Formal Programming Language Semantics note 7

Static semantics: scoping and typing rules

So far, we have concentrated on describing the *dynamic* (i.e. run-time) semantics of programming languages. We will now examine how one might formally express *static* aspects of a language (things that would typically be checked at compile-time), such as scoping and typing constraints.

A block-structured language. We will consider a simple example of a language in which programs are structured in (possibly nested) *blocks* which statically determine the scoping of variables. Virtually all imperative languages from Algol (1960) onwards are of this kind. In fact, we will introduce a language **IMP**^{b} which extends **IMP** with several new features:

1. A "block" construct (written `let var` $X = a$ `in` $\cdots$ `end`) for declaring local variables with delimited scope.
2. Program variables of boolean as well as integer type — this will allow us to illustrate the idea of *typing rules*.
3. A distinction between *variables* (which can be updated) and *constants* (which can't). This just illustrates another kind of static constraint.

We will need a bit more notation. Let $\mathbb{U}$ be the set $\{\texttt{int}, \texttt{bool}, \texttt{com}\}$, which we will think of as a set of *types*. (We would like to have used $\mathbb{T}$, but that's already something else.) We also write $\mathbb{U}_E$ for the set $\{\texttt{int}, \texttt{bool}\}$ of *expression types*. We use u and its decorated variants to range over $\mathbb{U}$ and $\mathbb{U}_E$. We also let $\mathbb{D}$ be the set $\{\texttt{var}, \texttt{const}\}$, which we will call the set of *status descriptors*; we use d and its decorated variants to range over $\mathbb{D}$.

To give the syntax of **IMP**^{b}, it will make life a bit easier to combine the phrase categories Aexp and Bexp into a single category Exp of general *expressions*, ranged over by e. (Don't worry - we will be able to recover this distinction once we have given the typing rules.) The full syntax of **IMP**^{b} may now be given by the following grammar:

$$\begin{array}{lll} \mathsf{Exp}: & e ::= & X \mid n \mid e_0 \texttt{ - } e_1 \mid e_0 \texttt{ * } e_1 \mid \\ & & t \mid e_0 \texttt{ <= } e_1 \mid \texttt{not } e \mid e_0 \texttt{ and } e_1 \\ \\ \mathsf{Com}: & c ::= & \texttt{skip} \mid X \texttt{ := } e \mid c_0 \texttt{ ; } c_1 \mid \texttt{if } e \texttt{ then } c_0 \texttt{ else } c_1 \mid \texttt{while } e \texttt{ do } c \mid \\ & & \texttt{let } d\ X = e \texttt{ in } c \texttt{ end} \end{array}$$

We will also write Phrase for the set of all phrases generated by this grammar: $\mathsf{Phrase} = \mathsf{Aexp} \sqcup \mathsf{Exp}$.

Static semantics. Not all phrases generated by the above grammar are (or should be) legal program fragments in **IMP**^{b}. In order to specify which are and which aren't, we will introduce a set of static *typing rules*, somewhat analogous to the dynamic semantic rules considered earlier. The key idea is this: whether a program phrase is legal will typically depend on the "environment" in which it is used — that is, on which variables are in existence, and what their type and status is, at the point in the program where the phrase occurs. We can capture all this by means of a *typing relation* of the form

$$\Gamma \vdash P : u$$

where P is a program phrase, u is a type (one of `int`, `bool`, `com`), and Γ is a *static environment* which records information about the variables that exist and their type and status. The informal reading of this relation is "in the environment Γ, P is a well-formed program phrase of type u."

What exactly is a static environment? In this case, it will be a finite list of identifiers together with associated types and status descriptors. An example of a static environment would be the list $[(\mathtt{X}, \mathtt{int}, \mathtt{const}),\ (\mathtt{Y}, \mathtt{bool}, \mathtt{var})]$. In general, a static environment will be an element of the set

$$\mathbb{SE} \ = \ \mathsf{List}(\mathbb{I} \times \mathbb{U}_E \times \mathbb{D}).$$

We use Γ to range over static environments. (Note that static environments are very similar to what are often called *contexts* in logic and type theory, but we have already used this word to mean something else: see Note 6.) The typing relation will therefore be a certain subset of $\mathbb{SE} \times \mathsf{Phrase} \times \mathbb{U}$.

Like the evaluation relation, the typing relation will be defined inductively by means of a set of syntax-directed *derivation rules*. (The way in which a set of rules gives rise to a relation will be exactly as explained in detail in Note 4.) In fact, the typing rules will be even more strictly syntax-directed than the evaluation rules — each rule will specify the type of some phrase in terms of strictly smaller phrases, so we won't have the equivalent of "non-terminating computations" — the practical upshot of this is that the typing relation is *decidable* via a simple algorithm.

A bit more notation involving static environments is needed before we give the rules. For environment update, we write $\Gamma[X \mapsto (u, d)]$ (where $u \in \mathbb{U}_E$) to mean $\Gamma; (X, u, d)$ — that is, the list obtained by adding the element $(X, u,)$ to the end of Γ. For looking up a particular identifier in an environment, we define $\Gamma(\!|X|\!)$ recursively as follows:

$$[\,](\!|Y|\!) \ = \ ? \ \text{('undefined')}, \qquad (\Gamma; (X, u, d))(\!|Y|\!) \ = \ \begin{cases} (u, d) & \text{if } Y = X, \\ \Gamma(\!|Y|\!) & \text{otherwise.} \end{cases}$$

Note that if Γ contains more than one entry for a certain identifier X, this definition of $\Gamma(\!|X|\!)$ will give us the pair (u, d) associated with the *last* such entry.

[Exercise: In what ways do these definitions differ from the definitions of state update and lookup? Can you see the reason for these differences, and for the above choice of definition of $\Gamma(\!|X|\!)$?]

Typing rules for IMP^b. We may now give the rules for the typing relation. There is exactly one rule for each syntactic construct. First let us give the rules that make use of static environments in an interesting way. The rule for variable expressions says that the type of such expressions is that determined by the current static environment:

$$\frac{}{\Gamma \vdash X : u} \quad \Gamma(\!|X|\!) = (u, d)$$

The rule for assignments says that an assignment is a legal phrase of type `com`, but only if the types match up correctly *and* the identifier has variable status:

$$\frac{\Gamma \vdash e : u}{\Gamma \vdash X := e \ : \texttt{com}} \quad \Gamma(\!|X|\!) = (u, \texttt{var})$$

Next, the crucial rule for blocks, which captures the idea that within the body of a block we are allowed to use the newly declared variable with the appropriate type and status, but nowhere outside the block can we refer to this variable. Note that the initialization expression e has to make sense in the "outer environment" Γ in which the block occurs.

$$\frac{\Gamma \vdash e : u \qquad \Gamma' \vdash c : \texttt{com}}{\Gamma \vdash \texttt{let}\ d\ X = e\ \texttt{in}\ c\ \texttt{end} \ : \texttt{com}} \quad \Gamma' = \Gamma; (X, u, d)$$

(There is quite a lot packed into this rule, so it is worth studying it carefully!)

The remaining rules give the types of other expression forms, but do not make use of the static environment in any interesting way. For readability, we give them here in a reduced form, writing just $P : u$ everywhere in place of $\Gamma \vdash P : u$ (you can call this the *static environment convention* if you like!).

$$\frac{}{n : \texttt{int}} \qquad \frac{e_0 : \texttt{int} \quad e_1 : \texttt{int}}{e_0\ \texttt{-}\ e_1 : \texttt{int}} \qquad \frac{e_0 : \texttt{int} \quad e_1 : \texttt{int}}{e_0\ \texttt{*}\ e_1 : \texttt{int}}$$

$$\frac{}{t : \texttt{bool}} \qquad \frac{e_0 : \texttt{int} \quad e_1 : \texttt{int}}{e_0\ \texttt{<=}\ e_1 : \texttt{bool}}$$

$$\frac{e : \texttt{bool}}{\texttt{not}\ e : \texttt{bool}} \qquad \frac{e_0 : \texttt{bool} \quad e_1 : \texttt{bool}}{e_0\ \texttt{and}\ e_1 : \texttt{bool}}$$

$$\frac{}{\texttt{skip} : \texttt{com}} \qquad \frac{c_0 : \texttt{com} \quad c_1 : \texttt{com}}{c_0\ \texttt{;}\ c_1 : \texttt{com}}$$

$$\frac{e : \texttt{bool} \quad c_0 : \texttt{com} \quad c_1 : \texttt{com}}{\texttt{if}\ e\ \texttt{then}\ c_0\ \texttt{else}\ c_1 : \texttt{com}} \qquad \frac{e : \texttt{bool} \quad c : \texttt{com}}{\texttt{while}\ e\ \texttt{do}\ c : \texttt{com}}$$

Finer points. A few further comments and observations on the above:

1. All this is closely analogous to what we did in dynamic semantics. In fact, one can think of a *state* as a kind of environment relative to which a program may be run. Indeed, in much of the literature, what we have called

states are referred to as *dynamic environments*; a letter like E is used in place of σ; and the evaluation relation is written as $E \vdash P \Downarrow R$ rather than $\langle P, \sigma \rangle \Downarrow R$. However, it is important to recognize that quite different kinds of information are recorded by static and dynamic environments: typically, the former will record statically determined information such as the *type* of a variable at a certain place in the program text, while the latter will record its *value* at a certain point in time during the execution.

2. The language **IMP**^{b} illustrates the difference between the concepts of *scope* and *visibility*. Consider the program

```
 let var X = true in
   let var X = 3 in X := X+1 end
end
```

   The two declarations here really declare two different variables called `X` (with different types). The *scope* of the outer declaration of `X` — the region in which the boolean variable "exists" — consists of the whole of the outer block, though it is not *visible* throughout the whole of this region since it is *shadowed* by the inner declaration. Our semantics makes precise the idea that within nested blocks like this, a variable expression refers to the *innermost* enclosing declaration for the corresponding identifier.

   [Exercise: In many languages we are allowed declarations in the middle of a block — their scope extends from the point of declaration to the end of the block. Show how one can give a static semantics for such a language.]

3. Whilst it may appear that our typing rules are partly just making up what we lost by amalgamating arithmetical and boolean expressions in the grammar of the language, it is fairly clear that *no* (finite) context-free grammar by itself would suffice by itself to capture exactly the well-formed phrases of **IMP**^{b}. So static semantics is really giving us something new here.

   Actually, the typing rules by themselves completely determine the set of well-formed phrases, so we could throw away the context-free grammar if we wanted! But there again, it's quite nice to construct a set of "potential terms" first and then say which ones are actually legal.

4. Notice that **IMP**^{b} has *implicit typing* — we don't have to annotate declarations with things like `:int`, but even so, the static semantics specifies how types are assigned to variables. An implementation of **IMP**^{b} would therefore have to do some simple *type inference* at compile time, in order to check that a program was well-typed.

John Longley