TD Types

Dans cette feuille de TD, on reprend comme base le mini-langage fonctionnel typé utilisé dans le cours. Rappel de la syntaxe des expressions e et des types τ :

$$
\begin{array}{lcl}
e & ::= & \mathsf{n} \\
 & | & e_1 + e_2 \\
 & | & x \\
 & | & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
 & | & \mathsf{fun}\ x \texttt{->} e \\
 & | & e_1\ e_2 \\
\\
\tau & ::= & \mathsf{int} \\
 & | & \tau_1 \to \tau_2
\end{array}
$$

Cette feuille contient de nombreux petits exercices, répartis en trois thèmes.

1 Expressions typables

Exercice 1 (Types simples) Les expressions suivantes sont-elles typables ? (sans polymorphisme, c'est-à-dire avec les règles de la page 42) Si oui donner une dérivation, et si non expliquer l'incohérence.

1. `let f = fun x -> x+1 in f (f 1)`
2. `let f = fun x -> x+1 in f f`
3. `let f = fun x -> fun y -> x in f 1`
4. `let f = fun x -> fun y -> x in f 1 2 3`
5. `let f = fun x -> fun y -> x in f f 2 3`
6. `let f = fun x -> fun y -> x in f (fun z -> z) 2 3`
7. `fun x -> fun y -> fun z -> x z (y z)`

□

Exercice 2 (Types polymorphes) Les expressions suivantes sont-elles typables dans le système de Hindley-Milner ? Ou avec des types polymorphes non restreints ? Fournir une dérivation le cas échéant.

1. `fun x -> x x`
2. `let f = fun x -> fun y -> x+1 in f f 1`
3. `let f = fun x -> fun y -> x+1 in f 1 f`
4. `let f = fun x -> fun y -> x in f f 2 3`

□

2 Extensions

Exercice 3 (Paires) On veut étendre le langage avec une notion de paire. On ajoute à la syntaxe des expressions les quatre constructions suivantes :

- (e_1, e_2) pour la construction d'une paire avec les valeurs des expressions e_1 et e_2,
- $\mathsf{fst}(e)$ pour l'extraction de la première composante de la paire obtenue en évaluant e,
- $\mathsf{snd}(e)$ pour l'extraction de la deuxième composante,
- $\mathsf{let}\ x, y = e_1\ \mathsf{in}\ e_2$ pour la définition simultanée de deux variables locales.

On étend également la syntaxe des types avec la construction :

- $\tau_1 \times \tau_2$ pour le type des paires dont la première composante a le type τ_1 et la deuxième composante a le type τ_2.

Donner des règles de typage pour ce langage étendu. □

Exercice 4 (Booléens) On veut étendre le langage avec des booléens. On ajoute :

- les constantes true et false,
- les opérations binaires $e_1 < e_2$, $e_1 = e_2$ et $e_1\ \&\&\ e_2$,
- une expression conditionnelle $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$.

Donner des règles de typage pour ce langage étendu. *Attention à ce que chaque opération soit aussi permissive que possible, mais pas plus que cela !* Que penser d'une expression de la forme $\mathsf{if}\ e_0\ \mathsf{then}\ e_1$? (vous pouvez réfléchir à ce qui se passe en caml avec une telle expression) □

Exercice 5 (Point fixe) On veut étendre le langage avec une définition récursive de variable locale :

— let rec $x = e_1$ in e_2

Donner une règle de typage pour cette nouvelle construction, et une dérivation de typage pour l'expression suivante :

```
let rec f =
  fun x -> 1 + f x
in
  f 0
```

□

Exercice 6 (Mise en œuvre (pour aller plus loin après la séance)) Étendre le vérificateur de types décrit dans le cours, section 5.3, pour intégrer ces extensions. Y a-t-il de nouveaux endroits où des annotations de types du programmeur sont nécessaires ?

Dans un deuxième temps, intégrer également ces extensions au moteur d'inférence décrit à la section 5.5. □

3 Raisonnements

Exercice 7 (Opérateurs paresseux) Dans le cadre de l'extension du langage avec les booléens, définir par des équations récursives une fonction F qui transforme une expression e en éliminant l'opérateur && : chaque opération e_1 && e_2 doit être remplacée par une expression conditionnelle.

Démontrer que si $\Gamma \vdash e : \tau$, alors $\Gamma \vdash F(e) : \tau$. □

Exercice 8 (Affaiblissement) Pour deux environnements Γ et Δ, on note $\Gamma \subseteq \Delta$ si pour tout $x \in \mathrm{dom}(\Gamma)$ on a $x \in \mathrm{dom}(\Delta)$ et $\Gamma(x) = \Delta(x)$.

Montrer que si $\Gamma \vdash e : \tau$ et $\Gamma \subseteq \Delta$, alors $\Delta \vdash e : \tau$. □

Exercice 9 (Adéquation) On définit l'opération de substitution $e[x := e']$ par les équations suivantes

$$
\begin{array}{rcll}
n[x := e'] & = & n & \\
(e_1 + e_2)[x := e'] & = & e_1[x := e'] + e_2[x := e'] & \\
x[x := e'] & = & e' & \\
y[x := e'] & = & y & \text{si } x \neq y \\
(\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2)[x := e'] & = & \mathsf{let}\ y = e_1[x := e']\ \mathsf{in}\ e_2[x := e'] & \text{si } x \neq y \text{ et } y \notin \mathsf{fv}(e') \\
(\mathsf{fun}\ y \to e)[x := e'] & = & \mathsf{fun}\ y \to e[x := e'] & \text{si } x \neq y \text{ et } y \notin \mathsf{fv}(e') \\
(e_1\ e_2)[x := e'] & = & e_1[x := e']\ e_2[x := e'] &
\end{array}
$$

On dira que la substitution n'est pas définie si la condition associée à un let ou un fun n'est pas remplie.

Montre que si on a les jugements $\Gamma \vdash e' : \tau'$ et $\Gamma, x : \tau' \vdash e : \tau$, et si la substitution $e[x := e']$ est définie, alors on a $\Gamma \vdash e[x := e'] : \tau$. □

Exercice 10 (Préservation des types) Définir par des équations récursives une fonction eval telle que $\mathrm{eval}(e, \rho)$ calcule la valeur de l'expression e dans l'environnement ρ, et démontrer que si $\emptyset \vdash e : \tau$ et $\mathrm{eval}(e, \emptyset) = v$, alors la valeur v est de type τ.

Indications : pour permettre la preuve par récurrence, il faudra énoncer une propriété plus générale. En outre, vous pouvez réutiliser la notion de substitution de l'exercice précédent pour éviter d'avoir à manipuler des clôtures. □