4 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 set of rules for deriving *typing judgments* 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 correct; 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 a simple expression language, **E**, as an illustration of the method that we will employ throughout this book.

4.1 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 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 **E**.

Typ	τ	::=	`num`	`num`	numbers
			`str`	`str`	strings
Exp	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 ^ e_2	concatenation
			`len`(e)	$\lvert e \rvert$	length
			`let`$(e_1; x.e_2)$	`let` x `be` e_1 `in` e_2	definition

https://doi.org/10.1017/CBO9781316576892.006 Published online by Cambridge University Press

This chart defines two sorts, Typ, ranged over by τ, and Exp, ranged over by e. The chart defines a set of operators and their arities. For example, it specifies that the operator `let` has arity (Exp, Exp.Exp)Exp, which specifies that it has two arguments of sort Exp, and binds a variable of sort Exp in the second argument.

4.2 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 the expression $\texttt{plus}(x;\texttt{num}[n])$ is sensible depends on whether the variable x is restricted 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 **E** consists of an inductive definition of generic hypothetical judgments 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 $x \in \vec{x}$. We rely on typographical conventions to determine the set of variables, using the letters x and y to stand for them. We write $x \notin dom(\Gamma)$ to say 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 **E** are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{4.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{4.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{4.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}} \tag{4.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1; e_2) : \texttt{num}} \tag{4.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{str}} \tag{4.1f}$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{num}} \tag{4.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) : \tau_2} \tag{4.1h}$$

In rule (4.1h), 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.

https://doi.org/10.1017/CBO9781316576892.006 Published online by Cambridge University Press

It is easy to check that every expression has at most one type by *induction on typing*, which is rule induction applied to rules (4.1).

Lemma 4.1 (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 (4.1), making use of the fact that variables have at most one type in any typing context. □

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

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

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

4.3 Structural Properties

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

Lemma 4.3 (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 (4.1h). We have that $e' = \mathtt{let}(e_1; z.e_2)$, where by the conventions on variables 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 (4.1h). □

Lemma 4.4 (Substitution)**.** *If* $\Gamma, x : \tau \vdash e' : \tau'$ *and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash [e/x]e' : \tau'$.

https://doi.org/10.1017/CBO9781316576892.006 Published online by Cambridge University Press

Proof By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. We again consider only rule (4.1h). As in the preceding case, $e' = \texttt{let}(e_1; z.e_2)$, where z is chosen so that $z \neq x$ and $z \notin dom(\Gamma)$. We have by induction and Lemma 4.3 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 (4.1h) that $\Gamma \vdash [e/x]\texttt{let}(e_1; z.e_2) : \tau'$, as desired. □

From a programming point of view, Lemma 4.3 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, introducing new variables beyond those required by an expression e does not invalidate e itself; it remains well-formed, with the same type.[1] More importantly, Lemma 4.4 expresses the important concepts of *modularity* and *linking*. We may think of the expressions e and e' as two *components* of a larger system in which e' is 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 to satisfy the assumptions of the client. If so, then we may link them to form the composite system $[e/x]e'$. This implementation 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 4.4 is called *decomposition*. It states that any (large) expression can be decomposed into a client and implementor by introducing a variable to mediate their interaction.

Lemma 4.5 (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. □

Lemma 4.5 tells us that any sub-expression can be isolated as a separate module of a larger system. This property 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 **E** given by rules (4.1) exemplifies a recurrent pattern. The constructs of a language are classified into one of two forms, the *introduction* and the *elimination*. The introduction forms for a type determine the *values*, or *canonical forms*, of that type. The elimination forms determine how to manipulate the values of a type to form a computation of another (possibly the same) type. In the language **E**, the introduction forms for the type `num` are the numerals, and those for the type `str` are the literals. The elimination forms for

https://doi.org/10.1017/CBO9781316576892.006 Published online by Cambridge University Press

the type `num` are addition and multiplication, and those for the type `str` are concatenation and length.

The importance of this classification will become clear once we have defined the dynamics of the language in Chapter 5. Then we will see that the elimination forms are *inverse* to the introduction forms in that they "take apart" what the introduction forms have "put together." The coherence of the statics and dynamics of a language expresses the concept of *type safety*, the subject of Chapter 6.

4.4 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 (Milner et al., 1997), building on earlier work by Church and others on the typed λ-calculus (Barendregt, 1992). The concept of introduction and elimination, and the associated inversion principle, was introduced by Gentzen in his pioneering work on natural deduction (Gentzen, 1969). These principles were applied to the structure of programming languages by Martin-Löf (1984, 1980).

Exercises

4.1. It is sometimes useful to give the typing judgment $\Gamma \vdash e : \tau$ an "operational" reading that specifies more precisely the flow of information required to derive a typing judgment (or determine that it is not derivable). The *analytic* mode corresponds to the context, expression, and type being given, with the goal to determine whether the typing judgment is derivable. The *synthetic* mode corresponds to the context and expression being given, with the goal to find the unique type τ, if any, possessed by the expression in that context. These two readings can be made explicit as judgments of the form $e \downarrow \tau$, corresponding to the analytic mode, and $e \uparrow \tau$, corresponding to the synthetic mode.

Give a simultaneous inductive definition of these two judgments according to the following guidelines:

(a) Variables are introduced in synthetic form.

(b) If we can synthesize a unique type for an expression, then we can analyze it with respect to a given type by checking type equality.

(c) Definitions need care, because the type of the defined expression is not given, even when the type of the result is given.

There is room for variation; the point of the exercise is to explore the possibilities.

https://doi.org/10.1017/CBO9781316576892.006 Published online by Cambridge University Press

4.2. One way to limit the range of possibilities in the solution to Exercise **4.1** is to restrict and extend the syntax of the language so that every expression is either synthetic or analytic according to the following suggestions:

(a) Variables are analytic.

(b) Introduction forms are analytic, elimination forms are synthetic.

(c) An analytic expression can be made synthetic by introducing a *type cast* of the form $\texttt{cast}\{\tau\}(e)$ specifying that e must check against the specified type τ, which is synthesized for the whole expression.

(d) The defining expression of a definition must be synthetic, but the scope of the definition can be either synthetic or analytic.

Reformulate your solution to Exercise **4.1** to take account of these guidelines.

Note

1 This point may seem so obvious that it is not worthy of mention, but, surprisingly, there are useful type systems that lack this property. Because they do not validate the structural principle of weakening, they are called *substructural* type systems.

https://doi.org/10.1017/CBO9781316576892.006 Published online by Cambridge University Press