Interpretation and Compilation of Programming Languages
Part 5 - Name declaration and binding (II)

João Costa Seco

March 26, 2014

In this lecture we discuss how we can define a semantic function in a language with variables (identifiers), that complies with the substitution principle, but does not require the rewriting of the abstract syntax tree.

We support the fundamental notions of binding and scope, by an auxiliary data structure, called environment, that maps identifiers to its meanings. We then study a type semantics for the language with identifiers, using the exact same data structure.

1 Evaluation Environment

The nested declaration of identifiers defines a mapping from identifiers to denotations, and at each point in a program or expression, a subset of these mappings is valid and visible. We call these subsets, the environment of an expression. Environments are actually observable in two situations. In a debugger, we can usually observe the visible identifiers and their values. In integrated development environments (IDE), the functionality called intellisense, that stands for context-aware code completion, is usually an instance of a typing environment.

Consider the code of Figure 1 written in C (C99). The evaluation environment of expression `j+y` contains the denotations of identifiers `f`, `x` (parameter), `z`, e `j`. Note that variable `x`, which is local in the loop, is not visible in this initializer expression, but is visible in the statement `z += x`. The declaration of the local variable shadows the parameter's (x) binding. Notice that a function declaration is implicitly recursive in `C`, hence identifier `f` is visible within the function body.

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

Figure 1: Code sample.

2 Operational semantics

In the previous lecture we defined the language *CALCI*, and an operational semantics for closed expressions. That definition is based on a substitution function that ensures the invariant, that expressions are closed. Expressions are evaluated, by evaluating (closed) sub-expressions.

An environment is a data structure that allows the definition of an operational semantics for open expressions.

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

This function is defined on pairs of expressions and environments, to denotations. We define an extended invariant, stating that all free names in the evaluating expression are included in the domain of the environment. The function is defined for closed pairs, and its definition mimics the substitution based definition.

We define the semantics of each construct in a compositional way, maintaining the semantics of the expressions of language *CALC*, and define the semantics for the new expressions, the declaration of an identifier (`decl` $x =$ E `in` E') and the corresponding use (x). The definition of the operational semantics is based on the environment. The environment is used to keep the current environment up-to-date. We define the following operations on environments:

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

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

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

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

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

```
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 = begin_scope env in
        let new_env' = assoc s v new_env in
        eval r new_env
        (* end_scope new_env *)
```

Figure 2: Operational semantics *CALCI*

Given a data structure such as this, we define the semantics using `OCaml` in Figure 3. In the case of the use of an identifier (x), the denotation is directly retrieved from the environment. In the case of an identifier declaration, the denotation of an expression `decl` $x = E$ `in` E' is given by the denotation of expression E', given an environment where identifier x is associated to the denotation of expression E (Instead of replacing it in place). The find operation (`find` x *env*) is used to get the denotation of identifier. The operation (`assoc` x v *env*) is used to create a new environment where identifier x is associated to the denotation v.

The scope of nested identifiers is represented in the layered structure of the environment data-structure. The `begin_scope` function initiates a new (inner) empty layer, and function `assoc` enriches that inner layer with a new association. Notice that, given the functional character of the `OCaml` language, the call to `end_scope` function is unnecessary.

An environment keeps information in a way, convenient for a recursive transversal of the AST where scopes, in a stack-like way (first-in, first-out). The recursive nature of the AST processing is accompanied by pushing it into the environment stack, it shadows the bindings of the outer scopes.

When using an imperative language like `Java`, we can represent its abstract data type through a generic interface like the one in Figure 3. We abstract the type of the denotation by the generic type, allowing it to support different semantic functions like evaluation, typing, and compilation.

```
interface Environment<T> {
  Environment<T> beginScope();
  Environment<T> endScope();
  T find(String x);
  void assoc(String x, T value);
}
```

Figure 3: Interface of an environment in `Java`

By using an environment, we can implement method `eval` of the nodes of the AST (`ASTNode`) by means of the (sample) code in Figure 4.

In the programming language *CALCI* above, each declaration defines a new scope, and new environment block, containing the definition of a single identifier. The scoping rules of all languages, can be represented by variants of this environment data type. We can allow more than one declaration by layer, recursive declarations, type declarations, etc.. The operations `beginScope` and `endScope` define the scope of identifiers, and operation `assoc` binds an identifier to a denotation, ensuring the absence of repetitions.

Figure 7 illustrates the semantics of a declaration `decl` for several identifiers. In this case function `beginScope` defines a new scope, and function `assoc` creates new bindings in the "closest" environment layer. Notice the iterative processing of the declaration lists, defined in `OCaml` by the function `fold_left` of the `List` module (see the official documentation for details).

```
class ASTNum {
  int n;
  ...
  Value eval(Environment<Value> env) {
    return IntegerValue.value(n);
  }
}

class ASTAdd {
  ASTNode left, right;
  ...
  Value eval(Environment<Value> env) {
    return IntegerValue.add(left.eval(env),right.eval(env));
  }
}

class ASTId {
  String x;
  ...
  Value eval(Environment<Value> env) {
    return env.find(x);
  }
}

class ASTDecl {
  String x;
  ASTNode def, body;
  ...
  Value eval(Environment<Value> env) {
    Environment<Value> 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;
  }
}
```

Figure 4: Operational semantics for language *CALCI* using `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
```

Figure 5: Operational semantics of *CALCI* with multiple declarations.

```
let rec typecheck e env =
  match e with
  | Number n -> IntType
  | Add (e,e') ->
    (match typecheck e env, typecheck e' env with
        IntType, IntType -> IntType
      | _,_ -> None)

  | Decl (x,e,e') -> typecheck e' (assoc x (typecheck e env) env)
  | Id(x) -> find x env ;;
```

Figure 6: Typing of *VALI*.

```
public Type typecheck(Environment<Type> env) throws TypeException {
  Type t = definition.typecheck(env);

  Environment<Type> new_env = env.beginScope();
  new_env.assoc(id,t);
  Type result = body.typecheck(new_env);
  new_env.endScope();
  return result;
}
```

Figure 7: Typing of *VALI*.

3 Typing using environments

Take the programming language *VAL* presented in previous lectures, and its typing procedure. Consider the extension of that language with identifiers (declaration and use), called *VALI*. Typing is a semantic function that is defined in the cases of the language expressions, like the operational semantics. Although the operational semantics and typing of some expressions are differently, according to its domain, the typing of identifier declarations and use is also defined in a very similar way. The rules for these expression essentially establish the scoping and visibility rules of identifiers, that are common to evaluation and typing functions. The difference is located in the denotation held by the (typing) environment.

The implementation using a language like `Java` takes the advantage of generic types, and adapts the kind of denotation is used in this semantic function.

4 Soundness of typing with identifiers

The typing of a language is defined in such a way that execution errors trapped in the definition of the operational semantics, do not occur in well-typed programs. We can use a version of the substitutivy principle, that ensures that substitution of identifiers by values in well-typed expressions (or programs), preserves typing.

We will recall this discussion in a future lecture.