Lecture Notes on Static and Dynamic Semantics

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 4
September 9, 2004

In this lecture we illustrate the basic concepts underlying the static and dynamic semantics of a programming language on a very simple example: the language of arithmetic expression augmented by variables and definitions.

The static and dynamic semantics are properties of the abstract syntax (terms) rather than the concrete syntax (strings). Therefore we will deal exclusively with abstract syntax here.

The static semantics can further be decomposed into two parts: variable scope and rules of typing. They determine how to interpret variables, and discern the meaningful expressions. As we saw in the last lecture, variable scope is encoded directly into the terms representing the abstract syntax. In this lecture we further discuss the laws governing variable binding on terms. The second step will be to give the rules of typing in the form of an inductively defined judgment. This is not very interesting for arithmetic expressions, comprising only a single type, but it serves to illustrate the ideas.

The dynamic semantics varies more greatly between different languages and different levels of abstraction. We will only give a very brief introduction here and continue the topic in the next lecture.

The basic principle of variable binding called *lexical scoping* is that the name of a bound variable should not matter. In other words, consistently renaming a variable in a program should not affect its meaning. Everything below will follow from this principle.

We now make this idea of "consistent renaming of variables" more precise. The development in [Ch. 5.3] takes simultaneous substitution as a primitive; we avoid the rather heavy notation by only dealing with a single substitution at a time. This goes hand in hand with the decision that binding prefixes such as $x.t$ only ever bind a single variable, and not multiple ones. We use the notation $\{y/x\}t$ to denote the result of substituting y for x in t, yet to be defined. With that we will define renaming of x to y with the equation

$$x.t =_\alpha y.\{y/x\}t$$

which can be applied multiple times, anywhere in a term. For this to preserve the meaning, y most not already occur free in $x.t$, because otherwise the free occurrence of y would be *captured* by the new binder.

As an example, consider the term

$$\mathtt{let}(\mathtt{num}(1), x.\mathtt{let}(\mathtt{plus}(x, \mathtt{num}(1)), y.\mathtt{plus}(y, x)))$$

which should evaluate to $\mathtt{num}(3)$. It should be clear that renaming y to x should be disallowed. The resulting term

$$\mathtt{let}(\mathtt{num}(1), x.\mathtt{let}(\mathtt{plus}(x, \mathtt{num}(1)), x.\mathtt{plus}(x, x)))$$

means something entirely different and would evaluate to $\mathtt{num}(4)$.

To make this side condition more formal, we define the set of free variables in a term.

$$\begin{array}{rcl} \mathrm{FV}(x) & = & \{\, x \,\} \\ \mathrm{FV}(o(t_1, \ldots, t_n)) & = & \bigcup_{1 \le i \le n} \mathrm{FV}(t_i) \\ \mathrm{FV}(x.t) & = & \mathrm{FV}(t) \setminus \{\, x \,\} \end{array}$$

So before defining the substitution $\{y/x\}t$ we restate the rule defining variable renaming, also called α-conversion, with the proper side condition:

$$x.t =_\alpha y.\{y/x\}t \qquad \text{provided } y \notin \mathrm{FV}(t)$$

Now back to the definition of substitution of one variable y for another variable x in a term t, $\{y/x\}t$. The definition recurses over the structure of a term.[1]

$$\begin{array}{rcll} \{y/x\}x & = & y & \\ \{y/x\}z & = & z & \text{provided } x \neq z \\ \{y/x\}o(t_1, \ldots, t_n) & = & o(\{y/x\}t_1, \ldots, \{y/x\}t_n) & \\ \{y/x\}x.t & = & x.t & \\ \{y/x\}z.t & = & x.\{y/x\}t & \text{provided } x \neq z \text{ and } y \neq z \\ \{y/x\}y.t & & \text{undefined} & \text{provided } x \neq y \end{array}$$

[1]It can in fact be seen as yet another form of inductive definition, but we will not formalize this here.

Note that substitution is a *partial* operation. The reason the last case must be undefined is because any occurrence of x in t would be replaced by y and thereby captured. As an example while this must be ruled out, reconsider

$$\texttt{let}(\texttt{num}(1), x.\texttt{let}(\texttt{plus}(x, \texttt{num}(1)), y.\texttt{plus}(y, x)))$$

which evaluates to $\texttt{num}(3)$. If we were allowed to rename x to y we would obtain

$$\texttt{let}(\texttt{num}(1), y.\texttt{let}(\texttt{plus}(y, \texttt{num}(1)), y.\texttt{plus}(y, y)))$$

which once again means something entirely different and would evaluate to $\texttt{num}(4)$.

In the operational semantics we need a more general substitution, because we need to substitute one term for a variable in another term. We generalize the definition above, taking care to rewrite the side condition on substitution in a slightly more general, but consistent form, in order to prohibit variable capture.

$$\begin{array}{rcll}
\{u/x\}x & = & u & \\
\{u/x\}z & = & z & \text{provided } x \neq z \\
\{u/x\}o(t_1, \ldots, t_n) & = & o(\{u/x\}t_1, \ldots, \{u/x\}t_n) & \\
\{u/x\}x.t & = & x.t & \\
\{u/x\}z.t & = & z.\{u/x\}t & \text{provided } x \neq z \text{ and } z \notin \mathrm{FV}(u) \\
\{u/x\}z.t & & \text{undefined} & \text{provided } x \neq z \text{ and } z \in \mathrm{FV}(u)
\end{array}$$

In practice we would like to treat substitution as a total operation. This cannot be justified on terms, but, surprisingly, it works on α-equivalence classes of terms! Since we want to identify terms that only differ in the names of their bound variables, this is sufficient for all purposes in the theory of programming languages. More formally, the following theorem (which we will not prove) justifies treating substitution as a total operation.

Theorem 1 (Substitution and α-Conversion)

(i) If $u =_\alpha u'$, $t =_\alpha t'$, and $\{u/x\}t$ and $\{u'/x\}t'$ are both defined, then $\{u/x\}t =_\alpha \{u'/x\}t'$.

(ii) Given u, x, and t, then there always exists a $t' =_\alpha t$ such that $\{u/x\}t'$ is defined.

We sketch the proof of part *(ii)*, which proceeds by induction on the size of t. If $\{u/x\}t$ is defined we choose t' to be t. Otherwise, then somewhere the last clause in the definition of substitution applies and there is a binder

$z.t_1$ in t such that $z \in \mathrm{FV}(u)$. Then we can rename z to a new variable z' which occurs neither in free in u nor free in $z.t_1$ to obtain $z'.t_1'$. Now we can continue with $z'.\{u/x\}t_1'$. by an appeal to the induction hypothesis.

The algorithm described in this proof is in fact the definition of *capture-avoiding substitution* which makes sense whenever we are working modulo α-equivalence classes of terms. Fortunately, this will always be the case for the remainder of this course.

With the variable binding, renaming, and substitution understood, we can now formulate a first version of the typing rules for this language. Because there is only one type, `nat`, the rules are somewhat trivialized. Their only purpose for this small language is to verify that an expression e is *closed*, that is, $\mathrm{FV}(e) = \{\,\}$.

A first judgmental way to express this would be the following[2]:

$$\frac{k\ \mathsf{nat}}{\mathtt{num}(k) : \mathtt{nat}}$$

$$\frac{e_1 : \mathtt{nat} \quad e_2 : \mathtt{nat}}{\mathtt{plus}(e_1, e_2) : \mathtt{nat}} \qquad \frac{e_1 : \mathtt{nat} \quad e_2 : \mathtt{nat}}{\mathtt{times}(e_1, e_2) : \mathtt{nat}}$$

$$\frac{e_1 : \mathtt{nat} \quad \{e_1/x\}e_2 : \mathtt{nat}}{\mathtt{let}(e_1, x.e_2) : \mathtt{nat}}$$

While this is perfectly correct, it has the potential problem that it re-checks e_1 for every occurrence of x in e_2. This could be avoided by substituting a fixed value such as $\mathtt{num}(0)$ for x and checking the result.

A more common (and more scalable) alternative is to use a new judgment form, a so-called *hypothetical judgment*. We write it as

$$J_1, \ldots, J_n \vdash J$$

which means that J follows from assumptions $J_1, \ldots, J_n$. Its most basic property is that

$$J_1, \ldots, J_i, \ldots J_n \vdash J_i$$

always holds, which should be obvious: if an assumption is identical to the judgment we are trying to derive, we are done. We will nonetheless restate instances of this general principle for each case.

The particular form of hypothetical judgment we consider is

$$x_1{:}\mathtt{nat}, \ldots, x_n{:}\mathtt{nat} \vdash e : \mathtt{nat}$$

which should be read:

[2]suggested by a student in class

> *Under the assumption that variables $x_1, \ldots, x_n$ stand for natural numbers, e has the type of natural number.*

We usually abbreviate a whole sequence of assumptions with the letter Γ.[3] We write '$\cdot$' for an empty collection of assumptions, and we abbreviate $\cdot, x{:}\texttt{nat}$ by $x{:}\texttt{nat}$. In order to avoid ambiguities, we always assume that all variables declared in a context are distinct.

The typing judgment is defined by the following rules.

$$\frac{x{:}\texttt{nat} \in \Gamma}{\Gamma \vdash x : \texttt{nat}} \qquad \frac{k\ \mathsf{nat}}{\Gamma \vdash \texttt{num}(k) : \texttt{nat}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{nat} \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{plus}(e_1, e_2) : \texttt{nat}} \qquad \frac{\Gamma \vdash e_1 : \texttt{nat} \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{times}(e_1, e_2) : \texttt{nat}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{nat} \quad \Gamma, x{:}\texttt{nat} \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{let}(e_1, x.e_2) : \texttt{nat}}$$

In the last rule some care has to be taken to make sure that x is not declared twice in the context. If the variable x bound in $\texttt{let}(e_1, x.e_2)$ is already declared, we use the assumption that we work modulo α-equivalence classes and rename the variable x to a fresh variable x' before applying the rule.

The point of being interested in typing for this small language is only to guarantee that there are no free variables in a term to the evaluation will not get stuck. This property can easily be verified.

Theorem 2
If $\cdot \vdash e : \texttt{nat}$ *then* $\mathrm{FV}(e) = \{\,\}$.

Proof: We cannot prove this directly by rule induction, since the second premise of the rule for `let` introduces an assumption. So we generalizing to

> *If* $x_1{:}\texttt{nat}, \ldots, x_n{:}\texttt{nat} \vdash e : \texttt{nat}$ *then* $\mathrm{FV}(e) \subseteq \{x_1, \ldots, x_n\}$.

This generalized statement can be proved easily by rule rule induction. ■

Next we would like to give the operational semantics, specifying the value of an expression. We represent values also as expressions, although they are restricted to have the form $\texttt{num}(k)$. Generally, when we write an

[3]In [Ch. 6] this is written instead as $\Gamma \vdash e\ \mathsf{ok}$, where Γ is a set of variables. Since there is only one type, the two formulations are clearly equivalent.

expression as v we imply that it is a value and therefore has the form $\mathtt{num}(k)$ for some k.

There are multiple ways to specify the operational semantics, for example as a structured operational semantics [Ch. 7.1] or as an evaluation semantics [Ch. 7.2]. We give two forms of evaluation semantics here, which directly relate an expression to its value.

The first way employs a hypothetical judgment in which we make assumptions about the values of variables. It is written as

$$x_1 {\Downarrow} v_1, \ldots, x_n {\Downarrow} v_n \vdash e \Downarrow v.$$

We call $x_1 {\Downarrow} v_1, \ldots, x_n {\Downarrow} v_n$ an *environment* and denote an environment by η. It is important that all variables x_i in an environment are distinct so that the value of a variable is uniquely determined.

$$\frac{x {\Downarrow} v \in \eta}{\eta \vdash x \Downarrow v} \qquad \frac{}{\eta \vdash \mathtt{num}(k) \Downarrow \mathtt{num}(k)}$$

$$\frac{\eta \vdash e_1 \Downarrow \mathtt{num}(k_1) \quad \eta \vdash e_2 \Downarrow \mathtt{num}(k_2)}{\eta \vdash \mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}(k_1 + k_2)} \quad \frac{\eta \vdash e_1 \Downarrow \mathtt{num}(k_1) \quad \eta \vdash e_2 \Downarrow \mathtt{num}(k_2)}{\eta \vdash \mathtt{times}(e_1, e_2) \Downarrow \mathtt{num}(k_1 \times k_2)}$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, x {\Downarrow} v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow v_2} \ (x \text{ not declared in } \eta)$$

In the rule for `let` we make the assumption that the value of x is v_1 while evaluating e_2. One may be concerned that this operational semantics is partial, in case bound variables with the same name occur nested in a term. However, since we working with α-equivalences classes of terms we can always rename the inner bound variable to that the rule for `let` applies. We will henceforth not make such a side condition explicit, using the general convention that we rename bound variables as necessary so that contexts or environment declare only distinct variables.

An alternative semantics uses substitution instead of environments. For this judgment we evaluate only closed terms, so no hypothetical judgment is needed.

$$\textit{No rule for variables } x \qquad \frac{}{\mathtt{num}(k) \Downarrow \mathtt{num}(k)}$$

$$\frac{e_1 \Downarrow \mathtt{num}(k_1) \quad e_2 \Downarrow \mathtt{num}(k_2)}{\mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}(k_1 + k_2)} \qquad \frac{e_1 \Downarrow \mathtt{num}(k_1) \quad e_2 \Downarrow \mathtt{num}(k_2)}{\mathtt{times}(e_1, e_2) \Downarrow \mathtt{num}(k_1 \times k_2)}$$

$$\frac{e_1 \Downarrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\mathtt{let}(e_1, x.e_2) \Downarrow v_2}$$

We postpone a discussion on the relationship between the two forms of semantics, but we will considered how typing is related to the operational semantics. Clearly, we cannot pick rules arbitrarily, but typing must reflect the operational behavior of programs and, conversely, the operational semantics must reflect typing. As first example of this relationship we show that well-typed and closed arithmetic expressions always evaluate to a value.

If $\cdot \vdash e : \mathtt{nat}$ *then* $\cdot \vdash e \Downarrow \mathtt{num}(k)$ *for some* k.

We cannot prove this directly by induction, since the second premise in the case of the typing rule for $\mathtt{let}(e_1, x.e_2)$ would have the form $\cdot, x{:}\mathtt{nat} \vdash e_2 : \mathtt{nat}$. This does not match the induction hypothesis where the context is required to be empty.

If we look at the rules for typing and evaluation side-by-side, we see that if we start with an empty context and environment, the set of variables in the derivations always correspond. We define that η *matches* Γ if η defines values for the same variables as are declared in Γ.

Lemma 3 (Evaluation in Environment)
If $\Gamma \vdash e : \mathtt{nat}$ *and* η *matches* Γ *then* $\eta \vdash e \Downarrow \mathtt{num}(k)$ *for some* k.

Proof: By rule induction on the derivation of $\Gamma \vdash e : \mathtt{nat}$.

(Rule for $\mathtt{num}(k)$) Then

$\eta \vdash \mathtt{num}(k) \Downarrow \mathtt{num}(k)$	By rule

(Rule for $\mathtt{plus}(e_1, e_2)$) Then

$\Gamma \vdash e_1 : \mathtt{nat}$	Subderivation
$\Gamma \vdash e_2 : \mathtt{nat}$	Subderivation
η matches Γ	Assumption
$\eta \vdash e_1 \Downarrow \mathtt{num}(k_1)$ for some k_1	By i.h.
$\eta \vdash e_2 \Downarrow \mathtt{num}(k_2)$ for some k_2	By i.h.
$\eta \vdash \mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}(k_1 + k_2)$	By rule

(Rule for $\mathtt{times}(e_1, e_2)$) Analogous to previous case.

(Rule for variable x) Then

$x{:}\mathtt{nat} \in \Gamma$	Subderivation
η matches Γ	Assumption
$x{\Downarrow}\mathtt{num}(k)$ for some k	By defn. of matching
$\eta \vdash x \Downarrow \mathtt{num}(k)$	By rule

(Rule for $\mathtt{let}(e_1, x.e_2)$) Then

$\Gamma \vdash e_1 : \mathtt{nat}$	Subderivation
η matches Γ	Assumption
$\eta \vdash e_1 \Downarrow \mathtt{num}(k)$	By i.h.
$\eta, x{\Downarrow}\mathtt{num}(k)$ matches $\Gamma, x{:}\mathtt{nat}$	By defn. of matching
$\Gamma, x{:}\mathtt{nat} \vdash e_2 : \mathtt{nat}$	Subderivation
$\eta, x{\Downarrow}\mathtt{num}(k) \vdash e_2 \Downarrow \mathtt{num}(k_2)$	By i.h.
$\eta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow \mathtt{num}(k_2)$	By rule

■