Interpretação e Compilação de Linguagens de Programação
Semântica *CALCI* com Ambiente

14 de Março de 2013

Nesta aula apresenta-se uma definição mais eficiente da semântica operacional da linguagem *CALCI*. A semântica operacional que recorre à função de substituição, apresentada na aula anterior, é matematicamente conveniente mas pressupõe a manipulação da sintaxe abstracta do programa durante a execução, o que se torna pouco prático e eficiente. Uma implementação mais pragmática recorre a uma estrutura de dados auxiliar que mantém as ligações dos vários identificadores.

1 Ambiente de avaliação

A declaração estruturada e embricada de identificadores define implicitamente um conjunto de associações entre identificadores e correspondentes denotações que são válidas ou visíveis num determinado ponto de programa ou sub-expressão de acordo com as regras sintáticas da linguagem em questão. A esse conjunto dá-se o nome de ambiente ou contexto. Note-se que o ambiente depende do ponto do programa (a subexpressão) que se está a considerar.

Tomando como exemplo o código da Figura 1 na linguagem C (C99), o ambiente de avaliação da expressão `j+y` contém as denotações dos identificadores `f`, `x` (parâmetro), `z`, e `j`. Note-se que a variável `x`, local ao ciclo, não é visível nesta expressão, mas é visível no comando `z += x`, escondendo por si a associação do parâmetro `x`. Note-se ainda que a definição de uma função em C é implicitamente recursiva, logo o identificador `f` é visível em todo o corpo da função.

No caso da linguagem *CALCI*, pode definir-se uma função de avaliação I para expressões abertas com o seguinte domínio e contradomínio:

$$I : CALCI \times ENV \rightarrow Integer$$

A função de avaliação está definida para os pares expressão/ambiente em que o ambiente define uma denotação (inteira) para todos os identificadores livres

```
int f(int x)
{
  int z = x+1;
  for(int j=0; j<10; j++){
    int x=j+y;
    z += x;
  }
  return z;
}
```

Figura 1: Âmbito de um identificador

da expressão. Conceptualmente, os pares expressão/ambiente do domínio são fechados, e é possível definir uma função de avaliação seguindo os princípios já apresentados para expressões fechadas da linguagem *CALCI*.

A definição desta função é feita de forma composicional, ou seja, de forma independente para cada construção e dependendo apenas da avaliação das suas sub-expressões. A definição da semântica da linguagem *CALCI* deve ser igual à da semântica da linguagem *CALC* para as expressões que são comuns às duas linguagens. Para as expressões de declaração de um identificador (`decl` $x = E$ `in` E') e uso de um identificador (x) a semântica depende do ambiente. Na definição do algoritmo de avaliação utiliza-se uma estrutura de dados para manter o ambiente corrente. Para essa estrutura de dados definem-se as operações

$$\texttt{new} : \mathit{Void} \rightarrow \mathit{ENV}$$

$$\texttt{beginScope} : \mathit{ENV} \rightarrow \mathit{ENV}$$

$$\texttt{endScope} : \mathit{ENV} \rightarrow \mathit{ENV}$$

$$\texttt{find} : \mathit{ENV} \times \mathit{String} \rightarrow \mathit{Integer}$$

$$\texttt{assoc} : \mathit{ENV} \times \mathit{String} \times \mathit{Integer} \rightarrow \mathit{ENV}$$

Dada uma estrutura de dados como esta, a definição da semântica na linguagem OCaml, poderia ser definida de forma simplificada como na Figura 3. No caso da expressão de uso de um identificador x, a sua denotação é dada diretamente pelo ambiente. No caso da expressão de declaração de identificador a denotação da expressão `decl` $x = E$ `in` E' é a denotação da expressão E', dado o ambiente em que x tem a denotação da expressão E (Em vez de recorrer à substituição direta). Nesta definição da semântica, a operação (`find` x) é usada para obter a denotação de um identificador num determinado ambiente. A operação (`assoc` x v) é usada para criar um novo ambiente onde o identificador x é associado ao valor v.

Na linguagem Java para implementar o interpretador, teríamos um ambiente representado num objeto com o interface da Figura 3, e a implementação

```
let rec eval e env =
  match e with
  | Number n -> n
  | Add (l,r) -> (eval l env) + (eval r env)
  | Sub (l,r) -> (eval l env) - (eval r env)
  | Mul (l,r) -> (eval l env) * (eval r env)
  | Div (l,r) -> (eval l env) / (eval r env)

  | Id s -> find s env
  | Decl (s,l,r) ->
        let v = eval l env in
        let new_env = assoc s v env in
        eval r new_env
```

Figura 2: Semântica *CALCI*

```
interface Environment {
  Environment beginScope();
  Environment endScope();
  int find(String x);
  void assoc(String x, int v);
}
```

Figura 3: Interface de um ambiente implementado em Java

do método eval exemplificado na Figura 4.

Na linguagem *CALCI* cada declaração define um bloco novo e ao mesmo tempo o âmbito para o identificador declarado. Em muitas linguagens, os blocos de programa onde podem ser declarados nomes locais onde não pode haver repetições de nomes declarados. As operações `beginScope` e `endScope` definem âmbitos de declaração de nomes e dentro desses âmbitos a operação `assoc` cria a ligação entre um identificador e um valor, verificando a ausência de repetições. A Figura 5 ilustra a semântica de uma expressão `decl` com várias declarações. É criado um novo nível no ambiente (com a função `beginScope`) e nesse nível são criadas várias ligações (com a função `assoc`). Note-se a iteração da lista feita com a função OCaml `fold_left`.

```
class ASTNum {
  int n;
  ...
  int eval(Environment env) {
    return n;
  }
}

class ASTAdd {
  ASTNode left, right;
  ...
  int eval(Environment env) {
    return left.eval(env)+right.eval(env);
  }
}

class ASTId {
  String x;
  ...
  int eval(Environment env) {
    return env.find(x);
  }
}

class ASTNum {
  String x;
  ASTNode def, body;
  ...
  int eval(Environment env) {
    Environment newEnv;
    int value, result;

    value = def.eval(env);
    newEnv = env.beginScope();
    newEnv.assoc(x,value);
    result = body.eval(newEnv);
    //assert env == newEnv.endScope();
    return result;
  }
}
```

Figura 4: Semântica da linguagem *CALCI* implementada em Java

```
type ast = ...
  | Id of string
  | Decl of (string * ast) list * ast

...

let rec eval e env =
  match e with
  | Number n -> n
  | Add (l,r) -> (eval l env) + (eval r env)
  | Sub (l,r) -> (eval l env) - (eval r env)
  | Mul (l,r) -> (eval l env) * (eval r env)
  | Div (l,r) -> (eval l env) / (eval r env)

  | Id s -> find s env
  | Decl (decls,r) ->
        let f = fun env (x,l) ->
           let v = eval l env in
           let new_env = assoc x v env in new_env
        in
        let new_env = beginScope env in
        let last_env = List.fold_left f env decls in
        let result = eval r last_env in
        assert (env = endScope new_env) ;
        result
```

Figura 5: Semântica $CALCI$ com multiplas declarações

A Código Haskell

```
module CalcIEnv where

{- ©2013 João Costa Seco, ICL DI-FCT-UNL -}

import StateMaybeMonad

{- Definition of the Abstract Syntax of the language CALCI -}

data CALCI =
    Num Int
  | Add CALCI CALCI
  | Sub CALCI CALCI
  | Mul CALCI CALCI
  | Div CALCI CALCI

  | Id String
  | Decl String CALCI CALCI
  deriving Show

{- Examples -}s

a = Id "x"

b = Decl "x" (Num 1) (Add a (Num 1))

c = Decl "x" (Id "x") b

-- decl x = 1 in x
-- decl x = x in decl x = 1 in x

 ------------------------------------------------------------------------------
{- Semantics for open expressions with environments                           -}
 ------------------------------------------------------------------------------

{- Environment managment -}

find :: String -> [(String,Int)] -> Maybe Int

find x env = if l == [] then Nothing else Just (head l)
             where l = [ v | (k,v) <- env, k == x]
```

```
assoc :: [(String,Int)] -> String -> Int -> [(String,Int)]

assoc env x l = ((x,l):env)

{- -}

evalEnv :: CALCI -> [(String,Int)] -> StateMaybe Int Int

evalEnv (Id x) env =
        case find x env of
          Nothing -> raise_error
          Just v -> return v

evalEnv (Decl x e e') env =
        do
        l <- evalEnv e env
        evalEnv e' (assoc env x l)

evalEnv (Num n) env = return n

evalEnv (Add e e') env =
     do
     l <- (evalEnv e env)
     r <- (evalEnv e' env)
     return (l+r)

evalEnv (Sub e e') env =
     do
     l <- (evalEnv e env)
     r <- (evalEnv e' env)
     return (l-r)

evalEnv (Mul e e') env =
     do
     l <- (evalEnv e env)
     r <- (evalEnv e' env)
     return (l*r)

evalEnv (Div e e') env =
     do
     l <- (evalEnv e env)
     r <- (evalEnv e' env)
     if r == 0 then raise_error
     else return (div l r)
```

```
runstateEnv :: CALCI -> [(String,Int)] -> Maybe Int
runstateEnv e env = let SM(f) = (evalEnv e env) in snd (f 0)

-- runstateEnv (Decl "x" (Num 1) (Add (Id "x") (Num 1))) []

-- runstateEnv (Decl "x" (Num 3) (Add b (Id "x"))) []
```