Introduction au typage

F. Barthélemy

6 avril 2005

1 Introduction

Le typage est un mécanisme qui permet d'assurer une propriété de **correction partielle** des programmes.

Pour la programmation impérative, cette propriété est que les opérateurs et sous-programmes sont appliqués à des opérandes appartenant au bon domaine de valeurs.

Pour la programmation objet, la propriété essentielle recherchée par le typage est que lorsqu'on appelle une méthode sur un objet, cet objet contient bien cette méthode et les paramètres donnés appartiennent au bon domaine de valeur.

2 Premier exemple

Nous allons traiter un premier exemple simple qui va nous permettre d'aborder les notions de base que nous utiliserons plus tard pour le typage de Java. Ces notions sont celles de type, d'environnement de typage, de règles de typage des instructions.

Le premier exemple que nous allons traiter est un tout petit sous-ensemble non objet de Java, en faisant des hypothèses simplificatrices. Nous allons nous restreindre à deux types primitifs : int et boolean, à quelques opérateurs (`!`, `&&`, `||`, `+`, `*`, `-`, `/`, `<`, `>`, `==`) et à quelques méthodes statiques prédéfinies.

Toujours dans un but de simplification, nous allons ignorer les conversions de type automatiques et la surcharge d'opérateurs. Nous allons supposer l'existence de méthodes statiques d'entrées-sorties pour les deux types que l'on considère :

- `Terminal.readInt()` et `Terminal.readBoolean()` pour des saisies au clavier
- `Terminal.printInt(int x)` et `Terminal.printBoolean(boolean b)` pour des affichages à l'écran.

Nous allons nous restreindre à des programmes sans méthodes, si ce n'est la méthode `main`.

Prenons un premier exemple de programme :

```
class TypageEx1{
    public static void main(String[] args){
        int val, somme;
        somme = 0;
        val = 0;
        do{
            somme = somme+val;
            val = EntreesSorties.readInt();
```

```
        }while (val > 0);
        EntreesSorties.printInt(somme);
    }
}
```

Ce programme calcule la somme d'une suite d'entiers positifs saisis au clavier. La saisie s'arrête lorsqu'un entier négatif ou nul est entré.

Dans ce genre de programmes, les erreurs de typage qui peuvent se produire sont par exemple :

– l'emploi d'un opérateur avec des opérandes du mauvais type. Par exemple `true+12`.

– l'appel d'un méthode statique prédéfinie avec un paramètre du mauvais type : `EntreesSorties.printIn`

Le typage va détecter automatiquement ces erreurs à la compilation, à condition que le programme soit correct syntaxiquement. Le typage est une vérification qui est exercée sur un programme déjà vérifié du point de vue de sa syntaxe.

2.1 Types et déclarations de type

Un type est un ensemble de valeurs. Le type `boolean` est l'ensemble `{true, false}`. Le type `int` contient un grand nombre de valeurs :
`{Integer.MIN_value,...,-2,-1,0,1,2,...Integer.MAX_VALUE}`.

Dans ce monde restreint, les types sont disjoints, chaque valeur a un et un seul type. Sont des valeurs les éléments de `int`∪`boolean`. Pour connaître le type d'une valeur, il faut tester son appartenance à l'un ou l'autre ensemble.

Les variables ont un type qui leur est donné par une déclaration. Ce type restreint le domaine de valeurs que peut contenir la variable.

Les opérateurs et méthodes statiques ont également un type qui résume le type des paramètres et le type du résultat. Par exemple : $+$: `int` $\times$ `int` $\rightarrow$ *int*
`EntreesSorties.printInt` : `void` $\rightarrow$ `void`
`EntreesSorties.ecrireInt` : `int` $\rightarrow$ `void`

On note `void` l'absence de valeur (pas de paramètre ou pas de valeur retournée). On peut considérer `void`comme un type, un ensemble de valeurs qui ne contient aucune valeur, l'ensemble vide.

2.2 Règles de typage

Les règles de typage sont des règles qui définissent ce qui est correctement typé ou non dans les instructions du programme. Ces règles font références aux types des variables, littéraux (valeurs comme 12 ou `true`), opérateurs et méthodes qui apparaissent dans l'instruction.

Par exemple, la règle de typage de l'affectation (pour le sous-ensemble de java isomé dans ce premier exemple) est la suivante : le type de la variables située à gauche du signe = doit être égal au type de l'expression située à droite du signe =.

Autre exemple : une instruction `if` est correctement typée si

1. la condition entre parenthèses est une expression de type `boolean`.
2. les instructions contenues dans les deux branches du `if` sont correctement typées.

2.3 Environnement de typage

L'environnement de typage est une mémoire utilisée pour enregistrer les types des variables, des méthodes et des opérateurs. Il est indispensable pour connaître le type au moment où l'on veut utiliser les règles de typage.

Par exemple, le type d'une variable est déclaré à un certain endroit, mais il est utilisé à un autre endroit (dans les instructions où la variable apparaît).

L'environnement de typage est une liste de couples `(nom, type)`. Nous le noterons entre crochets, les couples séparés par une virgule.

L'environnement de typage initial contient les opérateurs et méthodes prédéfinis. Dans notre exemple :
$[(+, \texttt{int} \times \texttt{int} \rightarrow int), (\texttt{EntreesSorties.printInt}, \texttt{void} \rightarrow \texttt{void}), (\texttt{EntreesSorties.ecrireInt}, \texttt{int} \rightarrow \texttt{void}), ...]$

Puis l'environnement de typage évolue au fil des déclarations.

Reprenons notre premier exemple : Prenons un premier exemple de programme :

```
class TypageEx1{
    public static void main(String[] args){
        // Environnement = environnement initial EI
        int val, somme;
        // ajout de (val,int) et (somme,int) à l'environnement
        somme = 0;
        val = 0;
        do{
            somme = somme+val;
            val = EntreesSorties.readInt();
        }while (val > 0);
        EntreesSorties.printInt(somme);
    }
}
```

Prenons un exemple un peu plus compliqué.

```
class TypageEx2{
    public static void main(String[] args){
        // environnement initial
        int nb, somme; // ajout de (nb, int) et (somme,int) à l'environnement
        somme = 0;
        nb = EntreesSorties.readInt();
        if (nb > 0){
            int val; // ajout de (val,int) à l'environnement
            for (int i=0;  // ajout de (i,int) à l'environnement
                 i<nb;
                 i++){
                val = EntreesSorties.readInt();
                somme = somme+val;
            } // retrait de (i,int) de l'environnement
        } // retrait de (val,int) de l'environnement
        EntreesSorties.printInt(somme);
    }
}
```

L'évolution de l'environnement se fait avec l'ajout progressif de couples au fil des déclarations et le retrait par paquet : tout ce qui a été déclaré dans un bloc est enlevé une fois arrivé à la fin du bloc. Pour modéliser ce fonctionnement, nous allons utiliser un fonctionnement de pile. Au lieu d'une liste, nous aurons une pile de listes, avec à sa base, l'environnement initial. Lorsque s'ouvre un nouveau bloc, une nouvelle liste est empilée. Lorsqu'un bloc se ferme, la liste située en sommet de pile est dépilée.

Opérations sur les environnements :

- pushNew : crée une liste vide et la place en sommet de pile. Paramètre : la pile.
- pop : dépile la liste située en sommet de pile. Paramètre : la pile.
- add : ajoute un couple à la liste située en sommet de pile. Paramètres : la pile et le couple.
- lookup : recherche le type associé à un nom dans la pile. La recherche commence par la liste en sommet de pile, puis en cas d'échec, dans la deuxième liste de la pile, etc. Paramètres : la pile et le nom.

```
class TypageEx2{
    // environnement initial: EI environnement courant: EC
    // EC=EI;
    public static void main(String[] args){
        // EC=pushNew(EC);
        int nb, somme; // add(EC,(nb, int)); add(EC,(somme,int))
        somme = 0;
        nb = EntreesSorties.readInt();
        if (nb > 0){
            // EC = pushNew(EC);
            int val; // add(EC,(val,int));
            for (int i=0;  // EC = pushNew(EC);
                           // add(EC,(i,int));
                 i<nb;
                 i++){
                val = EntreesSorties.readInt();
                somme = somme+val;
            } // pop(EC);
        } // pop(EC);
        EntreesSorties.printInt(somme);
    } // pop(EC);
}
```

2.4 Retour sur les règles de typage

2.4.1 Typage des expressions

Dans notre langage, une expression peut être :

- un littéral : ex : true, 45.
- une variable. Ex : x, nb, somme.
- une expression avec opérateur. Ex : x + 3.
- un appel de méthode statique. Ex : `EntreesSorties.readInt()`.

Il faut une règle de typage pour chaque cas.

Application d'opérateur binaire : `operande1 operateur operande2`

- typer `operande1` et `operande2`. Cela donne deux type t_1 et t_2.

– rechercher le type d'`operateur` dans l'environnement courant (`lookup(EC,operateur)`). Cela donne un type $t_3 \times t_4 \rightarrow t_5$
– le typage est correct si $t_1 = t_3$ et $t_2 = t_4$. Sinon l'expression est mal typée.
– le type de l'expression est t_5

Exemple : (5*3) + 5.

Application d'opérateur unaire : idem avec un seul opérande.

Typage d'une variable : on recherche le type de la variable dans l'environnement courant. Conséquence : une variable ne peut être utilisée qu'après avoir été déclarée.

Typage d'un littéral : on teste l'appartenance du littéral à chacun des types considérés. Autre technique possible : ajouter les littéraux dans l'environnement initial.

Typage de l'appel d'une méthode statique : `Classe.nomMethode`$(e_1, \ldots, e_n)$
– chercher les types des paramètres $e_1, \ldots e_n$. Cela donne n types $t_1, \ldots t_n$.
– chercher le type de `Classe.nomMethode` dans l'environnement courant. `lookup(EC,Classe.nomMethode)`. Cela donne le type $t'_1 \times \ldots \times t'_m \rightarrow t_0$.
– l'expression est bien typée si $n = m$ et $\forall i, 1 \leq i \leq n, t_i = t'_i$.
– le type de l'expression est t_0

Typage d'expressions complexes :
$EntreesSorties.printBoolean(x + 17 < EntreesSorties.readInt())$

On applique la règle à l'expression englobante (le plus externe), puis à ses opérandes, etc.

2.4.2 Typage d'instruction

Les instructions sont correctes ou non, mais elles n'ont pas de type en elle-mêmes.

Typage de l'affectation : déjà vu, égalité des types des deux côtés du signe =.

Typage du if et du while :
– que la condition soit du type `boolean`
– que les instructions imbriquées soient bien typées.

Typage du `for` : ouverture d'un nouvel environnement. Prise en compte des déclarations pour typer la condition et l'incrémentation.

3 Ajout des méthodes statiques

On ajoute à notre petit langage de base la possibilité d'écrire des méthodes statiques ou méthodes de classe. On va se limiter à des méthodes définies dans la même classe que `main`.

Exemple :

```
public class ValAbsFunc {
    static int valeurAbsolue(int n){
        int res;
        if (n > 0) {
            res = n;
        } else if (n < 0) {
            res = -n;
        } else {
            res = 0;
        }
        return res;
    }
    public static void main (String args[]) {
```

```
        int  x, abs;
        x = EntreesSorties.readInt();
        EntreesSorties.printInt(valeurAbsolue(x));
    }
}
```

Chaque méthode doit être typée comme la méthode `main`, sauf qu'il faut prendre en compte les paramètres dans l'environnement. Par ailleurs, il faut également prendre en compte le type de la méthode dans l'environnement.

Dans beaucoup de langages, il faut déclarer une méthode avant de pouvoir l'utiliser. En Java, ce n'est pas le cas. Toutes les méthodes d'une classe peuvent appeler toutes les autres méthodes, qu'elles soient déclarées avant ou après.

Exemple :

```
class TypageEx3{
    public static void main(String[] args){
        int x;
        x = EntreesSorties.readInt();
        EntreesSorties.printBoolean(premier(x));
    }
    static boolean premier(int n){
        for (int i=2; i< (n/2)+1; i=i+1){
            if (divisible(n,i))
                return false;
        }
        return true;
    }
    static boolean divisible(int x, int y){
        return ((x / y) * y) == x;
    }
}
```

Il faut donc un processus de typage en deux temps : d'abord, un environnement de typage est construit qui comporte toutes les méthodes avec leurs types déclarés, puis chaque méthode est typée dans cet environnement de typage.

Voyons cet environnement sur notre exemple, en appelant E_i l'environnement initial. Pour le typage de la classe, un nouvel environnement est créé :

```
Ec = pushNew(Ei);
add(Ec,(main,String[] -> void));
add(Ec,(premier,int -> boolean));
add(Ec,(divisible,int -> boolean));
```

Cela nous donne l'environnement où la liste `[(divisible,int  -> boolean),(premier,int -> boolean`
est mise par-dessus l'environnement initial.

La règle de typage d'une méthode est ensuite la suivante : soit une méthode `t0 nomDeMethode(t1 p1,...,`
à typer dans l'environnement courant `Ec`.

- réaliser les opérations suivantes sur l'environnement :

  ```
  Ec = pushNew(Ec);
  add(Ec,(p1,t1));
  add(Ec,(p2,t2));
  ```

```
...
add(Ec,(pn,tn));
```

- typer dans cet environnement typer les instructions i1, i2,...im.
- pour chaque instruction `return exp` dans i1...im, vérifier que `exp` a le type `t0`.