Part III

Static and Dynamic Semantics

Chapter 9

Static Semantics

Most programming languages exhibit a *phase distinction* between the *static* and *dynamic* phases of processing. The static phase consists of parsing and type checking to ensure that the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be *safe* exactly when well-formed programs are well-behaved when executed.

The static phase is specified by a *static semantics* comprising a collection of rules for deriving *typing judgements* stating that an expression is well-formed of a certain type. Types mediate the interaction between the constituent parts of a program by "predicting" some aspects of the execution behavior of the parts so that we may ensure they fit together properly at run-time. Type safety tells us that these predictions are accurate; if not, the static semantics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

In this chapter we present the static semantics of the language $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ as an illustration of the methodology that we shall employ throughout this book.

9.1 Type System

Recall that the abstract syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is given by Grammar (8.4), which we repeat here for convenience:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`num`	`num`
		|	`str`	`str`
Expr	e	::=	x	x
		|	`num[`n`]`	n
		|	`str[`s`]`	`"`s`"`
		|	`plus(`e_1`;`e_2`)`	e_1 `+` e_2
		|	`times(`e_1`;`e_2`)`	e_1 `*` e_2
		|	`cat(`e_1`;`e_2`)`	e_1 `^` e_2
		|	`len(`e`)`	`|`e`|`
		|	`let(`e_1`;`x`.`e_2`)`	`let` x `be` e_1 `in` e_2

According to the conventions discussed in Chapter 8, this grammar defines two judgements, τ type defining the category of types, and e exp defining the category of expressions.

The role of a static semantics is to impose constraints on the formations of phrases that are sensitive to the context in which they occur. For example, whether or not the expression `plus(`x`;num[`n`])` is sensib le depends on whether or not the variable x is declared to have type `num` in the surrounding context of the expression. This example is, in fact, illustrative of the general case, in that the *only* information required about the context of an expression is the type of the variables within whose scope the expression lies. Consequently, the static semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ consists of an inductive definition of generic hypothetical judgements of the form

$$\mathcal{X} \mid \Gamma \vdash e : \tau,$$

where $\mathcal{X}$ is a finite set of variables, and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$ with $x \in \mathcal{X}$. We rely on typographical conventions to determine the set of parameters, using the letters x and y for variables that serve as parameters of the generic judgement. We write $x \# \Gamma$ to indicate that there is no assumption in Γ of the form $x : \tau$ for any type τ, in which case we say that the variable x is *fresh* for Γ.

The rules defining the static semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{9.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{9.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{9.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}} \tag{9.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1; e_2) : \texttt{num}} \tag{9.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{str}} \tag{9.1f}$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{num}} \tag{9.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) : \tau_2} \tag{9.1h}$$

In Rule (9.1h) we tacitly assume that the variable, x, is not already declared in Γ. This condition may always be met by choosing a suitable representative of the α-equivalence class of the `let` expression.

Rules (9.1) illustrate an important organizational principle, called the *principle of introduction and elimination*, for a type system. The constructs of the language may be classified into one of two forms associated with each type. The *introductory* forms of a type are the means by which values of that type are created, or introduced. In the case of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, the introductory forms for the type `num` are the numerals, `num[`n`]`, and for the type `str` are the literals, `str[`s`]`. The *eliminatory* forms of a type are the means by which we may compute with values of that type to obtain values of some (possibly different) type. In the present case the eliminatory forms for the type `num` are addition and multiplication, and for the type `str` are concatenation and length. Each eliminatory form has one or more *principal* arguments of associated type, and zero or more *non-principal* arguments. In the present case all arguments for each of the eliminatory forms is principal, but we shall later see examples in which there are also non-principal arguments for eliminatory forms.

It is easy to check that every expression has at most one type.

Lemma 9.1 (Unicity of Typing)**.** *For every typing context Γ and expression e, there exists at most one τ such that $\Gamma \vdash e : \tau$.*

Proof. By rule induction on Rules (9.1). □

The typing rules are *syntax-directed* in the sense that there is exactly one rule for each form of expression. Consequently it is easy to give necessary conditions for typing an expression that invert the sufficient conditions expressed by the corresponding typing rule.

Lemma 9.2 (Inversion for Typing)**.** *Suppose that* $\Gamma \vdash e : \tau$*. If* $e = \texttt{plus}(e_1; e_2)$*, then* $\tau = \texttt{num}$*,* $\Gamma \vdash e_1 : \texttt{num}$*, and* $\Gamma \vdash e_2 : \texttt{num}$*, and similarly for the other constructs of the language.*

Proof. These may all be proved by induction on the derivation of the typing judgement $\Gamma \vdash e : \tau$. □

In richer languages such inversion principles are more difficult to state and to prove.

9.2 Structural Properties

The static semantics enjoys the structural properties of the hypothetical and generic judgements. We will focus our attention here on two key properties, the combination of proliferation (Rule (3.8a)) and weakening (Rule (2.12b)), and substitution, which generalizes transitivity (Rule (2.12c)).

Lemma 9.3 (Weakening)**.** *If* $\Gamma \vdash e' : \tau'$*, then* $\Gamma, x : \tau \vdash e' : \tau'$ *for any* $x \# \Gamma$ *and any* τ *type.*

Proof. By induction on the derivation of $\Gamma \vdash e' : \tau'$. We will give one case here, for rule (9.1h). We have that $e' = \texttt{let}(e_1; z.e_2)$, where by the conventions on parameters we may assume z is chosen such that $z \# \Gamma$ and $z \neq x$. By induction we have

1. $\Gamma, x : \tau \vdash e_1 : \tau_1$,
2. $\Gamma, x : \tau, z : \tau_1 \vdash e_2 : \tau'$,

from which the result follows by Rule (9.1h). □

Lemma 9.4 (Substitution)**.** *If* $\Gamma, x : \tau \vdash e' : \tau'$ *and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash [e/x]e' : \tau'$.

Proof. By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. We again consider only rule (9.1h). As in the preceding case, $e' = \texttt{let}(e_1; z.e_2)$, where z may be chosen so that $z \neq x$ and $z \# \Gamma$. We have by induction

1. $\Gamma \vdash [e/x]e_1 : \tau_1$,
2. $\Gamma, z : \tau_1 \vdash [e/x]e_2 : \tau'$.

By the choice of z we have

$$[e/x]\texttt{let}(e_1; z.e_2) = \texttt{let}([e/x]e_1; z.[e/x]e_2).$$

It follows by Rule (9.1h) that $\Gamma \vdash [e/x]\texttt{let}(e_1; z.e_2) : \tau$, as desired. □

From a programming point of view, Lemma 9.3 on the preceding page allows us to use an expression in any context that binds its free variables: if e is well-typed in a context Γ, then we may "import" it into any context that includes the assumptions Γ. In other words the introduction of new variables beyond those required by an expression, e, does not invalidate e itself; it remains well-formed, with the same type.[1] More significantly, Lemma 9.4 on the facing page expresses the concepts of *modularity* and *linking*. We may think of the expressions e and e' as two *components* of a larger system in which the component e' is to be thought of as a *client* of the *implementation* e. The client declares a variable specifying the type of the implementation, and is type checked knowing only this information. The implementation must be of the specified type in order to satisfy the assumptions of the client. If so, then we may link them to form the composite system, $[e/x]e'$. This may itself be the client of another component, represented by a variable, y, that is replaced by that component during linking. When all such variables have been implemented, the result is a *closed expression* that is ready for execution (evaluation).

The converse of Lemma 9.4 on the preceding page is called *decomposition*. It states that any (large) expression may be decomposed into a client and implementor by introducing a variable to mediate their interaction.

Lemma 9.5 (Decomposition)**.** *If* $\Gamma \vdash [e/x]e' : \tau'$, *then for every type* τ *such that* $\Gamma \vdash e : \tau$, *we have* $\Gamma, x : \tau \vdash e' : \tau'$.

Proof. The typing of $[e/x]e'$ depends only on the type of e wherever it occurs, if at all. □

This lemma tells us that any sub-expression may be isolated as a separate module of a larger system. This is especially useful in when the variable x occurs more than once in e', because then one copy of e suffices for all occurrences of x in e'.

[1] This may seem so obvious as to be not worthy of mention, but, suprisingly, there are useful type systems that lack this property. Since they do not validate the structural principle of weakening, they are called *sub-structural* type systems.

9.3 Exercises

1. Show that the expression $e = \texttt{plus(num[7];str[}abc\texttt{])}$ is ill-typed in that there is no τ such that $e : \tau$.

Chapter 10

Dynamic Semantics

The *dynamic semantics* of a language specifies how programs are to be executed. One important method for specifying dynamic semantics is called *structural semantics*, which consists of a collection of rules defining a transition system whose states are expressions with no free variables. *Contextual semantics* may be viewed as an alternative presentation of the structural semantics of a language. Another important method for specifying dynamic semantics, called *evaluation semantics*, is the subject of Chapter 12.

10.1 Structural Semantics

A structural semantics for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ consists of a transition system whose states are closed expressions, all of which are initial states. The final states are the *closed values*, as defined by the following rules:

$$\frac{}{\texttt{num[}n\texttt{]}\ \mathsf{val}} \tag{10.1a}$$

$$\frac{}{\texttt{str[}s\texttt{]}\ \mathsf{val}} \tag{10.1b}$$

The transition judgement, $e \mapsto e'$, is also inductively defined.

$$\frac{n_1 + n_2 = n\ \mathsf{nat}}{\texttt{plus(num[}n_1\texttt{];num[}n_2\texttt{])} \mapsto \texttt{num[}n\texttt{]}} \tag{10.2a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{;}e_2\texttt{)}} \tag{10.2b}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1;e_2')} \tag{10.2c}$$

$$\frac{s_1 \hat{} s_2 = s \text{ str}}{\texttt{cat}(\texttt{str}[s_1];\texttt{str}[s_2]) \mapsto \texttt{str}[s]} \tag{10.2d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{cat}(e_1;e_2) \mapsto \texttt{cat}(e_1';e_2)} \tag{10.2e}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\texttt{cat}(e_1;e_2) \mapsto \texttt{cat}(e_1;e_2')} \tag{10.2f}$$

$$\frac{}{\texttt{let}(e_1;x.e_2) \mapsto [e_1/x]e_2} \tag{10.2g}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (10.2a), (10.2d), and (10.2g) are *instruction transitions*, since they correspond to the primitive steps of evaluation. The remaining rules are *search transitions* that determine the order in which instructions are executed.

Rules (10.2) exhibit structure arising from the principle of introduction and elimination discussed in Chapter 9. The instruction transitions express the *inversion principle*, which states that *eliminatory forms are inverse to introductory forms*. For example, Rule (10.2a) extracts the natural number from the introductory forms of its arguments, adds these two numbers, and yields the corresponding numeral as result. The search transitions specify that the principal arguments of each eliminatory form are to be evaluated. (When non-principal arguments are present, which is not the case here, there is discretion about whether to evaluate them or not.) This is essential, because it prepares for the instruction transitions, which expect their principal arguments to be introductory forms.

Rule (10.2g) specifies a *by-name* interpretation, in which the bound variable stands for the expression e_1 itself.[1] If x does not occur in e_2, the expression e_1 is never evaluated. If, on the other hand, it occurs more than once, then e_1 will be re-evaluated at each occurence. To avoid repeated work in the latter case, we may instead specify a *by-value* interpretation of binding by the following rules:

$$\frac{e_1 \text{ val}}{\texttt{let}(e_1;x.e_2) \mapsto [e_1/x]e_2} \tag{10.3a}$$

[1] The justification for the terminology "by name" is obscure, but as it is very well-established we shall stick with it.

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1;x.e_2) \mapsto \texttt{let}(e_1';x.e_2)} \tag{10.3b}$$

Rule (10.3b) is an additional search rule specifying that we may evaluate e_1 before e_2. Rule (10.3a) ensures that e_2 is not evaluated until evaluation of e_1 is complete.

A derivation sequence in a structural semantics has a two-dimensional structure, with the number of steps in the sequence being its "width" and the derivation tree for each step being its "height." For example, consider the following evaluation sequence.

$$\begin{aligned}
&\texttt{let(plus(num[1];num[2]);}x\texttt{.plus(plus(}x\texttt{;num[3]);num[4]))}\\
&\quad\mapsto\quad \texttt{let(num[3];}x\texttt{.plus(plus(}x\texttt{;num[3]);num[4]))}\\
&\quad\mapsto\quad \texttt{plus(plus(num[3];num[3]);num[4])}\\
&\quad\mapsto\quad \texttt{plus(num[6];num[4])}\\
&\quad\mapsto\quad \texttt{num[10]}
\end{aligned}$$

Each step in this sequence of transitions is justified by a derivation according to Rules (10.2). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\texttt{plus(num[3];num[3])} \mapsto \texttt{num[6]}}\ (10.2a)}{\texttt{plus(plus(num[3];num[3]);num[4])} \mapsto \texttt{plus(num[6];num[4])}}\ (10.2b)$$

The other steps are similarly justified by a composition of rules.

The principle of rule induction for the structural semantics of $\mathcal{L}\{\texttt{num str}\}$ states that to show $\mathcal{P}(e \mapsto e')$ whenever $e \mapsto e'$, it is sufficient to show that $\mathcal{P}$ is closed under Rules (10.2). For example, we may show by rule induction that structural semantics of $\mathcal{L}\{\texttt{num str}\}$ is *determinate*.

Lemma 10.1 (Determinacy)**.** *If $e \mapsto e'$ and $e \mapsto e''$, then e' and e'' are α-equivalent.*

Proof. By rule induction on the premises $e \mapsto e'$ and $e \mapsto e''$, carried out either simultaneously or in either order. Since only one rule applies to each form of expression, e, the result follows directly in each case. □

10.2 Contextual Semantics

A variant of structural semantics, called *contextual semantics*, is sometimes useful. There is no fundamental difference between the two approaches,

only a difference in the style of presentation. The main idea is to isolate instruction steps as a special form of judgement, called *instruction transition*, and to formalize the process of locating the next instruction using a device called an *evaluation context*. The judgement, e val, defining whether an expression is a value, remains unchanged.

The instruction transition judgement, $e_1 \rightsquigarrow e_2$, for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

$$\frac{m + n = p\ \mathsf{nat}}{\texttt{plus(num[}m\texttt{];num[}n\texttt{])} \rightsquigarrow \texttt{num[}p\texttt{]}} \tag{10.4a}$$

$$\frac{s \hat{} t = u\ \mathsf{str}}{\texttt{cat(str[}s\texttt{];str[}t\texttt{])} \rightsquigarrow \texttt{str[}u\texttt{]}} \tag{10.4b}$$

$$\frac{}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \rightsquigarrow [e_1/x]e_2} \tag{10.4c}$$

The judgement $\mathcal{E}$ ectxt determines the location of the next instruction to execute in a larger expression. The position of the next instruction step is specified by a "hole", written $\circ$, into which the next instruction is placed, as we shall detail shortly. (The rules for multiplication and length are omitted for concision, as they are handled similarly.)

$$\frac{}{\circ\ \mathsf{ectxt}} \tag{10.5a}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\texttt{plus(}\mathcal{E}_1\texttt{;}e_2\texttt{)}\ \mathsf{ectxt}} \tag{10.5b}$$

$$\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\texttt{plus(}e_1\texttt{;}\mathcal{E}_2\texttt{)}\ \mathsf{ectxt}} \tag{10.5c}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\texttt{let(}\mathcal{E}_1\texttt{;}x\texttt{.}e_2\texttt{)}\ \mathsf{ectxt}} \tag{10.5d}$$

The first rule for evaluation contexts specifies that the next instruction may occur "here", at the point of the occurrence of the hole. The remaining rules correspond one-for-one to the search rules of the structural semantics. For example, Rule (10.5c) states that in an expression $\texttt{plus(}e_1\texttt{;}e_2\texttt{)}$, if the first principal argument, e_1, is a value, then the next instruction step, if any, lies at or within the second principal argument, e_2.

An evaluation context is to be thought of as a template that is instantiated by replacing the hole with an instruction to be executed. The judgement $e' = \mathcal{E}\{e\}$ states that the expression e' is the result of filling the hole

in the evaluation context $\mathcal{E}$ with the expression e. It is inductively defined by the following rules:

$$\frac{}{e = \circ\{e\}} \tag{10.6a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\mathtt{plus}(e_1; e_2) = \mathtt{plus}(\mathcal{E}_1; e_2)\{e\}} \tag{10.6b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\mathtt{plus}(e_1; e_2) = \mathtt{plus}(e_1; \mathcal{E}_2)\{e\}} \tag{10.6c}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\mathtt{let}(e_1; x.e_2) = \mathtt{let}(\mathcal{E}_1; x.e_2)\{e\}} \tag{10.6d}$$

There is one rule for each form of evaluation context. Filling the hole with e results in e; otherwise we proceed inductively over the structure of the evaluation context.

Finally, the dynamic semantics for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ is defined using contextual semantics by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \rightsquigarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'} \tag{10.7}$$

Thus, a transition from e to e' consists of (1) decomposing e into an evaluation context and an instruction, (2) execution of that instruction, and (3) replacing the instruction by the result of its execution in the same spot within e to obtain e'.

The structural and contextual semantics define the same transition relation. For the sake of the proof, let us write $e \mapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural semantics (Rules (10.2)), and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual semantics (Rules (10.7)).

Theorem 10.2. *$e \mapsto_{\mathsf{s}} e'$ if, and only if, $e \mapsto_{\mathsf{c}} e'$.*

Proof. From left to right, proceed by rule induction on Rules (10.2). It is enough in each case to exhibit an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. For example, for Rule (10.2a), take $\mathcal{E} = \circ$, and observe that $e \rightsquigarrow e'$. For Rule (10.2b), we have by induction that there exists an evaluation context $\mathcal{E}_1$ such that $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. Take $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, and observe that $e = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0\}$ and $e' = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0'\}$ with $e_0 \rightsquigarrow e_0'$.

From right to left, observe that if $e \mapsto_{\mathsf{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. We prove by

induction on Rules (10.6) that $e \mapsto_{\mathtt{s}} e'$. For example, for Rule (10.6a), e_0 is e, e_0' is e', and $e \rightsquigarrow e'$. Hence $e \mapsto_{\mathtt{s}} e'$. For Rule (10.6b), we have that $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \mapsto_{\mathtt{s}} e_1'$. Therefore e is $\mathtt{plus}(e_1; e_2)$, e' is $\mathtt{plus}(e_1'; e_2)$, and therefore by Rule (10.2b), $e \mapsto_{\mathtt{s}} e'$.

□

Since the two transition judgements coincide, contextual semantics may be seen as an alternative way of presenting a structural semantics. It has two advantages over structural semantics, one relatively superficial, one rather less so. The superficial advantage stems from writing Rule (10.7) in the simpler form

$$\frac{e_0 \rightsquigarrow e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}}\ . \tag{10.8}$$

This formulation is simpler insofar as it leaves implicit the definition of the decomposition of the left- and right-hand sides. The deeper advantage, which we will exploit in Chapter 15, is that the transition judgement in contextual semantics applies only to closed expressions of a *fixed* type, whereas structural semantics transitions are necessarily defined over expressions of *every* type.

10.3 Equational Semantics

Another formulation of the dynamic semantics of a language is based on regarding computation as a form of equational deduction, much in the style of elementary algebra. For example, in algebra we may show that the polynomials $x^2 + 2\,x + 1$ and $(x + 1)^2$ are equivalent by a simple process of calculation and re-organization using the familiar laws of addition and multiplication. The same laws are sufficient to determine the value of any polynomial, given the values of its variables. So, for example, we may plug in 2 for x in the polynomial $x^2 + 2\,x + 1$ and calculate that $2^2 + 2\,2 + 1 = 9$, which is indeed $(2 + 1)^2$. This gives rise to a model of computation in which we may determine the value of a polynomial for a given value of its variable by substituting the given value for the variable and proving that the resulting expression is equal to its value.

Very similar ideas give rise to the concept of *definitional*, or *computational*, *equivalence* of expressions in $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, which we write as $\mathcal{X} \mid \Gamma \vdash e \equiv e' : \tau$, where Γ consists of one assumption of the form $x : \tau$ for each $x \in \mathcal{X}$. We only consider definitional equality of well-typed expressions, so that when considering the judgement $\Gamma \vdash e \equiv e' : \tau$, we tacitly assume

that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$. Here, as usual, we omit explicit mention of the parameters, $\mathcal{X}$, when they can be determined from the forms of the assumptions Γ.

Definitional equivalence of expressons in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{10.9a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{10.9b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{10.9c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) \equiv \texttt{plus}(e_1'; e_2') : \texttt{num}} \tag{10.9d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) \equiv \texttt{cat}(e_1'; e_2') : \texttt{str}} \tag{10.9e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv \texttt{let}(e_1'; x.e_2') : \tau_2} \tag{10.9f}$$

$$\frac{n_1 + n_2 = n \text{ nat}}{\Gamma \vdash \texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \equiv \texttt{num}[n] : \texttt{num}} \tag{10.9g}$$

$$\frac{s_1 \hat{} s_2 = s \text{ str}}{\Gamma \vdash \texttt{cat}(\texttt{str}[s_1]; \texttt{str}[s_2]) \equiv \texttt{str}[s] : \texttt{str}} \tag{10.9h}$$

$$\frac{}{\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{10.9i}$$

Rules (10.9a) through (10.9c) state that definitional equivalence is an *equivalence relation*. Rules (10.9d) through (10.9f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (10.9g) through (10.9i) specify the meanings of the primitive constructs of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$. For the sake of concision, Rules (10.9) may be characterized as defining the *strongest congruence* closed under Rules (10.9g), (10.9h), and (10.9i).

Rules (10.9) are sufficient to allow us to calculate the value of an expression by an equational deduction similar to that used in high school algebra. For example, we may derive the equation

$$\texttt{let}\, x\, \texttt{be}\, 1 + 2\, \texttt{in}\, x + 3 + 4 \equiv 10 : \texttt{num}$$

by applying Rules (10.9). Here, as in general, there may be many different ways to derive the same equation, but we need find only one derivation in order to carry out an evaluation.

Definitional equivalence is rather weak in that many equivalences that one might intuitively think are true are not derivable from Rules (10.9). A prototypical example is the putative equivalence

$$x : \mathtt{num}, y : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{10.10}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from Rules (10.9). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{10.11}$$

where n_1 nat and n_2 nat are particular numbers.

The "gap" between a general law, such as Equation (10.10), and all of its instances, given by Equation (10.11), may be filled by enriching the notion of equivalence to include a principal of proof by mathematical induction. Such a notion of equivalence is sometimes called *semantic*, or *observational*, *equivalence*, since it expresses relationships that hold by virtue of the semantics of the expressions involved.[2] Semantic equivalence is a *synthetic judgement*, one that requires proof. It is to be distinguished from definitional equivalence, which expresses an *analytic judgement*, one that is self-evident based solely on the dynamic semantics of the operations involved. As such definitional equivalence may be thought of as *symbolic evaluation*, which permits simplification according to the evaluation rules of a language, but which does not permit reasoning by induction.

Definitional equivalence is adequate for evaluation in that it permits the calculation of the value of any closed expression.

Theorem 10.3. *$e \equiv e' : \tau$ iff there exists e_0 val such that $e \mapsto^* e_0$ and $e' \mapsto^* e_0$.*

Proof. The proof from right to left is direct, since every transition step is a valid equation. The converse follows from the following, more general, proposition. If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \equiv e' : \tau$, then whenever $e_1 : \tau_1, \ldots, e_n : \tau_n$, if

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \equiv [e_1, \ldots, e_n / x_1, \ldots, x_n]e' : \tau,$$

then there exists e_0 val such that

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \mapsto^* e_0$$

[2] This rather vague concept of equivalence is developed rigorously in Chapter 50.

and

$$[e_1, \ldots, e_n / x_1, \ldots, x_n] e' \mapsto^* e_0.$$

This is proved by rule induction on Rules (10.9). □

The formulation of definitional equivalence for the by-value semantics of binding requires a bit of additional machinery. The key idea is motivated by the modifications required to Rule (10.9i) to express the requirement that e_1 be a value. As a first cut one might consider simply adding an additional premise to the rule:

$$\frac{e_1 \ \mathsf{val}}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{10.12}$$

This is almost correct, except that the judgement e val is defined only for *closed* expressions, whereas e_1 might well involve free variables in Γ. What is required is to extend the judgement e val to the hypothetical judgement

$$x_1 \ \mathsf{val}, \ldots, x_n \ \mathsf{val} \vdash e \ \mathsf{val}$$

in which the hypotheses express the assumption that variables are only ever bound to values, and hence can be regarded as values. To maintain this invariant, we must maintain a set, Ξ, of such hypotheses as part of definitional equivalence, writing $\Xi\,\Gamma \vdash e \equiv e' : \tau$, and modifying Rule (10.9f) as follows:

$$\frac{\Xi\,\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Xi, x \ \mathsf{val}\,\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\,\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv \mathtt{let}(e_1'; x.e_2') : \tau_2} \tag{10.13}$$

The other rules are correspondingly modified to simply carry along Ξ is an additional set of hypotheses of the inference.

10.4 Exercises

1. For the structural operational semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, prove that if $e \mapsto e_1$ and $e \mapsto e_2$, then $e_1 =_\alpha e_2$.

2. Formulate a variation of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ with both a by-name and a by-value `let` construct.

Chapter 11

Type Safety

Most contemporary programming languages are *safe* (or, *type safe*, or *strongly typed*). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, type safety for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ states that it will never arise that a number is to be added to a string, or that two numbers are to be concatenated, neither of which is meaningful.

In general type safety expresses the coherence between the static and the dynamic semantics. The static semantics may be seen as predicting that the value of an expression will have a certain form so that the dynamic semantics of that expression is well-defined. Consequently, evaluation cannot "get stuck" in a state for which no transition is possible, corresponding in implementation terms to the absence of "illegal instruction" errors at execution time. This is proved by showing that each step of transition preserves typability and by showing that typable states are well-defined. Consequently, evaluation can never "go off into the weeds," and hence can never encounter an illegal instruction.

More precisely, type safety for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ may be stated as follows:

Theorem 11.1 (Type Safety)**.** *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.

The first part, called *preservation*, says that the steps of evaluation preserve typing; the second, called *progress*, ensures that well-typed expressions are either values or can be further evaluated. Safety is the conjunction of preservation and progress.

We say that an expression, e, is *stuck* iff it is not a value, yet there is no e' such that $e \mapsto e'$. It follows from the safety theorem that a stuck state is

necessarily ill-typed. Or, putting it the other way around, that well-typed states do not get stuck.

11.1 Preservation

The preservation theorem for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ defined in Chapters 9 and 10 is proved by rule induction on the transition system (rules (10.2)).

Theorem 11.2 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. We will consider two cases, leaving the rest to the reader. Consider rule (10.2b),

$$\frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1';e_2)}\ .$$

Assume that $\texttt{plus}(e_1;e_2) : \tau$. By inversion for typing, we have that $\tau = \texttt{num}$, $e_1 : \texttt{num}$, and $e_2 : \texttt{num}$. By induction we have that $e_1' : \texttt{num}$, and hence $\texttt{plus}(e_1';e_2) : \texttt{num}$. The case for concatenation is handled similarly.

Now consider rule (10.2g),

$$\frac{e_1\ \mathsf{val}}{\texttt{let}(e_1;x.e_2) \mapsto [e_1/x]e_2}\ .$$

Assume that $\texttt{let}(e_1;x.e_2) : \tau_2$. By the inversion lemma 9.2 on page 76, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 9.4 on page 76 $[e_1/x]e_2 : \tau_2$, as desired. □

The proof of preservation is naturally structured as an induction on the transition judgement, since the argument hinges on examining all possible transitions from a given expression. In some cases one may manage to carry out a proof by structural induction on e, or by an induction on typing, but experience shows that this often leads to awkward arguments, or, in some cases, cannot be made to work at all.

11.2 Progress

The progress theorem captures the idea that well-typed programs cannot "get stuck". The proof depends crucially on the following lemma, which characterizes the values of each type.

Lemma 11.3 (Canonical Forms)**.** *If e val and $e : \tau$, then*

1. *If $\tau = \texttt{num}$, then $e = \texttt{num[}n\texttt{]}$ for some number n.*
2. *If $\tau = \texttt{str}$, then $e = \texttt{str[}s\texttt{]}$ for some string s.*

Proof. By induction on rules (9.1) and (10.1). □

Progress is proved by rule induction on rules (9.1) defining the static semantics of the language.

Theorem 11.4 (Progress)**.** *If $e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.*

Proof. The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (9.1d),

$$\frac{e_1 : \texttt{num} \quad e_2 : \texttt{num}}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} : \texttt{num}},$$

where the context is empty because we are considering only closed terms.

By induction we have that either e_1 val, or there exists e_1' such that $e_1 \mapsto e_1'$. In the latter case it follows that $\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{;}e_2\texttt{)}$, as required. In the former we also have by induction that either e_2 val, or there exists e_2' such that $e_2 \mapsto e_2'$. In the latter case we have that $\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1\texttt{;}e_2'\texttt{)}$, as required. In the former, we have, by the Canonical Forms Lemma 11.3 on the facing page, $e_1 = \texttt{num[}n_1\texttt{]}$ and $e_2 = \texttt{num[}n_2\texttt{]}$, and hence

$$\texttt{plus(num[}n_1\texttt{];num[}n_2\texttt{])} \mapsto \texttt{num[}n_1 + n_2\texttt{]}.$$

□

Since the typing rules for expressions are syntax-directed, the progress theorem could equally well be proved by induction on the structure of e, appealing to the inversion theorem at each step to characterize the types of the parts of e. But this approach breaks down when the typing rules are not syntax-directed, that is, when there may be more than one rule for a given expression form. No difficulty arises if the proof proceeds by induction on the typing rules.

Summing up, the combination of preservation and progress together constitute the proof of safety. The progress theorem ensures that well-typed expressions do not "get stuck" in an ill-defined state, and the preservation theorem ensures that if a step is taken, the result remains well-typed (with the same type). Thus the two parts work hand-in-hand to ensure that the static and dynamic semantics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.

11.3 Run-Time Errors

Suppose that we wish to extend $\mathcal{L}\{\texttt{num str}\}$ with, say, a quotient operation that is undefined for a zero divisor. The natural typing rule for quotients is given by the following rule:

$$\frac{e_1 : \texttt{num} \quad e_2 : \texttt{num}}{\texttt{div}(e_1;e_2) : \texttt{num}}\ .$$

But the expression `div(num[3];num[0])` is well-typed, yet stuck! We have two options to correct this situation:

1. Enhance the type system, so that no well-typed program may divide by zero.

2. Add dynamic checks, so that division by zero signals an error as the outcome of evaluation.

Either option is, in principle, viable, but the most common approach is the second. The first requires that the type checker prove that an expression be non-zero before permitting it to be used in the denominator of a quotient. It is difficult to do this without ruling out too many programs as ill-formed. This is because one cannot reliably predict statically whether an expression will turn out to be non-zero when executed (because this is an undecidable property). We therefore consider the second approach, which is typical of current practice.

The general idea is to distinguish *checked* from *unchecked* errors. An unchecked error is one that is ruled out by the type system. No run-time checking is performed to ensure that such an error does not occur, because the type system rules out the possibility of it arising. For example, the dynamic semantics need not check, when performing an addition, that its two arguments are, in fact, numbers, as opposed to strings, because the type system ensures that this is the case. On the other hand the dynamic semantics for quotient *must* check for a zero divisor, because the type system does not rule out the possibility.

One approach to modelling checked errors is to give an inductive definition of the judgment e err stating that the expression e incurs a checked run-time error, such as division by zero. Here are some representative rules that would appear in a full inductive definition of this judgement:

$$\frac{e_1 \ \mathsf{val}}{\texttt{div}(e_1;\texttt{num[0]})\ \mathsf{err}} \tag{11.1a}$$

$$\frac{e_1 \ \mathsf{err}}{\mathtt{plus}(e_1;e_2) \ \mathsf{err}} \tag{11.1b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{err}}{\mathtt{plus}(e_1;e_2) \ \mathsf{err}} \tag{11.1c}$$

Rule (11.1a) signals an error condition for division by zero. The other rules propagate this error upwards: if an evaluated sub-expression is a checked error, then so is the overall expression.

The preservation theorem is not affected by the presence of checked errors. However, the statement (and proof) of progress is modified to account for checked errors.

Theorem 11.5 (Progress With Error)**.** *If $e : \tau$, then either e err, or e val, or there exists e' such that $e \mapsto e'$.*

Proof. The proof is by induction on typing, and proceeds similarly to the proof given earlier, except that there are now three cases to consider at each point in the proof. □

A disadvantage of this approach to the formalization of error checking is that it appears to require a special set of evaluation rules to check for errors. An alternative is to fold in error checking with evaluation by enriching the language with a special error expression, `error`, which signals that an error has arisen. Since an error condition aborts the computation, the static semantics assigns an arbitrary type to `error`:

$$\frac{}{\mathtt{error} : \tau} \tag{11.2}$$

This rule destroys the unicity of typing property (Lemma 9.1 on page 75). This can be restored by introducing a special error expression for each type, but we shall not do so here for the sake of simplicity.

The dynamic semantics is augmented with rules that provoke a checked error (such as division by zero), plus rules that propagate the error through other language constructs.

$$\frac{e_1 \ \mathsf{val}}{\mathtt{div}(e_1;\mathtt{num[0]}) \mapsto \mathtt{error}} \tag{11.3a}$$

$$\frac{}{\mathtt{plus}(\mathtt{error};e_2) \mapsto \mathtt{error}} \tag{11.3b}$$

$$\frac{e_1 \ \mathsf{val}}{\mathtt{plus}(e_1;\mathtt{error}) \mapsto \mathtt{error}} \tag{11.3c}$$

There are similar error propagation rules for the other constructs of the language. By defining e err to hold exactly when $e = \texttt{error}$, the revised progress theorem continues to hold for this variant semantics.

11.4 Exercises

1. Complete the proof of preservation.
2. Complete the proof of progress.

Chapter 12

Evaluation Semantics

In Chapter 10 we defined the dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ using the method of structural semantics. This approach is useful as a foundation for proving properties of a language, but other methods are often more appropriate for other purposes, such as writing user manuals. Another method, called *evaluation semantics*, or *ES*, presents the dynamic semantics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Two variants of evaluation semantics are also considered, namely *environment semantics*, which delays substitution, and *cost semantics*, which records the number of steps that are required to evaluate an expression.

12.1 Evaluation Semantics

Another method for defining the dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, called *evaluation semantics*, consists of an inductive definition of the evaluation judgement, $e \Downarrow v$, stating that the closed expression, e, evaluates to the value, v.

$$\frac{}{\texttt{num}[n] \Downarrow \texttt{num}[n]} \tag{12.1a}$$

$$\frac{}{\texttt{str}[s] \Downarrow \texttt{str}[s]} \tag{12.1b}$$

$$\frac{e_1 \Downarrow \texttt{num}[n_1] \quad e_2 \Downarrow \texttt{num}[n_2] \quad n_1 + n_2 = n \ \mathsf{nat}}{\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]} \tag{12.1c}$$

$$\frac{e_1 \Downarrow \texttt{str}[s_1] \quad e_2 \Downarrow \texttt{str}[s_2] \quad s_1 \hat{} s_2 = s \ \mathsf{str}}{\texttt{cat}(e_1; e_2) \Downarrow \texttt{str}[s]} \tag{12.1d}$$

$$\frac{e \Downarrow \mathtt{str}[s] \quad |s| = n \ \mathsf{str}}{\mathtt{len}(e) \Downarrow \mathtt{num}[n]} \tag{12.1e}$$

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow v_2} \tag{12.1f}$$

The value of a `let` expression is determined by substitution of the binding into the body. The rules are therefore not syntax-directed, since the premise of Rule (12.1f) is not a sub-expression of the expression in the conclusion of that rule.

The evaluation judgement 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 (12.1):

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

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

Lemma 12.1. *If $e \Downarrow v$, then v val.*

Proof. By induction on Rules (12.1). All cases except Rule (12.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the second premise of the evaluation rule. □

12.2 Relating Transition and Evaluation Semantics

We have given two different forms of dynamic semantics for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$. 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 transition semantics describes a step-by-step process of execution, whereas the evaluation semantics suppresses the intermediate states, focussing attention on

the initial and final states alone. This suggests that the appropriate correspondence is between *complete* execution sequences in the transition semantics and the evaluation judgement in the evaluation semantics. (We will consider only numeric expressions, but analogous results hold also for string-valued expressions.)

Theorem 12.2. *For all closed expressions e and values v, $e \mapsto^* v$ iff $e \Downarrow v$.*

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

Lemma 12.3. *If $e \Downarrow v$, then $e \mapsto^* v$.*

Proof. By induction on the definition of the evaluation judgement. 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 $e_1 \mapsto^* \texttt{num}[n_1]$ and $e_2 \mapsto^* \texttt{num}[n_2]$. We reason as follows:

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

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

For the converse, recall from Chapter 4 the definitions of multi-step evaluation and complete evaluation. Since $v \Downarrow v$ whenever v val, it suffices to show that evaluation is closed under reverse execution.

Lemma 12.4. *If $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$.*

Proof. By induction on the definition of the transition judgement. For example, suppose that $\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1';e_2)$, where $e_1 \mapsto e_1'$. Suppose further that $\texttt{plus}(e_1';e_2) \Downarrow v$, so that $e_1' \Downarrow \texttt{num}[n_1]$, $e_2 \Downarrow \texttt{num}[n_2]$, $n_1 + n_2 = n$ 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. □

12.3 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\texttt{num str}\}$ (Theorem 11.1 on page 89) states that a language is safe iff it satisfies both preservation and progress. This formulation depends critically on the use of a transition system to specify the dynamic semantics. But what if we had instead specified the dynamic

semantics as an evaluation relation, instead of using a transition system? Can we state and prove safety in such a setting?

The answer, unfortunately, is that we cannot. While there is an analogue of the preservation property for an evaluation semantics, there is no clear analogue of the progress property. Preservation may be stated as saying that if $e \Downarrow v$ and $e : \tau$, then $v : \tau$. This can be readily proved by induction on the evaluation rules. But what is the analogue of progress? One might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some v. While this property is true for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, it demands much more than just progress — it requires that every expression evaluate to a value! If $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ were extended to admit operations that may result in an error (as discussed in Section 11.3 on page 92), 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 simply conclude that type safety cannot be properly discussed in the context of an evaluation semantics, but only by reference to a transition semantics. Another point of view is to instrument the semantics with explicit checks for run-time type errors, and to show that any expression with a type fault must be ill-typed. Re-stated in the contrapositive, this means that a well-typed program cannot incur a type error. A difficulty with this point of view is that one must explicitly account for a class of errors solely to prove that they cannot arise! Nevertheless, we will press on to show how a semblance of type safety can be established using evaluation semantics.

The main idea is to define a judgement $e \Uparrow$ stating, in the jargon of the literature, 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:

$$\frac{}{\texttt{plus(str[}s\texttt{];}e_2\texttt{)} \Uparrow} \tag{12.2a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{plus(}e_1\texttt{;str[}s\texttt{])} \Uparrow} \tag{12.2b}$$

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

Theorem 12.5. *If $e \Uparrow$, then there is no τ such that $e : \tau$.*

Proof. By rule induction on Rules (12.2). For example, for Rule (12.2a), we observe that $\texttt{str[}s\texttt{]} : \texttt{str}$, and hence $\texttt{plus(str[}s\texttt{];}e_2\texttt{)}$ is ill-typed. □

Corollary 12.6. *If $e : \tau$, then $\neg(e \Uparrow)$.*

Apart from the inconvenience of having to define the judgement $e \Uparrow$ 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 judgement $e \Uparrow$, the proof of Theorem 12.5 on the facing page 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 transition semantics 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 static semantics to ensure that these cannot arise, and assigns no meaning to any ill-typed program. Execution is therefore more efficient, and the language definition is simpler, an elegant win-win situation for both the semantics and the implementation.

12.4 Cost Semantics

A structural semantics provides a natural notion of *time complexity* for programs, namely the number of steps required to reach a final state. An evaluation semantics, on the other hand, does not provide such a direct notion of complexity. Since 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 semantics*.

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

$$\frac{}{\mathtt{num[}n\mathtt{]} \Downarrow^0 \mathtt{num[}n\mathtt{]}} \tag{12.3a}$$

$$\frac{e_1 \Downarrow^{k_1} \mathtt{num[}n_1\mathtt{]} \quad e_2 \Downarrow^{k_2} \mathtt{num[}n_2\mathtt{]}}{\mathtt{plus(}e_1\mathtt{;}e_2\mathtt{)} \Downarrow^{k_1+k_2+1} \mathtt{num[}n_1 + n_2\mathtt{]}} \tag{12.3b}$$

$$\frac{}{\mathtt{str[}s\mathtt{]} \Downarrow^0 \mathtt{str[}s\mathtt{]}} \tag{12.3c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\texttt{cat}(e_1;e_2) \Downarrow^{k_1+k_2+1} \texttt{str}[s_1\,\hat{}\,s_2]} \tag{12.3d}$$

$$\frac{[e_1/x]e_2 \Downarrow^{k_2} v_2}{\texttt{let}(e_1;x.e_2) \Downarrow^{k_2+1} v_2} \tag{12.3e}$$

Theorem 12.7. *For any closed expression e and closed value v of the same type, $e \Downarrow^k v$ iff $e \mapsto^k v$.*

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

12.5 Environment Semantics

Both the transition semantics and the evaluation semantics given earlier rely on substitution to replace `let`-bound variables by their bindings during evaluation. This approach maintains the invariant that only closed expressions are ever considered. However, in practice, we do not perform substitution, but rather record the bindings of variables in a data structure where they may be retrieved on demand. In this section we show how this can be expressed for a by-value interpretation of binding using hypothetical judgements. It is also possible to formulate an environment semantics for the by-name interpretation, at the cost of some additional complexity (see Chapter 42 for a full discussion of the issues involved).

The basic idea is to consider hypotheses of the form $x \Downarrow v$, where x is a variable and v is a closed value, such that no two hypotheses govern the same variable. Let Θ range over finite sets of such hypotheses, which we call an *environment*. We will consider judgements of the form $\Theta \vdash e \Downarrow v$, where Θ is an environment governing some finite set of variables.

$$\frac{}{\Theta, x \Downarrow v \vdash x \Downarrow v} \tag{12.4a}$$

$$\frac{\Theta \vdash e_1 \Downarrow \texttt{num}[n_1] \quad \Theta \vdash e_2 \Downarrow \texttt{num}[n_2]}{\Theta \vdash \texttt{plus}(e_1;e_2) \Downarrow \texttt{num}[n_1 + n_2]} \tag{12.4b}$$

$$\frac{\Theta \vdash e_1 \Downarrow \texttt{str}[s_1] \quad \Theta \vdash e_2 \Downarrow \texttt{str}[s_2]}{\Theta \vdash \texttt{cat}(e_1;e_2) \Downarrow \texttt{str}[s_1\,\hat{}\,s_2]} \tag{12.4c}$$

$$\frac{\Theta \vdash e_1 \Downarrow v_1 \quad \Theta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\Theta \vdash \texttt{let}(e_1;x.e_2) \Downarrow v_2} \tag{12.4d}$$

Rule (12.4a) is an instance of the general reflexivity rule for hypothetical judgements. The `let` rule augments the environment with a new assumption governing the bound variable, which may be chosen to be distinct from all other variables in Θ to avoid multiple assumptions for the same variable.

The environment semantics implements evaluation by deferred substitution.

Theorem 12.8. $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ *iff* $[v_1, \ldots, v_n / x_1, \ldots, x_n]e \Downarrow v$.

Proof. The left to right direction is proved by induction on the rules defining the evaluation semantics, making use of the definition of substitution and the definition of the evaluation semantics for closed expressions. The converse is proved by induction on the structure of e, again making use of the definition of substitution. Note that we must induct on e in order to detect occurrences of variables x_i in e, which are governed by a hypothesis in the environment semantics. □

12.6 Exercises

1. Prove that if $e \Downarrow v$, then v val.
2. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.
3. Complete the proof of equivalence of evaluation and transition semantics.
4. Prove preservation for the instrumented evaluation semantics, and conclude that well-typed programs cannot go wrong.
5. Is it possible to use environments in a structural semantics? What difficulties do you encounter?