Compilation : théorie, techniques et outils

Thème 5 : Gestion de type

Habib ABDULRAB (Institut National des Sciences Appliquées de Rouen)
Claude MOULIN (Université de Technologie de Compiègne)
Sid TOUATI (Université de Versailles Saint-Quentin en Yvelines)

Table des matières

Objectifs

Notions de types dans un compilateur. Analyse de type, vérification de types; exemple d'un système de typage simple.

Lectures conseillées :

- Aho, Sethi et Ullman : chapitre 6
- Compilateurs : chapitre 6.1.2

Les types, pourquoi ?

I

A.Les types, pourquoi ?

- Génie logiciel :
  - Facilité de programmation
  - Structuration des applications, plus de visibilité algorithmique.
  - Portabilité des structures de données.
- Compilation : détection d'erreurs, gestion des données (taille, etc.).
- Sécurité : limiter les bugs.

Système de typage

II

A.Système de typage

- Un système de typage permet de gérer les types (analyse sémantique, détection d'erreur).
- Un système de typage est composé de :
  - Les constructions syntaxiques du langage
  - Le concept et la notion de types
  - Des règles pour affecter des types aux constructions du langage.

- Extrait des manuels de référence de C et pascal

  « "Si les deux opérandes des opérateurs arithmétiques d'addition, de soustraction et de multiplication sont de type entier, alors les résultat est de type entier" »

  « "Le résultat de l'opérateur unaire & est un pointeur vers l'objet que désigne l'opérande. Si le type de l'opérande est X, alors le type du résultat est 'pointeur vers X'" »
- Un système de typage permet de formaliser et d'automatiser ce genre de spécifications sémantiques d'un langage.

Expressions de types

- Un type composé est défini par une **expression de type**.
- Une expression de type est :
  - Un type de base
  - Un nom de type défini par une expression de type.
  - Ou une application d'un constructeur (opérateur) de type sur d'autres expressions de types.

Expressions de types : types de base

- Types atomiques reconnus par le langage.
  - entiers

  - booléens
  - flottants
  - caractères,...
- erreur_de_type (`type_error`)
  - type spécial utilisé par le compilateur pour signaler et propager des erreurs de types dans les expressions.
- vide (`void`)
  - type de base spécifiant l'absence de type. Il est utilisé par le compilateur, et parfois par le langage.

Expressions de types : noms de types

- Les langages de programmation haut niveau permettent aux utilisateurs de déclarer de nouveaux types.
- Ex: `typedef enum {violet, jaune, bleu,rouge} couleurs;`
- `couleurs` devient donc un type par construction.

B.Construction de types

Constructeurs de types : le tableau

- Si T est une expression de type, alors $tableau(I,T)$ est une expression de type.
- Ce type dénote un tableau dont les éléments sont de type T et les indices de type I.
- I peut être un intervalle d'entiers.

Constructeurs de types : le produit

- Si T_1 et T_2 sont des expressions de types, alors $T_1 \times T_2$ est aussi une expression de types.
- A vrai dire, le produit des types n'est qu'une concaténation.
- Ex : une liste de paramètres $(p_1, ..., p_n)$ sont de type $T_1 \times ... \times T_n$.

Constructeurs de types : les structures

- Le type d'une structure est le produit des types de ses champs, en incluant les noms des champs.
- Ex : le type de cette structure est
  $structure((adress, int) \times (lexeme, tableau(10, char))$

```
struct{
int adress;
char lexeme[10]:
};
```

Constructeurs de types : les pointeurs

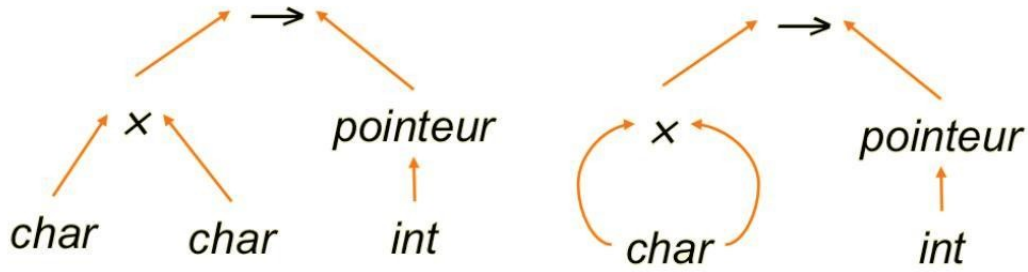
- Si T est une expression de type, alors $pointeur(T)$ est une expression de type.
- Exemple : `int *x;`
  - déclare un objet x de type "pointeur vers int"

Constructeurs de types : les fonctions

- Comme en mathématiques, une fonction fait correspondre des éléments d'un domaine (type) D vers des éléments d'un co-domaine (type) A.
- Le type d'une telle fonction est $D \longrightarrow A$
- Ex : `int *f(char a, char b)` déclare une fonction de type $char \times char \longrightarrow pointeur(int)$

C.Arbre ou DAG d'une expression de type

L'expression de type $char \times char \longrightarrow pointeur(int)$ est représentée comme une expression arithmétique



D.Contrôle de type

- Il peut se faire statiquement par le compilateur, ou dynamiquement lors de l'exécution du programme.
- Le système de typage doit être capable (idéalement) de dire si deux types sont identiques.
  - Cela n'est pas toujours facile.
  - Beaucoup de compilateurs usuels n'ont pas de systèmes de typage performants.
- Il est recommandé de prévoir des mécanismes de récupération sur erreur.

Complément

Ex de contrôle dynamique de types : frontières d'un tableau (pascal), mauvaise référence (java). Le compilateur introduit du code pour ce traitement directement dans le code du programme.

E.Table des types

- Pour éviter l'hétérogénéité de la table des symboles, on peut décider de rassembler les types d'un programme dans une **table de type**.
- Une entrée par type qui contient :
  - constructeur

- taille et alignement de ses variables
- liste des champs si c'est une structure
- dimensions si c'est un tableau, etc.

Un système de typage simple

III

A.Un système de typage simple

D → L **id**; D → L ***id***, | T L → ***int*** | ***float*** | ***char*** | ***array[nb] of*** *T* | **T*	D.type L.type T.type ***id***.nom ***nb.***val

Tableau 1 Système de typage simple

Production	Règle sémantique
D → L **id**;	TS.inserer(***id***.nom,L.type) D.type =L.Type;
L → L_1 ***id***,	TS.inserer(***id***.nom,L1.type) L.type=L1.type
L → T	L.type = T.type
T → ***int***	T.type= entier
T → ***char***	T.type=caractère
T → ***array[nb] of T1***	T.type=tableau(**nb**.val, T1.type)
T → $*T_1$	T.type=pointeur (T1.type)

Tableau 2 Système de typage simple

Complément

Ces règles sémantiques permettent d'affecter un type dans une déclaration de variables.

La traduction dirigée par la syntaxe ici est L-attribuée : en effet, il y a des attributs hérités. Par ex, la première production utilise l'attribut D.type qui est un attribut du membre gauche de la production, c'est un attribut hérité.

Contrôle de type dans des expressions arithmétiques

E → ***litteral*** E → ***nb*** E → ***id*** E → E ***mod*** E E → id ***[***E***]*** E → *E	E.type ***id***.type ***id***.nom

Tableau 3 Contrôle de type dans des expressions arithmétiques

Production	Règle sémantique
E → ***litteral***	E.type = caractère
E → ***nb***	E.type = entier
E → ***id***	E.type = TS.rechercher_type(***id***.nom)
E → E_1 ***mod*** E_2	**si** E1.type==E2.type==entier **alors** E.type= entier **sinon** E.type = erreur_de_type
E → id ***[***E_2***]***	**si** E2.type==s **et** id.type==tableau(s,t) **alors** E.type= t **sinon** E.type = erreur_de_type
E → *E_1	**si** E1.type==pointeur(t) **alors** E.type= t **sinon** E.type = erreur_de_type

Tableau 4 Contrôle de type dans des expressions arithmétiques

Complément

Ces règles sémantiques permettent de contrôler et d'affecter des types aux expressions arithmétiques.

C'est une traduction dirigée par la syntaxe S-attribuée, car tous les attributs sont synthétisés. En effet, dans chaque production, seul l'attribut du membre gauche est calculé en fonction des attributs des membres droit.

ATTENTION : l'attribut id.type est calculé en faisant une recherche dans la TS : id.type=TS.rechercher_type(id.nom)

Contrôle de type des instructions

I → ***id*** **=** E I → ***if*** E ***then*** I I → ***while*** E ***do*** I I → I;I	E.type I.type ***id***.type

Tableau 5 Contrôle de type des instructions

Production	Règle sémantique
I → ***id*** **=** E	**si** ***id***.type==E.type **alors** I.type= vide **sinon** I.type = *erreur_de_type*
I → ***if*** E ***then*** I_1	**si** E.type==booléen **alors** I.type= I_1.type **sinon** I.type = *erreur_de_type*
I → ***while*** E ***do*** I_1	**si** E.type==booléen **alors** I.type= I_1.type **Sinon** I.type = *erreur_de_type*
I → I_1 ; I_2	**si** I1.type==I_2.type==vide **alors** I.type= vide **sinon** I.type = *erreur_de_type*

Tableau 6 Contrôle de type des instructions

Complément

En général, les instructions n'ont pas de valeurs associées (pas de types). On leur affecte le type spécial void.

Si erreur dans l'instruction, on lui affecte le type erreur_de_type.

L'exemple ici est une traduction dirigée par la syntaxe S-attribuée, car tous les attribut sont synthétisés. En effet, dans chaque production, seul l'attribut du membre gauche est calculé en fonction des attributs des membres droit.

ATTENTION : l'attribut id.type est calculé en faisant une recherche dans la TS : id.type=TS.rechercher_type(id.nom)

Contrôle de type des appels de fonctions

E → ***id*** (E)	***id***.type

Tableau 7 Contrôle de type des appels de fonctions

Production	Règle sémantique
$E \rightarrow id\ (E_1)$	**si** E_1.type==s **et** ***id***.type == s → t **alors** E.type= *t* **sinon** E.type = *erreur_de_type*

Tableau 8 Contrôle de type des appels de fonctions

Complément

Dans le cas où nous avons plusieurs arguments, nous pouvons supposer que le type de la liste des arguments est le produit cartésien des types (concaténation des types). En effet, n arguments de type T_1, ..., T_n peuvent être considérés comme un seul argument de type T_1 X...X T_n.

ATTENTION : l'attribut id.type est calculé en faisant une recherche dans la TS : **id**.type=TS.rechercher_type(id.nom)

Équivalences entre les types

IV

A. Équivalence de noms

Équivalence entre les expressions de types

- Le problème est de déterminer avec exactitude si deux types sont équivalents.
- Deux types sont équivalents si l'on peut substituer un des deux types avec l'autre sans altérer le résultat du programme.
  - Cette définition n'est pas toujours respectée !
- Beaucoup de compilateurs ne savent pas répondre parfaitement à l'équivalence des types.

Complément

Si un compilateur n'arrive pas à déterminer si deux types sont équivalents,

- Il émet un signal d'erreur s'il est conservatif.
- Si le compilateur est laxiste, il peut simplement émettre un avertissement (warning).

Équivalence entre les expressions de types

- Les deux types "couleurs" et "codes" sont-ils équivalents ?

```
typedef enum couleurs {bleu, vert, rouge};
typedef enum codes {QUIT,KILL,STOP};
```

- Je dirai oui, et gcc répond oui.
- Et si j'écris ce qui suit ?

```
typedef enum couleurs {bleu, vert, rouge};
typedef enum codes {QUIT=4,KILL,STOP, SUSP};
```

- J'aurais dit non, mais gcc dit oui tout de même. Il a un système de typage faible.

Complément

Un système de typage faible laisse passer beaucoup de sources de bugs. Le programmeur doit savoir ce qu'il fait.

Un langage avec un système de typage fort comme caml laisse passer peu de sources de bugs, et permet beaucoup de vérifications formelles du programme. Hélas, le développement logiciel sera plus lent (même s'il est plus sûr).

Équivalence de noms

- Si deux types ont le même nom, ils sont équivalents.
- Le nom d'un type peut être donné par le programmeur ou par le compilateur.

non équivalents par noms

```
type t1= tableau [int]  de int;
type t2= tableau [int]  de int;
```

équivalents par noms

```
type t3= tableau [int]  de int;
type t4= t3;
```

Graphique 1 Équivalence de noms

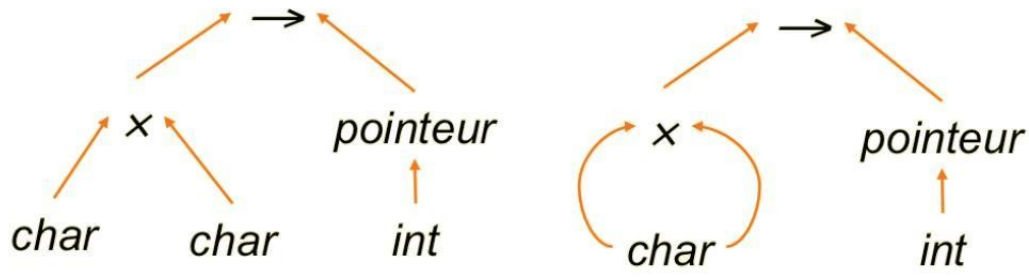
B.Équivalence structurelle

Équivalence structurelle : cas simple

- Deux expressions de types sont **structurellement équivalentes** ssi leurs arbres ou DAGs sont identiques.

Arbre ou DAG d'une expression de type

l'expression de type $char \times char \longrightarrow pointeur(int)$
est représentée comme une expression arithmétique



- Comparer exactement deux graphes peut être lent !
- Il existe des techniques « allégées » qui déterminent si deux graphes ne sont pas équivalents en comparant uniquement l'ordre des constructeurs dans les expressions des types.

Complément

Souvent, on a besoin d'alléger le test d'équivalence structurelle. Par exemple, on n'a pas besoin de tester les bornes des tableaux, etc.

Parfois, on ne teste même pas les noms des champs des structures.

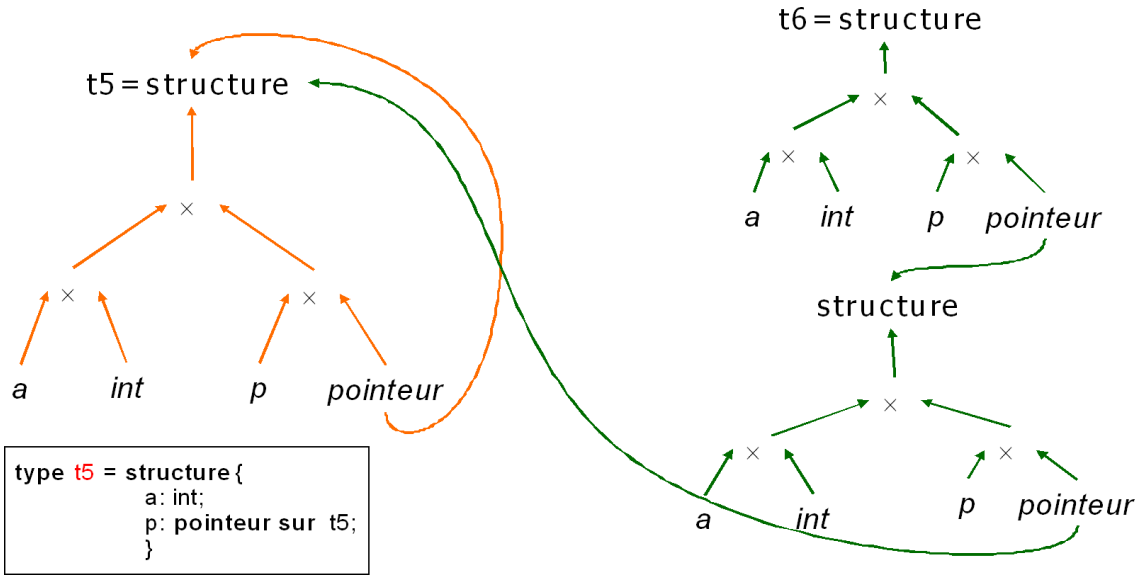
Équivalence structurelle : cas compliqué

- Dans le cas général, le problème d'équivalence structurelle est plus difficile car :
  - des expressions de types peuvent contenir des noms de types qui ne sont pas des types de base.
  - l'utilisation des pointeurs peut rendre cyclique la représentation d'un type.

Exemple d'équivalence structurelle compliquée

```
type t5   = structure{
            a : int ;
            p : pointeur sur t5 ;
}
type t6   = structure{
            a : int;
            p : pointeur sur structure{
                a : int;
                p : pointeur sur t5
                }
}
```

Exemple d'équivalence structurelle compliquée




```
type t5 = structure {
            a: int;
            p: pointeur sur t5;
            }
```

Complément

Les types t5 et t6 sont équivalents mais il n'est pas aisé de le déduire en comparant ces deux graphes (le vert et l'orange)

Conversions des types

V

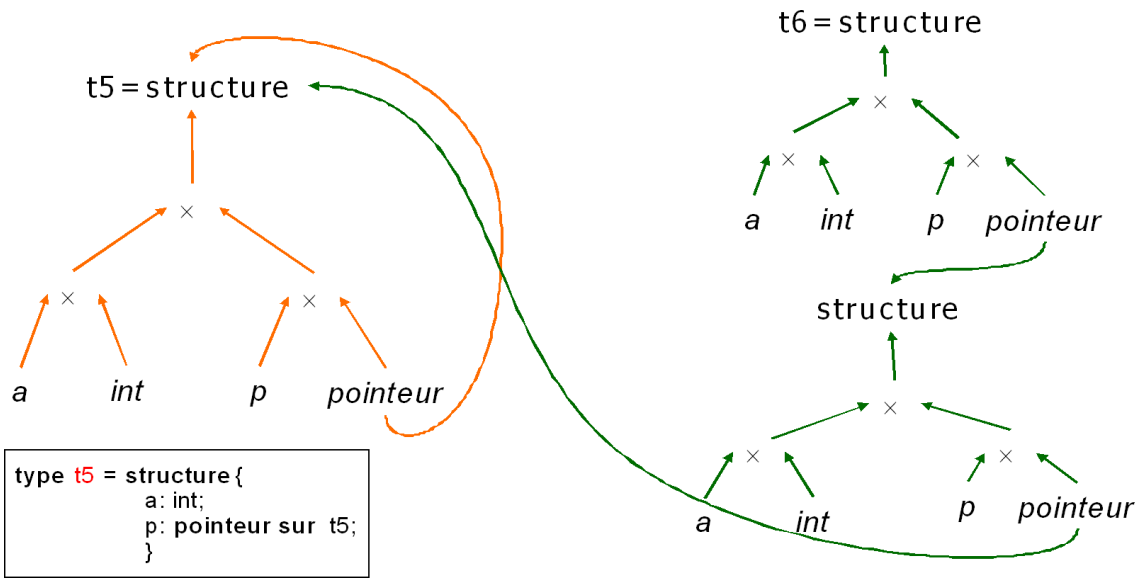
A. Conversion implicite

Conversion de types

- La conversion de types s'effectue car :
  - Les compilateurs essayent de générer des codes effectuant des calculs entre données de types différents.
  - Quand l'équivalence de types échoue, le compilateur tente de corriger l'erreur et continue d'analyser le programme.

- Elle est également appelée **coercition**.
- Les règles sont définies par le langage
  - Ex : flottant + entier = flottant.
- En général, les conversions implicites de types se donnent le principe de ne pas perdre d'information.
  - Ne pas transformer une donnée stockée sur n bits en une autre stockée sur $m<n$ bits.

Quels sont les conversions de types que l'ont doit appliquer à cet arbre abstrait ?



Complément

Dans le cas général, trouver les conversions implicites dans un arbre abstrait avec des types et opérations quelconques est un problème difficile. En effet, il faut trouver une séquence de conversions de types afin de pouvoir évaluer l'arbre abstrait.

B.Conversion explicite

Forçage explicite de type

- C'est un mécanisme prévu par certains langages pour indiquer (forcer) le type d'une expression dans un programme.
- Ex : le cast en langage C

  `(type) expr`
- Ce mécanisme est prévu pour pallier les faiblesses du système de typage. Mais en pratique, on en abuse !
  - Rétrécissement des données : double to int.
  - Conséquence: des bugs difficilement détectables.

Conversion et forçage explicite de type

- Il y a subtilité entre une conversion explicite de type et un forçage de type.
- Une conversion de type introduit une fonction (opération) qui transforme la donnée en une autre. Ex : transformer un entier en flottant.
  - Cette fonction obéit aux normes du langage.
- Un forçage de type ne transforme pas la donnée, mais seulement son type, qui est une information utilisée par le compilateur.

Conclusion

- Le typage est un concept très important dans les langages haut niveau.
  - Il définit une partie de la sémantique et permet de structurer nos programmes.
  - Il élimine des sources d'erreurs.
- Il existe beaucoup de travaux de recherche qui tentent de définir des systèmes de typages forts, cohérents et efficaces. En général, ils utilisent des méthodes formelles.