Typage de programmes ML

Julien Forget
julien.forget@onera.fr

April 30, 2009

Langages de programmation et types

- Les types et les systèmes de types constituent la technique principale de validation d'un programme.
- Ces notions sont liées à des algorithmes et des techniques de preuve.
- Un système de types n'effectue qu'une validation partielle.
- Un système de types contraint, régit et complique les moyens d'expression d'un langage.

$\Rightarrow$ Il n'y a pas de solution unique. Nous étudions ici le typage des langages de la famille ML.

Base des langages fonctionnels : le λ-calcul (1)

Soit V un ensemble de variables et C un ensemble de constantes.
Le langage L des λ-expressions sur V et C est tel que :

- Si $x \in V \cup C$, alors $x \in L$.
- Si $x \in V$ et $e \in L$, alors $\lambda x.e \in L$ (abstraction).
- Si $e_1, e_2 \in L$, alors $e_1 e_2 \in L$ (application).

Le λ-calcul (2)

- L'occurrence d'une variable x dans une λ-expression e est dite liée (bound) si $e = \lambda x.e'$ et $x \in e'$. Elle est libre (free/unbound) sinon.
- La β-réduction de $(\lambda x.e_1)\ e_2$ consiste à substituer toutes les occurrences de x liées dans e_1 par e_2.

β-réduction

$$(\lambda x.x)\ 1 \longrightarrow 1$$
$$(\lambda x.x)\ (\lambda x.x) \longrightarrow (\lambda x.x)$$
$$(\lambda x.y)\ 1 \longrightarrow y$$

Correspond à l'application de fonction en ML.

Typage ?

La version du λ-calcul que nous venons d'étudier est un exemple de langage non-typé, ce qui pose plusieurs questions :

- Les expressions correspondent-elles toutes à des valeurs ? (ex: $x\ x$)
- Peut-on regrouper des expressions partageant un ensemble de propriétés ?
- Comment distinguer des comportements suivant les propriétés d'une valeur (ex : addition d'entiers $\neq$ addition de flottants).

Les Types

- Organisent les valeurs traitées par les programmes en ensembles qui sont caractérisés par l'usage que l'on en fait.
- Permettent de vérifier la validité (une partie) des programmes lors de leur compilation et/ou de leur exécution (prouve l'absence de certains mauvais comportements).
- Documentent un programme, ses compilations, et/ou ses exécutions.
- Rationalisent la représentation des valeurs et leurs transformations au niveau machine.

Type : ensemble d'entités (ou valeurs) partageant les mêmes propriétés.

Système de type

- Jugement de type : $E \vdash e : t$, signifie “dans l’environnement (ou le contexte) E, l’expression e a pour type t”.
- Système de type : ensemble de types + règles permettant de déterminer si un programme/une expression est bien typée ou non.
- Vérification de type : l’affirmation $E \vdash e : t$ est-elle correcte ?
- Inférence de type : étant donné E peut-on trouver t tel que $E \vdash e : t$? (c’est le cas des langages ML)
- L’ensemble des règles d’un système de type forme un système de preuve permettant de prouver que le type d’une expression est correct ou non.

Règles de typage : règles d'inférence

Type d'une constante

$$\frac{c \in dom(E)}{E \vdash c : E(c)}$$

- E est un environnement associant un type à une expression.
- Une règle d'inférence dit que si les prémisses de la règle (partie haute) sont satisfaites alors on peut en déduire que la conclusion (partie basse) est vraie.
- Ci-dessus on déclare qu'une constante c a pour type $E(c)$ si elle fait partie du domaine de E.
- On dit ainsi que c est bien typée si elle est liée (déclarée) dans E.

Types de base ML

Les types de base d'un noyau ML sont construits à partir de constantes de types ($\mathcal{C}$), de variables de types ($\mathcal{V}$) et de constructeurs de types. Soit $\mathcal{T}$ l'ensemble des types ML.

- Constantes de types (`int`, `float`, ...) : si $t \in \mathcal{C}$, alors $t \in \mathcal{T}$.
- Variables de types : si $t \in \mathcal{V}$, alors $t \in \mathcal{T}$.
- Type produit : si $t_1,\ t_2 \in \mathcal{T}$, alors $t_1 * t_2 \in \mathcal{T}$.
- Type liste : si $t \in \mathcal{T}$, alors $t\ list \in \mathcal{T}$.
- Type fonctionnel : si $t_1,\ t_2 \in \mathcal{T}$, alors $t_1 \rightarrow t_2 \in \mathcal{T}$.

Schémas de types $\mathcal{S}$ (pour le polymorphisme), $\mathcal{T} \subset \mathcal{S}$.

- $\sigma = \forall \alpha_1,\ \ldots,\ \alpha_n, t$
- En OCaml le quantificateur est remplacé par la convention d'écriture $'t$, ex : $'a \rightarrow' b \rightarrow ('a *' b)$.

L'environnement initial

- Le typage s'effectue à partir d'un environnement de typage initial E_{init} contenant le type des constantes et des opérateurs prédéfinis du langage.
- $E_{init}(3) = int$
- $E_{init}(+) = int \rightarrow int.$
- $E_{init}(::) = \forall \alpha, \alpha \rightarrow \alpha\ list \rightarrow \alpha$

$\Rightarrow$ Le typage de "$+$" s'effectue à l'aide de la règle d'inférence sur les constantes vue précédemment (ou plutôt la règle sur les variables, très similaire, pour des questions de polymorphisme).

Application de fonction

Règle d'inférence

$$\frac{E \vdash e_1 : t_1 \longrightarrow t_2 \qquad E \vdash e_2 : t_1}{E \vdash e_1\ e_2 : t_2}$$

- $e_1\ e_2$ est l'application de la fonction e_1 à e_2.
- Pour que $e_1\ e_2$ soit bien typé, il faut que :
  - e_1 soit une fonction (prémisse gauche).
  - e_2 ait le type du paramètre attendu par e_1 (prémisse droite).
- $e_1\ e_2$ a alors pour type le type retourné par la fonction e_1.
- Exemple d'application :

```
# float_of_int 1;;
- : float = 1.
```

Définition de fonction

Règle d'inférence

$$\frac{E, x : t_1 \vdash e : t_2}{E \vdash (\mathit{fun}\ x \longrightarrow e) : t_1 \longrightarrow t_2}$$

- Pour pouvoir calculer le type de l'expression e, on rajoute x dans l'environnement avec son type (qui est souvent quelconque).
- Intuitivement, cela revient à déclarer les paramètres de la fonction dans l'environnement.
- Dans cet environnement enrichi, on calcule ensuite le type de e.
- Le type obtenu pour e sera le type de retour de la fonction.
- L'expression résultante a un type fonctionnel.

Unification

Une substitution de type est une application $s : \mathcal{V} \rightarrow \mathcal{T}$. Appliquer s à $\sigma \in \mathcal{T}$ donne un type σs :

- $xs = s(x)$
- $(\sigma \rightarrow t)s = (\sigma s) \rightarrow ts$
- $(\sigma * t)s = (\sigma s) * ts$
- $(\sigma\ list)s = (\sigma s)\ list$

Si $\sigma_1, \sigma_2 \in \mathcal{T}$, unifier σ_1 et σ_2 consiste à trouver une substitution s telle que $\sigma_1 s = \sigma_2 s$.

Exemple

$t_1 \rightarrow t_2$ et $t_1 \rightarrow t_3 \rightarrow t_4$ sont unifiables :
$s = \{t_2 \mapsto t_3 \rightarrow t_4\}$.

Preuve de type

- Une preuve de type consiste à utiliser les règles d'inférence récursivement afin de trouver un type pour une expression.
- A chaque niveau de récursion :
  1. On considère la règle dont la conclusion correspond à l'expression à typer.
  2. On cherche à prouver chacune des prémisses.
  3. La preuve d'une prémisse calcule elle-même un type.
  4. Le type calculé pour une prémisse doit être unifiable avec le type attendu par la règle pour cette prémisse.

Preuve de type (2)

- Si une unification échoue, la preuve est impossible, l'expression est donc mal typée.
- En OCaml les erreurs de type affichées par le compilateur sont le résultat d'un échec d'unification :

Erreur de type

"This expression has **type** int but is here used **with type** float"
$\Rightarrow$ L'unification a échoué car `int` et `float` ne sont pas unifiables.

Exercice : preuves de type

Ecrire les preuves de types des expressions suivantes :

- `(fun x -> x+1) 3`
- `fun x -> 1::x`
- `fun x -> x+(x,x)`

Déclaration locale

Règle d'inférence

$$\frac{E \vdash e_1 : t_1 \qquad \sigma = generalize(t_1) \qquad E, x : \sigma \vdash e_2 : t_2}{E \vdash let\ x = e_1\ in\ e_2 : t_2}$$

- Ressemble à la règle d'inférence de la fonction.
- Mais : la généralisation introduit du polymorphisme, ie e_1 peut avoir un type variable.
- La déclaration locale est le seul point d'introduction du polymorphisme !

Généralisation

- Au cours du typage, on distingue en fait deux types de variables de type. Pour simplifier :
  - Variables monomorphes : ce sont les t_1, t_2, ..., que nous avons utilisé dans les règles d'inférence. Elles permettent juste de désigner un type que l'on ne connaît pas encore (que l'on attend de calculer).
  - Variables polymorphes (ou variables universelles) : ce sont les variables paramétrant les schémas de type (les α dans $\forall\alpha.\alpha \to \alpha$). Elles apparaissent cette fois dans le résultat final d'un calcul de type et désignent n'importe quel type.
- La généralisation consiste à remplacer les variables monomorphes d'un type par des variables polymorphes (un type non contraint devient quelconque).

Instanciation

Règle d'inférence d'une variable

$$\frac{E(x) = \sigma \qquad t = instance(\sigma)}{E \vdash x : t}$$

- L'instanciation est l'opération inverse de la généralisation.
- L'instanciation remplace les variables polymorphes d'un schéma de type par des variables monomorphes.
- Attention : l'instanciation ne modifie pas le type de la variable dans l'environnement E. On prend bien une instance de cette variable.
- L'instance de cette variable a un type pour l'instant inconnu (variable monomorphe) mais pas quelconque (variable polymorphe).

Exercice : typage polymorphe

Ecrire les preuves de types des expressions suivantes :

- `let f = fun x -> fun y -> (x,y)`
- `let g = fun x -> fun y -> x::[y]`
- `(g 2 3, g 'a' 'b')`