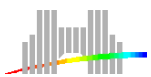
Master Informatique Fondamentale - M1 Compilation

Systèmes de types

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

8 février 2007



Système de types

- Un type est un ensemble. Exemples : les entiers, les caractères Unicode, les booléens ...
- Dans la mémoire d'un ordinateur, toute variable est représentée par une chaîne de bits. Dire qu'une variable est d'un certain type indique comment interpréter la chaîne de bits associée.
- Par exemple la chaîne `01100001` représente l'entier 97, ou le caractère a ou le booléen **vrai** (convention C).
- Connaître les types de données permet de savoir :
  - Quelles opérations utiliser (e.g. le choix entre opérateur virgule flottante ou opérateur entier)
  - Quelle place réserver en mémoire (e.g. le choix entre `char`, `short`, `long` et `long long` en C).
- Tous les langages ont un système de types, sauf les langages qui ne connaissent qu'un seul type (les langages de *script* – shell, awk – ne connaissent que le type chaîne de caractères).
- Le langage machine n'a pas de type.

Typage fort, typage faible

- Quand un langage a de nombreux types, il est souvent nécessaire de convertir entre types (e.g. passer d'un caractère à un entier).
- Un langage est à typage fort quand ces conversions doivent être indiquées explicitement ou sont interdites. Exemples : Pascal, Ada.
- Un langage est à typage faible quand ces conversions sont libres et implicites. Exemple : C

```
char c; int x;
x = c - '0';
```

- Prétexte : efficacité.

Surcharge

- Dans la plupart des langages, les opérateurs arithmétiques sont surchargés : `x + y` engendre une opération entière ou une opération en virgule flottante ou même une conversion suivant le type de `x` et `y`.
- On dit que l'opérateur + est surchargé.
- A partir de Ada, il est possible de surcharger une fonction, c'est à dire de la déclarer plusieurs fois à condition que les diverses versions puissent être distinguées par les types de arguments et du résultat.
- A partir de C++, il est possible de redéfinir, de la même façon, les opérateurs du langage (exemple : << peut être un opérateur de décalage ou un opérateur d'écriture suivant le type de son premier argument).

Contrôle ou inférence, il faut choisir

Un système de types peut être utilisé de deux façons différentes :

- En contrôle, tous les identificateurs sont typés ; le compilateur calcule le type de toutes les expressions du langage, et vérifie que les affectations sont cohérentes.
- En inférence, les identificateurs ne sont pas typés. Le compilateur infère les types essentiellement à partir des opérateurs utilisés, et vérifie que chaque identificateur est toujours utilisé avec le même type.
- Le seul langage à inférence de type est ML, Ocaml, etc.
- Comparaison :
  - L'inférence allège le travail du programmeur, mais les erreurs sont plus difficiles à corriger. On maintient en général la possibilité de typer certains identificateurs critiques (e.g. les arguments des fonctions).
  - En inférence, il est possible d'écrire du code générique.
  - L'inférence de type interdit en général la surcharge. Exemple : l'addition en virgule flottante de Ocaml : `+.`

Histoires de types

- Les types ont été inventés par Bertrand Russel pour éviter un paradoxe : l'ensemble des ensembles qui ne se contienennt pa comme élément n'est pas bien typé.
- Les premiers langages n'avaient que très peu de types fixes : entiers, flottant puis caractères.
- Sont venus ensuite les tableaux et les fonctions.
- C'est en Cobol qu'est apparu le premier constructeur de type, `record`.
- La surcharge vient de Ada et des langages à objets.
- Le système de types polymorphes de Ocaml vient du λ-calcul typé, lui même inventé pour garantir la normalisation forte.

Le système **F**

Le système **F**, qui sert à typer le λ-calcul, possède un certain nombre de types primitifs (par exemple `int`, `char`, `bool`) et un constructeur de type, $\rightarrow$.

- Si τ et σ sont deux types – deux ensembles – $\tau \rightarrow \sigma$ est le type des fonctions de τ vers σ.
- Si x est de type τ et si f est de type $\tau \rightarrow \sigma$, alors fx est de type σ :

$$\frac{x :: \tau \quad f :: \tau \rightarrow \sigma}{fx :: \sigma}\ (\mathbf{F})$$

- Par exemple, le type de $+$ est $\texttt{int} \rightarrow (\texttt{int} \rightarrow \texttt{int})$.
- Le type de $**$ (l'exponentiation) est $\texttt{float} \rightarrow (\texttt{int} \rightarrow \texttt{float})$

Types étendus : tableaux

- On type un tableau comme une fonction des entiers vers le type de ses valeurs. `char A[100]` est traité comme $\texttt{A} :: \texttt{int} \rightarrow \texttt{char}$.
- On remarque que l'information de taille est perdue. Il existe des systèmes de type plus complexes (types dépendants) qui la conservent.
- Le type de l'opérateur d'indexation `[]` est $(\texttt{int} \rightarrow \alpha) \rightarrow \texttt{int} \rightarrow \alpha$, où α est une variable de type.

$$\frac{[] :: (\texttt{int} \rightarrow \alpha) \rightarrow \texttt{int} \rightarrow \alpha \quad \texttt{A} :: \texttt{int} \rightarrow \texttt{char}}{\texttt{A}[] :: \texttt{int} \rightarrow \texttt{char} \quad \texttt{int} :: \texttt{i}}$$

$$\frac{}{\texttt{A[i]} :: \texttt{char}}$$

Types produits

Le type (l'ensemble) $\tau \times \sigma$ est le produit cartésien de τ et σ.

- Pour utiliser un type produit cartésien, on a besoin d'un constructeur, de type $\tau \longrightarrow \sigma \longrightarrow \tau \times \sigma$, et des deux projections de type $\tau \times \sigma \longrightarrow \tau$ et $\tau \times \sigma \longrightarrow \sigma$.
- Dans tous les langages depuis Cobol, un tel objet se nomme un record ou une structure, et on écrit :

```
struct A {
  t1 x; t2 y; ...} u;
```

  Le type de `u` est `t1` $\times$ `t2`. On peut considérer `x` et `y` comme les projecteurs, de type `t1` $\times$ `t2` $\longrightarrow$ `t1` et `t1` $\times$ `t2` $\longrightarrow$ `t2`.
- On peut définir de façon analogue des types somme (union en C).

Types pointeurs

- Pour définir un type pointeur, il faut définir un constructeur de type. En C, on utilise $*$, à la fois comme constructeur et comme opérateur d'adressage indirect.
- On a déja rencontré des constructeurs de type, comme $\times$ ou $\longrightarrow$.
- $*\alpha$ est le type des pointeurs vers des objets de type α.
- Le type de l'opérateur d'adressage indirect est :

$$* :: (*\alpha) \longrightarrow \alpha$$

- En Ocaml, on peut inventer des constructeurs de type arbitraires. Par exemple, le type des tables de hachage à clef de type α contenant des valeurs de type β est (α, β)`Hashtbl.t`.

Inclusion de types

Comme un type est aussi un ensemble, il peut y avoir inclusion entre deux types.

- Exemple : en C, les types `char`, `short`, `long` et `long long` peuvent être vu comme des sous-types du type `int`.
- Si $\alpha' \subseteq \alpha$ et si $\beta \subseteq \beta'$ alors une fonction de α vers β peut être également vue comme une fonction de α' vers β' :

$$\alpha \to \beta \subseteq \alpha' \to \beta'$$

On parle de covariance et de contravariance.

Coercions

Une coercion est une opération de changement de type, en général sans action sur la valeur de la donnée.

- Notation `(t1 :> t2)` (Ocaml) ou `(t2)` (C) parce que l'on suppose que le type de départ est clair.
- Une coercion d'un sous-type vers un sur-type est toujours légale.
- Une coercion vers un sous-type est dangereuse. On peut faire confiance au programmeur (C) ou vérifier à l'exécution (Java) ou interdire (Ocaml).

Représentation des types

Un type se représente facilement par un arbre, dont les feuilles sont les types primitifs et les nœuds sont les constructeurs.

- Dans certains langages (Ocaml, Pascal) la conversion est triviale.
- En C c'est plus complexe, parce que le type est divisé entre le déclarateur et le déclarande :

```
char * foo(int);
```

$$\texttt{int} \longrightarrow (*\texttt{char})$$

Tables des symboles

Il n'est pas praticable d'aller chercher les déclarations dans l'arbre syntaxique chaque fois que l'on en a besoin. On construit donc une table des symboles.

- Table hashcodée ; clef : le symbole ; valeur : le type et d'autres renseignement développées ultérieurement.
- Dans beaucoup de langages il y a plusieurs espaces de nom :
  - par fonction ou module
  - par genre d'information (données, membres de classes, etc).
- On peut batir de multiples tables, ou qualifier les symboles.

Un algorithme de vérification

- Le but est de détecter le maximum d'erreurs de typege.
- On calcule le type de toutes les expressions du programme (attribut synthétisé).
- On détecte les erreurs d'utilisation des opérateurs (e.g. une addition à un booléen), et les erreurs d'affectation.
- Quand les types se mélangent (exemple : entiers et flottants) on peut inférer automatiquement les conversions.
- On en profite pour résoudre les surcharges.
- On peut implémenter l'algorithme soit de façon *ad hoc* : on écrit un bout de code pour chaque opérateur,
- soit de façon générique : l'algorithme est dirigé par des tables donnant les types de chaque opérateur.

Un algorithme d'inférence, I

- Pour simplifier, on se place dans le cadre du système **F**.
- On suppose que les opérateurs peuvent avoir des types génériques (i.e. qui peuvent contenir des variables de type). Par exemple, suivant le langage, le type de l'opérateur d'affectation est $\alpha \to \alpha \to$ `unit` ou $\alpha \to \alpha \to \alpha$ (C).
- On représente le type inconnu de chaque identificateur et de chaque expression par une variable de type.
- Soit `a op b` une expression, τ_a, τ_b les types de `a`, `b`, et τ_{op} le type de l'opérateur. On a l'équation entre types :

$$\tau_{\text{op}} = \tau_a \to \tau_b.$$

- Soit `fun x -> e` une définition de fonction, de type τ_f, les types de `x`, `e` étant τ_x, τ_e :

$$\tau_f = \tau_x \to \tau_e$$

- Moyennant quelques précautions, on peut résoudre ces équations par une méthode analogue à l'élimination de Gauss.
- On doit s'assurer que tous les identificateurs utilisés sont distincts.

Un algorithme d'inférence, II

- On recherche une substitution S agissant sur les variables de type et qui transforme toutes les équations en identités.
- On sélectionne une équation $\sigma = \tau$
- On calcule l'unifiant le plus général s de σ et τ :
  - $s\sigma = s\tau$.
  - Tout t tel que $t\sigma = t\tau$ s'écrit $t = r \circ s$.
- Si s n'existe pas, le terme initial est mal typé.
- Sinon, on applique s à toutes les autres équations et on recommence.

Un exemple

Soit à typer `let f = let e = fst i in e + 1;;`

On a introduit des noms pour les sous-expressions de `fst i + 1`

On connait les types de `fst` : $\texttt{list}\ \alpha \to \alpha$, de 1 : `int` et de $+$: $\texttt{int} \to \texttt{int} \to \texttt{int}$.

Equations :

$$\begin{aligned} \tau_e \to \texttt{int} \to \tau_f &= \texttt{int} \to \texttt{int} \to \texttt{int} \\ \tau_i \to \tau_e &= \texttt{list}\ \alpha \to \alpha \\ s_1 &= [\tau_e \leftarrow \texttt{int}, \tau_f \leftarrow \texttt{int}] \\ \tau_i \to \texttt{int} &= \texttt{list}\ \alpha \to \alpha \\ s_2 &= [\alpha \leftarrow \texttt{int}, \tau_i \leftarrow \texttt{list int}] \end{aligned}$$

Types exotiques

- Les systèmes de type ont été utilisés pour toutes sortes d'analyses.
- Exemple : en synthèse, comme toutes les données sont des chaînes de bits, leur type est leur longueur.
- Le type d'une fonction peut être une estimation de sa complexité : 1, 2, 3, ..., récursif.
- Les systèmes d'effets on pour but d'identifier les effets de bord des constructions du programme.

Types manifestes

- Les types envisagés jusqu'ici restent implicites : ils ne figurent pas dans le code exécutable.
- Dans certaines situations, il peut être indispensable de matérialiser les types dans les structures de données. On parle de *tags* ou de types manifestes.
- Exemple :

```
type statement =
    Assign of string | Nop | If of string * statement * stat
;;
```

  Les constructeurs `Assign`, `Nop` et `If` doivent être codés dans la structure de donnée. Même remarque pour les records variants de Pascal et les unions de C (plus laxiste).
- Le langage Java admet les coercions d'une classe vers l'une de ses héritières : il faut pouvoir vérifier en codant le type.
- Les types manifestes sont spécialement utiles pour les langages faiblement typés.