Notes de cours
Semaine 27

Cours Turing

1 Typage

Aujourd'hui, nous allons nous intéresser à une technique très largement répandue dans les langages de programmation : le typage. De manière très générale, le typage consiste à utiliser des types, associés aux valeurs ou aux expressions syntaxiques manipulées par les programmes, pour détecter et reporter des opérations invalides. Les types permettent aussi de documenter le code, voire même de l'optimiser.

1.1 Le typage, qu'est-ce que c'est ?

Il est difficile de donner une définition précise du typage, car le terme recouvre des implémentations très différentes suivant les langages de programmation. Nous aurons l'occasion de discuter de ces différences et des grandes catégories qui en découlent dans la suite de ce document. Avant cela, nous allons essayer de donner une définition générale du typage.

Le typage se base sur la notion de *type*. De manière générale, un type correspond à un *ensemble* de valeurs manipulées par un programme. Par exemple, très souvent, on aura un type `int` qui correspond à l'ensemble des entiers, un type `float` qui correspond à l'ensemble des nombres à virgule flottante, un type `bool` qui correspond à l'ensemble des valeurs booléennes, etc. Il est aussi possible d'imaginer des types plus complexes, comme des types de listes ou des types de fonctions, par exemple.

Les types peuvent prendre des formes plus ou moins complexes suivant les langages de programmation. Ils gardent cependant généralement le même rôle, celui de décrire des ensembles de valeurs.

1.2 Pourquoi le typage ?

Le typage dans les langages de programmation est une technique qui remonte aux années 1950 avec le langage Fortran et plus particulièrement avec le langage Algol. Au fil du temps, les systèmes de types utilisés par les divers langages de programmation se sont complexifiés, permettant d'exprimer des propriétés de plus en plus sophistiquées sur les programmes.

Les systèmes de types ont été développés pour plusieurs raisons :

1. **Optimisation** : le typage permet d'optimiser le code. En connaissant les types des différentes expressions, on peut générer un code plus efficace, qui évite des conversions inutiles ou des tests redondants.

2. **Détection d'erreurs** : le typage permet de détecter des erreurs de programmation avant même l'exécution du programme. Par exemple, si on essaie de diviser une chaîne de caractères par un nombre, on peut détecter cette erreur de type avant même de lancer le programme.
3. **Documentation** : le typage permet de documenter le code. En indiquant les types des variables, des paramètres et des valeurs de retour, on peut comprendre plus facilement le code, et détecter des erreurs de conception.
4. **Sécurité** : le typage permet de sécuriser le code. Suivant l'expressivité du système de types, il est possible de prévenir des erreurs de sécurité, comme des débordements de tampon ou des fuites de mémoire.

Les langages avec systèmes de types sont très largement utilisés dans l'industrie, en particulier pour les applications critiques, où la sécurité et la fiabilité sont des enjeux majeurs (transports, finance, santé, etc.).

2 Catégorisation des systèmes de typage

Ce qu'on entend par typage peut être très différent d'un langage de programmation à un autre. Il existe en effet de nombreux systèmes de typage, qui peuvent être plus ou moins complexes, plus ou moins expressifs, plus ou moins contraignants, etc. Pour commencer, nous allons tenter de définir des catégories de systèmes de typage selon différents critères.

2.1 Typage dynamique et statique

Le premier critère que l'on peut utiliser pour classer les systèmes de typage est le moment où le typage est effectué. Il existe deux grandes familles de typage : le typage dynamique et le typage statique.

2.1.1 Typage dynamique

Dans le typage dynamique, les valeurs manipulées par les programmes ont un type qui est déterminé à l'exécution. C'est le cas par exemple de Python, JavaScript, Ruby, etc. Dans ce genre de langage, on ne vérifie généralement pas le code avant de l'exécuter, ce qui peut entraîner des erreurs de type à l'exécution. Lors de l'exécution, on vérifie que les opérandes des opérations ont de types compatibles, et on lève une exception si ce n'est pas le cas.

```
print(1 + "2")  # Lève une exception TypeError à l'exécution
```

Listing 1 – Exemple en Python, langage avec typage dynamique

Dans le typage dynamique, les types sont attachés aux *valeurs*, et non aux *expressions syntaxiques* du programme. Ils ne sont donc en principe pas connus avant l'exécution. Dans certains langages, comme Python, il est possible d'accéder au type d'une valeur à l'exécution, comme dans l'exemple ci-dessous.

```
for v in [1, "2", 3.0]:
    print(type(v))
```

```
# ^ Affiche <class 'int'>
#          <class 'str'>
#          <class 'float'>
```

Listing 2 – Exemple en Python, langage avec typage dynamique

2.1.2 Typage statique

Dans le typage statique, les valeurs manipulées par les programmes ont un type qui est déterminé avant l'exécution. C'est le cas par exemple de Java, C, C++, etc. Dans ce genre de langage, les différentes expressions syntaxiques du programme se voient attribuer un type avant l'exécution, et on vérifie que ces expressions sont utilisées de manière cohérente. À la différence du typage dynamique, les erreurs de type sont détectées avant même l'exécution du programme. Avec le typage statique, on analyse la structure du programme (l'AST) au moment de la compilation pour déterminer les types des différentes expressions, et on vérifie que ces types sont compatibles entre eux.

```
// Erreur de compilation, les types ne sont pas compatibles
// Le compilateur détecte cette erreur
// avant même l'exécution le programme
System.out.println(1 + "2");
```

Listing 3 – Exemple en Java, langage avec typage statique

Il est important de comprendre ici que les types sont attachés aux *expressions syntaxiques* du programme, et non aux *valeurs* manipulées par le programme. Les types sont donc connus avant l'exécution et sont utilisés pour détecter des erreurs de type.

Pour le projet, nous allons implémenter un système de typage statique. Nous analyserons l'AST du programme pour déterminer les types des différentes expressions, et nous vérifierons que ces types sont appropriés.

Parfois, les langages de programmation utilisent une combinaison de typage statique et dynamique. Par exemple, TypeScript permet de combiner les deux approches, en ajoutant un système de typage statique en plus du typage dynamique hérité de JavaScript.

Même dans le langage Python, il est possible d'ajouter des annotations de types pour documenter le code et détecter des erreurs de type à l'aide d'outils externes comme `mypy` [1].

2.2 Typage faible et fort

Un autre critère que l'on peut utiliser pour classer les systèmes de typage est l'action effectuée en cas d'erreur de type. Il existe également deux grandes familles de typage : le typage faible et le typage fort.

2.2.1 Typage faible

Dans le typage faible, en cas d'erreur de type, on essaie de convertir les valeurs pour les rendre compatibles. C'est le cas par exemple de JavaScript, qui peut, par exemple, convertir

1. Le projet `mypy` est accessible en ligne à l'adresse `https://mypy-lang.org/`

automatiquement un nombre en chaîne de caractères, ou une chaîne de caractères en nombre suivant le contexte.

```
console.log(1 + "2")   // Affiche "12"
```

Listing 4 – Exemple en JavaScript, langage avec typage faible

Dans cet exemple, JavaScript convertit automatiquement le nombre `1` en chaîne de caractères pour pouvoir concaténer les deux valeurs.

2.2.2 Typage fort

Au contraire, dans le typage fort, en cas d'erreur de type, on ne tente pas de faire de conversion mais on lève une exception.

```
print(1 + "2")   # Lève une exception TypeError
```

Listing 5 – Exemple en Python, langage avec typage fort

Pour le projet, nous allons nous intéresser à un système de typage fort. Lorsque nous détecterons une erreur de type lors de l'analyse de l'AST, nous lèverons une exception qui stoppera la compilation.

2.3 Typage explicite et implicite

Un dernier critère que l'on peut utiliser pour classer les systèmes de typage est la manière dont les types sont déclarés. Cette distinction se fait uniquement pour les systèmes de typage statique.

2.3.1 Typage explicite

Dans un système de typage explicite, les types sont déclarés explicitement par le programmeur à des endroits stratégiques. Souvent, il s'agira de déclarer le type des variables, des paramètres de fonctions, des valeurs de retour, etc. Dans l'exemple ci-dessous, en C, les types des paramètres et de la valeur de retour de la fonction `add` doivent être déclarés explicitement.

```
// Les types des paramètres et de la valeur de retour
// sont déclarés explicitement
int add(int a, int b) {
    return a + b;
}
```

Listing 6 – Exemple en C, langage avec types explicites

2.3.2 Typage implicite

Dans un système de typage implicite, les types sont déduits automatiquement par le compilateur. Les mécanismes d'*inférence* de ces types peuvent être plus ou moins complexes. Dans l'exemple ci-dessous, en Haskell, le type de la fonction `add` est déduit automatiquement par le compilateur.

```
add a b = a + b  -- Le type de add est déduit automatiquement
```

Listing 7 – Exemple en Haskell, langage avec inférence de types

Souvent, les langages avec inférence de types permettent tout de même de déclarer les types explicitement, par exemple dans le but de documenter le code.

```
add :: Int -> Int -> Int
add a b = a + b  -- Le type de add est annoté explicitement
```

Listing 8 – Exemple en Haskell, langage avec inférence de types

Pour le projet, nous allons nous intéresser à un système de typage explicite, où l'utilisateur doit déclarer les types des variables, des paramètres et des valeurs de retour.

3 Conclusion

Les systèmes de typage sont très variés, et peuvent être classés suivant de nombreux critères. Nous n'avons ici abordé que quelques-uns de ces critères, mais il en existe bien d'autres. Des systèmes de typage très complexes ont été développés, permettant d'exprimer des propriétés très sophistiquées sur les programmes. Par exemple, dans des langages comme Rust, il est possible de décrire des propriétés de concurrence, d'usage de la mémoire, etc.

À l'extrême, dans des langages comme Coq, les types permettent de décrire des énoncés de théorèmes, et de faire des preuves ces théorèmes à la compilation. Le système de type fait en sorte que seuls des théorèmes valides peuvent être prouvés !