TP 10: Inférence de type de MINIML

Ioana Pasca, Marc Lasson

1 Description des fichiers de l'archive

- `infer.ml` : C'est le seul fichier que vous aurez à compléter. Vous allez y implémenter l'inférence de type de MINIML en vous aidant du moteur d'unification implémenté au TP précédent.
- `term.ml`, `lexer.mll`, `parser.mly`, `conv.ml` : Ces fichiers sont tirés tout droit du TP8, on rappelle qu'ils implémentent les termes de MINIML et leur conversion en chaîne de caractères.
- `eval.ml` : C'est une correction de l'évaluateur du TP8 pour ceux d'entre vous qui ne l'auraient pas terminé. Si ça vous fait plaisir, vous êtes autorisé à mettre votre propre correction à la place.
- `sharing.ml` : C'est une version simplifiée et corrigée du fichier implémentant les expressions avec partage du TP9. Dans ce fichier, vous allez surtout utiliser la fonction `unify` dont on rappelle qu'elle est de type `Sharing.t -> Sharing.t -> unit`.
- `type.ml` : Ce module contient des fonctions pour simplifier la construction des expressions de types (qui seront décrites ci-dessous). Elle contient également une fonction de conversion des types en chaîne de caractère.
- `miniml.ml` : C'est le point d'entrée du programme il lance l'inférence de type et l'évaluateur sur une batterie d'exemples. Vous êtes encouragés à y rajouter vos propres exemples pour tester vos fonctions.

2 Représentation des types

Dans ce tp, les types seront générés par la grammaire suivante

$$\sigma, \tau \quad := \quad \alpha \quad | \quad \texttt{unit} \quad | \quad \texttt{bool} \quad | \quad \texttt{int} \quad | \quad \sigma \to \tau \quad | \quad \sigma \times \tau \quad | \quad \tau\,\texttt{ref}$$

et ils seront représentés par des valeurs de type `Sharing.t`. Il y aura donc six opérateurs (`Op`) différents. Trois opérateurs «zeroaires» représentés par les symboles `"unit"`, `"bool"` et `"int"`. Deux opérateurs binaires représentés par les symboles `"->"` et `"*"`. Et un opérateur unaire représenté par le symbole `"ref"`.

Le fichier `Type.ml` contient toutes les fonctions que l'on utilisera pour construire les types :

```
let new_var () = ref (Repr Var)
let type_unit = ref (Repr (Op ("unit",[])))
let type_bool = ref (Repr (Op ("bool",[])))
```

```
let type_int = ref (Repr (Op ("int",[])))
let build_arrow t1 t2 = ref (Repr (Op ("->",[t1;t2])))
let build_product t1 t2 = ref (Repr (Op ("*",[t1;t2])))
let build_ref t = ref (Repr (Op ("ref",[t])))
```

Le fichier contient également une fonction `string_of_type` de type `Sharing.t -> string` qui fabrique une chaîne de caractère représentant le type. Elle choisit elle-même le nom qu'elle va donner aux variables (contrairement à la fonction du TP précédent qui prenait en argument une liste d'association qui indiquait le nom des variables).

3 Inférence de type

C'est à vous de jouer maintenant. Vous devez compléter le fonction du fichier `infer.ml`. Elle a le type suivant.

```
infer : (var * Sharing.t) list -> Term.t -> Sharing.t
```

Le premier argument représente le type des environnements (c'est une liste d'association qui associe un type à chaque variable), le second est le terme à typer et la fonction retourne le type.

À titre d'exemple, nous vous donnons la solution pour le typage des variables et du `let`.

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

Solution :

```
| TVar x ->
      (try List.assoc x env with Not_found ->
        failwith (Printf.sprintf "infer: unbound variable '%s'" x))
```

$$\frac{\Gamma \vdash t_1 : \sigma \qquad \Gamma, x : \sigma \vdash t_2 : \tau}{\Gamma \vdash \texttt{let}\ x\ =\ t_1\ \texttt{in}\ t_2 : \tau}$$

Solution :

```
| TLet (x, t1, t2) ->
      let tau1 = infer env t1 in
      let env = (x, tau1)::env in
      infer env t2
```

Pour toutes les questions de cette section, vous devez compiler, tester et vérifier entre chaque question !

Question 1. Voici les règles de typage des constantes :

$$\frac{}{\vdash k : \texttt{int}} \qquad \frac{}{\vdash () : \texttt{unit}} \qquad \frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

Implémentez les cas `TCst (VInt _)`,`TCst (VBool _)`, `TCst VUnit`.

Question 2. Voici la règle de typage pour les opérations arithmétiques :

$$\frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 * t_2 : \texttt{int}}$$

Implémentez les cas `TAdd`, `TSub`, `TTMul`. Indice : il faut inférer les types des t_i et lancer la procédure d'unification (`unify`) avec `int` pour vérifier que les types inférés sont corrects.

Question 3. Voici les règles de opérations de comparaison

$$\frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 = t_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 < t_2 : \texttt{bool}}$$

Implémentez ces règles (cas `TEq`, `Lt`).

Question 4. Implémentez le cas `TIf`. N'oubliez pas que les deux cas du `if` doivent avoir le même type.

$$\frac{\Gamma \vdash t_1 : \texttt{bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : \tau}$$

Question 5. Implémentez les cas `TLetTuple` et `TPair`.

$$\frac{\Gamma \vdash t_1 : \sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (t_1, t_2) : \sigma \times \tau}$$

$$\frac{\Gamma \vdash t_1 : \sigma_1 \times \sigma_2 \qquad \Gamma, x : \sigma_1, y : \sigma_2 \vdash t_2 : \tau}{\Gamma \vdash \texttt{let}\ x, y\ =\ t_1\ \texttt{in}\, t_2 : \tau}$$

Indice : Pour le cas du `let`, vous n'avez pas à faire de «matching» vous-même afin de vérifier que le type inféré de t_1 est bien un type de la forme $\cdot \times \cdot$. Nous vous conseillons plutôt de créer deux nouvelles variables de types α et β puis de vérifier que le type de t_1 s'unifie bien avec $\alpha \times \beta$. Ensuite vous pouvez utilisez α et β pour inférer le type de t_2.

Question 6. Vous avez maintenant toutes les armes pour faire les cas `TFun` et `TApp`.

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \texttt{fun}\, x \to t : \sigma \to \tau} \qquad \frac{\Gamma \vdash s : \sigma \to \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash (s\, t) : \tau}$$

Question 7. Implémentez le cas `TFix`.

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash t : \tau}{\Gamma \vdash \texttt{fix}\ f\, x \to t : \sigma \to \tau}$$

Question 8. Implémentez le cas `TSeq`.

$$\frac{\Gamma \vdash t_1 : \sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1; t_2 : \tau}$$

Question 9. Implémentez les cas `TRef`, `TDeref`, `TAss`.

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \texttt{ref}\ t : \tau\ \texttt{ref}} \qquad \frac{\Gamma \vdash t : \texttt{ref}\,\tau}{\Gamma \vdash !t : \tau} \qquad \frac{\Gamma \vdash t_1 : \texttt{ref}\,\tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 := t_2 : \texttt{unit}}$$

Question 10. Rajoutez l'exemple suivant :

```
let delta x = x x in delta
```

Expliquez ce qu'il se passe. De quel endroit du code vient le problème ? Comment y remédier ? Essayez dans `Caml` ! Et avec l'option `-rectypes` ?

Question 11. Rajoutez l'exemple suivant :

```
let id x = x in
id 3, id true
```

Expliquez ce qu'il se passe. Est-ce que `Caml` se comporte de la même façon ?

Question 12 (Polymorphisme à la ML, difficile). Implémentez une solution au problème de la question précédente. Conseils :

- Vos environnements n'associent plus à une variable un type mais un «schéma de type» c'est-à-dire un couple (l, τ) où l est la liste des variables généralisables de τ.
- La liste des variables généralisables s'obtient en récupérant les variables libres inférés au niveau des cas `TLet` et `TLetTuple`. Elle est vide dans le cas des fonctions [1].
- Il reste à modifier le cas variable qui doit «instancier» les schémas de type. Cela revient à copier (à réallouer avec `ref`) toutes les variables du type qui apparaissent dans la liste du schéma. Et le tour est joué.

1. Si on autorisait la généralisation au niveau des fonctions (comme c'est le cas dans coq), alors l'inférence de type serait indécidable.