Cours de Compilation-Exercices

Typage

Master d'Informatique M1 2008–2009

14 octobre 2008

1 Typage de Pti-PtiCaml

Nous considérons ici un sous-ensemble du langage PtiCaml étudié dans le projet de programmation. Nous nous restreindrons uniquement aux expressions suivantes :

$\langle expr \rangle$	:=	$\langle simple\_expr \rangle$ | `function` $\langle ident \rangle$ `->` $\langle expr \rangle$ | $\langle expr \rangle$ $\langle op \rangle$ $\langle expr \rangle$ | $\langle simple\_expr \rangle$ $\langle simple\_expr \rangle^+$ | `let` $\langle ident \rangle$ `=` $\langle expr \rangle$ `in` $\langle expr \rangle$ | `if` $\langle expr \rangle$ `then` $\langle expr \rangle$ `else` $\langle expr \rangle$
$\langle simple\_expr \rangle$	:=	`(` $\langle expr \rangle$ `)` | $\langle ident \rangle$ | $\langle const \rangle$
$\langle op \rangle$	:=	`+` | `-` | `+.` | `-.` | `=`
$\langle const \rangle$	::=	`true` | `false` | $\langle entier \rangle$ | $\langle réel \rangle$
$\langle type \rangle$	:=	`bool` | `int` | `float` | α | $\langle type \rangle$ `->` $\langle type \rangle$

1. Définir en OCaml l'arbre de syntaxe abstraite des expressions et des types.

1.1 Typage monomorphe

Dans cette première partie, nous nous limitons au typage sans polymorphisme.

2. Quel est le type de l'expression suivante :

```
let id = function x -> x in
let a = id 1 in
id true
```

3. Définir le système de type pour Pti-PtiCaml monomorphe.
4. Écrire un typeur pour Pti-PtiCaml avec typage monomorphe et dans lequel les paramètres de fonctions sont annotés par leur type (`function (`$\langle ident \rangle$ `:` $\langle type \rangle$`) ->` $\langle expr \rangle$).

Nous voulons maintenant réaliser un typeur avec inférence de type (les arguments des fonctions ne sont plus annotés avec leur type).

5. Donner précisément la dérivation de typage de l'expression suivante :

```
function g -> function h -> function x -> g ((h x) + 1) + h (x + 2)
```

6. On représentera une substitution par une liste d'association. Définir une fonction `subst` qui applique une substitution à un type, une fonction `subst_env` qui applique une substitution à tous les types d'un environnement de typage et une fonction `compose` qui compose deux substitutions.
7. Définir une fonction `unify` qui calcule la substitution qui unifie deux types. `unify t1 t2` retourne une substitution `s` telle que `subst s t1 = subst s t2`.
8. Écrire un typeur pour Pti-PtiCaml avec typage monomorphe.

1.2 Typage polymorphe

Nous voulons maintenant introduire du typage polymorphe. Pour cela, l'environnement de typage associe à chaque variable un schéma de type (de la forme $\forall\alpha_1,\ldots,\alpha_n.\tau$) et les règles de typage des variables et du `let` deviennent :

$$\frac{\tau \in \texttt{inst}(\Gamma(x))}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall\bar{\alpha}.\tau_1 \vdash e_2 : \tau_2 \quad \bar{\alpha} = \texttt{vars}(\tau_1) \setminus \texttt{fv}(\Gamma)}{\Gamma \vdash \texttt{let } x{=}e_1 \texttt{ in } e_2 : \tau_2}$$

où `vars` est l'ensemble des variables apparaissant dans un type, `fv` est l'union des ensembles des variables libres des schémas de type présents dans un environnement et où $\texttt{inst}(\forall\alpha_1,\ldots,\alpha_n.\tau)$ est l'ensemble des types obtenus en substituant dans τ les variables α_i par des types quelconques.

9. Définir un type `schema` qui représente un schéma de type.
10. Écrire une fonction `specialize` qui à partir d'un schéma de type calcule un nouveau type où toutes les variables de type quantifiées universellement sont remplacées par des variables fraîches.
11. Écrire une fonction `generalize` qui à partir d'un environnement de typage et d'un type va calculer le schéma de type correspondant.
12. Définir la fonction `subst_sc` qui applique une substitution à un schéma de type.
13. Écrire un typeur pour Pti-PtiCaml avec typage polymorphe.

1.3 Typage avec niveaux

Avec le système de type précédent, l'opération de généralisation peut coûter cher. En effet, il faut parcourir tous les types de l'environnement pour déterminer les variables que l'on peut généraliser. Nous allons étudier ici un système de type où il suffit de parcourir le type que l'on souhaite généraliser.

Chaque variable de type est associée au niveau auquel elle a été créée (α^n). Les jugements de typage sont maintenant établis à un niveau courant. Ainsi, ils ont la forme suivante : $\Gamma \vdash_n e : \tau$.

Un type τ est dit de niveau n et noté $\tau \in \tau^n$ s'il ne comporte que des variables créées à un niveau au plus égal à n.

On note $\mathcal{V}^{\geq n}(\tau)$ l'ensemble des variables de type présentes dans τ dont le niveau est supérieur ou égal à n.

Les règles de typage des variables, des fonctions et du `let` sont les suivantes dans ce système :

$$\frac{\tau \in \texttt{inst}(n, \Gamma(x))}{\Gamma \vdash_n x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_n e : \tau_2 \quad \tau_1 \in \tau^n}{\Gamma \vdash_n \texttt{function } x \to e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash_{n+1} e_1 : \tau_1 \quad \Gamma, x : \forall\bar{\alpha}.\tau_1 \vdash_n e_2 : \tau_2 \quad \bar{\alpha} = \mathcal{V}^{\geq n+1}(\tau_1)}{\Gamma \vdash_n \texttt{let } x{=}e_1 \texttt{ in } e_2 : \tau_2}$$

où $\texttt{inst}(n, \forall\alpha_1,\ldots,\alpha_n.\tau)$ est l'ensemble des types obtenus en substituant dans τ les variables α_i par des types quelconques de niveau au plus n.

14. Redéfinir l'arbre de syntaxe abstraite des types.
15. Redéfinir la fonction `unify` de telle sorte que lors de l'unification d'une variable α^n avec un type τ, toutes les variables de τ soient mises à un niveau au plus n.
16. Redéfinir les fonctions `specialize` et `generalize`.
17. Écrire un typeur pour Pti-PtiCaml avec typage polymorphe avec niveaux.