cases} \Longrightarrow e_0 :: t_0$$

— e_0 est une expression du langage.

— les e_i sont de sous-expressions de e_0.

— les t_i sont des expressions de type.

Les $e_i :: t_i$ sont les prémisses et $e_0 :: t_0$ est la conclusion.

La règle stipule que si chaque e_i est du type t_i alors e_0 est du type t_0.

Quelques règles d'inférence de type

Lambda-expression : $\begin{cases} e_1 :: t_1 \\ e_2 :: t_2 \end{cases} \overset{\text{fun}}{\Longrightarrow} \backslash e_1 \to e_2 :: t_1 \to t_2$

Application de fonction : $\begin{cases} e_1 :: t_1 \to t_2 \\ e_2 :: t_1 \end{cases} \overset{\text{app}}{\Longrightarrow} e_1\ e_2 :: t_2$

Gardes : $\begin{cases} e_0 :: t_1 \to t_2 \\ e_1 :: t_1 \\ e_{11} :: \text{Bool} \\ \dots \\ e_{1n} :: \text{Bool} \\ e_{21} :: t_2 \\ \dots \\ e_{2n} :: t_2 \end{cases} \overset{\text{guard}}{\Longrightarrow} e_0\ e_1 \mid e_{11} = e_{21} \mid \cdots \mid e_{1n} = e_{2n} :: t_1 \to t_2$

Définition locale : $\begin{cases} e_1 :: t \\ e_2 :: t \end{cases} \overset{\text{let}}{\Longrightarrow} e_1 = e_2 :: t$

Opérateurs binaires `==` `/=` : $\begin{cases} e_1 :: \text{Eq}\ \ t \Rightarrow t \\ e_2 :: \text{Eq}\ \ t \Rightarrow t \end{cases} \overset{==}{\Longrightarrow} e_1 == e_2 :: \text{Bool}$

Opérateurs binaires `< <= > >=` :

$$\begin{cases} e_1 :: \text{Ord } a_1 \Rightarrow a_1 \\ e_2 :: \text{Ord } a_2 \Rightarrow a_2 \end{cases} \overset{<}{\Longrightarrow} \texttt{e}_1 < \texttt{e}_2 :: \text{Bool}$$

Opérateurs binaires `+ - *` :

$$\begin{cases} e_1 :: \text{Num } a_1 \Rightarrow a_1 \\ e_2 :: \text{Num } a_2 \Rightarrow a_2 \end{cases} \overset{+}{\Longrightarrow} e_1 + e_2 :: \text{Num } a_3 \Rightarrow a_3$$

Inférence de type

Exemple : Inférence du type de l'expression $1 + 2$:

$$\begin{cases} 1 :: \text{Num } a_1 \Rightarrow a_1 \\ 2 :: \text{Num } a_2 \Rightarrow a_2 \end{cases} \overset{+}{\Longrightarrow} 1 + 2 :: \text{Num } a => a$$

Inférence de type

Exemple : `myAbs x | x >= 0 = x | otherwise = (-x)`

1. $\begin{cases} \texttt{myAbs} :: t_1 \to t_2 \\ \texttt{x} :: t_1 \\ \texttt{x} >= \texttt{0} :: \text{Bool} \\ \texttt{otherwise} :: \text{Bool} \\ \texttt{x} :: t_2 \\ (-\texttt{x}) :: t_2 \end{cases} \overset{\text{guard}}{\Longrightarrow} \texttt{myAbs x} \mid \ldots :: t_1 \to t_2$
2. $\begin{cases} \texttt{x} :: \text{Ord } a_1 \Rightarrow a_1 \\ \texttt{0} :: \text{Ord } a_2 \Rightarrow a_2 \end{cases} \overset{>=}{\Longrightarrow} \texttt{x} >= \texttt{0} :: \text{Bool}$
3. $\begin{cases} - :: \text{Num } a_3 \Rightarrow a_3 \to a_3 \\ \texttt{x} :: \text{Num } a_4 \Rightarrow a_4 \end{cases} \overset{-}{\Longrightarrow} -\texttt{x} :: \text{Num } a_5 \Rightarrow a_5$

Inférence de type

Nous avons :

4. $t_1 = t_2$ (1)
5. $t_2 = \text{Ord } a \Rightarrow a$ (1, 2)
6. $t_2 = (\text{Ord } a, \text{Num } a) \Rightarrow a$ (3, 5)
7. $t_1 = (\text{Ord } a, \text{Num } a) \Rightarrow a$ (4, 6)

Donc, le type de l'expression :

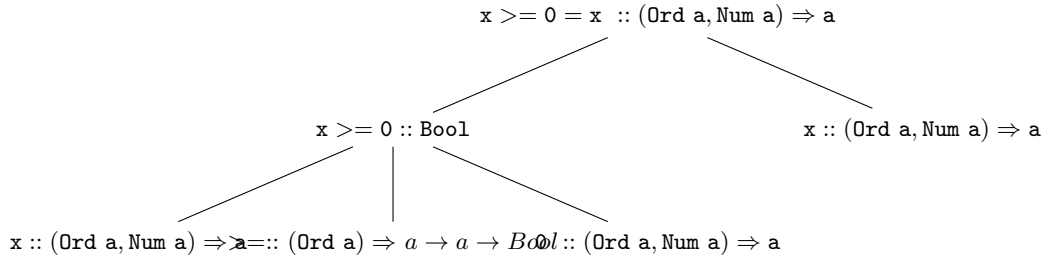
$$\texttt{myAbs x} \mid \texttt{x} >= \texttt{0} = \texttt{x} \mid \texttt{otherwise} = (-\texttt{x})$$

est :

$$(\text{Ord } a, \text{Num } a) \Rightarrow a \to a$$

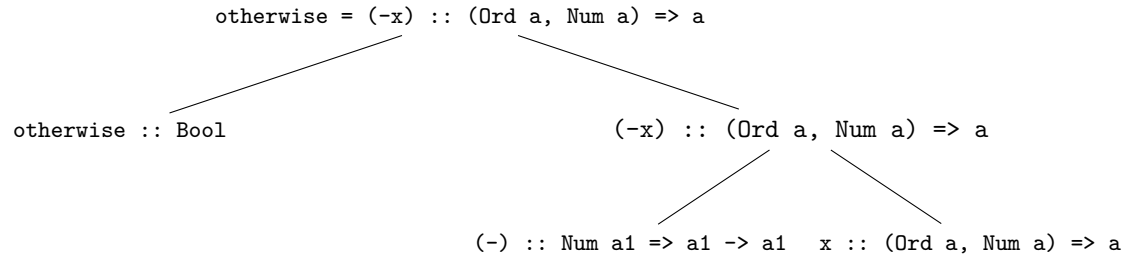
Inférence de type

Exemple : `myAbs x | x >= 0 = x | otherwise = (-x)`

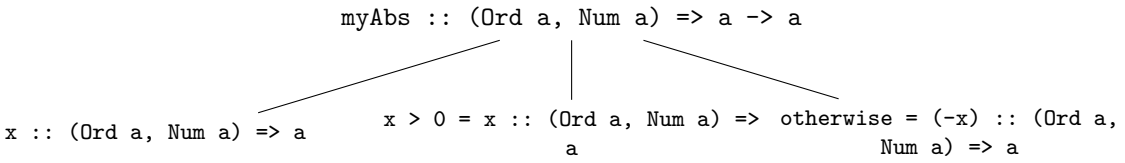


Inférence de type

Exemple : `myAbs x | x > 0 = x | otherwise = (-x)`



Exemple : `myAbs x | x > 0 = x | otherwise = (-x)`



Inférence de type

Exercice : Quel est le type de l'expression : `double = \f -> \x -> 2 * (f x)` ?

1. $\begin{cases} \texttt{double} : t_1 \\ \texttt{\textbackslash f -> \textbackslash x -> 2 * (f x)} :: t_1 \end{cases} \overset{let}{\Longrightarrow} \texttt{double = ...} :: t_1$

2. $\begin{cases} \texttt{f} :: t_2 \\ \texttt{\textbackslash x -> 2 * (f x)} :: t_3 \end{cases} \overset{fun}{\Longrightarrow} \texttt{\textbackslash f -> \textbackslash x -> 2 * (f x)} :: t_2 \to t_3$

3. $\begin{cases} \texttt{x} :: t_4 \\ \texttt{2 * (f x)} :: t_5 \end{cases} \overset{fun}{\Longrightarrow} \texttt{\textbackslash x -> 2 * (f x)} :: t_4 \to t_5$

4. $\begin{cases} \texttt{2} :: \texttt{Num} \\ \texttt{(f x)} :: \texttt{Num} \end{cases} \overset{*}{\Longrightarrow} \texttt{2 * (f x)} :: \texttt{Num}$

5. $\begin{cases} \texttt{f} :: t_4 \to t_6 \\ \texttt{x} :: t_4 \end{cases} \overset{app}{\Longrightarrow} \texttt{f x} :: t_6$

Inférence de type

Nous avons :

6.	$t_6 = \texttt{Num}$	$(4, 5)$
7.	$t_5 = \texttt{Num}$	$(3, 4)$
8.	$t_3 = t_4 \to t_5$	$(2, 3)$
9.	$t_3 = t_4 \to \texttt{Num}$	$(7, 8)$
10.	$t_2 = t_4 \to t_6$	$(2, 5)$
11.	$t_2 = t_4 \to \texttt{Num}$	$(6, 10)$
12.	$t_1 = t_2 \to t_3$	$(1, 2)$
13.	$t_1 = (t_4 \to \texttt{Num}) \to t_3$	$(11, 12)$
14.	$t_1 = (t_4 \to \texttt{Num}) \to (t_4 \to \texttt{Num})$	$(9, 13)$

Donc, le type de l'expression `double = \f -> \x -> 2 * (f x)` est :

$$(t_4 \to \texttt{Num}) \to t_4 \to \texttt{Num}$$

3.7 Exercices

Exercice 1. Pour chacune des expressions ci-dessous, essayez d'inférer son type sans vérifier sur le interpréteur. Ensuite, vérifier votre réponse. Si votre réponse n'est pas correcte, essayez d'en comprendre la raison :

1. `\x -> x + 1`
2. `\(x,y) -> g (x,y) where g (x, y) | x == 0 = y | otherwise = 0.5`
3. `\x -> \y -> g (x,y) where g (x,y) | x == 0 = y | otherwise = 0.5`
4. `\x -> \y -> x y`
5. `\x -> \y -> \z -> x (y,z)`
6. `\x -> \(y,z) -> x y z`

Exercice 2. Pour chacun des types suivants, écrivez une expression en Haskell ayant le type (vous n'avez pas droit de copier les expressions de l'exercice antérieur) :

1. `(a, b) -> a`
2. `(a, b) -> b`
3. `Num c => (a, b) -> c`
4. `Num a => (a, a) -> a`
5. `a -> b -> a`
6. `Num a => a -> a -> a`
7. `Num a => (a -> b) -> b`
8. `(Num a, Num b) -> (a -> b) -> b`

Pour tous les exercices qui suivent. N'oubliez pas d'indiquer le type de chaque fonction.

Exercice 3.

1. Écrire en Haskell une fonctionnelle `sommeTermes` qui calcule la somme des $n+1$ premiers termes d'une suite (u_n) de nombres réels qui est donnée comme argument.
2. Écrire en Haskell une expression (une fonction sans argument) qui calcule la somme des 100 premiers entiers naturels non nuls.
3. Écrire en Haskell une expression qui calcule la somme $1-2+3-4+5-6+...+99$. (Vérifier avec l'interpréteur que cette somme est égal à 50).
4. Définir la fonction `inv100` qui à tout nombre réel x associe la somme : $(1-x)^0+(1-x)^1+(1-x)^2+\cdots+(1-x)^{100}$.

Exercice 4.

1. Écrire une fonction `sommeFiltre` qui, étant donné un entier `n` et une fonction `f`, qui à tout entier associe un booléen, renvoie la somme de tous les entiers naturels `p` tel que `p` $\leq$ `n` et `f(p)` est vrai.
2. Un nombre parfait est un entier naturel non nul égal à la moitié de la somme de ses diviseurs. Par exemple, 6 est un nombre parfait car ses diviseurs sont 1, 2, 3 et 6, et $2\times 6 = 1+2+3+6$. Utiliser la fonction `sommeFiltre` écrite précédemment pour écrire la fonction `parfait` qui retourne si un nombre est parfait.
3. Écrire la fonction `parfaits` qui, étant donné un entier naturel `n`, retourne la liste de nombres parfaits de 0 jusqu'à `n`.

Exercice 5.

1. Écrire une fonction appelée `applyn` qui, étant donné un entier naturel `n`, une fonction `f` et une valeur `x` (les arguments sont à donner dans cet ordre), calcule le résultat de `f` appliquée `n` fois sur `x`, c'est à dire : $f^n(x) = f(...(f(x))...)$.
2. Utiliser la fonction `applyn` pour définir une fonction `power` qui calcule x^n, la n-ième puissance de x, où x est un entier, sans utiliser l'opérateur `^`. (Retourner 0 si `n` est inférieur à zéro).

Chapitre 4

Types structurés

4.1 Modules

Pour pouvoir encapuler des fonctions liées à un type structuré, nous pouvons utiliser un module.

Un fichier source Haskell peut contenir la définition d'un seul module. Le nom de base du fichier source doit être identique au nom du module.

Le fichier source d'un module débute avec la déclaration du module. Exemple :

```
1 module MonModule ( Personne (..),
2                    getPrenom,
3                    getAge        ) where
```

Après le nom du module, nous avons la liste des noms exportés entre parenthèses.

Le mot `where` indique que le corps du module suit.

La liste des noms exportés contient les noms qui seront visibles par d'autres modules.

La notation `(..)` indique que le type et tous ses constructeurs sont exportés.
Si la liste de noms exportés est omise, tous les noms sont exportés :

```
1 module ExporteTout where
```

Si la liste est vide, aucun nom n'est exporté (ce qui est rarement utile) :

```
1 module ExporteRien () where
```

Pour utiliser les nom exportés d'un module dans un autre module, il est nécessaire d'importer le premier dans le second :

```
1 module MonModule where
2
3 import AutreModule
```

4.2 Définition de types structurés

Voici la syntaxe de la définition d'un type structuré avec n constructeurs :

```
data <NomDeType> = <NomDeConstructeur1> <expressionDeType1>
                 | <NomDeConstructeur2> <expressionDeType2>
                        ⋮
                 | <NomDeConstructeurN> <expressionDeTypeN>
```

En Haskell, les noms de type, ainsi que les noms de constructeurs de type commencent obligatoirement par une majuscule.

Example 1. Le type des cercles peut être défini ainsi :

```
1 data Cercle = CercleCons ((Double, Double), Double)
```

Le constructeur `CercleCons` est une fonction qui prend en argument un tuple du type `((Double, Double), Doubl` et renvoie un objet du type `Cercle`.

Utilisation :

```
ghci> :t CercleCons
CercleCons :: ((Double, Double), Double) -> Cercle
ghci> c = CercleCons ((1,2), 3)
ghci> :t c
c :: Cercle
```

Affichage

L'affichage pour le type que nous avons créé n'est pas définie :

```
ghci> x = CercleCons ((1,2), 3)
ghci> x
<interactive>:6:1: error:
    . No instance for (Show Cercle) arising from a use of ‘print’
    . In a stmt of an interactive GHCi command: print it
```

Pour qu'une valeur soit affichée, il faut que la fonction `show` soit définie pour cette valeur.

La façon la plus simple (et la seule que nous connaissons pour l'instant) est de laisser Haskell générer cette fonction pour nous :

```
1 data Cercle = CercleCons ((Double, Double), Double)
2     deriving Show
```

Nous avons maintenant :

```
ghci> x = CercleCons ((1,2), 3)
ghci> x
CercleCons ((1.0,2.0),3.0)
```

Example 2. Le constructeur du type des cercles peut aussi être défini comme une fonction d'ordre supérieur :

```
data Cercle = CercleCons (Double, Double) Double
    deriving Show
```

```
ghci> :t CercleCons
CercleCons :: (Double, Double) -> Double -> Cercle
ghci> c = CercleCons (1,2) 3
ghci> c
CercleCons (1.0,2.0) 3.0
```

Dans ce cas, l'application partielle du constructeur `CercleCons` permet d'exprimer une famille de cercles. Par exemple, les cercles centrés à l'origine :

```
ghci> centreOrigine = CercleCons (0,0)
ghci> :t centreOrigine
centreOrigine :: Double -> Cercle
```

Un exemple avec plusieurs constructeurs différents.

Example 3. Les solutions d'une équation du second degré :

```
data Solution = AucuneRacine
              | RacineDouble Double
              | RacinesDistinctes (Double, Double)
                deriving Show
```

Utilisation :

```
f :: (Double, Double, Double) -> Solution
f (a, b, c)
    | delta <  0 = AucuneRacine
    | delta == 0 = RacineDouble ((-b) / (2*a))
    | otherwise  = RacinesDistinctes ( ((-b) + (sqrt delta) / (2*a)),
                                       ((-b) - (sqrt delta) / (2*a)) )
  where delta = (b*b) - (4*a*c)
```

Plusieurs résultats différents de la fonction `f` :

```
ghci> f (1,1,0)
RacinesDistinctes (-0.5,-1.5)
ghci> f (1,0,0)
RacineDouble (-0.0)
ghci> f (1,0,8)
AucuneRacine
```

Ici, `AucuneRacine` peut être vue comme une valeur d'erreur.

Notez donc comment cette technique peut être utilisée pour traiter les erreurs sans recourir à des exceptions.

Le programme ne « plante » jamais !

Division « safe »

Voici un autre exemple :

```
data FailableDouble = Failure
                    | OK Double
                      deriving Show

safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

```
ghci> safeDiv 1 0
Failure
ghci> safeDiv 1 2
OK 0.5
```

Sélecteurs

Pour récupérer les valeurs d'un type structuré, il est nécessaire de définir des sélecteurs.

```
-- Prénom et âge de la personne.
data Personne = PersonneCons String Int
    deriving Show

jean :: Personne
jean = PersonneCons "Jean" 31

pierre :: Personne
pierre = PersonneCons "Pierre" 94

getPrenom :: Personne -> String
getPrenom (PersonneCons p _) = p

getAge :: Personne -> Int
getAge (PersonneCons _ a) = a
```

NB : Le type et le constructeur peuvent avoir le même nom (ici, `Personne`).

D'autres exemples :

```
data Cercle = CercleCons ((Double, Double), Double)
    deriving Show

centre :: Cercle -> (Double, Double)
centre CercleCons c = fst c

rayon :: Cercle -> Double
rayon CercleCons c = snd c
```

Observateurs

La définition des observateurs est similaire :

```
nbSolutions :: Solution -> Int
nbSolutions AucuneRacine = 0
nbSolutions (RacineDouble _) = 1
nbSolutions (RacinesDistinctes _) = 2
```

Test d'égalité

Il est souvent important de pouvoir comparer des valeurs.

Pourtant, à exemple de `show`, l'opérateur `==` n'est pas défini automatiquement pour les types structurés :

```
ghci> x = CercleCons ((1,2),3)
ghci> y = CercleCons ((1,2),3)
ghci> x == y

<interactive>:7:1: error:
    . No instance for (Eq Cercle) arising from a use of ‘==’
    . In the expression: x == y
      In an equation for ‘it’: it = x == y
```

Encore une fois, la façon la plus simple (et la seule que nous connaissons pour l'instant) est de demander à Haskell de le générer pour nous :

```
data Cercle = CercleCons ((Double, Double), Double)
    deriving (Eq, Show)
```

4.3 Catégories de types structurés

Types somme

Un **type somme** est l'union disjointe (ou somme cartésienne) de plusieurs ensembles.

Un type somme possède plusieurs constructeurs de valeurs.

Example 4. Le type `Solution` vu précédemment est un type somme.

```
data Solution = AucuneRacine
              | RacineDouble Double
              | RacinesDistinctes (Double, Double)
                deriving Show
```

Types produit

Un **type produit** est le produit cartésien de plusieurs ensembles.

Un type produit possède un seul constructeur de valeurs. Il est possible d'extraire les valeurs en utilisant des fonctions d'accès ou sélecteurs.

Example 5. Le type `Cercle` vu précédemment est un type produit.

```
data Cercle = CercleCons ((Double, Double), Double)
    deriving (Eq, Show)
```

Type énumérés

Un **type énuméré** est un type structuré dont les constructeurs sont constants (c.-à-d., ils ne prennent pas d'argument).

Example 6.

```
data Couleur = Rouge
             | Jaune
             | Bleu
```

En Haskell, le type `Bool` est un type énuméré, défini comme suit :

```
data Bool = True | False
```

Types récursifs

Un **type récursif** t est un type dont l'un de ses constructeurs prend en argument un objet du type t.

Example 7. Le type de listes d'entiers :

```
data IntListe = Vide
              | Cons Int IntListe
                deriving (Eq, Show)
```

Utilisation :

```
ghci> x = Cons 1 (Cons 2 (Cons 3 Vide))
ghci> :t x
x :: IntListe
ghci> x
Cons 1 (Cons 2 (Cons 3 Vide))
```

Le type de liste pré-définie de Haskell utilise `[]` et `:` au lieu de `Vide` et `Cons`.

Un type structuré peut être **polymorphe**.

Dans ce cas, la définition de type est paramétrée par une ou plusieurs variables de type.

Syntaxe :

```
data <NomDeType> v1 ... vK = <NomDeConstructeur1> <ExpressionDeType1>
                           | <NomDeConstructeur2> <ExpressionDeType2>
                                   ⋮
                           | <NomDeConstructeurN> <ExpressionDeTypeN>
```

Chaque expression de type peut contenir une ou plusieurs variables de type `v1, ..., vK`.

Example 8. Le type de listes polymorphes :

```
data Liste a = Vide
             | Cons a (Liste a)
               deriving (Eq, Show)
```

Utilisation :

```
Prelude> x = Cons 'a' (Cons 'b' (Cons 'c' Vide))
Prelude> y = Cons 1 (Cons 2 (Cons 3 Vide))
Prelude> :t x
x :: Liste Char
Prelude> :t y
y :: Num a => Liste a
```

Exercice : Pour le type listes polymorphes défini précédemment :

1. Écrivez la fonction `len` qui calcule la longueur d'une liste.
2. Écrivez la fonction `conc` qui concatène deux listes.

Longueur :

```
len :: Liste a -> Integer
len Vide = 0
len (Cons _ xs) = 1 + len xs
```

Concaténation :

```
conc :: Liste a -> Liste a -> Liste a
conc Vide ys = ys
conc (Cons x Vide) ys = (Cons x ys)
conc (Cons x xs) ys = (Cons x (conc xs ys))
```

Maybe

Voici un type polymorphe très utile :

```
data Maybe a = Nothing
             | Just a
               deriving (Eq, Ord, Show)
```

Avec ce type, nous pouvons définir des fonctions qui ne plantent jamais (fonctions pures) :

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

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs
```

En effet, le type `Maybe` existe déjà en Haskell !

Pourquoi utiliser `Maybe` et les fonctions « safe » ?

— Les fonctions définies ainsi sont pures (elle ne plantent jamais).
— Le type de la fonction rend évident qu'elle peut retourner une erreur pour certains arguments.
— L'inférence de type du langage oblige que les utilisateurs de ce type de fonction vérifient le retour et traitent les possibles erreurs.

Comment utiliser un `Maybe` ?

Les valeurs du type `Maybe` peuvent être comparés :

```
Prelude> x = Just 0.5
Prelude> y = Just 1
Prelude> x == y
False
Prelude> x < y
True
Prelude> w = Nothing
Prelude> x == w
False
Prelude> x > w
True
```

Mais nous ne pouvons pas faire de calculs avec eux :

```
Prelude> x + y

<interactive>:146:1: error:
    . Non type-variable argument in the constraint: Num (Maybe a)
    ...
```

Nous devons extraire la valeur du type `Maybe` d'abord :

```
1 fromMaybe :: a -> Maybe a -> a
2 fromMaybe d Nothing  = d
3 fromMaybe _ (Just x) = x
```

La fonction `fromMaybe` renvoie la valeur par défaut `d` si le deuxième arguement est `Nothing`.

La fonction renvoie la valeur de `x` si le deuxième argument `Just x`.

Nous pouvons faire, par exemple :

```
Prelude fromMaybe 0 x
0.5
Prelude> (fromMaybe 0 x) + (fromMaybe 0 y)
1.5
Prelude> (fromMaybe 0 x) + (fromMaybe 0 w)
0.5
```

Ici, si l'une des valeurs est `Nothing`, alors nous effectuons le calcul avec `0`.

Bien sûr, la valeur par défaut peut varier selon le programme.

Liste non-vide

Une autre alternative est l'utilisation des listes non-vides.

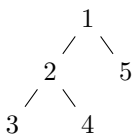
```haskell
data NonEmptyList a = NEL a [a]
    deriving (Eq, Show)

nelHead :: NonEmptyList a -> a
nelHead (NEL x _) = x

nelTail :: NonEmptyList a -> [a]
nelTail (NEL _ xs) = xs

nelToList :: NonEmptyList a -> [a]
nelToList (NEL x xs) = x:xs

listToNel :: [a] -> Maybe (NonEmptyList a)
listToNel []     = Nothing
listToNel (x:xs) = Just (NEL x xs)
```

4.4 Exemple

Nous pouvons définir un arbre binaire ainsi :

```haskell
data Arbre a = Vide
             | Noeud a (Arbre a) (Arbre a)
               deriving Show
```

Exemple :




```haskell
monArbre :: Arbre
monArbre = Noeud 1
               (Noeud 2
                   (Noeud 3 Vide Vide)
                   (Noeud 4 Vide Vide))
               (Noeud 5 Vide Vide)
```

Exercices : Pour le type `Arbre` défini précédemment :

1. Écrire une fonction qui calcule la hauteur de l'arbre.
2. Écrire une fonction qui calcule le nombre de feuilles de l'arbre.
3. Écrire une fonction qui renvoie une liste contenant le parcours infixe de l'arbre.

Hauteur :

```haskell
hauteur :: Arbre a -> Int
hauteur Vide  = 0
hauteur (Noeud _ g d) = 1 + (max (hauteur g) (hauteur d))
```

Nombre de feuilles :

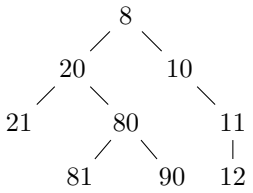
```haskell
nbFeuilles :: Arbre a -> Int
nbFeuilles Vide = 0
nbFeuilles (Noeud _ Vide Vide) = 1
nbFeuilles (Noeud _ g d) = (nbFeuilles g) + (nbFeuilles d)
```

Parcours infixe :

```haskell
parcoursInfixe :: Arbre a -> [a]
parcoursInfixe Vide = []
parcoursInfixe (Noeud n g d) = (parcoursInfixe g) ++
                               [n] ++
                               (parcoursInfixe d)
```

4.5 Exercices

Exercice 1. Un « Heap » (*Tas* en français) est un arbre binaire tel que chaque fils a une valeur supérieure ou égale à celle de son père (voir la figure ci-dessous).



1. Définissez un type polymorphe représentant un Arbre binaire.
2. Écrivez une fonction `isHeap` qui vérifie qu'un arbre binaire est un tas.
3. Écrivez une fonction `insert` qui forme un nouveau tas en insérant un nouvel élément à sa bonne place dans le tas.
4. Écrivez une fonction `fromList` qui, étant donné une liste, renvoie un tas dont les éléments sont ceux de la liste.
5. Écrivez une fonction `toList` qui, étant donné un tas, retourne une liste contenant les éléments du tas triés par ordre croissant.
6. Écrivez une fonction `sort` qui tri une liste en utilisant un tas.

Chapitre 5

Schémas de programme

5.1 Introduction

Introduction

Exercice : Écrivez une fonction qui calcule la somme des éléments d'une liste :

```
somme :: Num a => [a] -> a
somme [] = 0
somme (x:xs) = x + (somme xs)
```

Exercice : Écrivez une fonction qui calcule le produit des éléments d'une liste :

```
produit :: Num a => [a] -> a
produit [] = 1
produit (x:xs) = x * (produit xs)
```

Introduction

Plusieurs fonctions sur les listes se ressemblent.

Nous pouvons dire que ces fonctions suivent un même **schéma**.

Une idée intéressante est donc des les exprimer une fois pour toutes.

Un autre intérêt d'un schéma de programmes est qu'il correspond à un algorithme.

Puisque plusieurs algorithmes sont possibles pour résoudre un même problème, nous pouvons changer d'algorithme en remplaçant un schéma par un autre.

Schémas de programme

Les définitions de fonctions f précédentes, qui prennent une liste l en argument, suivent le même schéma récursif suivant :

— Si l est vide alors le résultat est une valeur a qui ne dépend pas de l (cas de base).

— Sinon, soit l de la forme $x : xs$, alors le résultat est $x \; op \; f(xs)$, où op est une opération binaire (cas récursif).

En Haskell :

```
f [] = a
f (x:xs) = op x (f xs)
```

Note : Ici l'opération binaire *op* est indiquée en forme préfixée.

Schémas de programme

Exercice : Soit le schéma :

```
f [] = a
f (x:xs) = op x (f xs)
```

Complétez le tableau :

f	a	op
somme	0	(+)
produit	1	(*)
longueur	?	?
conc	?	?

Extraction de l'opération binaire

La fonction `somme` s'écrit comme suit en Haskell :

```
somme [] = 0
somme (x:xs) == x + (somme xs)
```

Donc :

- `f` = `somme`
- `a` = `0`
- `op` :
  - Nous avons l'opération : `x + (somme xs)`
  - `x` est l'élément courant : `e`
  - `(somme xs)` est le résultat cumulé : `a`
  - Nous avons `x + (somme xs)` = `e + a`
  - Nous avons `op` = `\e -> \a -> e + a`

Extraction de l'opération binaire

Donc, nous pouvons réécrire la fonction `somme` :

```
somme [] = 0
somme (x:xs) = (\e -> \a -> e + a) x (somme xs)
```

Le cas de la fonction `produit` est analogue.

Extraction de l'opération binaire

La fonction `longueur` s'écrit comme suit en Haskell :

```
longueur [] = 0
longueur (_:xs) = 1 + (longueur xs)
```

Donc :

- `f` = `longueur`
- `a` = `0`
- `op` :
  - Nous avons l'opération : `1 + (longueur xs)`
  - `x` est l'élément courant : `e`
  - `(longueur xs)` est le résultat accumulé : `a`
  - Nous avons `1 + (longueur x)` = `1 + a` (`e` n'est pas utilisé)
  - Nous avons `op` = `\e -> \a -> 1 + a`

Extraction de l'opération binaire

Donc, nous pouvons réécrire la fonction `longueur` :

```
longueur [] = 0
longueur (x:xs) = (\e -> \a -> 1 + a) x (longueur xs)
```

Extraction de l'opération binaire

La fonction `conc` s'écrit comme suit en Haskell :

```
conc [] ys = ys
conc (x:xs) ys = x : (conc xs ys)
```

Donc :

- `f` = `conc`
- `a` = `ys`
- `op` :
  - Nous avons l'opération : `x : (conc xs ys)`
  - `x` est l'élément courant : `e`
  - `(conc xs ys)` est le résultat accumulé : `a`
  - Nous avons `x : (conc xs ys)` = `e : a`
  - Nous avons `op` = `\e -> \a -> e : a`

Extraction de l'opération binaire

Donc, nous pouvons réécrire la fonction `conc` :

```
conc [] ys = ys
conc (x:xs) ys = (\e -> \a -> e : a) x (conc xs ys)
```

Schémas de programme

Réponse de l'exercice : Soit le schéma :

```
f [] = a
f (x:xs) = op x (f xs)
```

Complétez le tableau :

f	a	op
somme	0	(+)
produit	1	(*)
longueur	0	\e -> \a -> 1 + a
conc	ys	(:)

5.2 Schéma réduction

Le schéma réduction

Le schéma que nous venons de voir s'appelle **réduction**.

Nous pouvons le capturer en définissant une fonction d'ordre supérieur.

La fonction `reduce` prend en arguments `op`, `a` et `l` et calcule l'opération désirée :

```
reduce op a [] = a
reduce op a (x:xs) = op x (reduce op a xs)
```

```
ghci> reduce (\e -> \a -> e + a) 0 [1, 2, 3]
6
```

Nous appelons cette technique « abstraction ».

Elle peut être utilisée pour définir d'autres schémas de programme.

Le schéma réduction

Haskell possède aussi la fonction `(+)` qui est la version préfixée et curryfiée de l'opérateur d'addition.

```
ghci> (+) 1 2
3
```

Et même chose pour `(*)`, `(/)`, `(:)`, `(&&)`, `(||)`, etc. Donc, nous avons :

```
ghci> reduce (+) 0 [1, 2, 3]
6
ghci> reduce (*) 1 [3, 4, 5]
60
ghci> reduce (:) [4, 5, 6] [1, 2, 3]
[1, 2, 3, 4, 5, 6]
ghci> reduce (\_ -> \a -> 1 + a) 0 [1, 2, 3]
3
```

Le schéma réduction

Nous pouvons donc définir les fonctions vues précédemment comme suit :

```
somme = reduce (+) 0
produit = reduce (*) 1
conc = reduce (:)
longueur = reduce (\_ -> \a -> 1 + a) 0
```

Utilisation :

```
ghci> somme [1, 2, 3]
6
ghci> produit [3, 4, 5]
60
ghci> longueur [1, 2, 3]
3
ghci> conc [4, 5, 6] [1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

Pour information, en Haskell :

— `somme` = `sum`

— `produit` = `product`

— `longueur` = `length`

— `conc` = `(++)`

Exercice

Exercice : Quelle est le type de `reduce` ?

Réponse : `(t1 -> t2 -> t2) -> t2 -> [t1] -> t2`

5.3 Schémas plier

Les schémas plier

Le résultat du schéma réduction sur une liste $l = [e_1, e_2, \ldots, e_n]$ est :

$$e_1 \ op \ (e_2 \ op \ \ldots (e_n \ op \ a) \ldots)$$

Nous calculons donc d'abord $e_n \ op \ a$, ensuite $(e_{n-1} \ op \ (e_n \ op \ a))$, etc.

Mais nous pourrions aussi faire :

$$(\ldots ((a \ op \ e_1) \ op \ e_2) \ldots) \ op \ e_n$$

C'est-à-dire, calculer d'abord $a \ op \ e_1$, ensuite $(a \ op \ e_1) \ op \ e_2$, etc.

Le schéma plier à droite

Le premier schéma, où nous utilisons les éléments dans l'ordre inverse, s'appelle aussi **plier à droite** et correspond exactement au schéma réduction que nous avons déjà vu.

```
foldr _  a [] = a
foldr op a (x:xs) = op x (foldr op a xs)
```

Par exemple :

```
foldr (+) 0 [1, 2, 3]
= 1 + foldr (+) 0 [2, 3]
= 1 + (2 + foldr (+) 0 [3])
= 1 + (2 + (3 + foldr (+) 0 []))
= 1 + (2 + (3 + 0))
= 6
```

Le schéma plier à gauche

L'autre schéma, où on utilise les éléments dans l'ordre s'appelle **plier à gauche** et correspond à un schéma différent du schéma réduction.

Exercice : Écrivez une fonction d'ordre supérieur qui correspond au schéma **plier à gauche**.
Réponse :

```
foldl _  a [] = a
foldl op a (x:xs) = foldl op (op a x) xs
```

Par exemple :

```
foldl (+) 0 [1, 2, 3]
= foldl (+) (0 + 1) [2, 3]
= foldl (+) ((0 + 1) + 2) [3]
= foldl (+) (((0 + 1) + 2) + 3) []
= (((0 + 1) + 2) + 3)
= 6
```

Les schémas plier

La fonction `foldl` est récursive terminale et donc plus efficace que `foldr`.

Exercice : Écrivez la fonction `longueur` en utilisant `foldl`.

Nous voulons le comportement suivant :

```
foldl 0 [1, 2, 3]
= foldl (0 + 1) [2, 3]
= foldl ((0 + 1) + 1) [3]
= foldl (((0 + 1) + 1) + 1) []
= (((0 + 1) + 1) + 1)
= 3
```

Donc, nous devons trouver la fonction `op` correspondante. Notez que l'accumulation est maintenant à gauche. Donc :

```
op = \a -> \_ -> a + 1
```

```
1 longueur = foldl (\a -> \_ -> a + 1) 0
```

Les schémas plier

Notez pourtant que nous ne pouvons pas utiliser `foldl` pour implémenter `conc`.

Par exemple, le déroulement de `conc` avec `foldl` aurais été *(ceci ne fonctionne pas)* :

```
foldl [4,5,6] [1,2,3]
= foldl (:) ([4,5,6] : 1) [2,3]
= foldl (:) (([4,5,6] : 1) : 2) [3]
= foldl (:) ((([4,5,6] : 1) : 2) : 3) []
= ((([4,5,6] : 1) : 2) : 3) : []
= ??
```

5.4 Schéma map

Le schéma map

Le schéma **map** consiste à appliquer une même fonction à tous les éléments d'une liste.

Par exemple, soit la liste $[e_1, e_2, \ldots, e_n]$ et la fonction f, le résultat de map est la liste $[f(e_1), f(e_2), \ldots, f(e_n)]$.

Exercice : Écrivez la fonction d'ordre supérieur qui correspond au schéma map :

Réponse :

```
1 map _ [] = []
2 map f (x:xs) = f x : (map f xs)
```

Le schéma map

Exercice : Écrivez la fonction qui prend une liste d'entiers en argument et retourne la même liste où les entiers sont multipliés par 2.

Exemple d'utilisation :

```
ghci> fois2 [1, 3, 4]
[2, 6, 8]
```

Réponse :

```
1 fois2 = map ((*) 2) l
```

Le schéma map

Regardez bien le déroulement du schéma map. Par exemple :

```
map ((*) 2) [1, 2, 3]
= 2 : (map ((*) 2) [2, 3])
= 2 : (4 : (map ((*) 2) [3]))
= 2 : (4 : (6 : map ((*) 2) []))
= 2 : (4 : (6 : []))
```

Cela ne vous rappelle pas quelque chose que nous avons déjà vu ?

Exercice : Re-écrivez la fonction qui correspond au schéma map en utilisant le schéma `foldr`.

Réponse :

```
1 map f = foldr (\e -> \a -> (f e) : a) []
```

Le schéma map2

Le schéma **map2** généralise le schéma map en utilisant deux listes comme argument.

Soit les listes $[a_1, \ldots, a_n]$ et $[b_1, \ldots, b_n]$ et la fonction f. Le résultat de `map2` est la liste $[f(a_1, b_1), \ldots, f(a_n, b_n)]$.

Exercice : Écrivez une fonction d'ordre supérieur qui correspond à map2 :

```
1 map2 _ [] _ = []
2 map2 _ _ [] = []
3 map2 f (x:xs) (y:ys) = f x y : (map2 f xs ys)
```

Le schéma map2 est aussi appelé zip.

Évidement, il est aussi possible de généraliser le schéma map à n listes.

5.5 Schéma filtrer

Soit la liste $[a_1, \ldots, a_n]$ et la fonction f. Le résultat de `filter` est une nouvelle liste contenant uniquement les éléments a_i tél que $f(a_i)$ est vrai.

Exercice : Écrivez une fonction d'ordre supérieur qui correspond à `filter`.

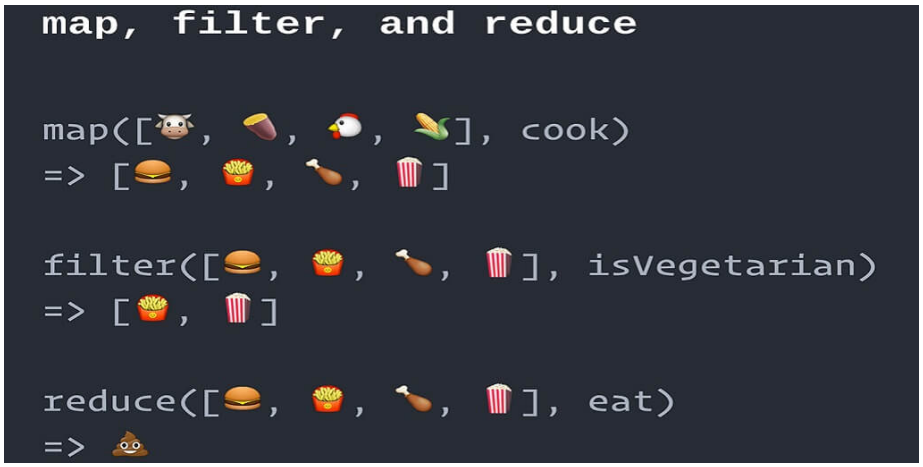
Réponse :

```
1 filter _ [] = []
2 filter f (x:xs)
3     | f x       = x : (filter f xs)
4     | otherwise = filter f xs
```

Exemples :

```
ghci> even = filter (\e -> e `mod` 2 == 0)
ghci> odd = filter (\e -> e `mod` 2 /= 0)
```

Résumé



5.6 Exercices

Exercice 1. Écrivez le schéma `myZipWith3` qui, pour une fonction `f` et trois listes `[a1, ..., an]`, `[b1, ..., bn]`, et `[c1, ..., cn]` passées en argument, renvoie la liste `[(f a1 b1 c1), ..., (f an bn c2)]`. Exemple :

```
myZipWith3 (\x -> \y -> \z -> x+y+z) [1,2] [3,4] [7,9] == [11,15]
```

Exercice 2. Utilisez le schéma `foldr` pour écrire le schéma `myFilter` qui, pour une condition et une liste passées en argument, retourne la sous-liste des éléments qui satisfont la condition. Exemple :

```
myFilter even [1,2,3,4] == [2,4]
```

Exercice 3. Utilisez le schéma `foldl` pour écrire le schéma `myAll` qui, pour une condition (fonction booléenne) et une liste passées en argument, revoie `True` si tous les éléments de la liste satisfont la condition et `False` sinon. Exemples :

```
myAll even []
myAll even [4,6,2]
not (myAll even [4,3,2])
```

Exercice 4. Utilisez le schéma `foldl` pour écrire le schéma `myAny` qui, pour une condition (fonction booléenne) et une liste passées en argument, renvoie `True` s'il existe un élément dans la liste qui satisfait la condition et `False` sinon. Exemples :

```
not (myAny even [])
not (myAny even [3,7,1])
myAny even [4,3,2]
```

Exercice 5. Maybe.

1. Écrivez la fonction `catMaybes` qui, pour une liste de type `[Maybe a]` passée en argument, renvoie une liste de type `[a]` contenant uniquement les éléments `Just`. Exemple :

   ```
   catMaybes [Just 1, Nothing, Just 2, Nothing] == [1,2]
   ```

2. Écrivez la fonction `mapMaybe` qui, pour une fonction `f` de type `a -> Maybe b` et une liste `[a1, ..., an]` passées en argument, renvoie la liste `[(f a1), ... (f an)]` contenant uniquement les éléments `(f ai)` différents de `Nothing`. Exemple :

   ```
   mapMaybe safeHead [[],[1],[],[2]] == [1,2]
   ```

Exercice 6. Tri de listes.

1. Écrivez la fonction `insere` qui, étant donnée une valeur e et une liste l ordonnée, insère e dans l de façon à ce que l reste ordonnée.
2. Utilisez le schéma `foldr` pour écrire la fonction `sort`, qui ordonne une liste passée en argument

Exercice 7. Utilisez le schéma `map` pour écrire la fonction `prefixes` qui renvoie la liste de tous les préfixes d'une liste. Exemple :

```
prefixes [1,2,3] == [[],[1],[1,2],[1,2,3]]
```

Exercice 8. On appelle sous-liste d'une liste `l`, une liste obtenue à partir de `l` en enlevant un nombre quelconque d'éléments et en conservant l'ordre. Utiliser le schéma `map` pour écrire une fonction `sousListes` qui renvoie la liste des toutes les sous-listes d'une liste.

```
sousListes [1,2,3] == [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
```

Exercice 9. Générateurs de n-uplets.

1. Écrire la fonction `partition2` qui, étant donnée un entier naturel n, retourne la liste de toutes les paires (x, y) telles que $n = x + y$. Exemple :

   ```
   partition2 2 == [(0,2),(1,1),(2,0)]
   ```

2. Écrire la fonction `partition3` qui, étant donné un entier naturel n, retourne la liste de touts les triplets (x, y, z) tels que $n = x + y + z$. Exemple :

   ```
   partition3 2 == [(0,0,2),(0,1,1),(0,2,0),(1,1,0),(1,0,1),(2,0,0)]
   ```

Chapitre 6

Classes de type

6.1 Polymorphisme paramétrique

Le polymorphisme de Hakell est dit **paramétrique**.

Essentiellement, cela veut dire que les fonctions polymorphes doivent fonctionner de façon *uniforme* pour tout type d'entrée.

Cela a des implications sur les fonctions.
Par exemple, quelles sont les fonctions qui peuvent avoir le type `a -> a -> a` ?

Est-ce que cette définition fonctionne ?

```
1 f :: a -> a -> a
2 f x y = x && y
```

En effet, non :

```
<interactive>:5:9: error:
    . Couldn't match expected type 'Bool' with actual type 'a'
      'a' is a rigid type variable bound by
        the type signature for:
          f :: forall a. a -> a -> a
        at <interactive>:4:1-12
    . In the first argument of '(&&)', namely 'x'
      In the expression: x && y
      In an equation for 'f': f x y = x && y
      ...
```

La raison est que l'utilisateur de la fonction polymorphe est celui qui choisi son type.

Dans l'exemple, le programmeur a utilisé l'opérateur `&&` qui doit être utilisé avec un type spécifique, `Bool`.

Pourtant, le type `a -> a -> a` est une « promesse » que la fonction accepte n'importe quelle type.

Donc, ces deux choses ne vont pas ensemble.

Pour essayer de contourner cela, nous pouvons imaginer une sorte de vérification du type des arguments. Plus au moins comme suit *(ceci ne fonctionne pas)* :

```
f :: a -> a -> a
f a1 a2
    | instanceof a1 == Int  = a1 + a2
    | instanceof a1 == Bool = a1 && a2
    | otherwise             = a1
```

Pourtant, Haskell n'a pas une fonction `instanceof` (comme Java).

L'une des raisons est que le compilateur supprime les types du code, une fois ceci compilé.

En effet, les types polymorphes de Haskell ressemblent plus au « generics » de Java.
Donc, quelles sont les fonctions qui peuvent avoir le type `a -> a -> a` ?

En effet, il en a que deux !!

```
f1 :: a -> a -> a
f1 x y = x

f2 :: a -> a -> a
f2 x y = y
```

Donc, les types de Haskell sont très restrictifs !

Un autre point de vue

Les concepteurs de Haskell ne voient pas ceci comme une restriction, mais plutôt comme une *garantie.*

Les types de Haskell donnent des garanties sur le comportement des fonctions.

Mais, comment est-il possible de faire, par exemple, l'addition sur plusieurs types comme `Int`, `Integer` et `Double` ?

La réponse vient des « classes de types ».

Par exemple, notez le `Num` dans le type de l'opérateur `(+)` :

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

Notez aussi le `Eq` dans le type de l'opérateur `(==)` :

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

6.2 Classes de types

Classes de types

Comme nous avons déjà vu, `Num` et `Eq` sont des classes de types.

Intuitivement, elles correspondent à des ensembles de types définissant des opérations.

Les fonctions polymorphes fonctionnent uniquement pour les types qui sont des instances des classes de types correspondantes.

Par exemple, voici une partie de la définition de la classe `Eq` :

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

Tout type `a` qui se veut une instance de `Eq` doit implémenter les opérateurs `(==)` et `(/=)`.

Quand une méthode d'une classe est utilisée, le compilateur utilise l'inférence de types pour déterminer quelle implémentation de la méthode sera choisie.

Si nous comparons encore une fois avec Java, ceci fonctionne plutôt comme une surcharge.

6.3 Instances

Voici comment nous pouvons construire un type qui est une instance de `Eq` :

```
data Truc = F Int | G Char

instance Eq Truc where
    (F i1) == (F i2) = i1 == i2
    (G c1) == (G c2) = c1 == c2
    _ == _ = False

    truc1 /= truc2 = not (truc1 == truc2)
```

```
Prelude> x = F 1
Prelude> :t x
x :: Truc
Prelude> y = G 'a'
Prelude> :t y
y :: Truc
Prelude> x == y
False
```

En effet, nous n'avons pas besoin de définir les deux opérateurs `(==)` et `(/=)`, parce que la classe `Eq` contient les définitions suivantes :

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Ceci veut dire qu'il suffit de définir un seul des deux opérateurs.

Mais attention, si nous ne définissons aucun des deux, nous avons une boucle infinie.

Classes spéciales

La classe `Eq` est une des classes dites spéciales en Haskell.

Pour ces classes, le compilateur est capable de générer des instances automatiquement avec le mot clé `deriving` :

```
1 data Truc' = F Int | G' Char
2     deriving (Eq, Ord, Show)
```

Voici une liste de classes pour lesquelles Haskell peut générer du code automatiquement :

Bounded Définit `minBound` et `maxBound` comme les valeurs minimum et le maximum du type.

Eq Définit les opérations d'égalité `(==)` et inégalité `(/=)`.

Enum Uniquement pour les énumérations. Permet la création des listes en utilisant la syntaxe `[x .. y]`.

Ord Définit les opérateurs de comparaison : `<`, `<=`, `>`, `>=`, `min` et `max`.

Show Définit la fonction `show`, qui convertit une valeur en `String`.

Read Définit la fonction `read`, qui convertit une `String` en une valeur.

Read

La fonction la plus utile de la classe `Read` est `read`. Voici son type :

```
Prelude> :t read
read :: Read a => String -> a
```

Utilisation :

```
Prelude> (read "1.5")::Double
1.5
```

Notez que nous avons donnée le type explicite `Double` à l'expression ci-dessus.

Autrement, le compilateur doit deviner le type de l'expression et il va se tromper souvent :

```
Prelude> read "1.5"
*** Exception: Prelude.read: no parse
```

Dans certains cas, plus d'un type est possible :

```
Prelude> (read "1")::Integer
1
Prelude> (read "1")::Double
1.0
```

6.4 Exemple

Maintenant, nous allons créer notre propre classe de types.

L'idée est de créer une classe de types `Listables` dont les instances peuvent être convertis en liste de `Int`.

```
class Listable a where
    toList :: a -> [Int]
```

Voyons quelle est le type de `toList` :

```
Prelude> :t toList
toList :: Listable a => a -> [Int]
```

Le type de toute fonction qui utilise `toList` a la contrainte de type `Listable` :

```
sumL x = sum (toList x)
truc x y = sum (toList x) == sum (toList y) || x < y
```

```
*Main> :t sumL
sumL :: Listable a => a -> Int
*Main> :t truc
truc :: (Listable a, Ord a) => a -> a -> Bool
```

Nous pouvons maintenant définir quelques instances de `Listable` :

```
instance Listable Int where
    toList x = [x]

instance Listable Bool where
    toList True = [1]
    toList False = [0]

instance Listable [Int] where
    toList = id
```

```
*Main> toList (1::Int)
[1]
*Main> toList True
[1]
*Main> toList [1::Int]
[1]
```

Il est possible de créer des instances plus compliquées.

```
{-# LANGUAGE FlexibleInstances #-}

data Tree a = Empty | Node a (Tree a) (Tree a)
    deriving (Eq, Show)

```

```
instance Listable (Tree Int) where
    toList Empty        = []
    toList (Node x l r) = toList l ++ [x] ++ toList r
```

(L'ajout du *pragma* `FlexibleInstances` est nécessaire dans ce cas.)

Utilisation :

```
*Main> toList (Node (1::Int) (Node (2::Int) Empty Empty)
              (Node (3::Int) Empty Empty))
[2,1,3]
```

Un autre exemple un peu plus complexe utilisant `Listable` :

```
instance (Listable a, Listable b) => Listable (a,b) where
    toList (x,y) = toList x ++ toList y
```

Dans l'exemple, un pair `(a,b)` est une instance de `Listable` si les types `a` et `b` le sont aussi.

Notez que cette version de la fonction `toList` *n'est pas récursive*, car elle appelle d'autres versions de la fonction `toList`.

6.5 Héritage

Une classe de types peut hériter d'une autre.

Voici une partie de la définition de la classe `Ord` :

```
class (Eq a) => Ord a where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

Cela veut dire que tout type qui est une instance de `Ord` est aussi une instance de `Eq`.

Donc, une instance de `Ord` doit également implémenter les opérateurs `(==)` et `(/=)`.

Une classe peut hériter de plusieurs autres classes :

```
class (Num a, Ord a) => Real a where
    ...
```

(Nous avons déjà vu le diagramme d'héritage de Haskell dans la Partie 3 du cours).

Conseils

Il est une erreur de penser que chaque fonctionnalité possiblement généralisable nécessite une classe.

Par exemple, s'il existe une seule instance d'une classe, alors implémentez la fonctionnalité en question dans le type. La classe n'a pas besoin d'être.

Une classe de types est bien utilisée quand elle possède plusieurs instances, et que nous voulons que les utilisateurs ne soient pas obligés de se préoccuper avec les différentes implémentations pour chaque type.

L'exemple le plus typique est `Show`, qui est très générale et possède un grand nombre d'instances.

Chapitre 7

Monades, entrées et sorties

7.1 Introduction

Introduction

Nous savons comment appliquer une fonction à une valeur :

```
ghci> (+3) 2
5
```

Mais, quand il s'agit d'une valeur dans un *contexte*, le résultat dépend du contexte. Par exemple :

— Appliquer `(+3)` à `Just 2` pour obtenir `Just 5`.

— Appliquer `(+3)` à `[2,3]` pour obtenir `[5,6]`.

L'utilisation des Foncteurs, Applicatifs et Monades permettent de simplifier ce type de tâche.

Une bonne partie de ce cours a été inspiré de l'adresse suivante : `https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

7.2 Foncteurs

Foncteurs

La fonction `fmap` permet d'appliquer une fonction à une valeur du type `Maybe` :

```
ghci> fmap (+3) (Just 2)
Just 5
```

Mais aussi à une valeur du type de listes :

```
ghci> fmap (+3) [2,3]
[5,6]
```

Comment cela fonctionne ?

Voici son type :

```
ghci> :type fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

Foncteur est une classe de types :

```
class Functor f where
    fmap :: fmap (a -> b) -> f a -> f b
```

La fonction `fmap` prend une fonction et un foncteur en arguments. Elle applique la fonction au foncteur et renvoie le résultat comme un foncteur.

Le type `Maybe` est un foncteur :

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing  = Nothing
```

La fonction `fmap` appliquée à `Just x` applique la fonction passée en argument à `x`, et `fmap` appliquée à `Nothing` renvoie toujours `Nothing` :

```
ghci> fmap (+3) Just 2
Just 5
ghci> fmap (+3) Nothing
Nothing
```

Nous avons vu que `fmap` peut être utilisée avec le type de listes.

Cela veut dire que le type de listes est une instance de foncteur aussi :

```
instance Functor [] where
    fmap = map
```

En effet, le type fonction est aussi une instance de foncteur :

```
ghci> f = fmap (+3) (+3)
ghci> f 2
8
```

Dans ce cas, `fmap` est la composition des fonctions :

```
instance Functor ((->) r) where
    fmap f g = f . g
```

Pour information, la fonction `fmap` possède comme synonyme l'opérateur `<$>` :

```
ghci> g = (+3) <$> (+3)
ghci> g 2
8
```

En effet, la classe foncteurs a plusieurs instances, notamment :

— `Either`
— `[]`
— `Maybe`
— `IO`
— `(->)`
— `(,)`

7.3 Applicatifs

Applicatifs

Maintenant, supposez que nous avons une fonction dans un contexte. Par exemple :

```
ghci> f = Just (+3)
ghci> :t f
f :: Num a => Maybe (a -> a)
```

Ici, la fonction `Just (+3)` est du type « maybe fonction ».

L'opérateur `<*>` permet d'utiliser cette fonction dans son contexte :

```
ghci> f <*> (Just 2)
Just 5
```

Ici, l'opérateur `<*>` applique la « maybe fonction » à un « maybe entier » et renvoie un « maybe entier ».

Voici le type de cet opérateur :

```
ghci> :t (<*>)
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

La classe des applicatifs permet d'utiliser une fonction dans un contexte à des arguments dans le même contexte.

```
1 class Functor f => Applicative (f :: * -> *) where
2   (<*>) :: f (a -> b) -> f a -> f b
```

Voici un exemple plus compliqué :

```
ghci> [(*2), (+1)] <*> [1,2,3]
[2,4,6,2,3,4]
```

Chaque fonction dans la première liste est appliquée à chaque élément de la deuxième liste. Les résultats sont tous renvoyés dans une liste.

Foncteurs vs. applicatifs

Supposez que nous voulons appliquer une fonction à deux arguments, chacun dans son contexte.

Par exemple, nous voulons faire la somme de deux maybe entiers :

```
ghci> fmap (+) (Just 2) (Just 3)

<interactive>:33:1: error:
...
```

`fmap` toute seule ne peux pas nous aider parce qu'elle ne prends qu'une fonction à un argument.

Mais il est possible d'exécuter cette tâche avec l'aide de l'opérateur `<*>` :

```
ghci> fmap (+) (Just 2) <*> (Just 3)
Just 5
```

Ici, `fmap (+) (Just 2)` génère `Just (+2)`. Ensuite, ceci est appliqué à `Just 3`, ce qui renvoie `Just 5`.

7.4 Monades

Monades

Soit la fonction :

```
1 half :: Int -> Maybe Int
2 half x
3     | even x    = Just (x `div` 2)
4     | otherwise = Nothing
```

Nous pouvons appliquer `half` à `2` et obtenir `Just 1`, mais nous ne pouvons pas l'appliquer à `Just 2`.

Encore une fois, Haskell nous fourni un opérateur que le fait pour nous :

```
ghci> Just 2 >>= half
Just 1
ghci> Just 3 >>= half
Nothing
```

L'opérateur `>>=` s'appelle *bind*. Voici son type :

```
ghci> :type (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

L'opérateur bind prend une monade et une fonction qui renvoie une monade en arguments et applique la fonction au premier argument.

Monades

Le terme **monade** est issue de la théorie des catégories en mathématiques.
Il s'agit d'une notion abstraite mais qui possède des nombreuses applications.

Intuitivement, une monade est un type à l'intérieur duquel nous pouvons injecter une valeur et composer des séquences d'actions.

En Haskell, nous pouvons définir une monade comme une classe de types :

```
class Applicative m => Monad (m :: * -> *) where
    return :: a -> m a                    -- Injection.
    (>>=) :: m a -> (a -> m b) -> m b     -- Chain (bind).
    (>>) :: m a -> m b                    -- Chain (then).
    x >> f = x >>= \_ -> f
    fail :: String -> m a                 -- Exécuté en cas d'erreur.
```

La fonction `return` permet d'injecter une valeur dans la monade.

Attention : La fonction `return` est une fonction comme une autre en Haskell. Cela ne contrôle absolument pas le flux d'exécution du programme.

L'opérateur `>>=`, appelé *bind*, permet de chaîner des actions.

L'opérateur `>>`, appelé *then*, permet aussi e chaîner des action, mais il ignore le résultat de la dernière opération.

Ce dernier opérateur n'a pas besoin d'être défini par les instances de `Monad`, car il a une définition par défaut en termes de `>>=`.

De plus, le lois suivantes doivent être satisfaites :

```
m >>= return    = m                -- Composition à droite par return.
return x >>= f  = f x              -- Composition à gauche par return.
(m >>= f) >>= g = m >>= (\x -> f x >>= g)           -- Associativité.
```

Par exemple, le type `Maybe` est une monade qui permet de représenter une suite de calculs :

```
instance Monad Maybe where
    return :: a -> Maybe a
    return x = Just x

    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= f = Nothing
    Just x  >>= f = f x

    fail :: String -> Maybe a
    fail _ = Nothing
```

La monade Maybe

Par exemple :

```
reverseM :: String -> Maybe String
reverseM = return . reverse
```

```
3
4 v1 :: Maybe String
5 v1 = return "hello"
6
7 v2 :: Maybe String
8 v2 = Nothing
```

```
ghci> v1 >>= reverseM
Just "olleh"
ghci> v2 >>= reverseM
Nothing
ghci> return "hello" >>= reverseM >>= safeHead
Just 'o'
ghci> return "" >>= reverseM >>= safeHead
Nothing
```

Haskell possède une fonction très pratique appelée `lookup` :

```
ghci> :type lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Par exemple, soit la liste associative :

```
1 phonebook :: [(String, String)]
2 phonebook = [ ("Bob",   "01788 665242"),
3               ("Fred",  "01624 556442"),
4               ("Alice", "01889 985333"),
5               ("Jane",  "01732 187565") ]
```

`lookup` permet de « chercher » des éléments dans la liste :

```
ghci> lookup "Bob" phonebook
Just 01788 665242
ghci> lookup "Machin" phonebook
Nothing
```

Maintenant, imaginez que nous avons plusieurs listes associatives :

```
1  phonebook :: [(String, String)]
2  phonebook = [ ("Bob",   "01788 665242"),
3                ("Fred",  "01624 556442"),
4                ("Alice", "01889 985333"),
5                ("Jane",  "01732 187565") ]
6
7  regNumber :: [(String, String)]
8  regNumber = [ ("01788 665242", "1"),
9                ("01624 556442", "2"),
10               ("01889 985333", "3"),
11               ("01732 187565", "4") ]
12
13 taxDatabase :: [(String, String)]
```

```
taxDatabase = [("1", 11.00),
               ("2", 22.50),
               ("3", 33.33) ]
```

Notre tâche est celle d'écrire une fonction qui trouve l'entrée correspondante à un nom donnée dans la liste `taxDatabase`.

Mais il faut retourner `Nothing` si la donnée est introuvable dans une de listes.
D'abord, il faut trouver le nom dans `phonebook` :

```
getPhoneNumber :: String -> Maybe String
getPhoneNumber name = lookup name phonebook
```

Ensuite, il faut trouver le numéro de téléphone dans `regNumber`.

Pour cela, nous avons d'abord besoin de « sortir » le numéro du `Maybe` :

```
fromMaybe :: a -> Maybe a -> a
fromMaybe x Nothing  = x
fromMaybe _ (Just y) = y
```

Trouver le numéro de téléphone dans `regNumber` :

```
getRegNumber :: String -> Maybe String
getRegNumber name =
    | number == Nothing = Nothing
    | otherwise         = lookup (fromMaybe "" number) regNumber
  where
    number = getPhoneNumber name
```

Maintenant, il faut trouver le numéro d'enregistrement dans `taxDatabase` :

```
getTaxOwed :: String -> Maybe Double
getTaxOwed name =
    | reg == Nothing = Nothing
    | otherwise      = lookup (fromMaybe "" reg) taxDatabase
  where
    reg = getRegNumber name
```

Vous avez certainement remarqué un schéma qui se répète.
En plus, comparée à ce que nous aurions écrit dans un langage impératif, le code en Haskell semble excessivement long :

```
double get_tax_owed( string name ) {
    x = lookup( name,  phonebook );
    if( not x ) return 0;
    y = lookup( x, govDatabase );
    if( not y ) return 0;
    z = lookup( y, taxDatabase );
    return z;
}
```

Chaînage d'actions

Nous pouvons généraliser le schéma qui se répète et améliorer notre code en Haskell :

```
chain :: Maybe a -> (a -> Maybe b) -> Maybe b
chain Nothing _  = Nothing
chain (Just x) f = f x

getRegNumber' :: String -> Maybe String
getRegNumber' x = chain (lookup x phonebook)
                        (\y -> lookup y govDatabase)

getTaxOwed' :: String -> Maybe Double
getTaxOwed' x = chain (getRegNumber' x) (\y -> lookup y taxDatabase)
```

En effet, la fonction `getTaxOwed` peut être encore plus simple :

```
getTaxOwed'' :: String -> Maybe Double
getTaxOwed'' = \x ->
    lookup x phonebook `chain` \y ->
    lookup y govDatabase `chain` \z ->
    lookup z taxDatabase
```

La dernière version de `getTaxOwed` ressemble beaucoup à sa version impérative, car elle donne l'idée des actions faites en séquence.

En effet, Haskell contient déjà la fonction `chain`.

En Haskell, la fonction `chain` est notée `>>=`, et elle est appelée *bind* :

```
getTaxOwed''' :: String -> Maybe Double
getTaxOwed''' = \x ->
    lookup x phonebook >>= \y ->
    lookup y govDatabase >>= \z ->
    lookup z taxDatabase
```

Notation do

La notation `do` permet de définir une séquence d'actions sur une forme volontairement proche des suites d'instructions des langages impératifs :

```
v3 :: Maybe Char
v3 = do x1 <- v1              -- Extrait "hello" de Just "hello".
        x2 <- reverseM x1     -- Applique et extrait le résultat.
        safeHead x2           -- Produit le résultat final.
```

Nous pouvons même utiliser point virgule et accolades !

```
v4 :: Maybe Char
v4 = do {
    let x1 = "" ;          -- Défini une variable locale.
    x2 <- reverseM x1 ;
    safeHead x2
}
```

```
ghci> v3
Just 'o'
ghci> v4
Nothing
```

notation do	**notation standard**
`do act`	`act`
`do act` `  suite`	`act >>` `do suite`

`do motif <- act`
`  suite`
	`let f motif = do suite`
	`  f _  = fail "..."`
	`in act >>= f`

`do let val1 = expr1`
`  ...`
`  valM = exprM`
`  suite`
	`let val1 = expr1`
	`  ...`
	`  valM = exprM`
	`in do suite`

Exemple

La définition :

```
v3 = do x1 <- v1              -- Extrait "hello" de Just "Hello".
       x2 <- reverseM x1      -- Applique et extrait le résultat.
       safeHead x2            -- Produit le résultat final.
```

sera traduite comme :

```
v3 = let f1 Just x1 = do x2 <- reverseM x1
                         safeHead x2
         f1 _  = fail "..."
     in v1 >>= f1
```

qui sera traduite comme :

```
v3 = let f1 x1 =
         let f2 Just x2 = do safeHead x2
             f2 _       = fail "..."
         in reverseM x1 >>= f2
         f1 _  = fail "..."
      in v1 >>= f1
```

qui sera traduite comme :

```
v3 = let f1 x1 =
         let f2 Just x2 = safeHead x2
```

```
3               f2 _        = fail "..."
4           in reverseM x1 >>= f2
5           f1 _  = fail "..."
6       in v1 >>= f1
```

Notez que cette dernière est équivalente à :

```
1 v3 = v1 >>= reverseM >>= safeHead
```

La monade Liste

Le type `[]` est aussi une monade !

Voici comment l'injection et le bind sont implémentés pour les listes :

```
1 instance Monad [] where
2     return :: a -> [a]
3     return x = [x]
4
5     (>>=) :: [a] -> (a -> [a]) -> [a]
6     xs >>= f = concat (map f xs)
```

L'injection injecte un élément dans une liste.

Le bind, exécute la fonction `f` pour chaque élément de la liste.

Exemple

```
1 fois2M :: Int -> [Int]
2 fois2M x = return (x*2)
3
4 plus1M :: Int -> [Int]
5 plus1M x = return (x+1)
```

```
ghci> [1..4] >>= fois2M >>= plus1M
[3,5,7,9]
```

7.5 Entrées et sorties

Entrées et sorties

Haskell est pure. Cela veut dire principalement :

— Les fonctions ne peuvent pas dépendre des entités extérieures. Elles doivent dépendre uniquement de leurs arguments. Une fonction appelée avec les mêmes arguments doit toujours retourner la même valeur.

Donc, nous sommes amenés à penser qu'il n'est pas possible de faire des entrés/sorties en Haskell.

Pourtant, cela est quand même possible, mais différent de la plus part des autres langages.

La monade `IO`

La solution au problème est l'utilisation d'une monade d'entrées et sorties. La fameuse monade `IO`.

Voici un exemple de son utilisation :

```
main :: IO ()
main = do
    putStrLn "Entrez votre prénom : "    -- Affiche à l'écran.
    txt <- getLine                       -- Saisit au clavier.
    putStrLn ("Hello " ++ txt ++ " !")   -- Affiche le résultat.
```

```
ghci> main
Entrez votre prénom :
Tiago
Hello Tiago !
```

Le même programme peut être écrit avec la notation standard :

```
main :: IO ()
main = putStrLn "Entrez votre prénom : " >>
       getLine >>=
       \txt -> putStrLn ("Hello " ++ txt ++ " !")
```

Le type de `main` in Haskell est `IO ()`.

Le `()` est appelé *unit*. Il s'agit d'un type de base qui représente « le type des effets de bords ».

La unit correspond au type `void` de C et C++, ainsi que le `None` de Python.

Donc, nous pouvons voir le type `IO ()` comme une monade sans valeur injectée.
Voici le type des fonctions utilisées dans l'exemple :

```
ghci> :type putStr
putStr :: String -> IO ()
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

Les fonction `putStr` et `putStrLn` affichent la chaîne de caractères à l'écran, sans et avec un retour à la ligne, respectivement.

La fonction `getLine` construit une valeur de type `IO String` à partir d'une saisie au clavier.

Il est important de noter que la valeur de `main` est toujours le même, c.-à-d., `IO ()` quoi qu'on saisisse au clavier.

C'est comme ça qu'on peut dire que `main` a toujours la même valeur. Cela ne dépend pas des facteurs externes au programme.

Compiler un programme Haskell

Si `main` est défini, alors le programme Haskell peut être exécuté directement de la ligne de commande :

```
$ runghc cm07-monade-io-1.hs
Entrez votre prénom :
Tiago
Hello Tiago !
$
```

Ainsi que compilé en programme exécutable :

```
$ ghc cm07-monade-io-2.hs
[1 of 1] Compiling Main             ( cm07-monade-io-2.hs, cm07-monade-io-2.o )
Linking cm07-monade-io-2 ...
$ ./cm07-monade-io-2
Entrez votre prénom :
Tiago
Hello Tiago !
```

Hello world en Haskell

Résumé

Quelles sont les différences entre les trois classes vues dans ce cours ?

— `Functor` : applique une fonction à une valeur dans un contexte :

```
((+3) <$> Just 2) == Just 5
```

— `Applicative` : applique une fonction dans un contexte à une valeur dans le même contexte :

```
(Just (+3) <*> Just 2) == Just 5
```

— `Monad` : applique une fonction qui renvoie une valeur dans un contexte à une valeur dans le même contexte :

```
(Just 2 >>= half) == Just 1
```

7.6 Exercices

Exercice 1. Un peu d'entrées/sorties :

1. Utilisez la fonction `readMaybe` de la bibliothèque `Text.Read` pour écrire une fonction qui :
   (a) Prend une chaîne de caractères saisie par l'utilisateur.
   (b) Essaye de lire la chaîne dans un `Double`.
   (c) Si la lecture est réussie, affiche le double du nombre ; sinon, affiche un message d'erreur et recommence.

   Exemple :

   ```
   > interactiveDoubling
   Saisissez un nombre : truc
   Saisi invalide.
   Saisissez un nombre : 4
   Le double de votre nombre est 8.0
   ```

2. Même chose, mais cette fois ci :
   (a) Prend une chaîne de caractères saisie par l'utilisateur.
   (b) Si l'utilisateur a saisi une chaîne vide, alors quitte et affiche la somme de tous les nombres saisies.
   (c) Sinon, essaye de lire la chaîne dans un `Double`.
   (d) Si la lecture est réussie, demande le prochain nombre ; sinon, affiche un message d'erreur et ressaye.

   Exemple :

   ```
   > interactiveSumming
   Saisissez un nombre (ENTRER pour finir) : 1
   Saisissez un nombre (ENTRER pour finir) : truc
   Saisi invalide.
   Saisissez un nombre (ENTRER pour finir) : 2
   Saisissez un nombre (ENTRER pour finir) : 3
   Saisissez un nombre (ENTRER pour finir) :
   La somme est : 6.0
   ```

Exercice 2. Pour s'entrainer et mieux comprendre les monades, nous allons créer notre propre monade liste. Soit le type récursif `List` définie comme suit :

```
data List a = Empty
            | Cons a (List a)
            deriving (Eq, Show)
```

1. Faites de sorte que le type `List` soit une instance de la classe `Functor` et implémentez la fonction `fmap`.
2. Écrivez la fonction
   `myConcat2 :: List a -> List a -> List a`
   qui prend deux listes en arguments et renvoie la concaténation des deux listes. Exemples :

```
myConcat2 Empty (Cons 1 (Cons 2 Empty)) == Cons 1 (Cons 2 Empty)
myConcat2 (Cons 1 (Cons 2 Empty) (Cons 3 Empty) == Cons 1 (Cons 2 (Cons 3 Empty))
```

3. Écrivez la fonction
   `myConcat :: List (List a) -> List a`
   qui prend une liste de listes en argument et renvoie une liste contenant la concaténation de toutes les listes.
4. Faites de sorte que le type `List` soit aussi une instance de la classe `Applicative` et implémentez la fonction `pure`, ainsi que l'opérateur `(<*>)`.
5. Faite de sorte que le type `List` soit une instance de la classe `Monad` et implémentez l'opérateur `(>>=)`.
6. Écrivez la fonction `replace :: Eq a => [(a,b)] -> List a -> List (Maybe b)` qui remplace les occurrences du type `a` dans la liste par des valeurs du type `Maybe b` correspondant. Exemples :

   ```
   replace [] (Cons 'a' Empty) == Cons Nothing Empty
   replace [('a', 3)] (Cons 'a' Empty) == Cons (Just 3) Empty
   replace [('a', 3)] (Cons 'a' (Cons 'b' Empty)) ==
       Cons (Just 3) (Cons Nothing Empty)
   ```

   Vous devez utiliser les fonctionnalités de la monade `List`. Vous avez droit d'utiliser la fonction `lookup`.

Chapitre 8

Lambda-calcul

8.1 Introduction

Le **lambda-calcul** est un langage et un système de réécriture imaginé par le mathématicien Alonzo Church en 1932.

Le lambda-calcul est un langage qui est :

— très petit : il ne comporte que deux constructions syntaxiques.

— très expressif : il est capable d'exprimer toutes les fonctions calculables.

8.2 Syntaxe

Les expressions du lambda-calcul sont appelée **lambda-expressions** ou **lambda-termes**.

Une **lambda-expression** est :

— soit une **constante** : 4, π, $+$,... (un nombre, un symbole ou un opérateur).

— soit une **variable** : x, y, z, ... (typiquement une seule lettre minuscule).

— soit une **abstraction** de la forme :

$$(\lambda x.M)$$

où x est une variable et M est une lambda-expression.

— soit une **application** de la forme :

$$(M\ N)$$

où M et N sont des lambda-expressions.

Conventions de parenthésage

Les lambda-expressions ainsi définies sont non ambigües. Mais le grand nombre de parenthèses rend la lecture des expressions difficile :

$$(((\lambda x.(\lambda y.((+\ x)\ y)))\ 1)\ 2)$$

Conventions :

1. Les parenthèses du début et de la fin sont optionnels.
2. L'application est plus prioritaire que l'abstraction.
3. L'opérateur lambda est associatif à droite.
4. L'application est associative à gauche.

Ceci équivaux à ces quatre règles :

$$(M) \equiv M \tag{8.1}$$
$$\lambda x.(M\ N) \equiv \lambda x.M\ N \tag{8.2}$$
$$\lambda x.(\lambda y.M) \equiv \lambda x.\lambda y.M \tag{8.3}$$
$$(M\ N)\ O \equiv M\ N\ O \tag{8.4}$$

Exercice

Enlever les parenthèses inutiles de la lambda-expression :

$$(((\lambda x.(\lambda y.((+\ x)\ y)))\ 1)\ 2)$$

Règle (1) :

$$((\lambda x.(\lambda y.((+\ x)\ y)))\ 1)\ 2$$

Règle (2) :

$$((\lambda x.(\lambda y.(+\ x)\ y))\ 1)\ 2$$

Règle (3) :

$$((\lambda x.\lambda y.(+\ x)\ y)\ 1)\ 2$$

Règle (4) (deux fois) :

$$(\lambda x.\lambda y.+\ x\ y)\ 1\ 2$$

8.3 Signification des lambda-expressions

Signification de l'abstraction

Une abstraction de la forme $\lambda x.M$ exprime la fonction anonyme qui à tout x associe M (M est l'image de x par cette fonction).

Dans ce cas, la lambda-expression M est appelée **corps** de l'abstraction.

Exemple : L'abstraction $\lambda x.x$ exprime la fonction qui à tout x associe x (fonction identité). (L'abstraction $\lambda y.y$ exprime la même fonction.)

Exemple : La fonction $\lambda x.\lambda y.y$ exprime la fonction d'ordre 2 qui à tout x associe la fonction identité.

Signification de l'application

Une application de la forme $M\ N$ exprime l'image de N par M.

Exemple : L'application $(\lambda x.x)\ 1$ exprime l'image du nombre 1 par la fonction identité.

Exemple : En lambda-calcul, la constante + exprime la fonction d'ordre 2 qui à tout x associe la fonction qui à tout y associe $x + y$. Donc, l'expression $+\ 1\ 2 \equiv (+\ 1)\ 2$ exprime le résultat de l'application de cette fonction d'ordre supérieur à 1 puis à 2.

8.4 Réduction

Notion de réduction

Intuitivement, une **réduction** en lambda-calcul est l'action de transformer une lambda-expression en une autre plus simple et ayant la même signification, et de répéter cette opération jusqu'à ce que la lambda-expression ne puisse plus être réduite.

Exemple : $(\lambda x. *\ 2\ x)\ 3 \rightarrow *\ 2\ 3 \rightarrow 6$

De manière générale, la réduction d'une lambda-expression s'interprète comme le calcul du résultat.

Une réduction peut comporter plusieurs étapes, ces étapes décrivent le déroulement du calcul.

Exemple : $(\lambda x.\lambda y. +\ x\ y)\ 1\ 2 \rightarrow (\lambda y. +\ 1\ y)\ 2 \rightarrow +\ 1\ 2 \rightarrow 3$

Bêta-réduction

Une expression de la forme $(\lambda x.M)\ N$ se réduit en M où les occurrences de la variable x sont remplacées par N. C'est à dire :

$$(\lambda x.M)\ N \xrightarrow{\beta} M[x := N]$$

Exemple :

$$\begin{aligned} & (\lambda x.\lambda y. +\ x\ y)\ 1\ 2 \equiv ((\lambda x.\lambda y. +\ x\ y)\ 1)\ 2 \\ \xrightarrow{\beta} \quad & ((\lambda y. +\ x\ y)[x := 1])\ 2 \equiv (\lambda y. +\ 1\ y)\ 2 \\ \xrightarrow{\beta} \quad & (+\ 1\ y)[y := 2] \equiv +\ 1\ 2 \end{aligned}$$

Delta-réduction

La δ-réduction modélise les opérations mathématiques classiques. C'est-à-dire :

$$op\ c_1\ c_2 \ldots c_n \xrightarrow{\delta} c_0$$

où op est un opérateur qui exprime une opération n-aire, chaque c_i est une constante qui exprime un argument, et c_0 est une constante qui exprime le résultat de l'opération.

Exemple :

$$+\ 1(*\ 2\ 3) \xrightarrow{\delta} +\ 1\ 6 \xrightarrow{\delta} 7$$

Problème de capture de variable

Soit la lambda-expression $(\lambda f.\lambda a.f\ a)$. Il s'agit de la fonction d'ordre supérieur qui reçoit une fonction et un argument et applique la fonction à l'argument.

Nous avons :

$$\begin{aligned}&((\lambda f.\lambda a.f\ a)\ g)\ x)\\ \xrightarrow{\beta}&(\lambda a.g\ a)\ x\\ \xrightarrow{\beta}&g\ x\end{aligned}$$

Pourtant, en remplaçant g par a nous avons :

$$\begin{aligned}&((\lambda f.\lambda a.f\ a)\ a)\ x)\\ \xrightarrow{\beta}&(\lambda a.a\ a)\ x\\ \xrightarrow{\beta}&x\ x\end{aligned}$$

C'est-à-dire, le résultat est incorrect.

Problème de la capture de variable

La variable a qui apparaît dans le corps de l'abstraction exprime un autre objet que celui exprimé par la variable a en argument.

On dit que a a été **capturée** dans le corps de l'abstraction après la β-réduction.

En conséquence, la β-réduction ne peut pas toujours être appliquée.

Il faut que les variables **libres** dans l'argument ne soient pas **liées** dans le corps de l'abstraction.

Variables libres et liées

Une occurrence d'une variable x est **liée** dans une lambda-expression M si elle apparaît dans M à l'intérieur d'une sous-expressions de la forme $\lambda x.E$.

Une occurrence d'une variable x est **libre** dans une lambda-expression M si elle n'est pas liée dans M.

Exemple :

$$(\lambda f.\lambda a.f\ a)\ b$$

La deuxième occurrence de a est liée dans l'expression. Cela veut dire que les deux occurrences de a désignent le même objet.

L'occurrence de b est libre dans l'expression, elle désigne un autre objet.

Retour à la bêta-réduction

Une expression de la forme $(\lambda x.M)\ N$ se réduit en M où les occurrences de la variable x sont remplacées par N. C'est à dire :

$$(\lambda x.M)\ N \xrightarrow{\beta} M[x := N]$$

s'il n'existe pas de variable dont une occurrence est libre dans N et une autre occurrence est liée dans M.

Alpha-réduction

La α-réduction est le renommage des variables dans une abstraction :

$$\lambda x.M \xrightarrow{\alpha} \lambda y.(M[x := y])$$

où y est une variable qui n'apparaît pas dans M.

Exemple :

$$\lambda a.f\ a \xrightarrow{\alpha} \lambda b.(f\ a)[a := b] \equiv \lambda b.f\ b$$

Réduction généralisée

Notez que nous pouvons appliquer les réductions à des sous-expressions.

Exemple : Nous pouvons faire :

$$\begin{aligned}&(\lambda x.(\lambda y.\lambda z.+\ z\ y)\ 4\ x)\ 3\\ \xrightarrow{\beta}&(\lambda y.\lambda z.+\ z\ y)\ 4\ 3)\end{aligned}$$

ou bien :

$$\begin{aligned}&(\lambda x.(\lambda y.\lambda z.+\ z\ y)\ 4\ x)\ 3\\ \xrightarrow{\beta}&(\lambda x.(\lambda z.+\ z\ 4)\ x)\ 3\end{aligned}$$

Redex

Une sous-expression que l'on peut choisir de réduire par β-réduction (donc de la forme $(\lambda x.M)\ N)$) est appelée **redex**.

Exemple : La lambda-expression de l'exemple précédent comporte donc deux redex :

$$(\lambda x.(\lambda y.\lambda z.+\ z\ y)_{\wedge}4\ x)_{\wedge}3$$

Le premier redex est plus à l'intérieur de l'expression que le deuxième.

Forme normale

Lorsqu'une lambda-expression ne peut plus se réduire autrement que par la α-réduction, alors elle est en **forme normale**.

Lorsqu'une lambda-expression M se réduit en une lambda-expression N et que N est en forme normale, alors N est la forme normale de M.

Théorème de Church-Rosser : Si une même lambda-expression M se réduit en une lambda-expression M_1 (en choisissant certains redex) et en une autre lambda-expression M_2 (en choisissant d'autres redex), alors il existe une autre lambda-expression N telle que M_1 et M_2 se réduisent en N.

Autrement dit, la réduction est **confluente**.

Forme normale

Donc, toutes les réductions d'une même lambda-expression aboutissent à une même forme normale (à des α-réductions près) **si elles terminent.**

Cependant, **une réduction peut ne pas terminer :**

Exemple : En choisissant toujours le redex le plus à l'intérieur :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_\wedge(\lambda z.z\ z)) \xrightarrow{\beta} (\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_\wedge(\lambda z.z\ z)) \xrightarrow{\beta} \dots$$

la réduction ne termine pas.

En choisissant toujours le redex le plus à l'extérieur :

$$(\lambda x.\lambda y.y)_\wedge((\lambda z.z\ z)\ (\lambda z.z\ z)) \xrightarrow{\beta} \lambda y.y$$

la réduction termine en une étape.

8.5 Stratégies de réduction

Stratégies de réduction

Une **stratégie de réduction** définie l'ordre dans lequel les redex sont utilisés.

La plupart des langages fonctionnels utilisent l'une de ces deux stratégies (avec quelques variantes) :

— **Ordre applicatif de réduction (AOR) :** consiste à choisir toujours le redex **interne**.

— **Ordre normal de réduction (NOR) :** consiste à choisir toujour le redex **externe**.

Stratégies de réduction

Exemple : Considérons l'expression :

$$(\lambda x.(\lambda a.*\ a\ a)\ x)\ ((\lambda y.y)\ 2)$$

Stratégie AOR :

$$(\lambda x.(\lambda a.*\ a\ a)_\wedge x)\ ((\lambda y.y)\ 2) \xrightarrow{\beta} (\lambda x.*\ x\ x)\ ((\lambda y.y)_\wedge 2) \xrightarrow{\beta} (\lambda x.*\ x\ x)_\wedge 2) \xrightarrow{\beta} *\ 2\ 2 \xrightarrow{\delta} 4$$

Stratégie NOR :

$$(\lambda x.(\lambda a.*\ a\ a)\ x)_\wedge((\lambda y.y)\ 2) \xrightarrow{\beta} (\lambda a.*\ a\ a)_\wedge((\lambda y.y)\ 2) \xrightarrow{\beta}$$
$$*\ ((\lambda y.y)_\wedge 2)\ ((\lambda y.y)\ 2) \xrightarrow{\beta} *\ 2\ ((\lambda y.y)_\wedge 2) \xrightarrow{\beta} *\ 2\ 2 \xrightarrow{\delta} 4$$

Passage des arguments

Avec AOR, l'argument est évalué avant l'application de la fonction. Ceci correspond au **passage par valeur** (le mode utilisé par le langage C, par exemple).

Avec NOR, la fonction est appliquée avant l'évaluation de l'argument. Ceci correspond au **passage par nom** (le mode utilisé par le langage FORTRAN, par exemple).

Évaluation

Avec AOR l'évaluation des arguments est faite dès que possible. Ceci correspond à l'**évaluation affairée (eager evaluation)**.

Avec NOR l'évaluation des arguments est faite le plus tard possible. Ceci correspond à l'**évaluation paresseuse (lazy evaluation)**.

AOR vs. NOR

AOR est généralement plus rapide que NOR.

La raison est qu'il arrive fréquemment la situation où nous devons réduire une expression de la forme $(\lambda x.M)\ N$ où M est sous forme normale et contient plusieurs occurrences libres de x.

Avec AOR, l'expression N est réduite en premier et donc l'argument est évalué une seule fois.

Exemple : Considérez l'expression :

$$(\lambda a.*\ a\ a)\ ((\lambda y.y)\ 2)$$

Avec AOR, la sous-expression $((\lambda y.y)\ 2)$ sera évalué une seule fois.

AOR vs. NOR

Pourtant, AOR ne garanti pas la terminaison :

Exemple : Rappelez-vous de l'expression :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

Nous avons vu que la réduction ne termine pas alors qu'une forme normale existe : $\lambda y.y$.

AOR vs. NOR

NOR est généralement moins efficace que AOR.

Pourtant, elle garanti la terminaison quand une forme normale existe.

Théorème de normalisation de Curry : L'ordre normal de réduction conduit à coup sûr à la forme normale lorsqu'elle existe.

Exemple : Nous avons vu que la forme normale de l'expression ci-dessous peut être trouvé avec NOR :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

AOR vs. NOR

Il existe certains cas où NOR est plus efficace que AOR.

Exemple : Soit $(\lambda x.M)\ N$ où M est sous forme normale et il n'y a aucune occurrence libre de x dans M, alors NOR permet d'attendre la forme normale en une étape.

Stratégie du langage OCaml

Le langage OCaml utilise AOR avec une variation.

Exemple : `(function x -> (function a -> a * a) x) ((function y -> y) 2) = (function x -> x * x) ((fun`
`(function x -> x * x) 2 = 2 * 2 = 4`

Stratégie du langage OCaml

Pourtant, OCaml associe à certaines constructions syntaxiques une évaluation spécifique.

L'évaluation de l'alternative `if`, de la disjonction et conjonction logique sont dites **non-strictes** (ou court-circuitée).

Exemple : La fonction récursive : `let rec f n = n = 1 || (n > 1 && f (n - 2))` est correcte et termine.

Pourtant, celle-ci *ne termine pas* : `let rec f n = n = 1 || (f (n - 2) && n > 1)`

Stratégie du langage Haskell

Le langage Haskell utilise NOR avec une variation :

— l'argument d'un redex est évalué au plus une fois.

Pour éviter d'évaluer plusieurs fois l'argument la β-réduction est réalisée avec **partage** de l'argument.

Exemple : `(\x -> (\a -> * a a) x) ((\y -> y) 2) = (\a -> * a a) ((\y -> y) 2) = * ((\y -> y) 2) ((\y -> y) 2)= * 2 2 = 4`

où l'expression `((\y -> y) 2)` est partagée en mémoire et donc calculée une seule fois.

Stratégie du langage Haskell

En Haskell, à cause de l'évaluation paresseuse, les listes peuvent être infinies !

Exemple :

```haskell
take :: Int -> [a] -> [a]
take n xs | 0 < n     = unsafeTake n xs
          | otherwise = []
unsafeTake _ []     = []
unsafeTake 1 (x:_)  = [x]
unsafeTake n (x:xs) = x : unsafeTake (n-1) xs

nats :: [Integer]
nats = 0 : map (+1) nats

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```
Prelude> take 10 nats
[0,1,2,3,4,5,6,7,8,9]
Prelude> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

Q'est-ce qu'est est en train de se passer, en effet ?

```
take 3 nats = take 3 (0 : map (+1) nats)
            = 0 : take 2 (map (+1) nats)
            = 0 : take 2 (map (+1) (0 : map (+1) nats))
```

```
= 0 : take 2 (1 : map (+1) (map (+1) nats))
= 0 : 1 : take 1 (map (+1) (map (+1) nats))
= 0 : 1 : take 1 (map (+1) (map (+1) (0 : map (+1) nats)))
= 0 : 1 : take 1 (map (+1) 1 : (map (+1) (map (+1) nats)))
= 0 : 1 : take 1 (2 : map (+1) (map (+1) (map (+1) nats)))
= 0 : 1 : 2 : take 0 (map (+1) (map (+1) (map (+1) nats)))
= 0 : 1 : 2 : []
= [0,1,2]
```

OCaml vs. Haskell

À cause de la différence de stratégie entre OCaml et Haskell, les deux programmes ci-dessous ne sont pas équivalents.

En OCaml, ce programme *ne termine pas* :

```
let rec f x = f x;;
let g = (fun x -> 1) (f 0);;
```

En Haskell, ce programme termine et donne le bon résultat :

```
1 f x = f x
2 g = (\x -> 1) (f 0)
```

```
Prelude> g
1
```

Nous avons que Haskell est « plus expressive » qu'OCaml (c.-à-d., nous pouvons écrire plus de programmes qui fonctionnent).

8.6 Exercices

Exercice 1. Pour chacune des lambda-expressions suivantes, indiquer tous les redex qu'elle contient.

1. $(\lambda x.((\lambda z.z)\ x)\ y)\ \lambda y.y$
2. $\lambda x.\lambda y.z\ \lambda z.z\ \lambda x.y$
3. $(\lambda y.+\ ((\lambda x.x)\ y)\ y)\ ((\lambda y.*\ 2\ y)\ 1)$
4. $(((\lambda f.\lambda x.f\ x)\ \lambda x.\lambda a.x\ a)\ \lambda x.x)\ \lambda y.y$
5. $(\lambda x.\lambda y.x\ z\ (y\ z))\ \lambda x.(\lambda y.y)\ y$
6. $((\lambda h.(\lambda x.h\ (x\ x))\ \lambda x.h\ (x\ x))\ \lambda f.\lambda x.x)\ (+\ 1\ 5)$

Exercice 2. Réduire chacune des expressions de l'exercice précédent à une expression en forme normale, si elle existe, en indiquant les redex avec le symbole $_\wedge$ et en utilisant la α-réduction uniquement si nécessaire :

1. En utilisant NOR.
2. En utilisant AOR.

Exercice 3. (Les lambda-booléens) Nous pouvons exprimer les booléens et la conditionnelle en lambda calcul par :

$$\text{True} \equiv \lambda x.\lambda y.x \qquad \text{False} \equiv \lambda x.\lambda y.y \qquad \text{Cond} \equiv \lambda c.\lambda v.\lambda f.c\ v\ f$$

1. Vérifiez que Cond se comporte de la bonne façon. C'est-à-dire, que Cond True E_1 E_2 se réduit en E_1 et que Cond False E_1 E_2 se réduit en E_2.
2. En utilisant ce modèle donnez une lambda-expression pour exprimer chacune des opérations booléennes classiques (négation, conjonction et disjonction).

Exercice 4. (Les lambda-entiers). Le mathématicien Alonzo Church a définie les nombres entiers en lambda-calcul de la façon suivante :

$$0 \equiv \lambda f.\lambda x.x \qquad 1 \equiv \lambda f.\lambda x.f\ x \qquad 2 \equiv \lambda f.\lambda x.f\ (f\ x)$$

En effet, nous pouvons définir l'arithmétique en lambda-calcul de la manière suivante :

$$\begin{aligned}\text{Succ} &\equiv \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)\\ \text{Add} &\equiv \lambda n.\lambda m.\lambda f.\lambda x.n\ f\ (m\ f\ x)\\ \text{Mult} &\equiv \lambda n.\lambda m.\lambda f.m\ (n\ f)\\ \text{Pow} &\equiv \lambda n.\lambda m.n\ m\end{aligned}$$

Testez les ce modèle en effectuant les calculs suivants (utiliser NOR ou AOR) :

1. Calculez le successeur de 2.
2. Calculez $3 + 2$.
3. Calculez $3 * 2$.
4. Calculez 3^2.