Chapter 6

Statics

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 *statics* 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 statics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

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

6.1 Syntax

When defining a language we shall be primarily concerned with its abstract syntax, specified by a collection of operators and their arities. The abstract syntax provides a systematic, unambiguous account of the hierarchical and binding structure of the language, and is therefore to be considered the official presentation of the language. However, for the sake of clarity, it is also useful to specify minimal concrete syntax conventions, without going through the trouble to set up a fully precise grammar for it.

We will accomplish both of these purposes with a *syntax chart*, whose meaning is best illustrated by example. The following chart summarizes the abstract and concrete syntax of $\mathcal{L}\{\texttt{num str}\}$, which was analyzed in detail in Chapters 4 and 5.

Type τ ::=	`num`	`num`	numbers
	`str`	`str`	strings
Expr e ::=	x	x	variable
	`num[`n`]`	n	numeral
	`str[`s`]`	$"s"$	literal
	`plus(`e_1`;`e_2`)`	$e_1 + e_2$	addition
	`times(`e_1`;`e_2`)`	$e_1 * e_2$	multiplication
	`cat(`e_1`;`e_2`)`	$e_1 \hat{} e_2$	concatenation
	`len(`e`)`	$\lvert e \rvert$	length
	`let(`e_1`;`x`.`e_2`)`	`let` x `be` e_1 `in` e_2	definition

This chart defines two sorts, Type, ranged over by τ, and Expr, ranged over by e. The chart defines a number of operators and their arities. For example, the operator `let` has arity $(\mathsf{Expr}, (\mathsf{Expr})\mathsf{Expr})$, which specifies that it has two arguments of sort Expr, and binds a variable of sort Expr in the second argument.

6.2 Type System

The role of a type system 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 sensible 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 statics of $\mathcal{L}\{\texttt{num str}\}$ consists of an inductive definition of generic hypothetical judgements of the form

$$\vec{x} \mid \Gamma \vdash e : \tau,$$

where $\vec{x}$ is a finite set of variables, and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$, one for each $x \in \mathcal{X}$. We rely on typographical conventions to determine the set of variables, using the letters x and y for variables that serve as parameters of the typing judgement. We write $x \notin$

$dom(\Gamma)$ 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 statics of $\mathcal{L}\{\texttt{num str}\}$ are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{6.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{6.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{6.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}} \tag{6.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1; e_2) : \texttt{num}} \tag{6.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{str}} \tag{6.1f}$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{num}} \tag{6.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) : \tau_2} \tag{6.1h}$$

In Rule (6.1h) 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.

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

Lemma 6.1 (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 (6.1). □

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 6.2 (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.

6.3 Structural Properties

The statics enjoys the structural properties of the generic hypothetical judgement.

Lemma 6.3 (Weakening). *If $\Gamma \vdash e' : \tau'$, then $\Gamma, x : \tau \vdash e' : \tau'$ for any $x \notin dom(\Gamma)$ and any type τ.*

Proof. By induction on the derivation of $\Gamma \vdash e' : \tau'$. We will give one case here, for rule (6.1h). 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 $z \notin dom(\Gamma)$ 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 (6.1h). □

Lemma 6.4 (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 (6.1h). As in the preceding case, $e' = \texttt{let}(e_1; z.e_2)$, where z may be chosen so that $z \neq x$ and $z \notin dom(\Gamma)$. We have by induction and Lemma 6.3 that

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 (6.1h) that $\Gamma \vdash [e/x]\texttt{let}(e_1; z.e_2) : \tau$, as desired. □

From a programming point of view, Lemma 6.3 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 6.4 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 6.4 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 6.5 (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 when the variable x occurs more than once in e', because then one copy of e suffices for all occurrences of x in e'.

The statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ given by Rules (6.1) exemplifies a recurrent pattern. The constructs of a language are classified into one of two forms, the *introductory* and the *eliminatory*. The introductory forms for a type determine the *values*, or *canonical forms*, of that type. The eliminatory forms determine how to manipulate the values of a type to form a computation of another (possibly the same) type. In $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ the introductory forms for the type `num` are the numerals, and those for the type `str` are the literals. The eliminatory forms for the type `num` are addition and multiplication, and those for the type `str` are concatenation and length.

The importance of this classification will become apparent once we have defined the dynamics of the language in Chapter 7. Then we will see that

[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.

the eliminatory forms are *inverse* to the introductory forms in that they "take apart" what the introductory forms have "put together." The coherence of the statics and dynamics of a language expresses the concept of *type safety*, the subject of Chapter 8.

6.4 Notes

The concept of the static semantics of a programming language was historically slow to develop, perhaps because the earliest languages had relatively few features and only very weak type systems. The concept of a static semantics in the sense considered here was introduced in the definition of the Standard ML programming language [60], building on much earlier work by Church and others on the typed λ-calculus [11]. The concept of introduction and elimination, and the associated inversion principle, was introduced by Gentzen in his pioneering work on natural deduction [29]. These principles were first explicitly applied to programming languages by Martin-Löf [54, 68].

Chapter 7

Dynamics

The *dynamics* of a language is a description of how programs are to be executed. The most important way to define the dynamics of a language is by the method of *structural dynamics*, which defines a *transition system* that inductively specifies the step-by-step process of executing a program. Another method for presenting dynamics, called *contextual dynamics*, is a variation of structural dynamics in which the transition rules are specified in a slightly different manner. An *equational dynamics* presents the dynamics of a language equationally by a collection of rules for deducing when one program is *definitionally equivalent* to another.

7.1 Transition Systems

A *transition system* is specified by the following four forms of judgment:

1. s state, asserting that s is a *state* of the transition system.
2. s final, where s state, asserting that s is a *final* state.
3. s initial, where s state, asserting that s is an *initial* state.
4. $s \mapsto s'$, where s state and s' state, asserting that state s may transition to state s'.

In practice we always arrange things so that no transition is possible from a final state: if s final, then there is no s' state such that $s \mapsto s'$. A state from which no transition is possible is sometimes said to be *stuck*. Whereas all final states are, by convention, stuck, there may be stuck states in a transition system that are not final. A transition system is *deterministic* iff for

every state s there exists at most one state s' such that $s \mapsto s'$, otherwise it is *non-deterministic*.

A *transition sequence* is a sequence of states $s_0, \ldots, s_n$ such that s_0 initial, and $s_i \mapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff there is no s such that $s_n \mapsto s$, and it is *complete* iff it is maximal and, in addition, s_n final. Thus every complete transition sequence is maximal, but maximal sequences are not necessarily complete. The judgement $s \downarrow$ means that there is a complete transition sequence starting from s, which is to say that there exists s' final such that $s \mapsto^* s'$.

The *iteration* of transition judgement, $s \mapsto^* s'$, is inductively defined by the following rules:

$$\frac{}{s \mapsto^* s} \tag{7.1a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^* s''}{s \mapsto^* s''} \tag{7.1b}$$

It is easy to show that iterated transition is transitive: if $s \mapsto^* s'$ and $s' \mapsto^* s''$, then $s \mapsto^* s''$.

When applied to the definition of iterated transition, the principle of rule induction states that to show that $P(s, s')$ holds whenever $s \mapsto^* s'$, it is enough to show these two properties of P:

1. $P(s, s)$.
2. if $s \mapsto s'$ and $P(s', s'')$, then $P(s, s'')$.

The first requirement is to show that P is reflexive. The second is to show that P is *closed under head expansion*, or *closed under inverse evaluation*. Using this principle, it is easy to prove that $\mapsto^*$ is reflexive and transitive.

The *n-times iterated* transition judgement, $s \mapsto^n s'$, where $n \geq 0$, is inductively defined by the following rules.

$$\frac{}{s \mapsto^0 s} \tag{7.2a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^n s''}{s \mapsto^{n+1} s''} \tag{7.2b}$$

Theorem 7.1. *For all states s and s', $s \mapsto^* s'$ iff $s \mapsto^k s'$ for some $k \geq 0$.*

7.2 Structural Dynamics

A *structural dynamics* for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ is given by a transition system whose states are closed expressions. All states are initial. The final states are the

(closed) values, which represent the completed computations. The judgement e val, which states that e is a value, is inductively defined by the following rules:

$$\frac{}{\texttt{num[}n\texttt{]}\ \mathsf{val}} \tag{7.3a}$$

$$\frac{}{\texttt{str[}s\texttt{]}\ \mathsf{val}} \tag{7.3b}$$

The transition judgement, $e \mapsto e'$, between states is inductively defined by the following rules:

$$\frac{n_1 + n_2 = n\ \mathsf{nat}}{\texttt{plus(num[}n_1\texttt{];num[}n_2\texttt{])} \mapsto \texttt{num[}n\texttt{]}} \tag{7.4a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{;}e_2\texttt{)}} \tag{7.4b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1\texttt{;}e_2'\texttt{)}} \tag{7.4c}$$

$$\frac{s_1 \hat{} s_2 = s\ \mathsf{str}}{\texttt{cat(str[}s_1\texttt{];str[}s_2\texttt{])} \mapsto \texttt{str[}s\texttt{]}} \tag{7.4d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1'\texttt{;}e_2\texttt{)}} \tag{7.4e}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1\texttt{;}e_2'\texttt{)}} \tag{7.4f}$$

$$\frac{}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto [e_1/x]e_2} \tag{7.4g}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (7.4a), (7.4d), and (7.4g) 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.

Rule (7.4g) 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:

$$\frac{e_1\ \mathsf{val}}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto [e_1/x]e_2} \tag{7.5a}$$

[1] The justification for the terminology "by name" is obscure, but the terminology is firmly established and cannot be changed.

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1;x.e_2) \mapsto \texttt{let}(e_1';x.e_2)} \tag{7.5b}$$

Rule (7.5b) is an additional search rule specifying that we may evaluate e_1 before e_2. Rule (7.5a) ensures that e_2 is not evaluated until evaluation of e_1 is complete.

A derivation sequence in a structural dynamics 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 (7.4). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\texttt{plus(num[3];num[3])} \mapsto \texttt{num[6]}}\ (7.4a)}{\texttt{plus(plus(num[3];num[3]);num[4])} \mapsto \texttt{plus(num[6];num[4])}}\ (7.4b)$$

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

The principle of rule induction for the structural dynamics 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 (7.4). For example, we may show by rule induction that structural dynamics of $\mathcal{L}\{\texttt{num str}\}$ is *determinate*, which means that an expression may transition to at most one other expression. The proof a simple lemma relating transition to values:

Lemma 7.2. *For no expression e do we have both e val and $e \mapsto e'$ for some e'.*

Proof. By rule induction on Rules (7.3) and (7.4). □

Lemma 7.3 (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. It is assumed that the primitive operators, such as addition, have a unique value when applied to values. □

Rules (7.4) exemplify the *inversion principle* of language design, which states that the eliminatory forms are *inverse* to the introductory forms of a language. The search rules determine the *principal arguments* of each eliminatory form, and the instruction rules specify how to evaluate an eliminatory form when all of its principal arguments are in introductory form. For example, Rules (7.4) specify that both argument of addition are principal, and specify how to evaluate an addition once its principal arguments are evaluated to numerals. The inversion principle is central to ensuring that a programming language is properly defined, the exact statement of which is given in Chapter 8.

7.3 Contextual Dynamics

A variant of structural dynamics, called *contextual dynamics*, 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, $e_1 \leadsto e_2$, for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

$$\frac{m + n = p\ \mathsf{nat}}{\mathtt{plus(num[}m\mathtt{];num[}n\mathtt{])} \leadsto \mathtt{num[}p\mathtt{]}} \tag{7.6a}$$

$$\frac{s \hat{} t = u\ \mathsf{str}}{\mathtt{cat(str[}s\mathtt{];str[}t\mathtt{])} \leadsto \mathtt{str[}u\mathtt{]}} \tag{7.6b}$$

$$\frac{}{\mathtt{let}(e_1; x.e_2) \leadsto [e_1/x]e_2} \tag{7.6c}$$

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.)

$$\frac{}{\circ\ \mathsf{ectxt}} \tag{7.7a}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\mathtt{plus}(\mathcal{E}_1; e_2)\ \mathsf{ectxt}} \tag{7.7b}$$

$$\frac{e_1 \text{ val} \quad \mathcal{E}_2 \text{ ectxt}}{\texttt{plus}(e_1;\mathcal{E}_2) \text{ ectxt}} \tag{7.7c}$$

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 dynamics. For example, Rule (7.7c) states that in an expression $\texttt{plus}(e_1;e_2)$, if the first argument, e_1, is a value, then the next instruction step, if any, lies at or within the second 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:

$$\frac{}{e = \circ\{e\}} \tag{7.8a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(\mathcal{E}_1;e_2)\{e\}} \tag{7.8b}$$

$$\frac{e_1 \text{ val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(e_1;\mathcal{E}_2)\{e\}} \tag{7.8c}$$

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 contextual dynamics for $\mathcal{L}\{\texttt{num str}\}$ is defined by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \rightsquigarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'} \tag{7.9}$$

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 dynamics 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 dynamics (Rules (7.4)), and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual dynamics (Rules (7.9)).

Theorem 7.4. *$e \mapsto_{\mathsf{s}} e'$ if, and only if, $e \mapsto_{\mathsf{c}} e'$.*

Proof. From left to right, proceed by rule induction on Rules (7.4). 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 (7.4a), take $\mathcal{E} = \circ$, and

observe that $e \rightsquigarrow e'$. For Rule (7.4b), 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 $\mathcal{E} = \texttt{plus}(\mathcal{E}_1; e_2)$, and observe that $e = \texttt{plus}(\mathcal{E}_1; e_2)\{e_0\}$ and $e' = \texttt{plus}(\mathcal{E}_1; e_2)\{e_0'\}$ 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 (7.8) that $e \mapsto_{\mathsf{s}} e'$. For example, for Rule (7.8a), e_0 is e, e_0' is e', and $e \rightsquigarrow e'$. Hence $e \mapsto_{\mathsf{s}} e'$. For Rule (7.8b), we have that $\mathcal{E} = \texttt{plus}(\mathcal{E}_1; e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \mapsto_{\mathsf{s}} e_1'$. Therefore e is $\texttt{plus}(e_1; e_2)$, e' is $\texttt{plus}(e_1'; e_2)$, and therefore by Rule (7.4b), $e \mapsto_{\mathsf{s}} e'$.

□

Since the two transition judgements coincide, contextual dynamics may be seen as an alternative way of presenting a structural dynamics. It has two advantages over structural dynamics, one relatively superficial, one rather less so. The superficial advantage stems from writing Rule (7.9) in the simpler form

$$\frac{e_0 \rightsquigarrow e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}}. \tag{7.10}$$

This formulation is superficially simpler in that it does not make explicit how an expression is to be decomposed into an evaluation context and a reducible expression.

7.4 Equational Dynamics

Another formulation of the dynamics 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 $\mathcal{L}\{\texttt{num str}\}$, 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 expressions in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{7.11a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{7.11b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{7.11c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) \equiv \texttt{plus}(e_1'; e_2') : \texttt{num}} \tag{7.11d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) \equiv \texttt{cat}(e_1'; e_2') : \texttt{str}} \tag{7.11e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv \texttt{let}(e_1'; x.e_2') : \tau_2} \tag{7.11f}$$

$$\frac{n_1 + n_2 = n \text{ nat}}{\Gamma \vdash \texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \equiv \texttt{num}[n] : \texttt{num}} \tag{7.11g}$$

$$\frac{s_1 \hat{} s_2 = s \text{ str}}{\Gamma \vdash \texttt{cat}(\texttt{str}[s_1]; \texttt{str}[s_2]) \equiv \texttt{str}[s] : \texttt{str}} \tag{7.11h}$$

$$\frac{}{\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{7.11i}$$

Rules (7.11a) through (7.11c) state that definitional equivalence is an *equivalence relation*. Rules (7.11d) through (7.11f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (7.11g) through (7.11i) specify the meanings of the primitive constructs of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$. For the sake of concision, Rules (7.11) may be characterized as defining the *strongest congruence* closed under Rules (7.11g), (7.11h), and (7.11i).

Rules (7.11) 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 (7.11). 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 (7.11). A prototypical example is the putative equivalence

$$x : \mathtt{num}, y : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{7.12}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from Rules (7.11). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{7.13}$$

where n_1 nat and n_2 nat are particular numbers.

The "gap" between a general law, such as Equation (7.12), and all of its instances, given by Equation (7.13), may be filled by enriching the notion of equivalence to include a principle 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 dynamics of the expressions involved. (Semantic equivalence is developed rigorously for a related language in Chapter 49.)

Definitional equivalence is sometimes called *symbolic execution*, since it allows any subexpression to be replaced by the result of evaluating it according to the rules of the dynamics of the language.

Theorem 7.5. *$e \equiv e' : \tau$ 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

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

and

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

This is proved by rule induction on Rules (7.11). □

The formulation of definitional equivalence for the by-value dynamics of binding requires a bit of additional machinery. The key idea is motivated by the modifications required to Rule (7.11i) 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:

$$\frac{e_1 \text{ val}}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{7.14}$$

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

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

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 (7.11f) as follows:

$$\frac{\Xi\,\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Xi, x \text{ val}\,\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\,\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv \mathtt{let}(e_1'; x.e_2') : \tau_2} \tag{7.15}$$

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

7.5 Notes

The use of transition systems to specify the behavior of programs goes back to the early work of Church and Turing on computability. Turing's approach emphasized the concept of an abstract machine consisting of a finite program together with unbounded memory. Computation proceeds by changing the memory in accordance with the instructions in the program. Much early work on the operational semantics of programming languages, such as Landin's SECD machine [49], emphasized machine models. Church's approach emphasized the language for expressing computations, and defined execution in terms of the programs themselves, rather than in terms of auxiliary concepts such as memories or tapes. Plotkin's elegant formulation of structural operational semantics [75], which we use heavily throughout this book, was inspired by Church's and Landin's ideas [77].

Contextual semantics, which was introduced by Felleisen [27], may be seen as an alternative formulation of structural semantics in which "search rules" are replaced by "context matching". Computation viewed as equational deduction goes back to the early work of Herbrand and Gödel. (In fact, this style of dynamics is sometimes called *Herbrand-Gödel equations*). Church's original formulation of the λ-calculus [20] was based on equational deduction of the kind considered here.

Chapter 8

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 $\mathcal{L}\{\texttt{num str}\}$ 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 statics and the dynamics. The statics may be seen as predicting that the value of an expression will have a certain form so that the dynamics 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 $\mathcal{L}\{\texttt{num str}\}$ may be stated as follows:

Theorem 8.1 (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.

8.1 Preservation

The preservation theorem for $\mathcal{L}\{\texttt{num str}\}$ defined in Chapters 6 and 7 is proved by rule induction on the transition system (rules (7.4)).

Theorem 8.2 (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 (7.4b),

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

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 (7.4g),

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

Assume that $\texttt{let}(e_1;x.e_2) : \tau_2$. By the inversion lemma 6.2 on page 59, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 6.4 on page 60 $[e_1/x]e_2 : \tau_2$, as desired.

It is easy to check that the primitive operations are all type-preserving; for example, if a nat and b nat and $a + b = c$ nat, then c nat.

□

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.

8.2 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 8.3 (Canonical Forms). *If e val and $e : \tau$, then*

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

Proof. By induction on rules (6.1) and (7.3). □

Progress is proved by rule induction on rules (6.1) defining the statics of the language.

Theorem 8.4 (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 (6.1d),

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

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 $\mathtt{plus}(e_1; e_2) \mapsto \mathtt{plus}(e_1'; e_2)$, 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 $\mathtt{plus}(e_1; e_2) \mapsto \mathtt{plus}(e_1; e_2')$, as required. In the former, we have, by the Canonical Forms Lemma 8.3, $e_1 = \mathtt{num}[n_1]$ and $e_2 = \mathtt{num}[n_2]$, and hence

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

□

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 statics and dynamics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.

8.3 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 dynamics 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 dynamics 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:

$$\frac{e_1 \text{ val}}{\texttt{div}(e_1;\texttt{num[0]}) \text{ err}} \tag{8.1a}$$

$$\frac{e_1 \ \mathtt{err}}{\mathtt{plus}(e_1;e_2) \ \mathtt{err}} \tag{8.1b}$$

$$\frac{e_1 \ \mathtt{val} \quad e_2 \ \mathtt{err}}{\mathtt{plus}(e_1;e_2) \ \mathtt{err}} \tag{8.1c}$$

Rule (8.1a) 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.

Once the error judgement is available, we may also consider an expression, `error`, which forcibly induces an error, with the following static and dynamic semantics:

$$\frac{}{\Gamma \vdash \mathtt{error} : \tau} \tag{8.2a}$$

$$\frac{}{\mathtt{error} \ \mathtt{err}} \tag{8.2b}$$

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 8.5 (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. □

8.4 Notes

The concept of type safety as it is understood today was first formulated by Milner in his study of the ML type system [58]. This work gave rise to the slogan "well-typed programs cannot go wrong." Whereas Milner relied on an explicit notion of "going wrong" to express the concept of a type error, Wright and Felleisen observed that one can instead show that ill-defined states cannot arise in a well-typed program [97], giving rise to the slogan "well-typed programs cannot get stuck." However, their formulation relied on a backward analysis showing that a stuck state cannot be well-typed. The formulation given here reformulates this as the progress theorem, which itself relies on the concept of canonical forms of a type introduced by Martin-Löf [68]. The informal concept of type safety is therefore formalized as the conjunction of progress and preservation, which ensure that the state of the dynamics is always well-defined.

Chapter 9

Evaluation Dynamics

In Chapter 7 we defined the evaluation of $\mathcal{L}\{\texttt{num str}\}$ expression using the method of structural dynamics. 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 dynamics* presents the dynamics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Evaluation dynamics suppresses the step-by-step details of determining the value of an expression, and hence does not provide any useful notion of the time complexity of a program. *Cost dynamics* rectifies this by augmenting evaluation dynamics with a *cost measure*. Various cost measures may be assigned to an expression. One example is the number of steps in the structural dynamics required for an expression to reach a value.

9.1 Evaluation Dynamics

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

$$\frac{}{\texttt{num}[n] \Downarrow \texttt{num}[n]} \tag{9.1a}$$

$$\frac{}{\texttt{str}[s] \Downarrow \texttt{str}[s]} \tag{9.1b}$$

$$\frac{e_1 \Downarrow \texttt{num}[n_1] \quad e_2 \Downarrow \texttt{num}[n_2] \quad n_1 + n_2 = n \text{ nat}}{\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]} \tag{9.1c}$$

$$\frac{e_1 \Downarrow \texttt{str}[s_1] \quad e_2 \Downarrow \texttt{str}[s_2] \quad s_1 \hat{} s_2 = s \text{ str}}{\texttt{cat}(e_1; e_2) \Downarrow \texttt{str}[s]} \tag{9.1d}$$

$$\frac{e \Downarrow \mathtt{str}[s] \quad |s| = n \text{ str}}{\mathtt{len}(e) \Downarrow \mathtt{num}[n]} \tag{9.1e}$$

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow v_2} \tag{9.1f}$$

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 (9.1f) 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 (9.1):

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 9.1. *If $e \Downarrow v$, then v val.*

Proof. By induction on Rules (9.1). All cases except Rule (9.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the premise of the evaluation rule. □

9.2 Relating Structural and Evaluation Dynamics

We have given two different forms of dynamics 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 structural dynamics describes a step-by-step process of execution, whereas the evaluation dynamics 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 structural dynamics and the evaluation judgement in the evaluation dynamics. (We will consider only numeric expressions, but analogous results hold also for string-valued expressions.)

Theorem 9.2. *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 9.3. *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 7 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 9.4. *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. □

9.3 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\texttt{num str}\}$ (Theorem 8.1 on page 75) 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 dynamics. But what if we had instead specified the dynamics 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 dynamics, 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 $\mathcal{L}\{\texttt{num str}\}$, it demands much more than just progress — it requires that every expression evaluate to a value! If $\mathcal{L}\{\texttt{num str}\}$ were extended to admit operations that may result in an error (as discussed in Section 8.3 on page 78), 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 dynamics, but only by reference to a structural dynamics. Another point of view is to instrument the dynamics 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 form of error solely to prove that it cannot arise! Nevertheless, we will press on to show how a semblance of type safety can be established using evaluation dynamics.

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:

$$\frac{}{\texttt{plus(str[}s\texttt{];}e_2\texttt{)}\Uparrow} \tag{9.2a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{plus(}e_1\texttt{;str[}s\texttt{])}\Uparrow} \tag{9.2b}$$

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

Theorem 9.5. *If $e\Uparrow$, then there is no τ such that $e : \tau$.*

Proof. By rule induction on Rules (9.2). For example, for Rule (9.2a), we observe that $\texttt{str[}s\texttt{]} : \texttt{str}$, and hence $\texttt{plus(str[}s\texttt{];}e_2\texttt{)}$ is ill-typed. □

Corollary 9.6. *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 9.5 on the preceding 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 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. Execution is therefore more efficient, and the language definition is simpler, an elegant win-win situation for both the dynamics and the implementation.

9.4 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, 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 dynamics*.

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

$$\frac{}{\mathtt{num}[n] \Downarrow^0 \mathtt{num}[n]} \tag{9.3a}$$

$$\frac{e_1 \Downarrow^{k_1} \mathtt{num}[n_1] \quad e_2 \Downarrow^{k_2} \mathtt{num}[n_2]}{\mathtt{plus}(e_1;e_2) \Downarrow^{k_1+k_2+1} \mathtt{num}[n_1 + n_2]} \tag{9.3b}$$

$$\frac{}{\mathtt{str}[s] \Downarrow^0 \mathtt{str}[s]} \tag{9.3c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\mathtt{cat}(e_1;e_2) \Downarrow^{k_1+k_2+1} \mathtt{str}[s_1 \hat{} s_2]} \tag{9.3d}$$

$$\frac{[e_1/x]e_2 \Downarrow^{k_2} v_2}{\texttt{let}(e_1; x.e_2) \Downarrow^{k_2+1} v_2} \tag{9.3e}$$

Theorem 9.7. *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 dynamics. From right to left proceed by induction on k, with an inner rule induction on the definition of the structural dynamics. □

9.5 Notes

The structural similarity between evaluation dynamics and typing rules was first developed in the definition of Standard ML [60]. Martin-Löf's formulation of type theory as a programming language [68] also stressed evaluation dynamics to define computation. The advantage of evaluation semantics is that it directly defines the relation of interest, that between a program and its outcome. The disadvantage is that it is not as well-suited to metatheory as structural semantics, precisely because it glosses over the fine structure of computation. The concept of a cost dynamics was introduced by Blelloch and Greiner in their study of parallel computation [14, 34]. The sequential cost semantics given here sets the stage for the treatment of parallelism in Chapter 41.