Compilation

Types et typage

Jules Chouquet



SOM2IF15 – 2025

Point d'étape

Ce que l'on a traité :

- Lecture d'une suite de symboles et sa transformation en donnée structurée.
- Représentation du programme sous forme d'arbre de syntaxe abstrait.
- Mise en place d'une méthode générique pour appliquer des algorithmes à ces arbres

Point d'étape

Ce que l'on a traité :

- Lecture d'une suite de symboles et sa transformation en donnée structurée.
- Représentation du programme sous forme d'arbre de syntaxe abstrait.
- Mise en place d'une méthode générique pour appliquer des algorithmes à ces arbres

Ce qui nous reste à traiter :

- **Vérifications** du code source. (Gestion d'erreurs non syntaxiques).
- Transformation de l'arbre en structures plus élémentaires
- Génération de code assembleur.

Que doit-on vérifier ?

Le typage du programme [1]

Avant de voir comment s'implémente la vérification de typage dans notre compilateur [2], revenons aux fondamentaux :

- Qu'est-ce qu'un type ?
- Comment définir formellement le lien entre les bouts de programme et les types ?

1. Qui, dans un style java ou C, implique de vérifier aussi que toutes les variables utilisées sont déclarées au préalable. Donc la vérification couvre des erreurs qui sont plus générales que le typage en lui-même.
2. Cours suivant, table des symboles etc. . .

Une classification de tous les éléments du programme [3].

Attention, ici, si notre programme contient `print(1)`, on distingue `1` (de type entier) et `print(1)` (de "type" instruction) comme deux éléments. [4]

Autrement dit, tout nœud de l'arbre de syntaxe abstrait est considéré comme un élément, même s'il est constitué d'autres éléments.

3. Élément : lexème, expression, instruction...
4. Mais `print` tout seul n'est pas un élément bien sûr...

Incidences concrètes du typage

Le type des données détermine la taille qu'elles occuperont en mémoire : Entiers, courts ou longs,flottants, adresses mémoire.

Deux grandes écoles :

Typage	**Statique**	**Dynamique**
Type déterminé	à la compilation	à l'exécution
Type des variables	fixe, connu	inconnu
Déclaration de variables	selon les langage	jamais
`fact([])`	Erreur à la compilation	Exception à l'exéc.
Performances	Ça dépend	Forcément moins bon

Remarque

Une partie de la vérification de type est faite par certains IDE, (qui connaissent l'AST)

Déclaration de variables dans les langage à typage statique

- Obligatoire en l'absence de mécanisme d' **inférence de types** (Fortran, Pascal, C).
- Possible en C++(>11) [5], en Java [6]
- Systématique en Ocaml, Haskell, Rust

L'inférence de types

Un algorithme qui détermine le type de chaque expression (en particulier de chaque variable) en fonction du contexte.

(Il faut en général avoir une bonne compréhension du typage pour être capable de lire et écrire de bons programmes dans lesquels le type n'est pas déclaré.)

5. `int a = 1; auto b = a + 1;`
6. `var x=...`, ou `l=new List<>();`

Unicité du typage

Association : une expression = un type

Dans un langage typé dynamiquement, il est impossible d'associer un type à chaque expression de l'AST sans exécuter le programme.

Mais un élément peut aussi avoir plusieurs types. C'est le cas avec des alias (en C : `typedef int bool;`, en OCaml : `type entier = int;;`). Mais également avec l'héritage de classes (en Java : `"tiki taka"` est de type `String`, et aussi de type `Object`).

Pour le cours et les TP, on considèrera qu'un élément ne peut avoir qu'un seul type (unicité).

En résumé : le typage c'est important

Dans la conception d'un langage[7], comment présente-t-on la façon dont le typage va se comporter ?

On va voir les **systèmes de types** , qui sont des présentations formelles[8] du lien entre les éléments de programmes et les types.

7. À typage statique
8. C'est-à-dire mathématiques, indépendantes des détails de l'implémentation

Ensemble des termes, ensembles des types

Soit $\mathcal{N}$ l'ensemble des nœuds possible de l'AST de notre langage [9], et $\mathcal{T}$ l'ensemble des types [10]

Relation de typage

Le **typage** d'un langage est défini comme une relation $R \subseteq \mathcal{N} \times \mathcal{T}$.
Si $(p, t) \in R$, on écrit $p : t$, qui se lit "p est de type t".

Chaque langage définit sa relation de typage, c'est une partie importante de sa documentation : elle détermine les programmes qu'on a le droit d'écrire, et les autres, qui seront rejetés par le compilateur.

9. infini, forcément
10. Peut être fini (`int`,`bool`,...) si l'on n'a pas de types composés, infini sinon : `int[]`,`int[][]`,... sont des types.

Comment savoir si l'expression `x+1` est bien typée ?
On considère dans le système de types un *environnement* (aussi appelé *contexte*) de typage [11].

Environnements

Un environnement Γ est un sous ensemble de $\mathcal{V} \times \mathcal{T}$ où $\mathcal{V}$ est l'ensemble des **variables** du langage [a] .

a. Chaque variable ne peut apparaître qu'une fois, si l'on veut l'unicité du typage. L'environnement peut donc aussi être défini comme une fonction (partielle).

Exemple

Si $(\texttt{x}, \texttt{int}) \in \Gamma$, alors `x` sera considérée comme une expression de type `int` dans **chacune** de ses apparitions sous le contexte Γ.

11. On en a vu au cours précédent, dans l'exemple `interpWhile.ml`

Typage des expressions

Règles

Pour spécifier un langage, on donne un ensemble de règles de typage. Pour les expressions, ces règles prennent la forme suivante [12] et permettent de vérifier si elles sont bien typées.

$$\frac{}{\Gamma \cup \{x : t\} \vdash x : t} \qquad \frac{}{\Gamma \vdash \texttt{0:int}} \qquad \frac{}{\Gamma \vdash \texttt{true:bool}}$$

$$\frac{\Gamma \vdash \texttt{t:A} \qquad \Gamma \vdash \texttt{u:B}}{\Gamma \vdash (t, u) : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \texttt{fst(t)} : A}$$

$$\frac{\Gamma \vdash \texttt{n:int} \qquad \Gamma \vdash \texttt{m:int}}{\Gamma \vdash \texttt{n+m:int}} \qquad \frac{\Gamma \vdash \texttt{e:bool}}{\Gamma \vdash \texttt{not e:bool}}$$

Exemple

Si x : `int` $\in \Gamma$, alors `x+0` est bien typée, mais pas `not x`.

12. Liste non exhaustive à titre d'exemple, les règles varient selon les choix de conception du langage.

Typage des expressions

Vérification

Pour vérifier qu'une expression est bien typée, on cherche à appliquer les règles précédentes tant que c'est possible.
Par exemple, pour $\Gamma = \{x : \texttt{bool} \times \texttt{int}, t : \texttt{int} \times \texttt{int}\}$, on a la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{bool} \times \texttt{int}}}{\Gamma \vdash \texttt{fst(x)} : \texttt{bool}}}{\Gamma \vdash \texttt{not fst}(x) : \texttt{bool}} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{bool} \times \texttt{int}}}{\Gamma \vdash \texttt{snd}(x) : \texttt{int}} \qquad \dfrac{\dfrac{}{\Gamma \vdash 42 : \texttt{int}} \qquad \dfrac{\dfrac{}{\Gamma \vdash t : \texttt{int} \times \texttt{int}}}{\Gamma \vdash \texttt{fst}(t) : \texttt{int}}}{\Gamma \vdash 42 + \texttt{fst}(t) : \texttt{int}}}{\Gamma \vdash \texttt{snd}(x) + (42 + \texttt{fst}(t)) : \texttt{int}}}{\Gamma \vdash (\texttt{not fst}(x), \texttt{snd}(x) + (42 + \texttt{fst}(t))) : \texttt{bool} \times \texttt{int}}$$

Remarque

On peut avoir des règles à nombre de prémisses variables. Par exemple, pour des tableaux en Java, on pourrait écrire :

$$\frac{\Gamma \vdash v_1 : A \qquad \ldots \qquad \Gamma \vdash v_n : A}{\Gamma \vdash \{v_1, \ldots, v_n\} : A[]} \; n \in \mathbb{N}$$

Remarque

On peut avoir des règles à nombre de prémisses variables. Par exemple, pour des tableaux en Java, on pourrait écrire :

$$\frac{\Gamma \vdash v_1 : A \qquad \ldots \qquad \Gamma \vdash v_n : A}{\Gamma \vdash \{v_1, \ldots, v_n\} : A[]}\ n \in \mathbb{N} \qquad \text{puis :} \quad \frac{\Gamma \vdash t : A[]}{\Gamma \vdash t[i] : A}\ i \in \mathbb{N}$$

n.b. l'expression `t[n]` peut-être bien typée même dans le cas d'un indice hors de portée, qui ne pourrait pas de toute façon être détecté à la compilation

Règles de typage dans la vraie vie

En pratique, le contexte sera déterminé par les déclarations du programme et par les constantes du langage (`1`, `2`, `true`, ...). En particulier, `x+1` sera bien typé si une déclaration `int x` précède[13].

Le typage est *contextuel* : "`int x; x+1`" est un programme bien typé, mais pas "`bool x; x+1`" (le fait que `x` ne soit pas initialisé n'est pas un problème de typage).

La vérification de types doit donc passer par le parcours de toutes les déclarations, de façon à ce que le typage des expressions contenant des variables soit cohérent.

13. Dans certains langages comme JavaScript, la déclaration peut se faire après l'utilisation. Le compilateur doit alors déplacer les déclarations avant le début de leur portée : c'est le *hoisting*.

Typer les instructions

Exemples

$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash t : A}{\Gamma \vdash x = t : \texttt{inst}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \qquad \Gamma \vdash s : \texttt{inst}}{\Gamma \vdash \texttt{while(b) s} : \texttt{inst}}$$

$$\frac{\Gamma \vdash s : \texttt{inst} \qquad \Gamma \vdash t : \texttt{inst}}{\Gamma \vdash \texttt{s;t} : \texttt{inst}} \qquad \frac{\Gamma \cup \{x : A\} \vdash \texttt{s} : \texttt{inst}}{\Gamma \vdash \texttt{A x;s} : \texttt{inst}}$$

[Exemples au tableau]

Remarque

Cela ne marche que pour les programmes monoblocs, (comme l'interpréteur vu au cours précédent). Pour un système moins simpliste, il faut des environnements imbriqués représentant la hiérarchie des blocs (voir prochain cours sur la table des symboles)

Remarque

Dans un langage fonctionnel comme OCaml, `inst` est un type comme les autres (`unit`), et les instructions sont des expressions comme les autres.

Autre remarque

La vérification du type des instructions, même en présence d'un seul type, permet de détecter des erreurs d'écriture de code, et de faire des choix de conception :

- 1 ;[14]
- x=print(y) ;[15]

14. Autorisé en C, mais en Java : erreur "not a statement"
15. Rarement autorisé, sauf en fonctionnel complet où les expressions peuvent être de type `inst`.

Écrire la dérivation de typage d'un programme revient à vérifier qu'il est entièrement bien typé. Si c'est impossible, le programme est mal typé.
Si un seul élément est mal typé, le programme entier n'a aucun sens, et il ne faudra pas essayer de l'exécuter.

Complexité

Si, pour une instruction donnée, une seule règle de typage s'applique, elle correspond à un parcours (en profondeur) de l'arbre syntaxique.

Pourquoi tout ce formalisme pénible ?

Ces représentations des programmes et leurs manipulations permettent de les analyser de façon *scientifique*[16]. Cela pour élever le niveau de garantie et de confiance dans les logiciels.

Quelles propriétés veut-on prouver ?

Que des erreurs critiques ne vont pas se produire à l'exécution (divisions par 0 par exemple, ou erreur d'accès à la mémoire).

Comment on s'y prend ?

On passe en M2, et on va au cours de M. Dabrowski sur l'analyse statique.

16. *i.e.* On veut des théorèmes, pas des tests (même nombreux).

Dans la suite du cours

- Implémentation de la vérification de types. Ce sera un visiteur de l'AST, pour lequel le parcours récursif correspondra vraiment à la dérivation de typage que l'on a vue. (la différence sera essentiellement la gestion des environnements dans les blocs, avec le choix de la portée pour la déclaration des variables).
- Définition formelle du langage, étape 2 : la **sémantique** (opérationnelle). Après le typage et les programmes qu'on a le droit d'écrire, on va formaliser le comportement qu'on attend du programme.
- Puis traductions vers du code de moins en moins structuré [17]

17. Le typage devra avoir été traité, il faut savoir quelle place en mémoire prendront les données avant de traduire vers du code qui parle de registres, etc. . .