Cours de Compilation-Exercices

Analyse sémantique

Master d'Informatique M1 2010–2011

11 octobre 2010

4 Analyse sémantique élémentaire

4.1 Arbre de syntaxe abstraite-Portée des identificateurs-Typage

Soit un langage de programmation dans lequel un programme est constitué d'une suite de déclarations de variables et de fonctions.

Le but de cet exercice est d'explorer des représentations possibles pour les arbres de syntaxe abstraite de ce langage dans le but d'implanter des fonctions de vérification de la bonne formation des programmes.

Types Les variables, paramètres ou résultat de fonctions ont pour type soit `int` le type des entiers, soit `bool` le type des valeurs booléennes.

Déclarations Une *déclaration de variable* s'écrit : **let** $id = exp$ avec id le nom de la variable et exp la valeur initiale.

Une *déclaration de fonction* s'écrit : **let** $id\ (id_1 : type_1, \ldots, id_n : type_n) : type = exp$
avec id le nom de la fonction, id_i le nom du i-ème paramètre, $type_i$ le type du paramètre et $type$ le type du résultat de la fonction. Une fonction peut être récursive.

Expressions Une *expression* peut être :

- une variable, déclarée globalement ou localement ou bien comme paramètre d'une fonction.
- une valeur entière ou booléenne (`true` ou `false`).
- une expression exp_1 `op` exp_2 formée d'un opérateur binaire prédéfini `op` appliqué à deux expressions exp_1 et exp_2 avec `op` $\in \{<, >, \leq, \geq, +, *, /, -, =\}$
- un appel de fonction $id(exp_1, \ldots, exp_n)$
- une expression conditionnelle :
  if exp **then** exp_1 **else** exp_2.
- une expression avec déclaration locale **let** $id = exp_1$ **in** exp_2

Un *programme* est une suite de déclarations de variables et de fonctions séparés par des point-virgules.

Portée Les *règles de portée* définissent la durée de vie des identificateurs introduits dans les déclarations. Elles permettent de relier chaque utilisation d'un identificateur apparaissant dans le corps du programme à la déclaration dont il dépend.

Les règles de portée de notre langage sont les suivantes :

- dans une déclaration globale de valeur **let** $id = exp$ la portée de id est l'ensemble des déclarations qui suivent dans le programme, en particulier id n'est pas visible dans exp.

- dans une déclaration globale de fonction
  let id $(id_1 : type_1, \ldots, id_n : type_n) : type = exp$
  la portée des paramètres formels id_i est limitée à l'expression exp; la portée de id est l'ensemble des déclarations de fonctions dans le programme; id est en particulier visible dans exp;
- dans une déclaration locale **let** $id = exp_1$ **in** exp_2, la portée de l'identificateur id introduit est limitée à exp_2
- un paramètre ou une variable peut être caché par la redéfinition d'un paramètre ou d'une variable de même nom dans un bloc plus profond.

Questions

1. Dire pour chacune des déclarations du programme suivant si elles respectent les règles de portée et si oui à quelle déclaration correspond chaque utilisation d'un identificateur.

```
let x = 1
let y = x+1
let f(y:int,z:int):int= x + y + let y = y+3 in g(z+y)
let g(x:int,y:int):int = f(x-1,y-1)
let h(x) = let z = x+z in f(y)
```

2. Pour représenter les arbres de syntaxe abstraite de ce langage, on introduit les types CAML suivants :

```
type ident = string
type asa_typ = TypInt | TypBool
type cte = Int of int | Bool of bool
type op = Plus | Mult | Div | Minus | Leq | Geq | Lt | Gt | Eq
```

   Compléter cette définition par les définitions des types `asa_decl`, `asa_exp` et `asa_prog` permettant de coder les arbres de syntaxe abstraite des déclarations, des expressions et des programmes.

3. Lorsqu'une erreur de portée ou de typage se produit, il est nécessaire de produire un message d'erreur significatif. Pour cela on doit associer à un arbre de syntaxe abstraite, la suite de caractères correspondante dans le fichier initial.

   Pour construire la nouvelle structure d'arbres localisés en CAML, on peut procéder de la manière suivante :
   - Le type CAML `Lexing.position` permet de conserver les informations relatives à une position dans un fichier. On définit un type `location` qui représente une localisation par deux positions (début et fin).
   - On introduit un type polymorphe `'a loc` permettant d'ajouter une localisation à n'importe quelle valeur de type `'a`;
   - On introduit pour chaque type d'arbre de syntaxe abstraite `asa`, deux types distincts, l'un `asa_raw` qui contient un objet non-localisé (mais dont les sous expressions sont localisées) et un `asa` qui est défini comme `asa_raw loc`

   La structure CAML sera donc :

```
(* position du premier et dernier caractere de l'expression *)
type location = Lexing.position *  Lexing.position
(* Un type generique pour annoter un asa par une position *)
type 'a loc = {loc : location;asa : 'a}

type asa = asa_raw loc
and asa_raw = C1 of ... | C2 of ...
```

   (a) Que faut-il changer aux définitions de la question précédente pour introduire les localisations ?

(b) On suppose que la grammaire comporte les règles suivantes :

```
decl:
    LET ID EQ expr              {}
expr:
   expr OP expr                 {}
```

Compléter les actions pour produire des arbres localisés. On pourra utiliser dans les actions de la grammaire les fonctions:
`Parsing.symbol_start_pos` et `Parsing.symbol_end_pos`
de type `unit -> Lexing.position` qui donnent les positions initiales et finales du symbole non-terminal à gauche de la production.

(c) On introduit une exception pour les erreurs de typage qui prend en argument une chaîne de caractères et une localisation.

```
exception TypeError of string * location
```

On suppose qu'une liste d'identificateurs `primitifs` est donnée qui correspondent à des fonctions primitives et qui ne doivent pas être redéfinies. Écrire une fonction CAML qui prend en argument un programme, vérifie que les déclarations globales respectent la condition et renvoie un message d'erreur aproprié si ce n'est pas le cas.

4. On se propose de décrire les règles de typage pour le langage. Pour cela on introduira un environnement qui associe aux noms de variables déclarées un type et aux noms de fonctions un profil. Un profil est de la forme $[\tau_1, \ldots, \tau_n] \to \tau$, avec τ_i le type des arguments et τ le type du résultat. Un environnement s'écrit $id_1 : \pi_1, \ldots, id_n : \pi_n$ où les π_i sont soit des types soit des profils. On écrira $f : [\texttt{int};\texttt{int}] \to \texttt{int}, x : \texttt{bool}$ un environnement qui définit une fonction f qui attend deux arguments de type entier et renvoie un entier ainsi qu'une variable x de type booléen.

   (a) Identifier pour les différentes classes syntaxiques (expressions, déclarations, programme) la forme du jugement qui dit qu'ils sont correctement formés.

   (b) Donner les règles de typage pour les programmes, les déclarations et les expressions.

5. On souhaite vérifier si un programme satisfait les règles de typage. Pour cela on doit utiliser localement une table de visibilité correspondant à l'environnement qui comporte tous les identificateurs visibles en un point du programme avec pour chacun son type ou son profil (types des arguments et du résultat) dans le cas d'une fonction. On se contentera pour cette question de spécifier les opérations nécessaires sur cette structure de données.

   (a) Ecrire des fonctions qui vérifient simplement les règles de typage.

   (b) Pour préparer les étapes de génération de code, on souhaite de plus transformer l'arbre de syntaxe abstraite pour mettre des identifiants uniques. De plus, on veut construire une table des symboles qui conserve pour chaque identifiant unique, le type ou le profil associé.
   Reprendre les programmes précédents pour procéder à ces transformations. Commencer par indiquer les changements dans les types des fonctions de vérification et de manipulation de la table de visibilité. On pourra construire la table des symboles par effet de bord de la fonction d'ajout dans la table de visibilité.

6. Proposer une manière d'implanter les structures de données précédentes (table de visibilité locale et tables des symboles globales) en utilisant les bibliothèques CAML suivantes :

   (a) `Hashtbl` qui implante des tables de hachage

```
create : int -> ('a,'b) t
add : ('a,'b) t -> 'a -> 'b -> unit
find : ('a,'b) t -> 'a -> 'b
mem : ('a,'b) t -> 'a -> bool
remove : ('a,'b) t -> 'a -> unit
```

(b) `Map` qui implante des ensembles d'association sous forme d'arbres AVL. Le module `Map` définit un module qui peut être instancié sur n'importe quel ensemble de clés muni d'une fonction de comparaison. Pour l'utiliser, on peut faire :

```
module Key = struct type t=ident let compare = compare end
module Env = Map.Make(Key)
Env.empty : 'b Env.t
Env.add : ident -> 'b -> 'b Env.t -> 'b Env.t
Env.find : ident -> 'b Env.t -> 'b
Env.mem : ident -> 'b Env.t -> bool
```

7. On étend le langage avec des déclarations de type record de la forme

$$\textbf{type } id = \{id_1 : T_1; \ldots; id_n : T_n\}$$

Les expressions peuvent maintenant désigner un accès à un champs d'une expression de la forme $e.id$. Modifier votre programme pour prendre en compte cette nouvelle fonctionnalité. On précisera ce qui change dans les arbres de syntaxe abstraite, les règles de typage, la table de visibilité et la table des symboles.