Programmation 1 – TP n° 10
InfÃl'rence de types

Au dernier TP, on a vu l'unification du premier ordre, avec les ensembles de Tarjan, etc... Cet algorithme est utilisé, entre autres pour l'inférence de types dans ML (à la Hindley-Milner). C'est ce qu'on va faire dans ce TP.

1 Introduction

La syntaxe et les règles de typage de miniML sont les suivantes :

$$\begin{array}{rl} \text{termes :} & u, v ::= x \mid u\ v \mid \lambda x.u \mid \texttt{let}\ x = u\ \texttt{in}\ v \mid \texttt{ref}\ u \mid !u \mid () \mid \mathbf{n} \\ \text{types :} & \sigma, \tau ::= \alpha \mid \sigma \to \tau \mid \tau\ \texttt{ref} \mid \texttt{unit} \mid \texttt{int} \\ \text{environnements :} & \Gamma ::= \emptyset \mid \Gamma, x : \tau \end{array}$$

$$\frac{\Gamma, x : \sigma \vdash u : \tau}{\Gamma \vdash \lambda x.u : \sigma \to \tau} \qquad \frac{\Gamma \vdash u : \sigma \to \tau \quad \Gamma \vdash v : \sigma}{\Gamma \vdash u\ v : \tau} \qquad \frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \sigma \vdash v : \tau}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash u : \tau}{\Gamma \vdash \texttt{ref}\ u : \tau\ \texttt{ref}} \qquad \frac{\Gamma \vdash u : \tau\ \texttt{ref}}{\Gamma \vdash !u : \tau} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}} \qquad \frac{}{\Gamma \vdash \mathbf{n} : \texttt{int}}$$

Dans ce système, les variables de types (α) ne sont pas mentionnées explicitement, et un terme comme l'identité ($\lambda x.x$) peut être typé de plusieurs façons : $\texttt{int} \to \texttt{int}$, $(\texttt{int} \to \alpha) \to (\texttt{int} \to \alpha)$... Cependant, tout terme u typable admet un unique *(schéma de) type principal* τ tel que tout autre type possible de u en soit une instance. L'objectif de l'inférence de type est de trouver ce type principal, quand il existe.

Q 1.1 Au flair, où voit-on qu'il va falloir unifier des choses (au sens du dernier TP), et que faudra-t-il unifier ?

Récupérez l'archive contenant l'interprète miniML sur la page des TPs. Elle est structurée comme suit.

Common :	type des expressions (AST),
Parser, Lexer :	analyses syntaxique et lexicale (TP 4),
Eval :	interprète (TPs 1 et 2),
Tipe :	inférence de types,
Main :	programme principal,
Makefile :	pour compiler le tout,
tests.ml :	des exemples à essayer.

Q 1.2 Survolez très rapidement ces fichiers pour vous convaincre qu'ils définissent bien un interprète de miniML.

Dans la suite, le seul fichier à modifier sera `tipe.ml`.

2 Type des types

Le point central de la solution qu'on propose ici est le type de données utilisé pour représenter les types de `miniml`. Il vous est donné :

```
type descr =
  | NoLink of constr
  | Link of typ
and typ = descr ref
and constr =
  | TFun of typ * typ
  | TRef of typ
  | TInt | TString | TUnit
  | Var
```

Les arbres de Tarjan sont intégrés à la représentation : un type (`typ`) est une référence pointant, soit vers un autre type, soit vers un constructeur de type (la partie *informative*). Comme on utilise une référence, on pourra faire de la compression de chemin.

On reconnaît dans les constructeurs les types standards de ML (flèche, référence, types de base) et les variables. Notez que l'on n'utilise pas de chaînes de caractères pour représenter ces variables de type ! En fait, les types seront comparés par égalité physique, et deux variables de type distinctes seront représentées par deux références distinctes (physiquement) vers le même constructeur `Var`.

Ne vous braquez pas trop sur ce type un peu tordu : vous le comprendrez peu à peu en l'utilisant.

Q 2.1 Pour vous familiariser, écrivez les cinq fonctions `tfun`, `tref`, `tint`, `tunit` et `tsring` dont les types sont donnés dans le fichier, et qui permettent de *fabriquer* simplement les représentations des type `miniml` correspondants.

Q 2.2 Quel doit être le type de `tvar`, qui permet d'obtenir la représentation d'une variable de type ?

Q 2.3 Écrivez `tvar`.

Il faut maintenant pouvoir accéder au représentant d'un type (au sens de Tarjan).

Q 2.4 Écrivez la fonction `repr: typ -> typ`, qui renvoie le représentant d'un type, en faisant la compression de chemin de Tarjan.

La fonction `find: typ -> typ*constr` qui suit permet alors de récupérer le constructeur correspondant à un type.

3 Unification

L'avantage de cette représentation est qu'elle permet d'unifier les types *en place* : lors de l'unification de $t_1 = \alpha \to$ `int` et $t_2 =$ (`int` $\to$ `unit`) $\to$ `int`, on va remplacer *physiquement* le nœud représentant α dans t_1 par `int` $\to$ `unit`.

De manière plus générale, dans un contexte où une variable α est utilisée à différents endroits, lorsque l'on va trouver une équation $\alpha = t$, on va modifier la référence qui représente α afin de la faire pointer sur t. Ainsi, la substitution prendra effet à tous les points où est utilisée la variable.

Q 3.1 Complétez la fonction `unify: typ -> typ -> unit` qui effectue l'unification en place de deux types. En cas de types non unifiables, vous lèverez l'exception `TypeError`.

4 Hindley Milner

Pour représenter les types polymorphes, il nous faut parler de *schémas de type*, i.e. un type et la liste de ses variables universellement quantifiées. Par exemple, pour l'identité, de type $\forall\alpha,\ \alpha \to \alpha$, on aura informellement « $[\alpha],\ \alpha \to \alpha$ ».

Q 4.1 Écrivez la représentation `t_id: scheme` du type de l'identité.

La fonction `instantiate: scheme -> typ` instancie les variables de type liées dans un schéma de type afin d'obtenir un type *simple* où toutes les variables de types précédemment liées sont fraîches. La fonction `generalize: typ -> scheme` fait exactement l'inverse : elle *ferme* un type par sa liste de variables de types. Remarquez que dans ces deux fonctions, il faut absolument utiliser `List.assq`, qui compare les éléments par égalité physique : toutes les variables sont égales structurellement.

On peut désormais écrire l'algorithme d'inférence des types. Comme d'habitude, on utilise un environnement de typage, qui nous permet d'associer un schéma de type à toute variable ($x, f, \ldots$) introduite par un `fun` ou un `let`.

Q 4.2 Complétez la fonction `infer: env -> expr -> scheme`, qui effectue l'inférence de types. Le cas de la variable est donné : on récupère son type dans l'environnement et on l'instancie.

Q 4.3 Vous pouvez alors compiler le projet avec la commande `make` (dans emacs, « Ctrl-c Ctrl-c », ou encore « M-x compile »), ce qui devrait créer un exécutable nommé `miniml`. Lancez-le dans un terminal (« `./miniml` »), il doit réagir plus ou moins comme ocaml. Ensuite vous pouvez ouvrir le fichier de tests (`tests.ml`) et l'évaluer dans votre interprète : fermez l'interprète ocaml courant s'il y en a un, puis redémarrez-le depuis tests.ml (« Ctrl-c Ctrl-e »), en spécifiant « `./miniml` » au lieu de « `ocaml` ».

4.1 Let rec

Q 4.4 L'évaluateur a été modifié pour accepter les définitions récursives (`let rec`, constructeur `ELetRec`). Proposez une règle de typage du `let rec`, puis étendez l'inférence de types pour qu'elle traite ce cas.

4.2 Occurs check

Il manque à ce niveau la règle d'élimination des occurrences récursives, qui peuvent apparaître lorsque l'on unifie par exemple α et $\alpha \to \alpha$.

Q 4.5 Observez ce qu'il se passe lors de cette unification, faites un dessin de la représentation du type résultant de cette unification.

Q 4.6 Écrivez le test d'occurrence récursive.

Q 4.7 Le test est utilisé à la fin de l'inférence de type, mais il faut aussi l'utiliser à chaque fois que l'on rencontre un `let`. Rajoutez cet appel.

4.3 Généralisation

Pour l'instant, on ne généralise les variables d'un type qu'à la fin du typage (pour l'expression globale, donnée à toplevel). C'est insuffisant, par exemple pour typer :

```
let k =
  let id z = z in
    id 5; id "hop"; 3;;
```

Mais c'est aussi faux, par exemple avec :

```
let y = ref (fun x -> x);;
y := (fun x -> x+1);;
y;;
let z = !y "chtok"
```

En fait, il faut faire trois choses :
- généraliser dès qu'on introduit une variable dans l'environnement via un `let` ;
- restreindre la généralisation aux termes qui ne sont pas une application (il faut virer aussi ceux de la forme « `let ... in` *application* ») ;
- annoter chaque variable de type d'un *niveau* correspondant au nombre de `let` rencontrés depuis le toplevel (profondeur) et ne généraliser que les variables dont le niveau est supérieur ou égal au niveau courant.

Q 4.8 Définissez la fonction `generalizable`.

Q 4.9 Modifiez le cas du `let` dans la fonction d'inférence afin qu'elle généralise le type attribué à `x` lorsque le corps de la définition (`a`) autorise la généralisation.

Q 4.10 Introduisez les niveaux.
- Au tout début du fichier, la ligne « `| Var` » devient « `| Var of int` ».
- La fonction `tvar` doit être modifiée.
- La fonction d'unification doit désormais faire attention au cas `Var i,Var j` : elle doit conserver le niveau le plus élevé.
- La fonction interne d'inférence prend un argument supplémentaire : ce niveau, et l'incrémente à chaque passage sous un `let`.
- La fonction d'instanciation prend en argument supplémentaire le niveau auquel instancier les variables.
- La fonction de généralisation prend aussi un argument supplémentaire, indiquant à partir de quel niveau généraliser (`generalize l t` quantifie toutes les variables de `t` de niveau supérieur ou égal à `l`).