# Types et polymorphisme

Jean-Marc Bourguet
jm@bourguet.org

4 septembre 2006

## 1 Introduction

Les mots *type* et *polymorphisme* sont beaucoup employé, mais chacun a l'air d'avoir sa propre définition plus ou moins différente de celle du voisin. Benjamin Pierce [Pie02] reconnait le fait se contente de dire qu'un *système de type* est un mécanisme syntaxique utilisé pour éviter certaines erreurs dans les programmes. Même en étant aussi vague, cela ne recouvre pas certaines acceptions utilisées, en particulier tout ce qui concerne le typage dynamique.

Ce document essaye de mettre au clair ma compréhension de ce que sont *type* et *polymorphisme*.

Tous commentaires – sur la forme comme sur le fond – sont les bienvenus.

## 2 Types et valeurs

Un ordinateur manipule des bits et suivant le contexte la même suite de bits va être interprétée de façon différente, comme un entier, un nombre en virgule flottante, comme un chaîne de caractères...

Le *type* est une indication de la manière dont on doit interpréter une représentation, et donc donne une certaine *valeur* à la représention. Il détermine donc aussi les opérations qui sont possibles en utilisant cette valeur. En prenant un point de vue légèrement différent, on peut considérer un type comme étant l'ensemble des représentations auxquelles il donne une valeur, ou encore comme l'ensemble des valeurs résultantes.

Quand un objet contient une représentation, comment savoir comment il faut interpréter cette représentation ? Différents langages offrent des réponses différentes dans les détails, mais ils peuvent être regroupés en trois grandes classes.

### 2.1 Langages non typés

Tout d'abord, on peut décider que ce sont les opérations qu'on fait qui vont fixer l'interprétation. Si la représentation n'a pas d'interprétation pour le type ou, peut-être pire, a une interprétation mais ce n'est pas ce que contient réellement l'object, c'est une erreur et les conséquences ne font plus réellement l'objet du langage.

Cette position est celle des *langages non typés*, qui comportent la plupart des langages machines et quelques autres langages comme BCPL ou BLISS.

## 2.2 Langages typés dynamiquement

Une autre technique est de placer dans la représentation une information sur la manière dont elle doit être interprétée. Lorsque l'on tente une opération, il est ainsi possible de vérifier le type des arguments et de générer une erreur à l'exécution s'ils ne sont pas les bons.

Cette position est celle des *langages typés dynamiquement*, par exemple le Lisp, Perl et Python.

## 2.3 Langages typés statiquement

La troisième technique est de pouvoir déduire les types par une analyse statique du programme, et de générer une erreur à la compilation s'il n'est pas possible de prouver qu'ils sont employés de manière cohérente. La manière dont la déduction des types s'opère varie suivant les langages. Cela va de l'exigence de déclarer les types de tous les objets, à une inférence complexe. Mais quelle que soit la méthode utilisée, il est des cas où elle va entraîner le refus de programmes qui s'ils auraient pu être exécutés n'aurait pas eu d'erreur de type.

Cette position est celle des *langages typés statiquement*, par exemple C, C++, Ada, ML, Haskell.

La plupart des langages statiquement typés permettent de contourner le système de type et de forcer la réinterprétation d'une représentation comme étant une valeur d'un autre type. Suivant que la chose soit facile ou pas, souvent nécessaire ou pas, souvent employée ou pas, on va dire que le langage est *faiblement* ou *fortement typé*.

# 3 Polymorphisme

Forcer chaque opération à n'accepter que des valeurs d'un seul type pour chaque argument, à n'avoir qu'une *signature*, c'est peut-être bien théoriquement, mais ce n'est pas très pratique. Il faut par exemple des notations différentes pour l'addition des entiers et l'addition des flottants.

Des opérations qui peuvent avoir plusieurs signatures, qui peuvent accepter plusieurs signatures sont dites *polymorphes* [Car85]. Parler de polymorphisme n'a de sens que dans un langage typé.

## 3.1 Polymorphisme *ad hoc*

La première technique qui vient à l'esprit, c'est de convertir. Si on a besoin d'un flottant et qu'on passe un entier, la conversion n'est pas bien difficile et il est naturel de la faire faire au langage sans qu'il y ait de signe de sa présence. Les *conversions implicites*, ou *coercions*, sont une première forme de polymorphisme présente dans pas mal de langages, au moins pour des types de base, certains permettant même à l'utilisateur de définir des conversions qui seront appliquées implicitement.

La deuxième technique, c'est de décrire précisément ce qu'il faut faire pour chacune des signatures. La *surcharge* est une deuxième forme de polymorphisme. Elle est aussi souvent présentes pour des fonctionnalités de base et certains langages permettent à l'utilisateur de définir des routines surchargées.

Ces deux techniques sont qualifiées de *polymorphisme ad hoc* par Cardelli. Elles ont en effet quelque chose d'artificiel, demandant de traiter particulièrement chaque nouveau

type, même si elles apportent un confort certain. Les deux techniques suivantes sont qualifiées d'*universelles* car elles traitent d'un coup un ensemble, potentiellement infini, de type ayant la même structure.

## 3.2 Polymorphisme universel

La troisième technique, le *polymorphisme paramétrique*, consiste à passer des paramètres supplémentaires à l'opération. Ces paramètres ne sont plus des valeurs, mais le type d'autres paramètres. Cette technique, aussi appelée *généricité*, permet aussi de construire des types à partir d'autre type.

La dernière technique, le *polymorphisme d'inclusion*, consiste à changer de point de vue et à dire que certaines valeurs ont plusieurs types et donc de permettre de les passer comme arguments à des opérations demandant l'un quelconque de ces types. Quand toutes les valeurs d'un type sont aussi des valeurs d'un autre type, le premier est un *sous-type* du second[1].

Dans les deux derniers cas, si l'opération polymorphe est une routine définie par l'utilisateur, il est souvent utile qu'elle puisse faire des opérations qui vont dépendre du type passé. On utilise pour cela un mécanisme ad hoc, généralement la surcharge.

Il faut remarquer que la classification n'est pas toujours nette. Par exemple on peut considérer certaines conversions implicites (d'entier vers flottants) comme du polymorphisme d'inclusion (les valeurs entières sont représentables aussi en flottant), mais ce n'est pas le cas de toutes (les conversions de flottant en entier perdent généralement de l'information et certaines représentations d'entiers permettent de représenter des valeurs entières qui ne peuvent être qu'approximées dans certaines représentations de flottants). De même, la distinction entre polymorphisme paramètrique et polymorphisme d'inclusion avec un typage dynamique est parfois difficile à faire, pour autant qu'elle ait un sens.

## 3.3 Premier exemple : le C++

Le C++ est un langage riche d'un point de vue polymorphisme : les quatres types donnés peuvent y être utilisés pour des types définis par l'utilisateur, et parfois de plusieurs façons.

### 3.3.1 Conversion implicite

Si on a une classe que l'on veut pouvoir convertir implicitement en un autre type, on peut déclarer un opérateur de conversion. Par exemple, si on est en train d'écrire une classe de nombres en virgule fixe permettant de stocker des centièmes, et qu'on veut fournir une conversion implicite vers des **double**[2], il suffit de définir un opérateur adéquat :

[1] Sous-type et sous-classe sont deux choses différentes mais pas toujours indépendantes. Un sous-type comme déjà dit est un type contenu dans un autre – si on considère les types comme des ensembles de valeurs. Une sous-classe c'est une classe formée à partir d'une autre. Elle en est généralement un sous-type, mais pas toujours : par exemple en C++ si on dérive de manière privée, on a utilisé le mécanisme des sous-classes, mais on n'a pas créé un sous-type… du moins pas dans tous les contextes.

[2] Ce qui n'est peut-être pas une bonne idée, la raison pour laquelle on écrit une telle classe étant vraissemblablement que les **double** ne permettent généralement pas de stocker un tel nombre précisément.

```
class scaled {
public:
   ...
   operator double() const { return double(value_)/100; }
   ...
private:
   ...
   int value_;
};
```

De même, si on écrit une classe d'entiers à représentation non bornée et qu'on veut pouvoir convertir implicitement les **int** dans cette classe[3], il suffit de fournir un constructeur n'ayant qu'un paramètre[4] :

```
class big_integer {
public:
   ...
   big_integer(int i) { ... }
   ...
};
```

### 3.3.2 Surcharge

Le C++ permet la surchage des fonctions libres de façon simple[5], il suffit de déclarer les fonctions avec leurs paramètres. Le C++ permet aussi la surcharge des fonctions membres.

Un deuxième type de surcharge est disponible en C++ en utilisant la spécialisation explicite de template.

Un troisième type de surcharge intervient en collaboration avec le polymorphisme d'inclusion. Le polymorphisme d'inclusion permet d'avoir des pointeurs qui pointent vers un sous-type du type déclaré. Si un membre est marqué virtuel, alors plutôt que d'appeler un membre de la classe, on appellera le membre correspondant du sous-type s'il a été supplanté.

### 3.3.3 Polymorphisme paramétrique

Les templates permettent en C++ du polymorphisme paramétrique tant qu'on n'utilise pas le mécanisme de spécialisation explicite.

### 3.3.4 Polymorphisme d'inclusion

En C++, tous les pointeurs vers des classes sont polymorphes et peuvent contenir des pointeurs vers n'importe quel sous-type de la classe. De même les références peuvent désigner n'importe quel objet d'un sous-type de leur classe.

[3] Il vaut mieux prévoir aussi des conversions pour les autres types entiers du C++.

[4] Pour éviter de définir une conversion implicite en écrivant un constructeur à un paramètre, il faut le qualifier de **explicit**.

[5] Les interactions entre la surcharge et les conversions implicites et les **namespace** ne sont pas aussi simples.

# Références


[Car85] Luca Cardelli. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, December 1985. Also available at `http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Pie05] Benjamin C. Pierce, editor. *Advanced Topices in Types and Programminh Languages*. The MIT Press, 2005.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.