Cours de lambda calcul et programmation fonctionnelle

Tiago de Lima

UNIVERSITÉ D'ARTOIS
UFR des Sciences

Licence Sciences et Technologie
mention Informatique
Semestre 6

Année universitaire 2018–2019

Section 1

Présentation de l'unité

Objectif de l'unité

L'**objectif** de cette unité est de faire acquérir aux étudiants des notions et concepts de base de la programmation fonctionnelle.

Seront abordés :

- Programmer avec des fonctions
- Types
- Récursivité
- Schéma de programmes
- Lambda calcul

Crédits et charge

ECTS : 5

CM : 1,5 h $\times$ 10 sem $=$ 15 h (Semaines 1–10)

TD : 1,5 h $\times$ 8 sem $=$ 15 h (Semaines 1–10)

TP : 2 h $\times$ 8 sem $=$ 16 h (Semaines 2–10)

Contenu de l'unité (INTENTION ! !)

Sem.	Contenu
1	Notions élémentaires
2	Récursivité
3	Types
4	Types (cont.)
5	**C1**
6	Schémas de programmes
7	Schémas de programmes (cont.)
8	Lambda calcul
9	Lambda calcul (cont.)
10	**C2**

Site web de l'unité

Moodle : **http://moodle.univ-artois.fr/**

Vous trouverez :

- support des cours
- exercices
- notes

Lecture supplémentaire

Site web : télécharger Haskell :
`http://www.haskell.org`

Site web : tester Haskell :
`http://tryhaskell.org`

Plus à venir...

Cours 1

Notions élémentaires

Sommaire

Section 2

Modes de programmation

Modes de programmation

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

Principaux modes de programmation :

- Impératif
- Orienté objet
- Logique
- Fonctionnel

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

Exemples :

- Python est impératif, orienté objet et contient l'opérateur lambda.
- OCaml est fonctionnel, orienté objet et impératif.
- C++
- Java 8

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.

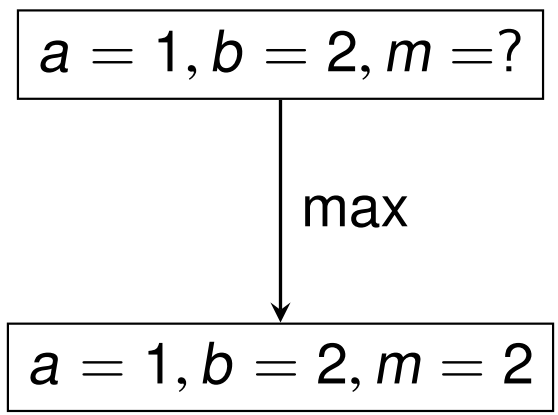
Les autres modes ont l'objectif de définir et organiser les calculs à réaliser par l'ordinateur ou la plateforme d'exécution.

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 (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 (Prolog) :

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

```
1 max (a, b)
2     | a >= b = a
3     | otherwise = b
```

$$\max : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$(a, b) \mapsto \begin{cases} a, & \text{si } a \geq b \\ b, & \text{sinon} \end{cases}$$

Exécution :

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

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

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.

On dit qu'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 associés aux objets de t correspondent à des opérations sur E.

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

Type

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.

Exemples in Haskell :

- `Integer` représente $\mathbb{Z}$.
- `Double` représente $\mathbb{R}$.
- `Bool` représente $\{\text{vrai}, \text{faux}\}$.
- etc.

Fonctions

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 :

$$f : A \to B$$
$$x \mapsto f(x)$$

Exemple :

$$suiv : \mathbb{N} \to \mathbb{N}$$
$$x \mapsto x + 1$$

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 :

$$0 \mapsto 1$$
$$1 \mapsto 2$$
$$2 \mapsto 3$$
$$\ldots$$

Un autre exemple :

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

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

Un premier programme fonctionnel

```
GHCi, version 8.0.1 ...
Prelude> suiv n = n + 1
Prelude> suiv 1
2
Prelude>
```

Fonctions partielles

Une **fonction partielle** sur un ensemble E est une fonction dont le domaine est une partie de E.

Une fonction est **totale** sur E lorsque son domaine égal à E.

Exemple : la fonction inverse $\frac{1}{x}$ est partielle sur $\mathbb{R}$.

Pour définir de telles fonctions, il faut convenir d'une valeur particulière appartenant à tous types et signifiant « indéfini ». Nous notons cette valeur $\perp$.

Dans nos programmes, nous allons utiliser une exception « error ».

En Haskell :

```
Prelude> error "message"
*** Exception: message
CallStack (from HasCallStack):
  error, called at <interactive>...
```

Conditionnelle

Une fonction partielle peut alors être définie en distinguant le cas où l'argument appartient au domaine du cas contraire.

Ceci est un procédé en programmation appelé **décomposition par cas**.

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

En Haskell :

```
Prelude> :{
Prelude| inverse x  =
Prelude|     if x /= 0 then 1 / x
Prelude|     else error "division by zero"
Prelude|:}
Prelude> inverse 1
1.0
Prelude> inverse 0
*** Exception: division by zero
CallStack (from HasCallStack):
  error, called at <interactive>...
```

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 :

```
Prelude> aire_carre r = r * r
Prelude> vol_cube r = r * aire_carre r
Prelude> vol_cube 10
1000
```

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 $n > 1$).

Fonctions sur n-uplets

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

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

```
Prelude> hauteur c  = fst c
Prelude> rayon c = snd c
Prelude> aire_disque r = 3.14 * r * r
Prelude> :{
Prelude| vol_cylindre c =
Prelude|     aire_disque (rayon c) * hauteur c
Prelude| :}
Prelude> vol_cylindre (3, 4)
150.72
```

Fonctions anonymes

En mathématique on peut exprimer une fonction sans la nommer :
« la fonction qui à tout x associe $2x + 1$ ».

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

Exemple :

```
Prelude> (\ x -> x + 1) 1
2
Prelude> (\ 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 :

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

Ou de manière équivalente :

```
Prelude> (\ 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.

Cours 2

Récursivité

Sommaire

Section 4

Définition de récursivité

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.

Récursivité

Exemple :

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

Exemple :

$$somme : \mathbb{N} \to \mathbb{N}$$

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

Forme générale d'une définition récursive

$$f : D \to C$$

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

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.

Forme générale d'une définition récursive

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 $D_1 = \mathbb{N}^*$

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 x
2     | x < 0 = error "negative value"
3     | x == 0 = 0
4     | x > 0 = somme (x - 1) + x
```

La première et deuxième lignes sont appelées **cas de base**.

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

Section 5

Création d'une définition récursive

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.

Création d'une définition récursive

Exemple : Créez la définition de *somme* :

1. Trois cas se présentent :
   - si $x < 0$, alors afficher une erreur.
   - si $x = 0$, alors la solution est 0.
   - si $x > 0$, alors le calcul de $0 + 1 + \ldots (x - 1)$ est un sous-problème de même nature. Il s'agit 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 x
2     | x < 0 = error "negative value"
3     | x == 0 = 0
4     | x > 0 = somme (x - 1) + x
```

Création d'une définition récursive

Exercice : Écrivez la fonction *somme_chiffres* 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 $n = c_0 c_1 \ldots c_k$:

1. Trois cas se présentent :
   - si $k < 0$, alors afficher une erreur.
   - 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 $c_0 + c_1 + \ldots c_{(k-1)}$ 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 *somme_chiffres*$(\lfloor n/10 \rfloor)$ est résolu, alors il suffit de faire $(n \mod 10) +$ *somme_chiffres*$(\lfloor n/10 \rfloor)$ pour calculer *somme_chiffres*(n).

Nous avons donc :

```
1 somme_chiffres n
2     | n < 0 = error "negative value"
3     | 0 <= n < 10 = n
4     | n >= 10 = (n `mod` 10) + somme_chiffres (n `div` 10
```

Section 6

Correction

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 le deux conditions suivantes sont vérifiées :

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

Alors $P(n)$ est vrai pour tout $n \in \mathbb{N}$.

Correction

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

Nous allons montrer par induction :

- **Cas de base :** $somme(0) = 0$ (par la définition de la fonction *somme*).
- **Cas récursif :** Soit $n \geq 0$.
  Si $somme(n) = 0 + 1 + \ldots n$, alors
  $somme(n + 1) = 0 + 1 + \cdots + n + (n + 1)$ (encore une fois, par la définition de la fonction *somme*).

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

Correction

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 !

Section 7

Terminaison

Terminaison

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

Exemple : (Ceci ne termine pas !)

```
1 pair n
2     | n == 0 = True
3     | n > 0 = pair (n - 2)
4     | n < 0 = pair (n + 2)
```

Exemple : (Ceci ne termine pas !)

```
1 moyenne (n, m)
2     | n == m = n
3     | n > m = moyenne (n - 1, m + 1)
4     | n < m = moyenne (n + 1, m - 1)
```

Terminaison

Il serait pratique de disposer d'un programme qui teste la terminaison de nos programmes.

Pourtant, un tel programme n'existe pas. Voici la preuve (Gödel, 1930) :

Supposons que nous avons créé un programme qui teste si tout programme termine. Appelons-le `termine`.

Donc, nous pouvons créer un autre programme. Appellon-le `absurde` :

```
1 absurde x =
2     if termine (absurde x)
3     then absurde x
4     else 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.

Terminaison

Heureusement, **dans certains cas particuliers**, nous pouvons prouver qu'un programme termine.

Soit un programme récursif 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 il termine :

1. Il 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 pas d'appel récursif.
2. Le cas de base est atteint après un nombre **fini** d'appels récursifs.
   - Pour cela, il faut s'assurer que tout appel récursif est toujours plus proche des cas de base que l'appel courant.

Terminaison

Exemple :

```
1 somme n
2     | n < 0 = error "negative value"
3     | n == 0 = 0
4     | n > 0 = somme (n - 1) + n
```

Ce programme termine pour tout $n \in \mathbb{Z}$ car :

- Il y a deux **cas de base :**
  - Si le programme est appelé avec $n < 0$, il retourne erreur et termine.
  - Si le programme est appelé avec $n = 0$, il 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 recursivement 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 succesifs tomberont nécessairement dans un des cas de base.
- Puisque tous les cas de base terminent, le programme termine. QED.

Terminaison

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

```
1 pgcd (a, b)
2     | a < 0 || b < 0 = error "negative values"
3     | a >= 0 && b == 0 = a
4     | a >= 0 && b > 0 = pgcd(b, a `mod` b)
```

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

- Il y a trois **cas de base :**
  - Si appelé avec $a < 0$, il retourne erreur et termine.
  - Si appelé avec $b < 0$, il retourne erreur et termine.
  - Si appelé avec $a \geq 0$ et $b = 0$, il retourne 0 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 $a \mod b \leq b$ 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, le programme termine. QED.

Terminaison

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

```
1 pair n
2     | n == 0 = True
3     | n > 0 = pair (n - 2)
4     | n < 0 = pair (n + 2)
```

Terminaison

Version corrigée :

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

Ce programme termine pour tout $n \in \mathbb{Z}$ car :

- **Cas de base :**
  - $n = 0$
  - $n = 1$
- **Cas récursifs :**
  - Si $n > 1$ alors $n - 2$. Plus proche des cas de base pour tout $n \in \mathbb{Z}$ tel que $n > 1$.
  - Si $n < 0$ alors $n + 2$. Plus proche des cas de base pour tout $n \in \mathit{Ints}$ tel que $n < 0$.

Section 8

Récursivité terminale

La récursivité est souvent moins efficace

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.

Récursivité terminale

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.
- $D_0 \cup D_1 = D$
- $D_0 \cap D_1 = \emptyset$

Récursivité terminale

Exemple :

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

Nous avons :

- La première et la deuxième lignes sont les cas de base.
- La troisième et la quatrième lignes sont les cas récursifs.
- $e_1 = n - 2$
- $e'_1 = n + 2$

Récursivité terminale

Exercice : Est-ce une récursivité terminale ?

```
1 pgcd (a, b) =
2     | a >= 0 && b == 0 = a
3     | a >= 0 && b > 0 = pgcd(a, a `mod` b)
```

Oui car nous avons :

- La première ligne est le cas de base.
- La deuxième ligne est le cas récursif.
- $e_1 = (a, a \mod b)$

Récursivité terminale

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

```
1 fac n
2     | n < 0 = error "nevative number"
3     | n == 0 = 1
4     | n > 0 = fac (n - 1) * n
```

Piste : vous avez droit d'en faire plusieurs fonctions.

Récursivité terminale

```
1 fac_aux (a, n)
2     | n < 0 = error "negative number"
3     | n == 0 = a
4     | n > 0 = fac_aux(a * n, n - 1)
5
6 fac n = fac_aux (1, n)
```

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

La fonction `fac_aux` est récursive et terminale car :

- La première et la deuxième lignes sont les cas de base.
- La troisième ligne est le cas récursif.
- $e_1 = (a \times n, n - 1)$.

Récursivité terminale

Avantages :

- Performance (même performance qu'avec un langage impératif).
- Absence de débordement de la pile.

Inconvénients :

- Manque de lisibilité
- Le gain de performance peut être illusoire.

Récursivité mutuelle (croisée)

Exemple :

```
1  pair n
2      | n < 0 = error "bottom"
3      | n == 0 = True
4      | n > 0 = impair (n - 1)
5
6  impair n
7      | n < 0 = error "bottom"
8      | n == 0 = False
9      | n == 1 = True
10     | n > 1 = pair (n - 1)
```

Cours 3

Types

Sommaire

Section 9

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`.

```
Prelude> :type 1
1 :: Num t => t
Prelude> :type 1 + 1.0
1 + 1.0 :: Fractional a => a
```

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

```
Prelude> :set +t
Prelude> circonf r = 2 * 3.14 * r
circonf :: Fractional a => a -> a
Prelude> circonf 1
6.28
it :: Fractional a => a
```

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.

Section 10

Constructeurs de type

Types de base

Il y a plusieurs types de base en Haskell. Voici quelques uns :

type	description
`Char`	caractères unicode
`Bool`	valeurs logique
`Int`	entiers (taille fixe)
`Integer`	entiers (taille illimitée)
`Float`	virgules flottante (non recommandé)
`Double`	virgules flottante
`Rational`	rationnels (deux entiers)
`Fractional`	(polymorphe) `Double`, `Float` ou `Rational`
`Num`	(polymorphe) numérique

Listes

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

Il y a plus d'une manière de créer de listes en Haskell :

```
Prelude> [1, 2, 3]
[1, 2, 3]
Prelude> 1:2:3:[]
1:2:3:[]
Prelude> []
[]
```

```
Prelude> :type []
[] :: [t]
Prelude> :type [1, 2, 3]
[1, 2, 3] :: Num t => [t]
Prelude> :type 1:2:3:[]
1:2:3:[] :: Num a => [a]
Prelude> :type "chaine"
"chaine" :: [Char]
```

Listes

Nous pouvons utiliser les fonctions `head` et `tail` respectivement pour avoir la tête et la queue de la liste.

```
Prelude> head [1, 2, 3]
1
Prelude> tail [1, 2, 3]
[2,3]
Prelude> head (1:2:3:[])
1
Prelude> tail (1:2:3:[])
[2,3]
Prelude> head "chaine"
'c'
Prelude> tail "chaine"
"haine"
Prelude>
```

Exercice

Écrivez la fonction `longueur` qui calcule la longueur d'une liste.

```
1 longueur l
2     | l == [] = 0
3     | otherwise = 1 + longueur (tail l)
```

Écrivez la fonction `concat` qui concatène deux listes.

```
1 concat l1 l2
2     | l1 == [] = l2
3     | otherwise = (head l1) : concat (tail l1) l2
```

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 Haskel :

```
case e of {
    p1 -> e1;
    p2 -> e2;
    ...
    pn -> en
}
```

Filtrage par motif

Exemple :

```
1 longueur l =
2     case l of {
3         [] -> 0;
4         _:q -> 1 + (longueur q)
5     }
```

Exercice : Écrivez la fonction `concat` en utilisant le filtrage par motif.

```
1 concat l1 l2 =
2     case l1 of {
3         [] -> l2
4         t:q -> t : (concat q l2)
5     }
```

Constructeurs de type

Un **constructeur de type** est un opérateur qui prend en argument un ou plusieurs types, et a pour résultat un type.

Il y a deux constructeurs de type fondamentaux : **produit** et **flèche**.

Type produit

Soit $A_1, \ldots A_n$ 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 $(a_1, \ldots, a_n)$ 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 :

```
Prelude> :type (1, 2)
(1, 2) :: (Num t1, Num t) => (t, t1)
Prelude> :type ('a', 1)
('a', 1) :: Num t => (Char, t)
```

Type flèche

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.

En Haskell :

```
Prelude> circonf r = 2 * 3.14 * r
circonf :: Fractional a => a -> a
Prelude> somme (a, b) = a + b
somme :: Num a => (a, a) -> a
```

Priorité et associativité

L'opérateur $\times$ est prioritaire sur $\rightarrow$.

- $A \times B \rightarrow C = (A \times B) \rightarrow C$

L'opérateur $\rightarrow$ est associatif à droite.

- $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Attention : l'opérateur $\times$ n'est pas associatif.

- $(A \times B) \times C \neq A \times B \times C$
- $A \times (B \times C) \neq A \times B \times C$
- $(A \times B) \times C \neq A \times (B \times C)$

Section 11

Fonctionnelles et fonctions d'ordre supérieur

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 :

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

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

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

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

Nous avons, par exemple, $\mathit{raison}(\mathit{suite}) = 2$.

Fonctionnelles

En Haskell :

```
1 suite n
2     | n == 0 = 1
3     | n > 0 = suite (n - 1) + 2
4
5 raison f = (f 1) - (f 0)
```

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

Fonctionnelles

Nous pouvons utiliser raison avec d'autres suites arithmétiques :

```
1 suite2 n
2     | n == 0 = 1
3     | n == 1 = suite (n - 1) + 5
```

```
Prelude> suite2 0
1
Prelude> suite2 1
6
Prelude> 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 x est pair et $-x$ sinon).

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

où :

$$f_n(x) = (-1)^n x$$

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 :

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

```
Prelude> (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 :

```
Prelude> sign 5 3.5
-3.5
```

Fonctionnelles d'ordre supérieur

Voici un exemple plus pratique.

Nous savons que l'intégrale d'une fonction f dans un intervalle $[a; b]$ est une nouvelle fonction $\int_a^b f$.

Nous pouvons calculer cela avec l'approximation de Simpson :

$$\int_b^a f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

Nous pouvons donc définir la fonction *intégrale* qui, étant donné une fonction f intégrable sur $\mathbb{R}$, revoie la fonction qui à tout intervalle $[a; b]$ associe une valeur approchée de l'intégrale de a à b de f.

Fonctionnelles d'ordre supérieur

En Haskell :

```
1 integrale f = \ (a, b) =
2     ((b - a)/6) * ((f a) +
3                    4 * (f ((a + b) / 2)) +
4                    (f b))
```

```
Prelude> f x = x * x
Prelude> (integrale f) (0, 1)
0.3333333
```

Ou bien :

```
Prelude> (integrale (\x -> x * x)) (0, 1)
0.3333333
```

Fonctionnelles d'ordre supérieur

La fonction *dérivé* qui, étant donné une fonction f dérivable sur un intervalle de $\mathbb{R}$, renvoie une fonction qui à tout point x_0 associe une valeur approchée de $f'(x_0)$.

Nous pouvons utiliser la définition suivante :

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Fonctionnelles d'ordre supérieur

En prenant *h* suffisamment petit, nous avons en Haskell :

```
1 derivee f =
2     \ x0 -> let h = 0.001 in
3         ((f (x0 + h)) - (f x0)) / h
```

```
Prelude> f x = x * x
Prelude> (derivee f) 1
2.0009999
```

Ou bien :

```
Prelude> (derivee (\x -> x * x)) 1
2.000999
```

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.

```
Prelude> add =  \ x -> \ y -> x + y
Prelude> suiv = add 1
Prelude> 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 (a, b) = a + b
2 add2 a b = a + b
```

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

```
Prelude> :type add1
add1 :: Num a => (a, a) -> a
Prelude> :type add2
add2 :: Num a => a -> a -> a
```

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

```
1 curry f = \ a -> \ b -> f (a, b)
```

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

```
1 uncurry f = \ (a, b) -> f a b
```

Notez que maintenant nous pouvons faire :

```
Predlude> add1 (a, b) = a + b
Prelude> add2 = curry add1
```

ou bien :

```
Prelude> add2 a b = a + b
Prelude> add1 = uncurry add2
```

Section 12

Polymorphisme

Polymorphisme

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

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

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.

Polymorphisme

Exemple : La fonction identité :

```
1 identite x = x
```

```
Prelude> identite 1
1
Prelude> identite "oui"
"oui"
```

Exemple : La fonction « swap » :

```
1 swap (x, y) = (y, x)
```

```
Prelude> swap (1, 2)
(2,1)
Prelude> swap (1, "oui")
("oui",1)
```

Variable 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.

Variable de type

Exemple :

```
Prelude> identite x = x
Prelude> :type identite
identite :: t -> t
```

La variable de type `t` n'est pas instanciée. Donc, le type de la fonction `identite` est polymorphe.

Exemple :

```
Prelude> swap (x, y) = (y, x)
Prelude> :type swap
swap :: (t1, t) -> (t, t1)
```

Les variables de type `t` et `t1` ne sont pas instanciées. Donc, le type de la fonction `swap` est polymorphe.

Variable de type

Exemple :

```
Prelude> const_un x = 1
Prelude> :type const_un
const_un :: Num t1 => t -> t1
```

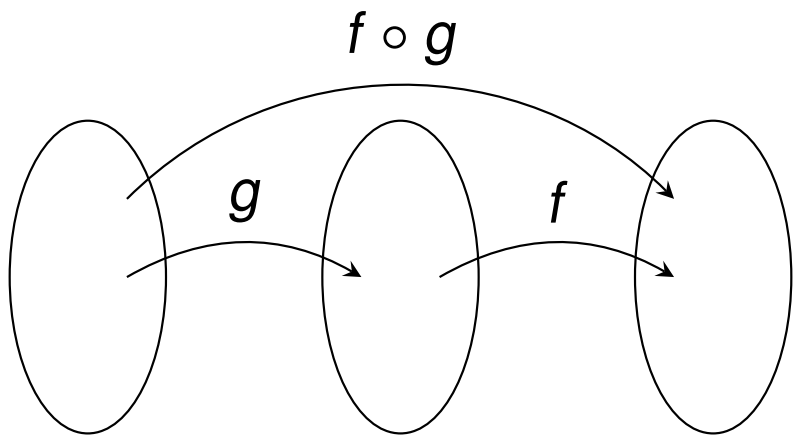
La variable de type `t1` est instanciée avec un type polymorphe et la variable `t` ne l'est pas. Donc, le type de la fonction `const_un` est polymorphe.

```
Prelude> const_a x = 'a'
Prelude> :type const_a
const_a :: t -> Char
```

La variable de type `t` n'est pas instanciée. Donc, le type de la fonction `const_a` est polymorphe.

Exercice

Un exemple classique de fonction polymorphe est la fonctionnelle qui, étant donné deux fonctions *f* et *g*, renvoie $f \circ g$, c'est à dire, la composée de *g* par *f*. Écrivez cette fonction en Haskell.




```
1 compose f g = \ x -> f (g x)
```

Quell est le type de `compose` ?

```
Prelude> :type compose
compose :: (t -> t1) -> (t2 -> t) -> t2 -> t1
```

Le type de la variable `f` est `t -> t1` et le type de `g` est `t2 -> t`.
Le type de retour de `g` est idéntique à celui de l'argument de `f`.

Fonction compose

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

```
Prelude> add1 x = x + 1
Prelude> add2 x = x + 2
Prelude> add3 = compose add1 add2
Prelude> 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`.

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

Section 13

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 :

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

- 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.

La règle dit que :
« Si e_1 est du type t_1 et . . .et e_n est du type t_n, alors e_0 est du type t_0. »

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

Quelques règles d'inférence de type

Définition de fonction `\ e1 -> e2` :

$$\begin{cases} e_1 : t_1 \\ e_2 : t_2 \end{cases} \overset{fun}{\Rightarrow} : t_1 \to t_2$$

Application de fonction `e1 e2` :

$$\begin{cases} e_1 : t_1 \to t_2 \\ e_2 : t_1 \end{cases} \overset{app}{\Rightarrow} : t_2$$

Conditionnelle `if e1 then e2 else e3` :

$$\begin{cases} e_1 : Bool \\ e_2 : t \\ e_3 : t \end{cases} \overset{cond}{\Rightarrow} : t$$

Opérateurs `e1 + e2` :

$$\begin{cases} e_1 : Num \\ e_2 : Num \end{cases} \overset{+}{\Rightarrow} : Num$$

Inférence de type

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

$$\begin{cases} \texttt{1} : \texttt{Num} \\ \texttt{2} : \texttt{Num} \end{cases} \overset{+}{\Rightarrow} \texttt{1 + 2} : \texttt{Num}$$

Inférence de type

Exemple : Soit l'expression
`abs = \ x -> if x > 0 then x else -x`.
D'abord nous appliquons la règle pour l'opérateur `let` :

$$1. \quad \begin{cases} \texttt{abs} : t_1 \\ \texttt{\textbackslash\ x -> if x > 0 then x else -x} : t_1 \end{cases} \overset{let}{\Rightarrow} : t_1$$

Ensuite, la règle pour l'opérateur lambda (\) :

$$2. \quad \begin{cases} \texttt{x} : t_2 \\ \texttt{if x > 0 then x else -x} : t_3 \end{cases} \overset{fun}{\Rightarrow} : t_2 \to t_3$$

Nous avons :

$$3. \quad t_1 = t_2 \to t_3 \quad (2, 3)$$

Inférence de type

Maintenant, la règle pour l'opérateur `if` :

$$4.\quad \begin{cases} \texttt{x} > \texttt{0} : t_4 \\ \texttt{x} : t_2 \\ \texttt{-x} : t_2 \end{cases} \overset{cond}{\Rightarrow} : t_2$$

Nous avons :

$$5.\quad t_3 = t_2 \qquad (2,4)$$
$$6.\quad t_1 = t_2 \to t_2 \qquad (3,5)$$

La règle pour l'opérateur $>$:

$$7.\quad \begin{cases} \texttt{x} : t_2 \\ \texttt{0} : t_2 \end{cases} \overset{>}{\Rightarrow} : \texttt{Bool}$$

Nous avons :

$$8.\quad t_4 = \texttt{Bool} \qquad (4,5)$$

Inférence de type

La règle pour les constantes :

$$9. \quad \{ \mathtt{0} : \mathtt{Num}$$

La règle pour l'opérateur – :

$$10. \quad \{ \mathtt{x} : \mathtt{Num} \quad \bar{\Rightarrow} : \mathtt{Num}$$

Ceci nous permet d'inférer :

11.	$t_2 = \mathtt{Num}$	$(7, 9)$ et aussi $(7, 10)$
12.	$t_1 = t_2 \to t_2$	$(3, 11)$
13.	$t_1 = \mathtt{Num} \to \mathtt{Num}$	$(11, 12)$

Donc, le type de l'expression :

```
abs = \ x -> if x > 0 then x else -x
```

est :

$$\mathtt{Num} \to \mathtt{Num}$$

Inférence de type

Exercice : Quel est le type de l'expression :

`double = \ f -> \ x -> 2 * (f x)` ?

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

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

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

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

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

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 :

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

Cours 4

Schémas de programmes

Sommaire

Section 14

Introduction

Introduction

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

```
somme l =
    case l of {
        [] -> 0;
        h:t -> h + (somme t)
    }
```

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

```
produit l =
    case l of {
        [] -> 1;
        h:t -> h * (produit t)
    }
```

Introduction

Plusieurs programmes sur les listes se ressemblent.

Nous pouvons dire que ces programmes 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 $h : t$, alors le résultat est $h\ op\ f(t)$, où *op* est une opération binaire (cas récursif).

En Haskell :

```
1 f l =
2     case l of {
3         [] -> a;
4         h:t -> op h (f t)
5     }
```

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

Schémas de programme

Exercice : Soit le schéma :

```
1 f l =
2     case l of {
3         [] -> a;
4         h:t -> op h (f t)
5     }
```

Complétez le tableau :

f	a	op
somme		
produit		
longueur		
conc		

Extraction de l'opération binaire

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

```
1 somme l =
2     case l of {
3         [] -> 0;
4         h:t -> h + (somme t)
5     }
```

Donc :

- `f` = somme
- `a` = `0`
- op :
  - Nous avons dans la fonction : `h + (somme t)`
  - `h` est l'élément courant : `e`
  - `(somme t)` est le résultat accumulé : `a`
  - Nous avons `h + (somme t)` = `e + a`
  - Nous avons op = `\ e -> \ a -> e + a`

Extraction de l'opération binaire

Donc, nous pouvons réécrire la fonction `somme` :

```
1 somme l =
2     case l of {
3         [] -> 0;
4         h:t -> (\ e -> \ a -> e + a) h (somme t)
5     }
```

Le cas de la fonction `produit` est analogue.

Extraction de l'opération binaire

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

```
1 longueur l =
2     case l of {
3         [] -> 0;
4         h:t -> 1 + (longueur t)
5     }
```

Donc :

- `f` = `longueur`
- `a` = `0`
- `op` :
  - Nous avons dans la fonction : `1 + (longueur t)`
  - `h` est l'élément courant : `e`
  - `(longueur t)` est le résultat accumulé : `a`
  - Nous avons `1 + (longueur t)` = `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` :

```
1 longueur l =
2     case l of {
3         [] -> 0;
4         h:t -> (\ e -> \ a -> 1 + a) h (longueur t)
5     }
```

Extraction de l'opération binaire

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

```
1 conc l1 l2 =
2     case l1 of {
3         [] -> l2;
4         h:t -> h : (conc t l2)
5     }
```

Donc :

- `f` = `conc`
- `a` = `l2`
- op :
  - Nous avons dans la fonction `h : (conc t l2)`
  - `h` est l'élément courant : `e`
  - `(conc t l2)` est le résultat accumulé : `a`
  - Nous avons `h : (conc t l2)` = `e : a`
  - Nous avons op = `\ e -> \ a -> e : a`

Extraction de l'opération binaire

Donc, nous pouvons réécrire la fonction conc :

```
1 conc l1 l2 =
2     case l1 of {
3         [] -> l2;
4         h:t -> (\ e -> \ a -> e : a) h (conc t l2)
5     }
```

Section 15

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 :

```
1 reduce op a l =
2     case l of {
3         [] -> a;
4         h:t -> op h (reduce op a t)
5     }
```

```
Prelude> 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.

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

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

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

Le schéma réduction

Nous pouvons donc définir les fonctions vues en haut comme suit :

```
1 somme = reduce (+) 0
2 produit = reduce (*) 1
3 conc = reduce (:)
4 longueur = reduce (\ e -> \ a -> 1 + a) 0
```

Utilisation :

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

Exercice

Quelle est le type de `reduce` ?

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

Section 16

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.

```
1 fold_right op a l =
2     case l of {
3         [] -> a;
4         h:t -> op h (fold_right op a t)
5     }
```

Par exemple :

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

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**.

```
1 fold_left op a l =
2     case l of {
3         [] -> a;
4         h:t -> fold_left op (op a h) t
5     }
```

Par exemple :

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

Les schémas plier

La fonction `fold_left` est récursive terminale et donc plus efficace que `fold_right`.

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

Nous voulons le comportement suivant :

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

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

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

```
1 longueur = fold_left (\ a -> \ e -> a + 1) 0
```

Les schémas plier

Notez pourtant que nous ne pouvons pas utiliser `fold_left` pour implémenter `conc`, car nous devons ajouter les éléments de la liste `l1` dans l'ordre inverse dans le résultat.

Section 17

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 :

```
1 map f l =
2     case l of {
3         [] -> [];
4         h:t -> f h : (map f t)
5     }
```

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.

```
1 fois_2 = map ((*) 2) l
```

Utilisation :

```
Prelude> fois_2 [1, 3, 4]
[2, 6, 8]
```

Le schéma map

Regardez bien le schéma map encore une fois.

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

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 : []))
```

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

```
1 map f = fold_right (\ 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 f l1 l2 =
2     case (l1, l2) of {
3         ([], _) -> [];
4         (_, []) -> [];
5         (h1:t1, h2:t2) -> f h1 h2 : (map2 f t1 t2)
6     }
```

Le schéma map2 est aussi appelé zip.

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

Cours 5

Lambda calcul

Sommaire

Section 18

Introduction

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.

Section 19

Syntaxe

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.
   (c.-à-d., la porté du λ va aussi loin que possible.)
3. L'application est associative à gauche.

Ceci équivaux à ces quatre règles :

$$(M) \equiv M \tag{1}$$

$$\lambda x.(M\ N) \equiv \lambda x.M\ N \tag{2}$$

$$\lambda x.(\lambda y.M) \equiv \lambda x.\lambda y.M \tag{3}$$

$$(M\ N)\ O \equiv M\ N\ O \tag{4}$$

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) :

$$(\lambda x.\lambda y.+\ \ x\ y)\ 1\ 2$$

Section 20

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.

Section 21

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 :

$$(\lambda x.\lambda y.+\ x\ y)\ 1\ 2 \equiv ((\lambda x.\lambda y.+\ x\ y)\ 1)\ 2$$
$$\xrightarrow{\beta} \quad ((\lambda y.+\ x\ y)[x := 1])\ 2 \equiv (\lambda y.\ 1\ y)\ 2$$
$$\xrightarrow{\beta} \quad (\lambda y.+\ 1\ y)[y := 2] \equiv +\ 1\ 2$$

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 :

$$(\lambda x.(\lambda y.\lambda z.+\ z\ y)\ 4\ x)\ 3$$
$$\xrightarrow{\beta}(\lambda y.\lambda z.+\ z\ y)\ 4\ 3)$$

ou bien :

$$(\lambda x.(\lambda y.\lambda z.+\ z\ y)\ 4\ x)\ 3$$
$$\xrightarrow{\beta}(\lambda x.(\lambda z.+\ z\ 4)\ x)\ 3$$

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 :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_\wedge(\lambda z.z\ z)) \xrightarrow{\beta} (\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_\wedge(\lambda z.z\ z)) \xrightarrow{\beta} \ldots$$

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.

Section 22

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) ((function y -> y) 2) =
(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 :

```
1 elem n l =
2     case (n, l) of
3         (0, h:_) = h
4         (n, _:t) = elem (n - 1) t
5
6 nats = 0:map ((+) 1) nats
7
8 fibs = 0:1:map2 (+) fibs (tail fibs)
```

Nous avons :

```
Prelude> elem 10 nats
10
Prelude> elem 5 fibs
5
```

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 let f x = f x
2 let g = (\ x -> 1) (f 0)
```

```
Prelude> g
1
```

Nous avons que Haskell est plus expressive qu'OCaml.