Compilation - TP5 - Analyse sémantique de *mini-Ocaml*

année universitaire 2013-2014

1 Introduction

L'interprète *mini-Ocaml* du TP4 n'est pas typé. Cela se manifeste pas le fait que l'évaluation de certains termes peut échouer. Nous allons effectuer les analyses sémantiques nécessaires pour détecter ces erreurs statiquement. Dans une mise en oeuvre réelle, ces analyses sémantiques sont mises à profit pour éviter des tests inutiles qui ne peuvent pas échoués.

Un squelette vous est fourni (`miniocamltp5.tgz` sous commun). Il fournit en particulier une mise en mise d'ensembles de chaînes `strSet.mli` et une interface pour manipuler des environnements de types `typEnv.mli`.

2 Variables libres

Pour obtenir la valeur d'une variable, l'interprète la recherche dans l'environnement. Pour assurer que la variable existe, il est nécessaire de s'assurer de l'absence de variables libres.

Question 2.1 *Modifier l'analyseur syntaxique de façon à calculer l'ensemble des variables libres d'un terme.*

Ce calcul devra être fait à l'aide d'un attribut synthétisé. Par conséquent, l'analyseur syntaxique retournera une paire `Ast.Ml_expr * StrSet.t`. Dans le cas où la liste de variables libres n'est pas vide, l'interprète doit retourner une erreur de la forme `Unbound variables .....`.

3 Typage

En général, l'inférence de type simple nécessite de collecter et résoudre des contraintes d'unification. Pour simplifier l'inférence, on va annoter le programme avec des types en des endroits stratégiques :

— Chaque occurrence de la liste vide `[]` doit être annotée par un type explicite ;

— Dans un motif, chaque occurrence d'une variable doit aussi être annotée par un type explicite ;

Par exemple, on écrira donc :

```
let x = 1 in function (y:int) -> x + y

(function ([]: int list)              -> 0
        | (hd:int) :: (tl:int list) -> hd + 1)
            (2 :: 3 :: 4 :: ([]:int list))
```

Question 3.1 *Modifier la grammaire pour pouvoir annoter les termes avec des types.*

On rappelle les priorités suivantes :

— l'opérateur `->` est associatif à droite ;

— les opérateurs `list` et `*` sont prioritaires par rapport à `->` ;

— l'opérateur `*` est prioritaire par rapport à `list`.

Par exemple : `int -> int -> int` se lit `int -> (int -> int)` ; `int -> int list` se lit `int -> (int list)` et `int * int list` se lit `(int * int) list`.

Question 3.2 *Compléter les fonctions de vérification de type (*`typing.ml`*).*

. La fonction `typ_of_pattern : ml_pattern -> (string * typ) * typ` prend un motif en entrée et retourne une paire contenant un environnement de typage et le type du motif.

Par exemple, le type du motif `((x:int),(y:int -> bool)) :: (tl:(int * (int -> bool)) list)` est `(int * (int -> bool)) list` et l'environnement est de la forme `[(x:int); (y:int -> bool) ; (tl : (int * (int -> bool)) list)]`.

La fonction `wt_expr : TypEnv.t -> Ast.ml_expr -> Ast.typ` retourne le type d'une expression étant donné un environnement de type pour les variables libres de l'expression. En cas d'erreur de type, la fonction retourne une simple exception `failwith "Type error"`.

Question 3.3 *Optionnel : Améliorer les messages d'erreur de façon à localiser et expliquer au mieux l'origine de l'erreur.*