W1-S4-C2-type-checking

December 15, 2014

1 Langages typés ou non

2 Typages statique et dynamique

2.1 Complément - niveau intermédiaire

Il est de tradition de distinguer, parmi les langages, ceux qui sont typés ou non. On parle aussi de langage à typage statique ou à typage dynamique. Ce notebook tente d'éclaircir ces notions pour ceux qui n'y sont pas familiers.

2.1.1 Typage statique

À une extrémité du spectre, on trouve les langages dits fortement typés, comme par exemple C/C++.

En C on écrira, par exemple, une version simpliste de la fonction factoriel comme ceci

```
#include <stdio.h>

int factoriel (int n) {
  int result = 1;
  int loop;
  for (loop = 1; loop <= n; loop ++) {
    result *= loop;
  }
  return result;
}
```

Comme vous pouvez le voir - ou le deviner - toutes les variables utilisées ici (comme par exemple `n`, `result` et `loop`) sont typées. * On doit appeler `factoriel` avec un argument `n` qui doit être un entier (`int` est le nom du type entier). * Les variables internes `result` et `loop` sont de type entier. * `factoriel` retourne une valeur de type entier.

C'est pourquoi on dit que C est fortement typé. Ces informations de type ont essentiellement trois fonctions. * En premier lieu, elles sont nécessaires au compilateur; en C si le programmeur ne précisait pas que `result` est de type entier, le compilateur n'aurait pas suffisamment d'éléments pour générer le code assembleur correspondant. * En contrepartie, le programmeur a un contrôle très fin de l'usage qu'il fait de la mémoire et du hardware. Il peut choisir d'utiliser un entier sur 32 ou 64 bits, signé ou pas, ou construire avec `struct` et `union` un arrangement de ses données. * Enfin, et surtout, ces informations de type permettent de faire un contrôle *a priori* de la validité du programme; par exemple si à un autre endroit dans le code on trouve

```
int main (int argc, char *argv[]) {
  /* le premier argument de la ligne de commande est argv[1] */
  char *input = argv[1];
  /* calculer son factoriel et afficher le resultat */
  printf ("Factoriel (%s) = %d\n",input,factoriel(input));
```

```
  /*                                                ^^^                */
  /* ici on appelle factoriel avec une entree 'chaine de caractere'   */
  return 0;
}
```

alors le compilateur va remarquer qu'on essaie d'appeler `factoriel` avec comme argument `input` qui, pour faire simple, est une chaîne de caractères; et comme `factoriel` s'attend à recevoir un entier ce programme n'a aucune chance de fonctionner.

On parle alors aussi de **typage statique**, en ce sens que chaque variable a exactement un type, qui est défini par le programmeur une bonne fois pour toutes.

C'est ce qu'on appelle le **contrôle de type**, ou *type-checking* en anglais. Si on ignore le point sur le contrôle fin de la mémoire, qui n'est pas crucial à notre sujet, le contrôle de type présente * l'**inconvénient** de demander davantage au programmeur, * et l'**avantage** de permettre un contrôle étendu, et surtout précoce (avant même de l'exécuter) de la bonne correction du programme.

Cela étant dit, le typage statique en C n'empêche pas le programmeur débutant d'essayer d'écrire dans la mémoire à partir d'un pointeur `NULL` - et le programme de s'interrompre brutalement. Il faut être conscient des limites du typage statique.

2.1.2 Typage dynamique

À l'autre bout du spectre, on trouve des langages comme, eh bien, python.

Pour comprendre cette notion de typage dynamique, regardons la fonction suivante `somme`:

```
In []: def somme(*largs):
           "retourne la somme de tous ses arguments"
           if not largs:
               return 0
           elif len(largs) == 1:
               return largs[0]
           else:
               result = largs[0] + largs[1]
               for i in range(2, len(largs)):
                   result += largs[i]
               return result
```

Naturellement vous n'êtes pas à ce stade en mesure de comprendre le fonctionnement intime de la fonction. Mais vous pouvez tout de même l'utiliser.

```
In []: somme(12, 14, 300)
```

```
In []: l1 = ['a', 'b', 'c']
       l2 = [0, 20, 30]
       l3 = ['spam', 'eggs']
       somme(l1, l2, l3)
```

Vous pouvez donc constater que `somme` peut fonctionner avec des objets de types différents. En fait, telle qu'elle est écrite, elle va fonctionner si il est possible de faire `+` entre ses arguments. Ainsi par exemple on pourrait même faire

```
In []: somme('abc', 'def')
```

Mais par contre on ne pourrait pas faire

```
In []: # ceci va déclencher une exception à run-time
       somme(12, [1, 2, 3])
```

On voit qu'on se trouve dans une situation très différente de celle du programmeur C++ en ce sens que * à l'écriture du programme il n'y aucun des surcoûts qu'on trouve avec C ou C++ en terme de définition de type, * aucun contrôle de type n'est effecué *a priori* par le langage au moment de la définition de la fonction `somme`, * par contre au moment de l'éxécution, si on retrouve en situation de faire une somme entre deux types qui ne peuvent pas être additionnés, comme ci-dessus avec un entier et une liste, le programme ne pourra pas se dérouler correctement.

Il y a deux points de vue vis à vis de la question du typage.

Les gens habitués au typage statique se plaignent du typage dynamique en disant qu'on peut écrire des programmes faux et qu'on s'en rend compte trop tard - à l'exécution, en insistant sur les parties qui traitent des cas particuliers, qui sont moins fréquemment utilisées, et qui donc nécessitent des tests plus approfondis pour s'assurer que la couverture des tests est complète.

À l'inverse les gens habitués au typage dynamique font valoir que le typage statique est très partiel; par exemple en C si on essaie d'écrire au bout d'un pointeur nul, l'OS ne le permet pas et le programme sort tout aussi brutalement.

Selon le point de vue le typage dynamique est vécu comme un inconvénient (pas assez de bonnes propriétés détectées par le langage) ou comme un avantage (pas besoin de passer du temps à déclarer le type des variables, ni à faire des conversions pour satisfaire le compilateur).

2.1.3 Inférence de type

Une approche, qu'on peut qualifier d'hybride entre ces deux approches, est connue sous le nom d'inférence de type, et a été mise en œuvre notamment dans le langage ML. L'idée est de laisser au langage le soin de déterminer le - ou les - type(s) qui sont permis à un endroit du code en fonction du contexte.

Pour ceux que ce sujet intéresse, sachez qu'il existe un projet baptisé `mypy` qui vise à creuser cette voie pour python.

Guido van Rossum a exposé son avis sur cette proposition; il semble qu'on puisse s'attendre dans le futur (mais dans le cadre de python-3) à avoir la possibilité (optionnelle) d'annoter le code d'une fonction avec des informations de type. On pourrait alors par exemple utiliser un outil comme `mypy` pour calculer de telles annotations.