7 Evaluation Dynamics

In Chapter 5, we defined evaluation of expressions in **E** using a structural dynamics. Structural dynamics is very useful for proving safety, but for some purposes, such as writing a user manual, another formulation, called *evaluation dynamics*, is preferable. An evaluation dynamics is a relation between a phrase and its value that is defined without detailing the step-by-step process of evaluation. A *cost dynamics* enriches an evaluation dynamics with a *cost measure* specifying the resource usage of evaluation. A prime example is time, measured as the number of transition steps required to evaluate an expression according to its structural dynamics.

7.1 Evaluation Dynamics

An *evaluation dynamics* consists of an inductive definition of the evaluation judgment $e \Downarrow v$ stating that the closed expression e evaluates to the value v. The evaluation dynamics of **E** is defined by the following rules:

$$\frac{}{\texttt{num}[n] \Downarrow \texttt{num}[n]} \tag{7.1a}$$

$$\frac{}{\texttt{str}[s] \Downarrow \texttt{str}[s]} \tag{7.1b}$$

$$\frac{e_1 \Downarrow \texttt{num}[n_1] \quad e_2 \Downarrow \texttt{num}[n_2] \quad n_1 + n_2 \text{ is } n \text{ nat}}{\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]} \tag{7.1c}$$

$$\frac{e_1 \Downarrow \texttt{str}[s_1] \quad e_2 \Downarrow \texttt{str}[s_2] \quad s_1 \hat{} s_2 = s \text{ str}}{\texttt{cat}(e_1; e_2) \Downarrow \texttt{str}[s]} \tag{7.1d}$$

$$\frac{e \Downarrow \texttt{str}[s] \quad |s| = n \text{ nat}}{\texttt{len}(e) \Downarrow \texttt{num}[n]} \tag{7.1e}$$

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\texttt{let}(e_1; x.e_2) \Downarrow v_2} \tag{7.1f}$$

The value of a `let` expression is determined by substitution of the binding into the body. The rules are not syntax-directed, because the premise of rule (7.1f) is not a sub-expression of the expression in the conclusion of that rule.

https://doi.org/10.1017/CBO9781316576892.009 Published online by Cambridge University Press

Rule (7.1f) specifies a by-name interpretation of definitions. For a by-value interpretation, the following rule should be used instead:

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v_2}{\texttt{let}(e_1; x.e_2) \Downarrow v_2} \tag{7.2}$$

Because the evaluation judgment is inductively defined, we prove properties of it by rule induction. Specifically, to show that the property $\mathcal{P}(e \Downarrow v)$ holds, it is enough to show that $\mathcal{P}$ is closed under rules (7.1):

1. Show that $\mathcal{P}(\texttt{num}[n] \Downarrow \texttt{num}[n])$.
2. Show that $\mathcal{P}(\texttt{str}[s] \Downarrow \texttt{str}[s])$.
3. Show that $\mathcal{P}(\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n])$, if $\mathcal{P}(e_1 \Downarrow \texttt{num}[n_1])$, $\mathcal{P}(e_2 \Downarrow \texttt{num}[n_2])$, and $n_1 + n_2$ is n nat.
4. Show that $\mathcal{P}(\texttt{cat}(e_1; e_2) \Downarrow \texttt{str}[s])$, if $\mathcal{P}(e_1 \Downarrow \texttt{str}[s_1])$, $\mathcal{P}(e_2 \Downarrow \texttt{str}[s_2])$, and $s_1 \hat{} s_2 = s$ str.
5. Show that $\mathcal{P}(\texttt{let}(e_1; x.e_2) \Downarrow v_2)$, if $\mathcal{P}([e_1/x]e_2 \Downarrow v_2)$.

This induction principle is *not* the same as structural induction on e itself, because the evaluation rules are not syntax-directed.

Lemma 7.1. *If $e \Downarrow v$, then v val.*

Proof By induction on rules (7.1). All cases except rule (7.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the premise of the evaluation rule. □

7.2 Relating Structural and Evaluation Dynamics

We have given two different forms of dynamics for **E**. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The structural dynamics describes a step-by-step process of execution, whereas the evaluation dynamics suppresses the intermediate states, focusing attention on the initial and final states alone. This remark suggests that the right correspondence is between *complete* execution sequences in the structural dynamics and the evaluation judgment in the evaluation dynamics.

Theorem 7.2. *For all closed expressions e and values v, $e \longmapsto^* v$ iff $e \Downarrow v$.*

How might we prove such a theorem? We will consider each direction separately. We consider the easier case first.

Lemma 7.3. *If $e \Downarrow v$, then $e \longmapsto^* v$.*

https://doi.org/10.1017/CBO9781316576892.009 Published online by Cambridge University Press

Proof By induction on the definition of the evaluation judgment. For example, suppose that $\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]$ by the rule for evaluating additions. By induction, we know that $e_1 \longmapsto^* \texttt{num}[n_1]$ and $e_2 \longmapsto^* \texttt{num}[n_2]$. We reason as follows:

$$\begin{array}{rcl} \texttt{plus}(e_1; e_2) & \longmapsto^* & \texttt{plus}(\texttt{num}[n_1]; e_2) \\ & \longmapsto^* & \texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \\ & \longmapsto & \texttt{num}[n_1 + n_2] \end{array}$$

Therefore, $\texttt{plus}(e_1; e_2) \longmapsto^* \texttt{num}[n_1 + n_2]$, as required. The other cases are handled similarly. □

For the converse, recall from Chapter 5 the definitions of multi-step evaluation and complete evaluation. Because $v \Downarrow v$ when v val, it suffices to show that evaluation is closed under converse evaluation:[1]

Lemma 7.4. *If $e \longmapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$.*

Proof By induction on the definition of the transition judgment. For example, suppose that $\texttt{plus}(e_1; e_2) \longmapsto \texttt{plus}(e_1'; e_2)$, where $e_1 \longmapsto e_1'$. Suppose further that $\texttt{plus}(e_1'; e_2) \Downarrow v$, so that $e_1' \Downarrow \texttt{num}[n_1]$, and $e_2 \Downarrow \texttt{num}[n_2]$, and $n_1 + n_2$ is n nat, and v is $\texttt{num}[n]$. By induction $e_1 \Downarrow \texttt{num}[n_1]$, and hence $\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]$, as required. □

7.3 Type Safety, Revisited

Type safety is defined in Chapter 6 as preservation and progress (Theorem 6.1). These concepts are meaningful when applied to a dynamics given by a transition system, as we shall do throughout this book. But what if we had instead given the dynamics as an evaluation relation? How is type safety proved in that case?

The answer, unfortunately, is that we cannot. Although there is an analog of the preservation property for an evaluation dynamics, there is no clear analog of the progress property. Preservation may be stated as saying that if $e \Downarrow v$ and $e : \tau$, then $v : \tau$. It can be readily proved by induction on the evaluation rules. But what is the analog of progress? We might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some v. Although this property is true for **E**, it demands much more than just progress—it requires that every expression evaluate to a value! If **E** were extended to admit operations that may result in an error (as discussed in Section 6.3), or to admit non-terminating expressions, then this property would fail, even though progress would remain valid.

One possible attitude towards this situation is to conclude that type safety cannot be properly discussed in the context of an evaluation dynamics, but only by reference to a structural dynamics. Another point of view is to instrument the dynamics with explicit checks for dynamic type errors, and to show that any expression with a dynamic type fault must be statically ill-typed. Re-stated in the contrapositive, this means that a statically well-typed program cannot incur a dynamic type error. A difficulty with this point of view

https://doi.org/10.1017/CBO9781316576892.009 Published online by Cambridge University Press

is that we must explicitly account for a form of error solely to prove that it cannot arise! Nevertheless, a semblance of type safety can be established using evaluation dynamics.

We define a judgment e err stating that the expression e *goes wrong* when executed. The exact definition of "going wrong" is given by a set of rules, but the intention is that it should cover all situations that correspond to type errors. The following rules are representative of the general case:

$$\frac{}{\mathtt{plus}(\mathtt{str}[s]; e_2)\ \mathsf{err}} \tag{7.3a}$$

$$\frac{e_1\ \mathsf{val}}{\mathtt{plus}(e_1; \mathtt{str}[s])\ \mathsf{err}} \tag{7.3b}$$

These rules explicitly check for the misapplication of addition to a string; similar rules govern each of the primitive constructs of the language.

Theorem 7.5. *If e err, then there is no τ such that $e : \tau$.*

Proof By rule induction on rules (7.3). For example, for rule (7.3a), we note that $\mathtt{str}[s] : \mathtt{str}$, and hence $\mathtt{plus}(\mathtt{str}[s]; e_2)$ is ill-typed. □

Corollary 7.6. *If $e : \tau$, then $\neg(e\ \mathsf{err})$.*

Apart from the inconvenience of having to define the judgment e err only to show that it is irrelevant for well-typed programs, this approach suffers a very significant methodological weakness. If we should omit one or more rules defining the judgment e err, the proof of Theorem 7.5 remains valid; there is nothing to ensure that we have included sufficiently many checks for run-time type errors. We can prove that the ones we define cannot arise in a well-typed program, but we cannot prove that we have covered all possible cases. By contrast the structural dynamics does not specify any behavior for ill-typed expressions. Consequently, any ill-typed expression will "get stuck" without our explicit intervention, and the progress theorem rules out all such cases. Moreover, the transition system corresponds more closely to implementation—a compiler need not make any provisions for checking for run-time type errors. Instead, it relies on the statics to ensure that these cannot arise, and assigns no meaning to any ill-typed program. Therefore, execution is more efficient, and the language definition is simpler.

7.4 Cost Dynamics

A structural dynamics provides a natural notion of *time complexity* for programs, namely the number of steps required to reach a final state. An evaluation dynamics, however, does not provide such a direct notion of time. Because the individual steps required to complete an evaluation are suppressed, we cannot directly read off the number of steps required to evaluate to a value. Instead, we must augment the evaluation relation with a cost measure, resulting in a *cost dynamics*.

https://doi.org/10.1017/CBO9781316576892.009 Published online by Cambridge University Press

Evaluation judgments have the form $e \Downarrow^k v$, with the meaning that e evaluates to v in k steps.

$$\frac{}{\mathtt{num}[n] \Downarrow^0 \mathtt{num}[n]} \tag{7.4a}$$

$$\frac{e_1 \Downarrow^{k_1} \mathtt{num}[n_1] \quad e_2 \Downarrow^{k_2} \mathtt{num}[n_2]}{\mathtt{plus}(e_1; e_2) \Downarrow^{k_1+k_2+1} \mathtt{num}[n_1 + n_2]} \tag{7.4b}$$

$$\frac{}{\mathtt{str}[s] \Downarrow^0 \mathtt{str}[s]} \tag{7.4c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\mathtt{cat}(e_1; e_2) \Downarrow^{k_1+k_2+1} \mathtt{str}[s_1 \hat{} s_2]} \tag{7.4d}$$

$$\frac{[e_1/x]e_2 \Downarrow^{k_2} v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow^{k_2+1} v_2} \tag{7.4e}$$

For a by-value interpretation of `let`, rule (7.4e) is replaced by the following rule:

$$\frac{e_1 \Downarrow^{k_1} v_1 \quad [v_1/x]e_2 \Downarrow^{k_2} v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow^{k_1+k_2+1} v_2} \tag{7.5}$$

Theorem 7.7. *For any closed expression e and closed value v of the same type, $e \Downarrow^k v$ iff $e \longmapsto^k v$.*

Proof From left to right, proceed by rule induction on the definition of the cost dynamics. From right to left, proceed by induction on k, with an inner rule induction on the definition of the structural dynamics. □

7.5 Notes

The structural similarity between evaluation dynamics and typing rules was first developed in *The Definition of Standard ML* (Milner et al., 1997). The advantage of evaluation semantics is its directness; its disadvantage is that it is not well-suited to proving properties such as type safety. Robin Milner introduced the apt phrase "going wrong" as a description of a type error. Cost dynamics was introduced by Blelloch and Greiner (1996) in a study of parallel computation (see Chapter 37).

Exercises

7.1. Show that evaluation is deterministic: if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.

7.2. Complete the proof of Lemma 7.3.

7.3. Complete the proof of Lemma 7.4. Then show that if $e \longmapsto^* e'$ with e' val, then $e \Downarrow e'$.

https://doi.org/10.1017/CBO9781316576892.009 Published online by Cambridge University Press

7.4. Augment the evaluation dynamics with checked errors, along the lines sketched in Chapter 5, using e err to say that e incurs a checked (or an unchecked) error. What remains unsatisfactory about the type safety proof? Can you think of a better alternative?

7.5. Consider generic hypothetical judgments of the form

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

where v_1 val, ..., v_n val, and v val. The hypotheses, written Δ, are called the *environment* of the evaluation; they provide the values of the free variables in e. The hypothetical judgment $\Delta \vdash e \Downarrow v$ is called an *environmental evaluation dynamics*.

Give a hypothetical inductive definition of the environmental evaluation dynamics *without making any use of substitution*. In particular, you should include the rule

$$\frac{}{\Delta, x \Downarrow v \vdash x \Downarrow v}$$

defining the evaluation of a free variable.

Show that $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ iff $[v_1, \ldots, v_n/x_1, \ldots, x_n]e \Downarrow v$ (using the by-value form of evaluation).

Note

1 Converse evaluation is also known as *head expansion*.

https://doi.org/10.1017/CBO9781316576892.009 Published online by Cambridge University Press