Formal Programming Language Semantics note 8

Evaluation and binding mechanisms

In this note we will consider several possible dynamic semantics for the language **IMP**^{b} of Note 7. These will illustrate some important distinctions, such as *lazy* vs. *eager* evaluation, and *static* vs. *dynamic* binding. We will also consider how the static and dynamic semantics may be related via by the property of *type safety*.

Eager evaluation. Let us consider one way to modify our operational semantics of **IMP** to give one for **IMP**^{b}. First, since we have lumped arithmetic and boolean expressions into a single class of *expressions* e, let us define the set $\mathbb{V}$ of *values* to be the disjoint union $\mathbb{Z} \sqcup \mathbb{T}$, and let us take the set $\mathbb{S}^b$ of *states* to be $(\mathbb{I} \times \mathbb{V})^*$, i.e. the set of finite lists of pairs associating values to identifiers. As in Note 7, the switch from partial functions to lists allows us to cope with the possibility of more than one variable with the same name, with one temporarily "hiding" the other. We use v to range over $\mathbb{V}$, and σ to range over $\mathbb{S}^b$. We will set $\sigma(\!|X|\!) = v$ iff (X, v) is the *rightmost* pair occurring in σ whose first component is v.

We now wish to give rules for deriving evaluation statements of the forms $\langle e, \sigma\rangle \Downarrow v$ and $\langle c, \sigma\rangle \Downarrow \sigma'$. Most of these rules are just simple adaptations of the corresponding rules for **IMP** (check that you know how they should go). The only new feature is the `let` construct, for which we may give the following rule:

$$\frac{\langle e, \sigma\rangle \Downarrow v \qquad \langle c, \sigma''\rangle \Downarrow \sigma'''}{\langle \texttt{let}\ DX = e\ \texttt{in}\ c\ \texttt{end}, \sigma\rangle \Downarrow \sigma'} \quad \sigma'' = \sigma; (X, v), \quad \sigma''' = \sigma'; (X, -)$$

This captures the idea that the original state σ is extended with a new variable called X to give a state σ''; the body of the `let` command (which may refer to X) is then run on this state yielding a new state σ'''; and finally we dispose of the local variable X to give the state σ'. The symbol "$-$" in the second side-condition means "anything" (we could also have used a dummy metavariable v').

This rule embodies a particular choice of evaluation behaviour of the `let` construct: the execution always begins by evaluating e and binding the resulting value to X, regardless of whether X is ever actually used in the body of the block. This evaluation strategy is often known as *eager evaluation* since we evaluate e "as soon as possible" without waiting to see whether the value is ever needed. Such a mechanism is also known as *call-by-value*, since occurrences of X within c effectively stand for a fixed value v which has been previously computed.

Lazy evaluation. We may contrast eager evaluation with various kinds of *lazy evaluation*, in which we do not evaluate e until we actually need to know the

value of X. We can distinguish two kinds of lazy evaluation: *call-by-name*, in which e is evaluated afresh each time we need to know the value of X (so we can think of occurrences of X in c as standing for a formal expression or "name" for X), and *call-by-need*, in which e is evaluated the *first* time we need to know the value of X, and the value is then stored for later use. It is easy to see that for a given program the call-by-value, call-by-name and call-by-need mechanisms could all lead to different results, since the value of e may depend on the current value of other variables.

To give a formal semantics that captures call-by-name evaluation, we need to change our notion of state once again. The idea is that an identifier will no longer be associated with a fixed value, but with a formal expression e which we can evaluate as required. Let us therefore take $\mathbb{S}^b = (\mathbb{I} \times \mathsf{Exp})^*$ as our set of states (recall that Exp is a phrase category in our grammar, and can be used to denote the corresponding set of phrases). Again, most of the operational rules are essentially as for **IMP**, but the rule for `let` now becomes:

$$\frac{\langle c, \sigma'' \rangle \Downarrow \sigma'''}{\langle \texttt{let}\ DX = e\ \texttt{in}\ c\ \texttt{end}, \sigma \rangle \Downarrow \sigma'} \quad \sigma'' = \sigma; (X, e), \quad \sigma''' = \sigma'; (X, -)$$

We also need to change the rule for variables, since in order to evaluate a variable we now have to evaluate the corresponding expression rather than just look up the value in the current state. One possibility is

$$\frac{\langle e, \sigma \rangle \Downarrow v}{\langle X, \sigma \rangle \Downarrow v} \quad \sigma(\!|X|\!) = e$$

This has the effect that e will be evaluated (relative to the current state) each time we encounter the expression X.

We can capture *call-by-need* in a very similar way, except that the first time we evaluate X we would like to replace the binding (X, e) by (X, v), so that e does not have to be evaluated the next time we encounter X. This means that the evaluation of expressions can sometimes have a side-effect on the state (as in **IMP**^{s}), so our evaluation statements will need to be of the form $\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$. Most of the rules only require trivial modifications to take account of this. However, the rule for evaluating a variable X now becomes

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle X, \sigma \rangle \Downarrow \langle v, \sigma'' \rangle} \quad \sigma(\!|X|\!) = e, \quad \sigma'' = \sigma'[X \mapsto v]$$

Static vs. dynamic binding. There is another subtle semantic choice implicit in the above rules. To see this, consider the following program, which we intend to be run in a state σ with $\sigma(\!|\mathsf{Y}|\!) = \sigma(\!|\mathsf{Z}|\!) = \mathtt{0}$:

```
let var X=Y-1 in let var Y=3 in Z := X end end
```

The first `let` construct will declare a local variable `X` and bind it to the expression `Y-1`, while the second will introduce a new local variable called `Y`. When we come

to evaluate the expression `X` in `Z := X`, we therefore have to evaluate `Y-1`. Now a question arises: does the `Y` here refer to the `Y` which was visible at the point of the declaration `X=Y+1` (i.e. the global variable `Y`), or to the `Y` which is visible at the point of the expression `X` which we are evaluating (i.e. the local `Y`)? In the former case, the value assigned to `Z` will be -1; in the latter case it will be 2.

These two possibilities are referred to as *static* and *dynamic* binding respectively: in static binding the locations referred to by any identifiers within a declaration are fixed at declaration time, whereas in dynamic binding these references are resolved at evaluation time, and thus may be different for different occurrences of the variable in question. (Of course, in either case, the *value* of the variable is that at evaluation time, assuming we are considering lazy evaluation.)

Let us look again at the above rule for variables under call-by-name:

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle X, \sigma\rangle \Downarrow v} \quad \sigma(\!|X|\!) = e$$

Since the state σ here is the evaluation time environment, we can see that this corresponds to a dynamic binding convention.[1] To capture a call-by-name strategy with static binding, we need to modify the premise so that e is evaluated relative to the declaration time environment. So let us write $\sigma\,|_X$ to mean the *restriction* of the state σ at X — that is, the portion of σ up to (but not including) the rightmost binding for X. The rule for variables is now

$$\frac{\langle e, \sigma'\rangle \Downarrow v}{\langle X, \sigma\rangle \Downarrow v} \quad \sigma(\!|X|\!) = e, \quad \sigma' = \sigma\,|_X$$

This has the effect that any variables occurring in e refer to the locations they would have referred to at the point at which e appears in the program, i.e. at the declaration $X = e$.

Notice that the static/dynamic distinction does not arise in the case of eager evaluation: if we evaluate the expression e at declaration time then we are in effect forced to adopt static binding. Note also that in all the above versions of the semantics we have (for convenience) retained the familiar treatment of assignments X `:=` e, so that in effect these work in an "eager" way, assigning to X the *value* of e rather than the expression e itself.

Exercises on static and dynamic binding.

(1) Formulate an appropriate rule for variables under a *call-by-need* strategy with dynamic binding.

(2) Give an example showing that under a dynamic binding convention the evaluation of an expression may "loop" indefinitely. How is this reflected in the evaluation relation defined by the operational rules?

(3) There is another way to capture call-by-name evaluation, using the idea of *substitution*. Let us write $c[e/X]$ for the command obtained from c by textually

[1] Notice that our language has static *scoping* but dynamic *binding*. The same is true for methods in Java, for instance.

replacing all occurrences of the expression X by the expression e (we do not replace occurrences of X on the left hand side of `:=`). Use this idea to formulate an alternative rule for `let` which captures call-by-name evaluation. Does your rule naturally give rise to static or dynamic binding? Can you modify it so that it gives the other one?

(4) The approach using substitution works well if we drop assignments from the language, but there is a problem in the presence of assignment. Give an example to illustrate the problem. (This suggests that the substitution idea is useful for *functional* languages, but not for imperative languages like **IMP**^{b}.)

Type safety. Notice that types do not feature at all in our various dynamic semantics of **IMP**^{b}: neither the states nor the operational rules make reference to the types of expressions. This reflects the idea that, in an implementation of **IMP**^{b}, we do not need to retain type information at runtime, so we can regard the type system as a kind of "scaffolding" that gets thrown away after compilation.[2]

Nevertheless, the type system does serve a purpose: as long as a program typechecks, it will be automatic that it produces results of the type we expect, so we do not need to perform runtime typechecks to ensure this. We can now give a precise formulation of this idea, which is known as *type safety*.[3]

Let us do this for the eager version of **IMP**^{b} as an example. First, define $[\![\,\texttt{int}\,]\!] = \mathbb{Z}$, $[\![\,\texttt{bool}\,]\!] = \mathbb{T}$ (we will see more of this notation later!). Next, let us say a state σ *conforms to* a static environment Γ if for some X_i, u_i, v_i we have

$$\Gamma \;=\; [(X_1, u_1), \ldots, (X_n, u_n)], \qquad \sigma \;=\; [(X_1, v_1), \ldots, (X_n, v_n)], \qquad \forall i.\ v_i \in [\![\, u_i \,]\!].$$

We may then say that (eager) **IMP**^{b} is *type-safe* if the following properties hold:

- If σ conforms to Γ, $\Gamma \vdash e : u$ and $\langle e, \sigma\rangle \Downarrow v$, then $v \in [\![\, u \,]\!]$.
- If σ conforms to Γ, $\Gamma \vdash c : \texttt{com}$ and $\langle c, \sigma\rangle \Downarrow \sigma'$, then σ' conforms to Γ.

For eager **IMP**^{b}, these properties can indeed be proved as theorems. The proofs work by induction on the derivations of $\langle e, \sigma\rangle \Downarrow v$ [resp. $\langle c, \sigma\rangle \Downarrow \sigma'$].

Exercises. (1) Satisfy yourself that you can see how to prove the above properties (a complete proof would be rather tedious). (2) The statement of type safety needs some minor modification for our other versions of **IMP**^{b} to cope with the different forms of evaluation statements. Give precise statements of type safety for call-by-name **IMP**^{b} with static and dynamic binding. Are these languages in fact type-safe? If not, can you see how one might fix the problem?

John Longley

[2] The same is true for Standard ML. In Java, however, some type information does need to be carried around at runtime, in order to achieve dynamic dispatch for methods.

[3] Type safety is a property that language designers and implementers really do care about in practice, and it is not always trivial — there have in the past been languages, such as Eiffel, that have later been discovered not to be type-safe. A considerable amount of effort has been put into checking formally that certain key fragments of Java are type-safe.