Compilation - TD7

Piles de langages impératifs et arbres de langages fonctionnels

1 Pour se réveiller, regardons fonctionner les compilateurs

1.1 Enregistrements d'activation

Voici un bout de code (disponible pour les paresseux sur `www.ens-lyon.fr/~fdedinec/enseignement/compil/TD7/exemples.txt` mais c'est plutôt plus long à taper).

```
int main()
{
  int i;
  for(i=0; i<10; i++)
    {
      int j;
      int k  ;
      k=i*j;
    }
}
```

Question 1-1 Passez-le à `gcc -S`, qui compile en code assembleur, et observez le fichier `.s` résultant. Montrez du doigt les variables `i`, `j`, `k`. Que contiennent les registres `%ebp` et `%esp` (sur les PC), respectivement `%fp` et `%sp` (sur les Sun) ?

Ajoutez après la ligne `k=i*j ;` le code suivant :

```
{
  int i;
  k=i+k;
}
```

Question 1-2 Que sont devenus les deux `i` ?

1.2 Avec des appels de fonction

Ici il est conseillé de compiler sur PC. Étudiez le code assembleur produit par le programme suivant.

```
int fonction (int x, int y)
{
  int z=3;
  return y*x+z;
}

int main()
{
  int i,j;
  i=42; j=43;
  j=fonction(i,j);
}
```

```
int fact (int x)
{
  int y = x-1;
  if (y==0)
    return 1;
  else
    return fact(y);
}
```

Question 1-3 Répondez aux questions suivantes :

- Où est placée la valeur de retour d'une fonction ?
- Où sont placés les paramètres de la fonction avant l'appel ? Où la fonction les récupère-t-elle ?
- Que fait l'instruction *call* au juste ?
- Que fait l'instruction *ret* ?

Simulez l'évolution de la pile lors de l'appel `fact(3)`.

2 Unification de type

La vérification de type dans les langages impératifs est relativement facile, elle se fait par des parcours des arbres d'expressions en remontant des feuilles (dont le type est connu) à la racine. Les opérations qui peuvent être insérées sont des conversions de type (d'entier en flottant, par exemple) et des déréférencements (de *lvalue* à *rvalue*).

Dans les langages fonctionnels purs les choses sont moins simples. La difficulté supplémentaire tient dans les types polymorphes des fonctions comme `map`.

On va travailler sur un langage fonctionnel simplifié défini avec les mains comme suit : un programme est une expression, et une expression peut utiliser, outre les habituelles opérations arithmétiques et logiques, les constructions suivantes :

- construction "`let` id *ids* `=` `expr` in `expr end`", où *ids* désigne une liste éventuellement vide d'identificateurs, pour définir des fonctions ou des variables ;
- application d'une fonction à ses arguments, de la forme "`(` expr *exprs* `)`", où *exprs* est une liste non vide d'expressions.

N.B. : on a une syntaxe un peu contraignante (`let` terminés par des `end`, application entourée de parenthèses) pour ne pas trop compliquer la définition de la grammaire.

Voici le type des programmes pour les AST de ce langage, fourni dans `types.ml` :

```
type identifier = string ;;

type operator = Plus | Moins | Fois | Divise ;;

type expression =
    Op of operator * expression * expression
  | Con of int
  | Var of identifier
  | Let of identifier * expression * expression
  | Fun of identifier * expression (* lambda abstraction *)
  | App of expression * expression
;;
```

Question 2-1 Demandez-vous comment les fonctions de plusieurs variables sont traduites en AST. Donnez les AST de quelques expressions compliquées.

Exercice à la maison : écrivez en `ocamlyacc/ocamllex` un parser pour ce langage, fournissant un AST du type `expr`. Dans ce TD nous travaillerons sur l'AST.

2.1 Jolie écriture

Écrivez un *pretty-printer* (le contraire d'un analyseur syntaxique) pour cet AST.

2.2 Analyse de la portée des variables

Question 2-2 Écrivez une fonction de parcours de l'arbre de syntaxe abstraite qui vérifie que toutes les variables utilisées sont dans la portée d'une déclaration faite à l'aide d'un `let` ou dans une définition de fonction[1]. Si cette condition n'est pas vérifiée, le programme affiche un message d'erreur sensé, sinon il dit que tout va bien.

Testez votre programme avec les expressions suivantes (dont vous construirez les AST vous-même pour vous motiver à écrire l'analyseur syntaxique pour la prochaine fois) :

```
let a = 3 in let b = 4 in a+b end end;;
let f x = x+3 in let f = 3 in f+2 end end;;
let f x = x+3 in let x y = y in (x f 51) end end;;
```

2.3 Inférence de types monomorphes

Étant donné un objet de type `expr`, on veut être capable d'en donner le type, dans un premier temps sans utiliser le polymorphisme.

Rappelons le principe des différentes étapes de l'inférence :

1. on commence par décorer le terme avec des variables de type qui sont des inconnues :
   – une variable de type par variable de terme ;
   – une variable de type par sous-expression de l'expression à typer.
   Dans le code qui vous est fourni, les types sont décrits par le type suivant :
   `type typ = INT | ARR of typ*typ | VAR of id`
   Vous disposez en outre de deux fonctions, `mk_id_var` et `new_var`, servant respectivement à engendrer une variable de type à partir d'un identificateur (variable de terme) et à engendrer une variable de type nouvelle.
2. on engendre ensuite un ensemble d'équations entre expressions de type, en descendant dans l'arbre de terme. Par exemple si le terme est de la forme `Pl(e,e')`, on peut noter que les variables de type correspondant au terme lui-même et aux deux sous-termes `e` et `e'` doivent être égales à `INT` (on ne s'occupe plus de surcharge comme lors du TD précédent).
3. on résoud les équations en faisant appel à une procédure d'unification du premier ordre. Le code correspondant est contenu dans le fichier `unif.ml` : la fonction `unify` prend en argument un problème d'unification (liste de couples de types), et renvoie une substitution (liste de couples (identificater,type)). L'expression `apply_one_subst x s` peut être utilisée pour appliquer la substitution `s` à l'expression de type `x`.

Question 2-3 Écrivez le code correspondant aux étapes 1 et 2 ci-dessus. On implémentera une fonction prenant en argument un programme sous forme d'AST, et renvoyant une liste de couples de type `typ * typ`, représentant les contraintes de types que l'on lit sur le terme.

On supposera pour simplifier le traitement que toutes les variables d'un terme à typer ont des noms distincts deux à deux (i.e. pas de terme de la forme `let x = 3 in let x = fun y -> y in ...`).

Question 2-4 Rassemblez les morceaux pour inférer le type d'un programme. Testez le tout sur les exemples ci-dessous :

```
let x = 3+2 in x*5 end;;
fun x -> fun y -> y end end;;
let f x = x in f end;;
let f x = x in (f f) end;;
```

[1] Certaines analyses que l'on implantera aujourd'hui pourraient être faites durant la phase d'analyse syntaxique. On choisit cependant de travailler sur l'arbre de syntaxe abstraite, pour accroître la clarté de la présentation.

```
(x x);;
let f x = x in (x 3)+x end;;
```

Comment se comporte votre procédure de typage vis-à-vis d'un terme comportant des variables libres (par exemple dans `x+2;;`)? Vous pouvez soit choisir de tolérer de tels termes, en décrétant que les variables libres sont soumises à des contraintes exprimées lors de la phase de typage, soit coupler l'inférence de type avec l'analyse statique définie à la question 1 pour ne typer que des termes *clos*.