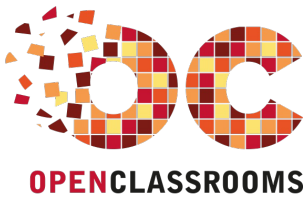
Le typage : présentation thématique et historique

Par bluestorm



www.openclassrooms.com

Sommaire

Le typage : présentation thématique et historique

Par  bluestorm

Mise à jour : 11/07/2009

Ce tutoriel est un peu particulier, car il n'est pas destiné à vous enseigner une technique particulière, ou un langage de programmation particulier.

Il s'agit d'un tuto sur le typage, un concept utile et très répandu dans l'ensemble des langages de programmation. Je compte expliquer les raisons qui ont motivé son apparition, ses principes fondamentaux, ses différentes formes et pourquoi pas, vers la fin, quelques fonctionnalités avancées.

Ne vous attendez donc pas à ressortir de la lecture (ou pas) de ce tuto avec de nouvelles connaissances concrètes, à mettre en oeuvre pour l'écriture d'applications. Attendez-vous à réfléchir sur un sujet souvent présenté comme allant de soi, qui concerne la pratique de la programmation en général, et, dans le meilleur des cas, qui vous aura permis de comprendre un peu mieux votre propre pratique de la programmation.

Les exemples seront pris dans des langages très divers (que vous n'aurez probablement pour la plupart jamais vus, mais c'est rigolo de découvrir de nouvelles choses, non ?), et j'ai un faible pour l'aspect historique des choses, même si la progression que suit le tuto est plus centrée sur la logique que sur la chronologie.
Sommaire du tutoriel :

Tout est nombre, pour l'ordinateur

Les ordinateurs manipulent des nombres. Ils stockent des nombres en mémoire, font transiter des nombres par leurs circuits, et envoient des informations aux autres ordinateurs sous forme de nombres. C'est l'ère du numérique.

La programmation, c'est (entre autres) le fait de dialoguer avec l'ordinateur, afin de lui faire faire les opérations que l'on veut : on décide ce qu'on veut, on le traduit dans un langage compréhensible pour l'ordinateur, et il l'exécute (programmer, c'est donc essentiellement traduire de la pensée en code).

Comme c'est un ordinateur, il ne manipule que des nombres, et ces opérations se transforment donc forcément, à un moment ou un autre, en opérations sur des nombres. Quand on veut faire faire à l'ordinateur des additions, multiplications ou autres opérations mathématiques (il fait ça très bien), ce n'est pas surprenant de savoir qu'il manipule des nombres. Mais quand on clique sur un bouton, qu'on choisit une couleur dans un logiciel de dessin ou qu'on écrit du texte dans un éditeur, est-ce que l'on conçoit clairement de quelle manière il va interpréter la chose en tant que nombre ? Non. Ce n'est d'ailleurs pas possible, parce que la manière dont est effectuée la traduction dépend de beaucoup de choses, que ce soit au niveau logiciel (encodage, format utilisé...) ou matériel (processeur, mémoire, etc.).

Quand tout est nombre pour le langage de programmation...

Les langages de bas niveau actuels, ainsi que les premiers langages de programmation sont très proches de la machine sur laquelle ils s'exécutent : les objets qu'ils manipulent sont, eux aussi, des nombres.

En Assembleur par exemple, la majorité des instructions consiste à modifier de manière arithmétique (en additionnant, multipliant, ou effectuant d'autres opérations de ce genre) le contenu des cases mémoire de l'ordinateur. Un exemple encore plus frappant est le Brainfuck, un langage dont les seules opérations (à part l'entrée-sortie) permettent soit de changer de case mémoire, soit de changer la valeur de la case actuelle (et de boucler tant que la case ne contient pas 0). Pour les curieux, voici une multiplication en Brainfuck :

Code : Autre

```
[->[->+>+<<]>[-<+>]<<]>>>
```

Des types pour plus de sûreté

Ces langages conviennent très bien quand il s'agit de manipuler des nombres (ils ont au départ été conçus pour des applications scientifiques), mais assez vite des problèmes se posent.
Mettons que vous ayez écrit une opération qui transforme une chaîne de caractères (une suite de nombres en mémoire) dans la même suite, mais où tous les caractères majuscules sont remplacés par leurs équivalents minuscules. Cela suppose de bien connaître la convention de conversion nombre <-> caractère de votre langage, mais ce n'est sans doute pas un problème pour vous.

Imaginez maintenant que vous appliquiez par erreur cette opération non pas sur une phrase, mais sur la liste des âges des membres de votre site. L'ordinateur n'y verra que du feu : il traduira cette liste en une liste assez ressemblante, mais dans laquelle, par exemple, toutes les personnes âgées de 65 à 90 ans se seront vues vieillies de 32 ans ! Heureusement que les visiteurs de votre site sont plutôt jeunes...

Cette erreur est assez courante, et même si elle peut être évitée en faisant attention, elle peut être gênante. Imaginez qu'au lieu de la liste des âges des membres, vous ayez mis la liste de vos mots de passe, voire la liste des clés de contrôle du réacteur nucléaire ( ) !

Peut-on l'éviter ?

Cette erreur est aisément repérable : un humain, si on lui demande de transformer en lettres majuscules une liste d'âges, va comprendre immédiatement qu'il y a quelque chose qui cloche. Pourquoi l'ordinateur ne pourrait-il pas faire pareil ?
C'est ce que se sont demandés très vite les ingénieurs concevant les ordinateurs et les langages de programmation, et ils ont vite décidé de mettre en place une gestion simple de ce genre d'erreur dans ces langages.

L'idée intuitive est qu'une lettre et un âge, ce "n'est pas la même chose". Comme vous l'a peut-être dit un jour votre professeur de physique, "on ne peut pas ajouter des patates et des carottes". On a donc introduit une sorte de *classification* des objets que manipule le langage : à chaque valeur du langage, on associe une étiquette qui dit "c'est une lettre", "c'est un âge" (ou "c'est un nombre"), "c'est une liste de lettres suivie par deux nombres entiers, puis un nombre à virgule", etc.
Cette étiquette s'appelle le **type** : le type d'une valeur est tout simplement le groupe d'objets auxquels il appartient.

Ensuite, on précise quand on déclare une variable à quel type elle appartient, et, quand on définit une opération (par exemple, la mise en minuscules), on précise (dans le code source du programme) sur quel type de données elle *a le droit* d'agir.

Alors, il suffit de mettre en place, pendant que l'ordinateur lit le texte de votre programme en prévision de son exécution, une étape très simple qui vérifie que chaque opération est bien utilisée sur les variables du type qui convient, et sinon de rapporter l'erreur au programmeur.

Le typage, c'est sémantique

La réaction des informaticiens face aux erreurs de ce genre (que l'on appelle depuis les **erreurs de typage**) est naturelle. Si vous aviez été ingénieur chez IBM dans les années 50, vous auriez sûrement réagi pareil.

Ce qui est intéressant, c'est que la démarche du typage s'inscrit dans une direction plus générale, que l'on peut observer en de nombreux endroits de l'histoire des langages de programmation : on a apporté du sens au langage.

Quand vous concevez votre programme de mise en minuscules, vous savez très bien quel type de variables vous allez manipuler.

L'ordinateur, lui, ne le savait pas. En introduisant le typage, on a permis au programmeur de rajouter des informations (dans le code source), pour en quelque sorte augmenter les connaissances de l'ordinateur au sujet du programme. Plus l'ordinateur est au courant, mieux il peut vérifier que ces informations sont cohérentes, et vous avertir si cela n'est pas le cas.
Le fait de modéliser le sens des objets manipulés par un programme est généralement appelé de la sémantique.
La plupart des langages informatiques que vous connaissez se sont préoccupés de sémantique (de la manière dont les programmes qu'on pouvait écrire reflétaient bien la signification des actions voulues par le programmeur). Le C (créé pour être portable, et donc avoir des instructions reflétant mieux (qu'en **assembleur**) le sens des fonctions désirées, en faisant abstraction de l'architecture matérielle de l'ordinateur) et le XHTML (dont une des grandes avancées a été de mettre l'accent sur les balises décrivant la structure des pages web - et donc leur sens -, au lieu de leur présentation) en sont de bons exemples.

Cela ajoute effectivement du sens au programme

Regardez le code de la fonction qui met en minuscule un caractère majuscule, en C :

Code : C

```
char tolower(char c)
{
   if (c >= 'A' && c <= 'Z')
      return (c - 'A' + 'a');
   else return c;
}
```

Si la lettre est comprise entre 'A' et 'Z', on la décale dans les minuscules (ce n'est pas grave si vous ne comprenez pas pourquoi ça marche, et d'ailleurs ça ne marchera pas forcément, car cela repose sur des fonctionnalités qui dépendent de l'environnement), sinon on la renvoie inchangée.

Si un informaticien lit ce code, il comprendra immédiatement ce que fait cette fonction. Mais un des intérêts du typage repose dans le fait qu'en pratique, on n'est pas obligé de lire tout le code !
En effet, il est possible en C d'insérer avant le code d'une fonction un *prototype*, qui la déclare à l'avance, et qui est souvent placé dans les fichiers de header (.h) :

Code : C

```
char tolower(char );
```

Ce qui est surprenant, c'est que cette ligne suffit pour comprendre ce que fait la fonction tolower ! Évidemment, cela suppose de connaître l'anglais, mais une fois que vous voyez le type (la fonction prend un char et renvoie un char), la signification est claire (de toute façon, en 1950, les ingénieurs d'IBM parlaient tous anglais).

Ainsi, le typage a effectivement apporté du sens dans le code : on peut se contenter de lire le prototype d'une fonction pour la comprendre ; c'est un gros gain de temps par rapport à l'époque où il fallait chercher le code source de la fonction dans tout le programme (bon, la documentation ça existe, mais les vrais hommes (et femmes !) chez IBM lisaient plutôt le code source, évidemment).

... mais ce n'est pas si nouveau !

Si vous avez déjà fait de la physique, vous avez sûrement remarqué l'importance qu'attachent les physiciens aux unités des valeurs qu'ils manipulent : pas question d'ajouter des mètres et des kilogrammes ! Cette restriction, cette classification des valeurs a de nombreux avantages : entre autres, elle permet d'obtenir très vite des informations sur une grandeur (c'est une distance, c'est une température...), et elle réduit les risques de faire des erreurs de calcul : il est très simple de vérifier qu'une formule est bien typée (on dit qu'elle est "homogène"), et si ce n'est pas le cas on sait tout de suite que la formule est fausse ! C'est un peu comme cela qu'on utilise en général les types en informatique.

Langages dynamiques, langages statiques

La description que j'ai faite du processus de vérification du typage est plus que simpliste. En fait, il existe grosso-modo deux manières répandues de procéder : le typage statique, et le typage dynamique.

La différence réside dans le moment où la vérification est effectuée : au moment où les opérations sont effectuées, ou plutôt une fois pour toutes, avant de lancer le programme ?

Un langage typé dynamiquement : Python

Le principe du typage dynamique, c'est de typer au besoin, pendant l'exécution du programme : on fait ce que demande le programme, et on vérifie à chaque opération que le type correspond bien (ce qui peut se faire assez rapidement si le système des types est simple).
C'est le typage le plus utilisé par les langages de script.
Voici un exemple en Python :

Code : Python

```
def essai(test):
    if test:
        return (1 + 2)
    else:
        return (1 + "essai")
```

On a défini une fonction qui prend un booléen en argument, et renvoie (1 + 2) s'il est vrai, et (1 + "essai") s'il est faux. Il y a une erreur de typage (on ne peut pas ajouter un entier et un mot !) mais l'interpréteur n'a pas protesté. On essaie ensuite d'utiliser la fonction :

Citation

```
>>> essai(True)
3
>>> essai(False)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

essai(True) renvoie un résultat correct, et seul essai(False) renvoie une erreur. Cela veut dire que tant qu'on n'est pas entré dans la branche "else" du if, aucune vérification de typage n'a été faite sur son contenu : on a véritablement un typage dynamique.

Ce typage est très simple à comprendre et à mettre en oeuvre. Cependant, un de ses désavantages apparents ici est le fait que pour être sûr que les types du programme sont corrects, il faut tester tous les "embranchements" du code. Ici, il faut faire deux tests, mais pour peu que vous ayez plusieurs if et des conditions un peu compliquées, tester l'ensemble du code devient vite long et difficile (même s'il existe des outils pour tenter d'automatiser le processus).
Avec un typage statique, comme vous allez le voir, toutes les erreurs seront relevées avant l'exécution du code.

Un langage typé statiquement : C

Un langage est typé statiquement si, avant de s'exécuter, votre code source passe par une phase de vérification des types. Si le langage est compilé, cette phase est souvent comprise dans le compilateur, mais il existe aussi des langages interprétés (ou des implémentations interprétées de langages, plus exactement) qui ont un typage statique.

L'exemple que j'ai choisi est le C. Ce langage n'a pas un système de types très évolué (il n'empêche pas beaucoup de choses), mais cela reste un typage statique, et il a inspiré les typages des langages qui l'ont suivi (le C++, principalement). De plus, indépendamment des contraintes de typage imposées par le langage, les compilateurs ont développé des techniques de recherche d'erreurs probables (si le code est valide, mais assez bizarre pour être une erreur dans la plupart des cas), qui donnent lieu à des avertissements, et peuvent donc être considérées comme une sécurité supplémentaire au niveau du typage.

Un exemple de problème de typage en C est la comparaison d'un entier de type "unsigned int" (un type qui ne peut contenir que des entiers positifs) et "int" (ou "signed int"), qui peut représenter des nombres négatifs.

Code : C

```
#include <stdio.h>

const unsigned int i = 23;
const int j = -5;

int main(void)
{
        if (i < j) printf("i < j\n");
```

```
                else printf("i > j\n");

                return 0;
        }
```

Si vous compilez ce code, vous verrez peut-être (si vous avez les *warning* activés) un message équivalent à celui-ci :

Citation

```
test.c: In function 'main':
test.c:8: warning: comparison between signed and unsigned
```

Le compilateur a repéré l'erreur de type, mais il compile le programme quand même. Si vous l'exécutez, vous verrez qu'il dit que i est plus petit que j !
23 < -5, ça ne ferait pas plaisir à votre prof de maths, mais c'est normal si l'on connaît les règles de conversion utilisées pour passer d'un int à un unsigned int.

L'avantage du typage statique, c'est qu'il nous prévient de l'erreur avant l'exécution. De plus, il vérifie tout le code, même celui qui se trouve dans des branches de "if".
Le typage statique présente aussi un avantage non négligeable du point de vue des performances : une fois que le test de typage a été effectué, le programme est supposé valide, et le compilateur peut alors se débarrasser des informations de typage qu'il a accumulées. À l'inverse, les typages dynamiques imposent de conserver pendant toute l'exécution les informations sur le type de chaque variable, et de faire des tests tout au long du programme, ce qui a un impact sur les performances.

Tout est dans la nuance

Dans le monde du typage, rien n'est tout blanc ni tout noir. Tout est dans la nuance : il existe un grand nombre de manières différentes de typer, et tous les langages font ça un peu différemment, avec plus ou moins de contraintes. Ainsi, même les langages que l'on désigne couramment comme "non typés" (comme Lisp) ont généralement une certaine forme (très réduite) de typage.

De même, aucun langage à ma connaissance n'est complètement rigoureux au niveau du typage. Par exemple, l'opération de division / peut s'effectuer sur des entiers, et provoque une erreur quand le deuxième argument est nul (division par zéro). Dans un langage complètement typé (au sens mathématique du terme), il faudrait définir un type "entier non nul" qui serait celui du deuxième argument de la division, et il faudra convertir, et n'autoriser que des variables de ce type (ou faire une conversion implicite).

Le typage se mesure donc en degrés, en niveaux, et assez approximativement. Les langages les plus typés actuellement sont désignés comme pratiquant un **typage fort**, alors qu'on parle pour les autres (par exemple pour le C ou le PHP) de **typage faible**. Ces notions sont indépendantes de l'aspect statique ou dynamique du typage : les *pythonneux* considèrent qu'ils ont un typage dynamique fort.

La différence entre le typage fort et le typage faible n'est pas très claire, car il n'y a pas de consensus à ce sujet : la définition dépend des gens, et si vous discutez de typage fort avec quelqu'un, demandez-lui celle qu'il utilise avant de provoquer des malentendus. Une définition possible du typage faible est la suivante : les langages faiblement typés sont les langages qui n'attachent pas une grande importance au typage. Par exemple, si vous faites une erreur de typage en PHP, dans la plupart des cas il utilisera un comportement par défaut (ignorer la variable qui pose problème, ou la convertir automatiquement vers un autre type, etc.), qui ne sera pas forcément celui que vous vouliez, mais il ne vous préviendra pas qu'il y a une incohérence, à moins que vous ne l'ayez configuré en mode ultra-paranoïaque.

Formalisons tout ça

Quand les premiers langages de programmation se sont mis au typage (bien que certain langages, comme Lisp, soient restés non typés, ou très peu typés, pendant très longtemps), chaque créateur de langage a mis au point sa propre recette.

Pendant quelques temps, le typage est resté une idée très concrète (on choisit un certain nombre de "groupes" dans lesquels se répartissent les variables), sans bénéficier d'une formalisation : on ne s'est globalement pas intéressé à ses aspects théoriques.

C'est vers les années 70, donc pendant ce qu'on appelle habituellement la deuxième génération des langages de programmation (la période qui a aussi vu naître, entre autres, le C et les langages objets), qu'ont commencé à se répandre des modèles formalisés de système de types de certains langages de programmation.

Le langage ML

Ainsi apparaît en 1973 le langage ML, dont le système de type est un précurseur d'absolument tous les systèmes de typage statique formalisés existant actuellement. Il existait déjà quelques systèmes formels de typage auparavant, mais ils n'étaient pas assez avancés pour avoir l'influence de celui de ML.

Ce langage a été créé par Robin Milner, pour être utilisé dans le démonstrateur automatique de preuves LCF, qu'il développait à l'époque à l'université d'Edimbourg. Un démonstrateur automatique de preuves est un logiciel qui prend en entrée un théorème mathématique et sa preuve (écrite dans un langage spécialisé), et capable de vérifier mécaniquement (informatiquement, sans intervention humaine) que la preuve est correcte (et correspond bien au théorème).

Comme vous pouvez le constater, on s'est éloigné des ingénieurs d'IBM : la formalisation du typage est apparue du côté de l'informatique théorique, dans un environnement très mathématique. Et ce n'est pas par hasard.

Nos cousins les mathématiciens

Ce n'est pas pour rien que les systèmes de typages évolués sont apparus en informatique sous l'influence des mathématiciens : comme beaucoup de concepts de l'informatique, ils ont en fait été inventés par eux, et ce bien avant l'apparition des premiers ordinateurs !

Il faut savoir qu'au début du 20^{e} siècle, les mathématiciens ont rencontré une série de problèmes liés à la théorie des ensembles, qui ont montré la faiblesse de ce qu'ils pensaient à l'époque être une formalisation satisfaisante des mathématiques. Un exemple très accessible et très connu de ces difficultés théoriques est le paradoxe découvert par Russel en 1901, et que l'on peut formuler ainsi : "le barbier rase tous ceux qui ne se rasent pas eux-mêmes (et personne d'autre). Le barbier se rase-t-il ?". La question amène à un paradoxe : essayez et vous verrez, quoi que fasse le barbier, on arrive à une contradiction :

- soit le barbier se rase, ce qui crée une contradiction car il ne doit raser que les personnes qui ne se rasent pas elles-mêmes ;
- soit le barbier ne se rase pas, et alors il devrait se raser, car il doit raser toutes les personnes qui ne se rasent pas.

Les mathématiciens ont résolu le problème en inventant des systèmes où le barbier (enfin, l'analogue mathématique du barbier) n'existe pas, c'est-à-dire où on ne peut pas construire des objets qui vérifient des propriétés aussi tordues que ça.
L'un de ces systèmes est une théorie des types, mise au point par Russel et Whitehead dans les années **1910** ! Elle est extrêmement complexe, et ce n'est donc pas le modèle utilisé de nos jours en mathématiques, mais historiquement c'est une des premières apparitions d'un typage proche de ce que nous connaissons dans la littérature scientifique.

Le typage a été réutilisé à nouveau dans les années 1940 (donc, encore avant l'informatique) par Church, un logicien qui est à la base du Lambda calcul, le premier langage de programmation (théorique ), inventé en 1930 pour résoudre un problème algorithmique d'origine mathématique : "quelles sont les fonctions que l'on peut calculer ?". Le Lambda calcul était initialement non typé, et il s'est très vite avéré que la plupart des termes (ou programmes) que l'on construisait ne terminaient pas, c'est-à-dire avaient un "temps d'exécution" infini. L'introduction du typage a permis l'ajout de contraintes garantissant qu'aucun programme ne boucle à l'infini ; malheureusement, ces contraintes font que certaines fonctions ne peuvent plus être construites par le langage. C'est un point théorique assez technique, et je n'en parlerai plus par la suite , rassurez-vous 

Une préférence pour le typage statique

Cette formalisation a permis la création d'algorithmes puissants permettant de traiter les types d'un programme. Certaines de ces fonctionnalités seront décrites dans les chapitres suivants.
Cette puissance supplémentaire a cependant un coût : elle rend plus complexe la phase de traitement des types. Elle est donc relativement inadaptée au typage dynamique (qui se fait "sur le pouce", à chaque exécution du programme, et a donc besoin d'être relativement léger) ; c'est pourquoi elle a été mise en place quasi-exclusivement pour les langages à typage statique.

Cela peut aussi être relié à l'influence des mathématiciens : ce qu'a apporté la formalisation du typage, c'est la possibilité de **prouver** qu'un programme est correctement typé. Quand on peut démontrer une partie de la sécurité (dans le sens "absence de bugs") d'un programme, vous pensez qu'on choisit de le faire plutôt avant de le lancer, ou pendant qu'il tourne ?

Les langages que je présenterai par la suite seront donc majoritairement statiquement typés, car ce sont eux qui ont les systèmes de typage les plus élaborés.

Inférence : quand le système se débrouille tout seul

L'inférence des types est une des commodités qu'a apporté la formalisation du système de typage des langages de programmation. Sa version simplifiée a été découverte par Feys et Curry en 1958, mais l'algorithme général, encore utilisé dans les langages modernes à inférence de types, a été découvert par Hindley en 1969, et redécouvert indépendamment par Milner en 1978, qui l'a alors intégré à son langage ML.

Le principe de l'inférence de type, c'est que l'ordinateur peut, dans la plupart des cas, **deviner** de quel type sont les valeurs manipulées.
Voici par exemple le programme ***périmètre_rectangle***, qui calcule le périmètre d'un rectangle dont on donne les deux dimensions, écrit en pseudo-code (un langage qui n'existe pas et qui sert juste à décrire les algorithmes) :

Citation

> Soit périmètre_rectangle (longueur, largeur) = 2 * (longueur + largeur)

Comme vous pouvez le voir, le code ne comporte aucune annotation précisant le type des valeurs 'longueur' et 'largeur', et indiquant le type de retour de la fonction. Mais sont-elles bien nécessaires ?
En fait, on peut s'en passer : on voit bien que les opérations (multiplier, ajouter) sont des opérations arithmétiques, et on en déduit tout naturellement que cette fonction s'applique à des nombres (le type numérique exact dépend du langage que vous utiliserez pour l'implémentation).

Comme c'était le cas pour les erreurs de typage, l'ordinateur passe pour l'instant à côté d'une information qui nous est accessible : nous pouvons deviner le type attendu de cette fonction. Eh bien l'algorithme de Hindley-Milner permet de faire exactement la même chose : s'il connaît le type des fonctions de base du langage, il peut en déduire le type des fonctions ou des valeurs que vous construisez.
Si pendant la phase où il infère (devine) les types présents dans le programme, le système de types découvre une contradiction, alors le programme n'est pas correctement typé, et il renvoie une erreur.

Voici un exemple du code en OCaml (le langage ML que j'utilise) :

Code : OCaml

```
let périmètre longueur largeur = 2 * (longueur + largeur)
```

Si l'on demande à OCaml le type de la fonction "périmètre", il renvoie ceci :

Citation

> val périmètre : int -> int -> int

La valeur périmètre est de type `int -> int -> int`, ce qui signifie qu'elle prend deux entiers (les deux premiers) et renvoie un entier (le dernier après la deuxième flèche).

On a donc combiné les avantages de sûreté (le compilateur découvre une partie des erreurs du programmeur, qui se manifestent par l'emploi de types contradictoires), de sémantique (on peut demander le type d'une fonction au compilateur, et cela aide pour comprendre sa signification) avec une légèreté fort agréable : plus besoin de rédiger à la main les types les plus évidents (on se doute bien que la fonction "multiplie" ne va pas renvoyer un couple de nombres !).
Encore une fois, on a pu reporter une partie du travail sur l'ordinateur, et en décharger le pauvre programmeur.

Cette fonctionnalité est longtemps restée confinée dans les langages dérivés du ML (les plus utilisés actuellement étant le SML et le OCaml). Récemment, des langages "mainstream" (c'est-à-dire moins utilisés - parce qu'ils sont moches - par les chercheurs en info, et plus par les ingénieurs informaticiens - idem) ont commencé à intégrer cette fonctionnalité : c'est le cas de Java et C#.

Cependant, la garantie d'une inférence totale de tous les types du programme par le compilateur (sans aide de la part du programmeur) impose des contraintes assez subtiles (et parfois assez restrictives) sur le langage, et n'est possible à ma connaissance que dans les langages ML.

Polymorphisme paramétrique

Le polymorphisme paramétrique, c'est le fait qu'une valeur donnée d'une fonction puisse, sur plusieurs appels différents de la fonction, avoir des types différents.
L'idée se voit très bien avec l'inférence de type ; mettons que je code la fonction identité (ou '`id`'), qui ne fait par définition *rien* à son argument, puisqu'elle le renvoie *à l'identique* (on dirait pas mais cette fonction est centrale en maths : si, si) :

Citation

> Soit identité x = x

En OCaml, cela s'écrit ainsi :

Code : OCaml

```
let id x = x
```

Comment le compilateur va-t-il deviner le type de la variable x ? Justement, il ne peut pas trop : les fonctions auxiliaires qui pourraient l'aider (comme **+** et ***** dans l'autre programme, qui permettent de comprendre qu'on manipule des nombres) ont disparu !

On est donc dans un cas où le compilateur ne "peut pas deviner" le type de la fonction. C'est dans ces cas-là qu'on rencontre le ***polymorphisme*** : le compilateur déclare "x est de type quelconque inconnu, que j'appellerai `'a`", et il fonctionne avec le type `'a` comme s'il s'agissait d'un type qu'il connaît.

Voici ce que répond le compilateur quand on lui demande le type de notre fonction id :

Citation

> val id : 'a -> 'a

Il a donc donné un nom au type inconnu qu'on lui avait présenté.

L'avantage du polymorphisme va plus loin : en plus d'être inconnu, ce type est quelconque : n'importe quelle valeur du langage pourrait être donnée à la fonction `id`.
Ainsi, si je donne un entier à `id` (par exemple '`id 3`' renvoie 3), `id` va rester de type `'a -> 'a` (ce n'est pas parce qu'on lui a donné un entier une fois qu'elle devient de type `int -> int`). On pourra toujours, dans la suite du programme, lui donner une chaîne de caractères, un tableau d'entiers, etc.

On a donc ajouté de la richesse à notre système de types : en plus d'avoir des types particuliers, on a aussi des types "généralistes" qui permettent d'être employés sur tous les types.

Vous allez me dire "ça existe aussi en C, on a `(void *)`". Il y a une différence notoire entre les deux : si vous construisez par exemple une liste chaînée dont le champ de données est de type `void*` (pour pouvoir y mettre n'importe quoi), quand vous mettez une valeur dedans, elle devient de type `void*` et **perd** son information de typage. Il est alors possible de la manipuler comme toutes les autres données de type `void*`, et à ce niveau-là de faire des erreurs.
Par exemple, si vous avez une liste chaînée qui contient des `strings`, et l'autre qui contient des tableaux d'entiers, les données seront forcées de devenir des `void*` en étant insérées dans la liste, et il sera possible quand vous les extrairez de ces listes (par exemple "donne-moi le premier élément de la liste des tableaux de notes") de les faire passer de `void*` à un type différent de celui qu'elles avaient à l'origine (par exemple, de mélanger des fonctions s'appliquant à la liste des tableaux d'entiers et à la liste des `strings`).

Le ***polymorphisme paramétrique*** est plus puissant que cela : le type de données générique est "`'a list`" (liste d'objets de n'importe quel type), mais si vous construisez une liste d'entiers, elle sera de type "`int list`". En effet, `'a` est une lettre qui désigne le type (inconnu) de la liste ; si, pour une liste particulière, le type que représente `'a` est connu, cette liste n'est plus polymorphe, et `'a` est remplacé par le véritable nom du type. `'a` joue un rôle de "paramètre de types", d'où le nom "polymorphisme paramétrique".

Ainsi, une liste de `strings` sera de type "`string list`", une liste de tableaux d'entiers "`int array list`" (on doit lire "`(int array) list`"), et vous ne pourrez pas les mélanger entre elles : le typage vous en empêche. Par contre, les fonctions qui prennent en entrée le type "`'a list`" (par exemple : "quelle est la taille de la liste donnée", car la taille de la liste ne dépend pas du type de ses éléments) pourront s'appliquer aux deux types de listes différents, puisque "`int array list`" et "`string list`" sont tous les deux des *cas particuliers* de "`'a list`" (où `'a` représente respectivement le type `int array` et le type `string`).

Le polymorphisme paramétrique n'est pas forcément évident à comprendre, alors voici deux petits exemples pour la route :

Code : OCaml

```
let gauche x y = x
```

Le type de cette fonction est "`'a -> 'b -> 'a`". Cela signifie que le premier argument est de type `'a`, le deuxième de type `'b`, et que la valeur de retour est de type `'a` (vu qu'on renvoie le premier argument). Que vient faire le `'b` ici ? Il indique que le deuxième argument est de type quelconque, et *non nécessairement* le même que le premier : pour différencier deux "types inconnus", on leur met un nom de paramètre différent.

Code : OCaml

```
let choisir test x y = if test then x else y
```

Le type de cette fonction est "`bool -> 'a -> 'a -> 'a`" : elle prend en argument un booléen (représentant une valeur de vérité, `true` ou `false`), un argument de type `'a`, et un autre argument de type `'a`, et renvoie soit le deuxième, soit le troisième argument, selon la valeur du booléen.
Pourquoi les deux arguments sont-ils du même type `'a` ? Parce que la valeur "choisir" ne peut avoir qu'un seul type de retour. Si le troisième argument était de type `'b`, le type de retour de la fonction devrait être "soit `'a`, soit `'b`", ce qui n'est pas possible. Par inférence, le compilateur a donc conclu que les deux arguments étaient nécessairement du même type.

Un nouveau point de vue : le type comme interface

Le travail sur le polymorphisme paramétrique permet un nouveau point de vue sur les types. Souvenez-vous, la raison initiale à l'introduction des types polymorphiques était l'existence d'une valeur qu'on "ne sait pas inférer".

Le point de vue est donc le suivant : le programmeur code son programme, manipule les valeurs comme il le souhaite, et après, le compilateur se débrouille pour trouver le type des variables comme il le peut. Quand il ne sait pas, il met un type polymorphe.

Ce point de vue n'est pas tout à fait exact, en tout cas il n'est pas suffisant, car il ne prend pas en compte les avantages du type polymorphique : en permettant la manipulation des données sans se soucier de leur type, mais en conservant la sûreté du typage, le polymorphisme permet d'introduire un degré d'abstraction et de **généricité** (de *générique*) supplémentaire. Le programmeur se trouve alors dans une autre position : il s'apprête à créer une fonction dont il souhaite qu'elle soit polymorphe, et choisit donc de restreindre les manipulations qu'il effectue dessus pour que (selon le premier point de vue) le compilateur "ne sache pas" donner un type particulier à la fonction.

On a ici changé de sens : ce ne sont plus les manipulations du programmeur qui imposent un type à la fonction, c'est le type que le programmeur désire qui le restreint dans les manipulations auxquelles il est autorisé : les types dirigent la manière dont on traite les valeurs.

Reprenons l'exemple "let id x = x". Vous avez remarqué qu'on "ne fait rien" à cette valeur x : on ne la manipule *pas du tout*. Et c'est pour cela qu'on a un type polymorphe. Si on l'avait discrètement additionnée, affichée ou copiée dans un tableau, on aurait fait une manipulation permettant au compilateur de déterminer son type concret, et le polymorphisme aurait disparu.

Une valeur est donc polymorphe dans un contexte où on ne la "manipule" pas trop. Pas trop, cela veut dire qu'on ne la touche pas (comme ici), ou qu'on ne lui applique que des fonctions polymorphes.
Par exemple, la fonction "let id2 x = id x" manipule la valeur x, puisqu'elle lui applique la fonction "id". Mais comme cette dernière est polymorphe (ne manipule pas trop x), la fonction id2 conserve le polymorphisme elle aussi. Rassurez-vous, il existe des fonctions polymorphes un peu plus utiles que id  (mais on les rencontre sur les objets plus évolués, comme les listes ou les tableaux).

Avec ce nouveau point de vue, on peut voir le type comme l'*interface* selon laquelle on manipule une valeur (on peut voir une interface comme la liste des fonctions qui sont autorisées) : si on applique la fonction (+) (l'addition), on utilise une interface d'entier, donc le type des deux arguments (a + b) est un entier. Si l'on applique la fonction "taille_liste", on utilise l'interface d'une 'a list, donc la fonction est une 'a list (et si, dans le reste de la fonction, on demande un élément et qu'on le manipule comme un entier, alors la liste sera une int list). Si on ne la manipule pas, on n'utilise aucune interface (donc, toutes les interfaces conviennent), donc on obtient un type polymorphe.

Ce nouveau point de vue est un peu compliqué, et il ne recèle pas plus de vérité que l'idée simple des types : il n'y en a pas un qui est juste, et l'autre, faux : ce sont deux manières d'envisager la chose ; selon la situation, il peut être utile pour raisonner sur les types de choisir l'un ou l'autre des points de vue, comme je le montrerai à la fin de cette partie.

On peut remarquer que le choix d'interface est assez limité : soit on en choisit une, et on a un type déterminé, soit on n'en choisit aucune et la valeur est de type polymorphe ; si on en choisit deux, on essaie d'utiliser la valeur avec des fonctions de deux types différents, et on a une erreur.

Ce concept d'interface a été repris et enrichi dans les langages objets, mais, pour la plupart, ils ne bénéficient pas d'un système de typage formalisé, ce qui réduit beaucoup leur intérêt d'un point de vue théorique (en ce qui concerne le typage). Des langages fonctionnels modernes, c'est-à-dire développés dans les années 90 (quelques variantes de Ocaml et SML, et Haskell) ont poussé l'idée dans un contexte formel, et cela a donné des résultats assez riches, mais qui sortent du cadre de ce tutoriel.

Les types sont universels

Un reproche que l'on entend souvent dans la bouche des programmeurs au sujet des langages non typés (ou plutôt peu typés, ou faiblement typés) est "votre langage a la contrainte additionnelle du typage, donc il est moins puissant". Rassurez-vous, ils se trompent ; et le nouveau point de vue permet de le comprendre.

Il faut savoir que les langages au système de typage riche ont développé des fonctionnalités qui permettent de construire des types complexes à partir des types simples fournis initialement par le langage : un peu à la manière (mais en beaucoup mieux ) des `struct` et des `enum` du C (mais je n'en parlerai pas dans ce tutoriel).
J'ai dit plus haut qu'il n'était pas possible de mélanger des manipulations issues de deux types différents. Il est cependant possible (avec la construction ressemblant à `enum`) de construire un troisième type, qui contient soit un objet du premier type, soit un objet du deuxième, et donc de choisir selon le cas la manipulation à effectuer.

Un exemple

Voici un exemple qui pourrait (à première vue seulement) mettre en difficulté un programmeur d'un langage typé : les listes d'un langage typé imposent que tous les éléments de la liste soient du même type. Si vous mettez par exemple un entier dans une liste, tous les autres éléments doivent obligatoirement être des entiers, faute de quoi vous aurez une erreur de typage.

Imaginez maintenant un programmeur dans un langage non typé qui a une liste de nombres, un nom (par exemple "liste de notes"), qui doit coder une fonction qui affiche le nom de la liste, puis son contenu, et qui a envie de produire du code laid. Il peut décider de mettre le nom de la liste en première position de la liste : ainsi, il ne donne que la liste à la fonction, qui récupère le premier élément (c'est le nom), le traite spécialement, et affiche normalement tous les éléments du reste de la liste.

Comme dans un langage typé, on ne peut mélanger des éléments de types différents dans une liste, cette solution n'est pas praticable. La première solution, la plus simple, consiste à fournir à la fonction d'affichage, au lieu d'une liste qui mélange tout, un couple (nom, liste), qui contient d'une part le nom, et d'autre part la liste des nombres. Cette solution convient tout à fait, mais le programmeur du langage non typé risque de vous dire que ce n'est pas le même algorithme, puisque lui n'envoie qu'une liste, et que vous avez triché parce qu'il n'existe pas de manière de créer des couples dans son langage à lui. On va donc produire une autre solution, moins élégante, mais qui reprend exactement le principe de son algorithme.

Quand il a mis le nom en premier élément de la liste, est-ce que cela veut dire qu'il a traité tous les éléments de la même manière ? Non, il ne peut pas, puisque le premier élément est une `string`, alors que les suivants sont des nombres : s'il avait essayé des fonctions d'affichage de nombres sur la `string`, ou d'affichage de `string` sur les nombres, il n'aurait pas eu le bon résultat. Le traitement qu'il effectue aux éléments de sa liste est un choix : si c'est le premier élément, l'afficher comme une `string`, si c'est un autre élément, l'afficher comme un nombre.

Pour reproduire son comportement, il nous suffit donc de créer un type qui représente ce choix :

Code : OCaml

```
type élément = Premier of string | Autre of nombre
```

C'est la syntaxe de déclaration des types ressemblant à `enum`, en ML. Le '|' représente le choix : une valeur de type élément est soit `Premier s`, où `s` est une `string`, soit `Autre n`, où `n` est un entier. Maintenant, quand on a un nom et une liste de nombres, on crée une liste de type élément (par exemple : de "`notes`" et la liste `[1; 2; 3]`, on crée la liste `[Premier "notes"; Autre 1; Autre 2; Autre 3]`, qui a bien tous les éléments du même type). Ensuite, on le donne à une fonction d'affichage. Pour l'exemple, voici le code d'une telle fonction :

Code : OCaml

```
let présente liste =
let affiche_élément = function
   | Premier titre -> printf "titre : %s\n" titre
   | Autre n -> printf "%d\t" n
in List.iter affiche_élément liste
```

(La fonction `List.iter` est une fonction prédéfinie du langage, qui applique une fonction à tous les éléments d'une liste.)
On a donc résolu le problème du programmeur sceptique, et de la même façon laide que lui, en stockant tout dans la liste.

Le cas général

Voyons maintenant le cas général : ils pensent qu'il existe des programmes qui ne peuvent pas être écrits dans un langage typé, mais qui seraient possibles dans le même langage, non typé. Il ne suffit pas de montrer que tous les codes qu'ils présentent sont traduisibles dans un langage typé (comme je viens de le faire pour l'exemple), mais il faut maintenant leur montrer qu'un tel code **n'existe pas**. C'est ce que je vais essayer (de manière très peu formelle) de faire.

Imaginez un programme de la sorte, écrit dans une variante non typée du langage : tout ce qu'il fait, c'est manipuler des valeurs. On peut donc pour chaque valeur regarder l'ensemble des manipulations (ou fonctions) qui lui sont appliquées : cela nous donne son interface. Et cette interface, aussi alambiquée soit-elle, peut être traduite (à l'aide des méthodes de constructions de nouveaux types) en un type de données.
Ainsi, n'importe quel programme écrit dans le langage non typé peut se traduire en un programme dans le langage typé, et le point de vue "type comme interface" le montre très clairement.
J'espère que cette petite ( ) dissertation sur le typage vous a intéressés.
C'est un sujet très large, très riche, mais trop souvent négligé des programmeurs qui, par manque d'information, de temps ou d'intérêt, ne se sont pas aventurés du côté des langages portés sur les développements théoriques de l'informatique.

La culture, c'est important, autant en informatique qu'ailleurs. Des connaissances historiques de base, ainsi qu'une vague idée de ce qui se fait dans les langages moins connus ne pourront qu'améliorer la manière dont vous concevez la programmation.