Si `lst` = `[e_1; e_2; ..., e_n]` est une liste non vide,

- l'élément `e_1` est la *tête* de `lst` ;
- la liste `[e_2; ...; e_n]` est la *queue* de `lst`.

La fonction

`List.hd : 'a list -> 'a`

renvoie la tête d'une liste non vide (ce qui suit le `:` est le type de la fonction). Ceci sera détaillé plus tard, mais pour le moment, nous pouvons retenir que cela signifie que la fonction accepte une valeur de type `'a list` et renvoie une valeur de `'a`.

La fonction

`List.tl : 'a list -> 'a list`

renvoie la queue d'une liste non vide.

L'*opérateur de construction* est l'opérateur infixe `::`. Il prend un élément `x` et une liste `lst` et renvoie la liste dont la tête est `x` et la queue est `lst`.

Cet opérateur appelle la fonction

`List.cons : 'a -> 'a list -> 'a list`

Exemples

- `43 :: [1; 4; 9]` est la liste `[43; 1; 4; 9]`.
- `1 :: 2 :: [3; 4; 5]` est la liste `[1; 2; 3; 4; 5]`.
- `List.cons 43 [1; 4; 9]` est la liste `[43; 1; 4; 9]`.
- `List.cons 1 (List.cons 2 [3; 4; 5])` est la liste `[1; 2; 3; 4; 5]`.

Il est possible de se servir de `::` dans un motif.

Exemple

```
let two_or_more lst =
    match lst with
        |x1 :: x2 :: lst' -> true
        |lst' -> false
```

Cette fonction teste si la liste `lst` possède au moins deux éléments.

Voici quelques exemples de fonctions sur les listes.

Exemple

```
let rec last lst =
    match lst with
        |[x] -> x
        |x :: lst' -> last lst'
```

Cette fonction renvoie le dernier élément de la liste non vide `lst`.

Exemple

```
let rec is_length_even lst =
    match lst with
        |[] -> true
        |x :: lst' -> not (is_length_even lst')
```

Cette fonction teste si la liste `lst` est de longueur paire.

Exemple

```
let rec member x lst =
    match lst with
        |[] -> false
        |x' :: lst' -> if x' = x then true else member x lst'
```

Cette fonction teste si la liste `lst` contient l'élément `x`.

4. Bases théoriques

4.1. Brève chronologie de la programmation

Préhistoire.

- 1801 : J. M. Jackard met au point la 1re machine programmable, un métier à tisser.
- 1837 : C. Babbage propose les plans de la machine analytique.
- 1843 : A. Lovelace propose le tout premier programme, destiné à la machine analytique.
- 1920 : M. Schönfinkel introduit la logique combinatoire.
- 1936 : A. Church introduit le λ-calcul.
- 1936 : A. Turing invente la machine de Turing.

Histoire.

- Années 1940 : débuts de la programmation, en assembleur et en langage machine.
- Années 1950-1960 : 1ers vrais langages de programmation : FORTRAN (1957), Lisp (1958) et COBOL (1959).
- Années 1960-1970 : apparition des paradigmes de programmation : impératif, fonctionnel, orienté objet, logique.

Période moderne.

- Années 1970-2000 : apparition des langages de programmation modernes : Pascal (1970), C (1972), ML (1973), Prolog (1973), C++ (1983), (O)Caml (1987, 1996), Python (1991) et Java (1995).
- Années 1970-2000 : apparition d'autres paradigmes de programmation et de méthodes de développement : programmation par réécriture, programmation réactive, programmation concurrente, programmation extrême, développement piloté par les tests.
- Années 1990-2020 : essor des bibliothèques : NumPy (1995), scikit-learn (2007), pandas (2008), PyTorch (2016).

4.2. Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

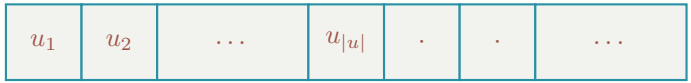
Une machine de Turing est un quadruplet (E, i, t, Δ) où

1. E est un ensemble fini d'états ;
2. $i \in E$ est l'état initial ;
3. $t \in E$ est l'état terminal ;
4. $\Delta : E \times \{\cdot, 0, 1\} \to E \times \{\cdot, 0, 1\} \times \{\mathrm{G}, \mathrm{D}\}$ est une fonction de transition.

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0,1\}^*$ un mot.

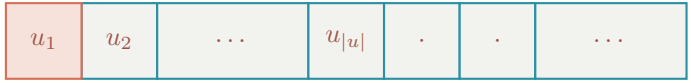
L'*exécution* de M sur u consiste à :

1. placer u le plus à gauche dans un tableau infini à droite, le *ruban* :



   Les cases à droite de u sont remplies jusqu'à l'infini de $\cdot$.

2. Placer la *tête de lecture/écriture* sur la 1re case du ruban :



   Elle pointe vers la case surlignée. On appelle a la lettre dans $\{\cdot, 0, 1\}$ indiquée par la tête de lecture/écriture à un instant donné.

3. Affecter au *registre d'état* e la valeur i (l'état initial).

4. Réaliser les actions dictées par Δ, le *programme*.

Pour réaliser les actions dictées par le programme de M, on procède itérativement comme suit :

1. calculer $(e', a', s') := \Delta(e, a)$.
2. Écrire a' dans la case du ruban indiquée par la tête de lecture/écriture.
3. Affecter au registre d'état e l'état e'.
4. Si $s' = \mathrm{D}$, déplacer la tête de lecture/écriture d'un pas vers la droite ; sinon, la déplacer si possible d'un pas vers la gauche.
5. Si $e = t$, alors l'exécution est terminée ; sinon, revenir en 1.

Le *résultat* de l'exécution de M sur u est le mot $M(u)$ défini comme étant **le plus court préfixe** du ruban qui contient tous ses 0 et ses 1 (le reste du ruban à droite est donc composé d'une infinité de $\cdot$).

Exemple

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$ $(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$ $(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$

Registre d'état	Ruban
e_1	0 0 1 · · ⋯
e_1	1 0 1 · · ⋯
e_1	1 1 1 · · ⋯
e_1	1 1 0 · · ⋯
e_2	1 1 0 · · ⋯

Voici les étapes du calcul de $M(001)$.

L'exécution de M sur u fournit ainsi le résultat $M(001) = 110$.

Ce programme Δ permet de calculer le complémentaire de tout mot $u \in \{0,1\}^*$ en entrée.

Il est possible de construire des machines de Turing telles que pour certaines entrées u, l'exécution de M sur u ne se termine pas.

Exemple

Soit la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$(e_1, 0) \mapsto (e_1, 0, \mathrm{D}),$ $(e_1, 1) \mapsto (e_2, 1, \mathrm{G}),$ $(e_1, \cdot) \mapsto (e_1, \cdot, \mathrm{D}).$

Calcul de $M(001)$:

Registre d'état	Ruban
e_1	0 0 1 · · …
e_1	0 0 1 · · …
e_1	0 0 1 · · …
e_2	0 0 1 · · …

On arrive à e_2 qui est l'état terminal : l'exécution est terminée.

Calcul de $M(000)$:

Registre d'état	Ruban
e_1	0 0 0 · · …
e_1	0 0 0 · · …
e_1	0 0 0 · · …
e_1	0 0 0 · · …
e_1	0 0 0 · · …

La tête de lecture/écriture part vers l'infini : l'exécution ne se termine pas.

Un langage de programmation L est *Turing-complet* s'il permet de simuler une machine de Turing.

Plus précisément, c'est le cas lorsqu'il est possible de développer dans L une fonction d'émulation `emul`, paramétrée par le codage d'une machine de Turing M et le codage d'un mot $u \in \{0,1\}^*$, et qui renvoie le codage de $M(u)$.

Ceci est un exercice très facile pour des langages impératifs comme C, Python ou Java. Ils sont donc Turing-complets. (Note : le ruban implanté ne peut pas être infini mais est considéré comme arbitrairement grand.)

Exemple

En Python, une telle fonction pourrait avoir l'allure suivante.

```
def emul(machine_encoding, input_encoding) :
    res = "" # A string of 0s and 1s.
    ...
    return res
```

Note : il existe des langages intéressants qui ne sont pas Turing-complets (comme les langages rationnels, les langages algébriques et certains langages d'assistants de preuve).

Une *machine de Turing universelle* est une machine de Turing M_{univ} telle que, pour toute machine de Turing M et tout mot $u \in \{0,1\}^*$,

$$M_{\text{univ}}(v) = M(u),$$

où $v \in \{0,1\}^*$ est le codage du couple (M, u).

En d'autres termes, M_{univ} est une machine de Turing qui permet d'**émuler n'importe quelle machine de Turing**.

Étant donné un langage de programmation L, il est facile de traduire les constructions qu'il propose (affectations, instructions de branchement, *etc.*) de sorte à pouvoir construire, pour tout programme p en L, une machine de Turing M_p qui émule l'exécution de p.

La **machine de Turing universelle** M_{univ} peut ainsi émuler M_p et donc, par transitivité, aussi p. Ainsi, M_{univ} **peut calculer tout ce qu'un programme peut calculer**.

4.3. Programme = entier

Nous associons à toute machine de Turing M un entier naturel $\text{code}(M)$ défini de la manière suivante :

1. nous fixons des conventions (arbitraires mais rigoureuses) pour écrire les définitions des machines de Turing avec des caractères ASCII ;
2. nous considérons la suite de caractères m ainsi obtenue qui code M ;
3. nous considérons la suite de bits v obtenue en remplaçant chaque caractère de m par sa représentation binaire ;
4. nous obtenons finalement l'entier naturel $\text{code}(M)$ en considérant l'entier dont v est la représentation binaire.

Exemple

Considérons la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, \mathrm{D}), \qquad (e_1, 1) \mapsto (e_2, 1, \mathrm{G}), \qquad (e_1, \cdot) \mapsto (e_1, \cdot, \mathrm{D}).$$

Cette machine de Turing M se code en caractères ASCII en le texte m suivant :

```
etats : e1, e2
initial : e1
terminal : e2
delta : (e1, 0) -> (e1, 0, D)
delta : (e1, 1) -> (e2, 1, G)
delta : (e1, .) -> (e1, ., D)
```

Nous en déduisons la représentation binaire

$$v = \texttt{01100101 01110100} \cdots \texttt{00101001}$$

et de celle-ci, l'entier naturel $\mathrm{code}(M)$.

En décimal, ce nombre est

$$\mathrm{code}(M) =$$

$$1195273483522463947698188428566573994862939056290801762903598135757140294873881005500260610437207957403372$$

$$2697918914577377247367371651684191013291324227514186378240511188416405854399907473618140642995536490442 52$$

$$2997516654502946689283241263412326824166734535399938860445815033689444918572055862472186488088289817569 69.$$

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application code est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

Nous pouvons ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leqslant M'$ si $\text{code}(M) \leqslant \text{code}(M')$.

L'ensemble de toutes les machines de Turing peut donc être ordonné en un segment infini

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le *rang* $\text{rang}(M)$ d'une machine de Turing M est la position de M dans ce segment (c.-à-d., $\text{rang}(M_n) = n$).

L'application rang fournit ainsi une **bijection** entre l'ensemble des machines de Turing et l'ensemble des entiers naturels.

En conclusion, un **programme** (une machine de Turing) est un **entier** et réciproquement.

4.4. Décidabilité et indécidabilité

Un *problème de décision* est une question P qui prend un mot de $\{0,1\}^*$ en entrée et qui répond « oui » ou « non ».

Exemples

- P : le mot est-il un palindrome ? Par exemple, $P(010010) = \text{oui}$, $P(011) = \text{non}$;
- P : la longueur du mot est-elle paire ? Par exemple, $P(\epsilon) = \text{oui}$, $P(1) = \text{non}$;
- P : l'entier en base deux codé par le mot est-il premier ? Par exemple, $P(111) = \text{oui}$, $P(100) = \text{non}$;
- P : le mot est-il le codage binaire d'un programme C accepté à la compilation par `gcc` avec l'option `-ansi` ?

Un problème de décision P est *décidable* s'il existe une machine de Turing M_P telle que pour toute entrée $u \in \{0,1\}^*$, l'exécution de M_P sur u se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oui}, \\ 0 & \text{sinon.} \end{cases}$$

Lorsqu'il n'existe pas de telle machine de Turing, P est dit *indécidable*.

Intuitivement, un problème de décision P est décidable s'il est possible d'écrire, dans un langage de programmation Turing-complet, une fonction f paramétrée par un objet u et renvoyant `true` si $P(u) = \text{oui}$ et `false` sinon.

Le *problème de l'arrêt* est le problème de décision Arr prenant en entrée le codage binaire d'un programme f (une fonction) et le codage binaire d'une entrée a (un argument) et renvoyant oui si l'exécution du calcul $f(a)$ se termine et non sinon.

Théorème [Turing, 1936]

Le problème de l'arrêt Arr est indécidable.

Intuitivement, ceci dit que pour tout langage de programmation L, il est impossible de concevoir une fonction `halt` en L qui accepte en entrée une autre fonction f et un argument a, et qui teste si l'exécution de $f(a)$ se termine.

Montrons le résultat précédent par l'absurde en supposant que `halt` existe.

En pseudo-code (langage Turing-complet), cette fonction s'exprime par

```
fonction halt(f, a) :
    si f(a) se termine :
        renvoyer oui
    sinon
        renvoyer non
```

Soit maintenant la fonction `delta` définie par

```
fonction delta(x) :
    si halt(x, x) = oui :
        renvoyer delta(x) (A)
    sinon
        renvoyer oui (B)
```

L'appel `delta(delta)` produit une absurdité :

- si `delta(delta)` se termine, par *(A)*, `delta(delta)` ne se termine pas ;
- si `delta(delta)` ne se termine pas, par *(B)*, `delta(delta)` se termine.

Nous faisons donc face à une absurdité qui montre que la fonction `halt` n'existe pas.

4.5. λ-calcul

Une *fonction récursive* est une fonction

$$f : \mathbb{N}^k \to \mathbb{N}, \qquad k \geqslant 0,$$

qui est intuitivement calculable.

Exemples

- $f : (n_1, n_2, n_3) \mapsto n_1 + n_2 + n_3$
- $f : n \mapsto 1$ si n est pair, 0 sinon
- $f : n \mapsto 1$ si $n \leqslant 1, n \times f(n-1)$ sinon

Ces fonctions sont récursives au sens précédent.

La dernière est même récursive au sens usuel.

Exemple

En revanche, la fonction $f : \mathbb{N} \to \mathbb{N}$ définie par

$$f(n) := \begin{cases} 1 & \text{si } \mathrm{Arr}(g) = \text{oui où } g \text{ est le prog. tq. } \mathrm{rang}(g) = n, \\ 0 & \text{sinon,} \end{cases}$$

n'est pas une fonction récursive (si elle était calculable, le problème de l'arrêt serait décidable).

Le *λ-calcul* a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions calculables.

En λ-calcul, la notion première est celle d'expression. Les fonctions sont des expressions particulières.

Une *expression* du λ-calcul (ou *λ-terme*) est

1. soit une *variable*, notée x, y, z, ... ;
2. soit l'*application* d'une expression f à une expression g, notée $f\,g$;
3. soit l'*abstraction* d'une expression f, notée $\lambda x.\, f$ où x est une variable.

Exemples

Quelques expressions :

- ☐ x
- ☐ $x\,x$
- ☐ $\lambda x.\, x$
- ☐ $\lambda x.\, \lambda y.\, x$
- ☐ $z\,(\lambda x.\, \lambda y.\, x)$
- ☐ $(\lambda z.\, z)\,((\lambda x.\, x)\,(\lambda y.\, ((\lambda x.\, x)\,y)))$

Une occurrence d'une variable x est *liée* si elle fait partie d'une sous-expression qui est dans la portée d'une abstraction en x.

Une occurrence d'une variable qui n'est pas liée est *libre*.

Une expression qui ne contient aucune variable libre est *close*.

Exemples

Les occurrences de variables qui sont soulignées sont liées. Les autres sont libres.

1. $\lambda x.\, \underline{x}$
2. $\lambda x.\, y$
3. $\lambda x.\, \lambda y.\, (\underline{y}\, \underline{x})$
4. $x\, (\lambda x.\, y\, (\lambda y.\, (\underline{x}\, \underline{y})))$
5. $(\lambda x.\, y\, \underline{x})\, (\lambda y.\, \underline{y}\, x))$
6. $\lambda z.\, (\lambda x.\, (y\, \underline{x})\, \underline{z})\, (\lambda y.\, (\underline{y}\, x)\, \underline{z})$

La 1re et la 3e expression ici sont closes. Aucune autre ne l'est.

La *β-substitution en racine* consiste, étant donnée une expression de la forme $(\lambda x.f)\,a$, à substituer a aux occurrences libres de x dans f. Ceci forme l'expression $f[x := a]$. On note $(\lambda x.f)\,a \;\to\; f[x := a]$.

Exemple

$$(\lambda x.\,(x\,y)\,x)\,(z\,z) \;\to\; ((z\,z)\,y)\,(z\,z)$$

Une *β-substitution* d'une expression e_1 est une expression e_2 obtenue en remplaçant une sous-expression e'_1 de e_1 par l'expression e'_2 obtenue par β-substitution en racine de e'_1. Par extension, on note $e_1 \to e_2$.

Exemple

$$x\,((\lambda x.\,(x\,y)\,x)\,(z\,z)) \;\to\; x\,(((z\,z)\,y)\,(z\,z))$$

Attention : il y a ici une difficulté cachée qui survient lorsque a possède des variables libres qui se font capturer après une β-substitution. Ceci sera élucidé dans un prochain chapitre.

Une *valeur* est une expression sur laquelle il n'est pas possible d'appliquer de β-substitution.

Exemples

- x est une valeur ;
- $\lambda x.\, x\, y$ est une valeur :
- $(\lambda x.\, x)\, y$ n'est pas une valeur :
- $\lambda x.\, (x\, (\lambda y.\, \lambda z.\, z\, (y\, x))\, z)$ n'est pas une valeur.

L'*évaluation* d'une expression e est l'expression obtenue en appliquant des β-substitutions sur e jusqu'à obtenir une valeur. Elle peut ne pas exister.

Exemple

Voici une telle suite de β-substitutions jusqu'à l'obtention d'une valeur :

$$((\lambda x.\, \lambda y.\, x)\, u)\, v \;\to\; (\lambda y.\, u)\, v \;\to\; u$$

Il existe des expressions qui n'aboutissent pas à des valeurs, comme $(\lambda x.\, (x\, x))\, (\lambda x.\, (x\, x))$.

5. Caractéristiques principales des langages de programmation

5.1. Paradigmes

Un *paradigme* est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un *paradigme de programmation* conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

Il existe deux principaux paradigmes de programmation :

1. le paradigme impératif ;
2. le paradigme fonctionnel.

Le 1er se base sur la machine de Turing, le 2e sur le λ-calcul.

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

Les éléments suivants sont constitutifs de ce paradigme :

1. instructions d'affectation ;
2. instructions de branchement ;
3. instructions de boucle ;
4. structures de données mutables.

Des instructions peuvent modifier l'état de la machine en altérant sa mémoire.

En programmation fonctionnelle, la notion de fonction est centrale.

Un programme est une expression, celle-ci étant une imbrication de définitions et d'appels de fonctions.

L'exécution d'un programme est l'évaluation de son expression en utilisant les définitions des fonctions pour la simplifier au maximum.

Les éléments suivants sont constitutifs de ce paradigme :

1. liaison d'un nom à une valeur ;
2. récursivité ;
3. instructions de branchement ;
4. structures de données non mutables.

Il n'y a pas de notion d'état de la machine car celui-ci ne peut pas être modifié. Toutes les données sont représentées dans l'expression en cours d'évaluation.

Dans ce cours, nous adopterons au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

1. d'utiliser des variables (et donc aussi de procéder à des affectations) ;
2. d'utiliser des instructions de boucle ;
3. de produire des effets secondaires (sauf éventuellement pour la gestion des entrées/sorties).

En revanche, nous utiliserons de manière courante

1. des **fonctions récursives** ;
2. des **fonctions locales**.

Note : une constante peut-être vue comme une **fonction d'arité zéro** (c.-à-d. une fonction qui ne prend pas d'entrée).

Le principe de *transparence référentielle* stipule que dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Exemple

Considérons le code C suivant :

```
int f(int n) {
    printf("a");
    return n;
}

int g(int n) {
    return n;
}
...
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche `a` mais pas `g(1)`.

Le principe de transparence référentielle n'est donc pas respecté en C.

Règle : en programmation fonctionnelle pure, le principe de transparence référentielle s'applique.

5.2. Vérification des types

Dans la plupart des langages de programmation, les expressions sont classifiées selon la nature de la valeur qu'elles dénotent. On parle alors de *type*.

Ceci permet, lorsque des fonctions sont associées (par le biais par exemple de l'application d'une fonction à des arguments) de s'assurer qu'il n'y a pas d'incohérence manifeste. L'avantage est de pouvoir capturer des erreurs le plus tôt possible.

Exemple

Dans un langage typé proposant une fonction `add` acceptant deux entiers et renvoyant un entier, l'expression `add(10, "3")` serait mal typée car le 2e argument n'est pas un entier.

Il existe plusieurs stratégies de vérification des types :

1. *vérification dynamique*, où les types sont vérifiés lors de l'exécution du programme ;
2. *vérification statique*, où les types sont vérifiés lors de la compilation du programme ;
3. *vérification hybride*, où certains aspects sont vérifiés lors de la compilation et d'autres, lors de l'exécution du programme.

Exemple

Considérons le programme PYTHON

```
n = int(input())
if n % 2 == 0 :
    print(n / 2)
else :
    print(n[1])
```

Lors de son exécution, des erreurs dans la vérification des types peuvent se produire :

- si l'utilisateur saisit un entier, alors l'expression `int(input())` en l. 1 est bien typée ; sinon, elle ne l'est pas ;
- si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée ; sinon, c'est l'expression `n[1]` qui est évaluée. Celle-ci n'est pas bien typée car `n` n'est pas un tableau.

  Cette information n'est donnée que lors de l'exécution :

```
Traceback (most recent call last):
    File "Prog.py", line 5, in <module>
        print(n[1])
TypeError: 'int' object has no attribute '__getitem__'
```

Exemple

Considérons le programme C, similaire au précédent,

```c
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

Lors de sa compilation, une erreur de typage se produit : `n` est une variable de type `int` mais elle est traitée en l. 8 comme une variable de type `int *`.

Cette information est donnée lors de la compilation :

```
Prog.c: In function ‘main’:
Prog.c:8:25: error: subscripted value is neither array nor pointer nor vector
        printf("%d\n", n[1]);
                         ^
```

Vérification des types dynamique.

- Avantage : grande flexibilité dans l'écriture des programmes. L'ensemble des expressions correctes est plus large.
- Inconvénients : les problèmes de types ne sont mis en évidence qu'à l'exécution. Le programmeur doit donc faire attention à tester tous les cas de figure. Il y a une perte d'efficacité lors de l'exécution à cause de la vérification des types qui se fait au même moment et qui demande des ressources.

Vérification des types statique.

- Avantages : grande sécurité lors de l'écriture du programme. Les erreurs les plus courantes sont détectées à la compilation. L'exécution n'est pas accompagnée de la vérification des types, ce qui peut la rendre plus rapide.
- Inconvénient : moins de flexibilité dans l'écriture des programmes. L'ensemble des expressions correctes est plus restreint.

5.3. Attribution des types

Dans un programme dans un langage donné, il y a deux manières principales d'attribuer un type à une expression ou à une sous-expression donnée.

1. L'*attribution explicite* (nommée parfois *Church-style*) demande que les types des variables et fonctions sont mentionnés dans le programme.

   Dans ce cas, la vérification des types s'appuie sur les annotations de types mentionnées dans le programme afin de voir si elles sont cohérentes.

2. L'*attribution implicite* (nommée parfois *Curry-style*) demande que les types des variables et fonctions ne sont pas mentionnés dans le programme.

   Ils sont devinés lors de la compilation (si vérification statique) ou lors de l'exécution (si vérification dynamique) en fonction du contexte.

   Ce mécanisme s'appelle l'*inférence des types*.

3. L'*attribution hybride*, une approche hybride, qui demande certaines annotations de types (pour les expressions atomiques notamment) et infère le type des expressions composées.

Exemple

Considérons le programme OCaml

```
let pair (x : int) (y : bool) : (int * bool) =
    (x, y)

let () =
    let p = pair 2 true in
    print_int (fst p)
```

Les annotations de types explicites indiquent que `pair` est une fonction acceptant un entier et un booléen et renvoyant un couple dont la 1re composante est un entier et la 2e est un booléen.

Ces annotations de types forment aussi des contraintes : par exemple, l'appel `pair 2 5` provoquerait une erreur lors de la vérification des types.

Exemple

Considérons le programme OCaml, similaire au précédent,

```
let pair x y =
    (x, y)

let () =
    let p1 = pair 2 true in
    print_int (fst p1);
    let p2 = pair 2 5 in
    print_int (snd p2)
```

Ici, les annotations de types sont implicites au niveau de la définition de la fonction `pair`.

Le système d'inférence des types lui attribue automatiquement un type. Il devine le type le plus général possible pour cette fonction.

Ce type est `'a -> 'b -> 'a * 'b`, mentionnant que `pair` est une fonction acceptant une valeur de n'importe quel type (`'a`), une valeur de n'importe quel type (`'b`, potentiellement différent de `'a`) et renvoyant un couple dont la 1re composante est de type `'a` et la 2e est de type `'b`.

Nous observons ici que la fonction `pair` est appelée deux fois et avec des types d'arguments différents.

Tout ceci est en lien avec le polymorphisme (notion qui sera vue en détail plus tard).