6 Type Safety

Most 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 **E** states that it will never arise that a number is added to a string, or that two numbers are 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. Safety 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.

Type safety for the language **E** is stated precisely as follows:

Theorem 6.1 (Type Safety)**.**

1. *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$.
2. *If* $e : \tau$*, then either* e val*, or there exists* e' *such that* $e \longmapsto e'$.

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 $e \longmapsto e'$. 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.

6.1 Preservation

The preservation theorem for **E** defined in Chapters 4 and 5 is proved by rule induction on the transition system (rules (5.4)).

Theorem 6.2 (Preservation)**.** *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$.

https://doi.org/10.1017/CBO9781316576892.008 Published online by Cambridge University Press

Proof We will give the proof in two cases, leaving the rest to the reader. Consider rule (5.4b),

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

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 (5.4h),

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

Assume that $\texttt{let}(e_1; x.e_2) : \tau_2$. By the inversion Lemma 4.2, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution Lemma 4.4 $[e_1/x]e_2 : \tau_2$, as desired.

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

The proof of preservation is naturally structured as an induction on the transition judgment, because the argument hinges on examining all possible transitions from a given expression. In some cases, we 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, sometimes, cannot be made to work at all.

6.2 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 6.3 (Canonical Forms)**.** *If e val and $e : \tau$, then*

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

Proof By induction on rules (4.1) and (5.3). □

Progress is proved by rule induction on rules (4.1) defining the statics of the language.

Theorem 6.4 (Progress)**.** *If $e : \tau$, then either e val, or there exists e' such that $e \longmapsto e'$.*

https://doi.org/10.1017/CBO9781316576892.008 Published online by Cambridge University Press

Proof The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (4.1d),

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

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 $e_1 \longmapsto e_1'$. In the latter case, it follows that $\mathtt{plus}(e_1; e_2) \longmapsto \mathtt{plus}(e_1'; e_2)$, as required. In the former, we also have by induction that either e_2 val, or there exists e_2' such that $e_2 \longmapsto e_2'$. In the latter case, we have that $\mathtt{plus}(e_1; e_2) \longmapsto \mathtt{plus}(e_1; e_2')$, as required. In the former, we have, by the Canonical Forms Lemma 6.3, $e_1 = \mathtt{num}[n_1]$ and $e_2 = \mathtt{num}[n_2]$, and hence

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

□

Because 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 is more than one rule for a given expression form. Such rules present no difficulties, so long as the proof proceeds by induction on the typing rules and not on the structure of the expression.

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 together 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.

6.3 Run-Time Errors

Suppose that we wish to extend **E** with, say, a quotient operation that is undefined for a zero divisor. The natural typing rule for quotients is given by the following rule:

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

But the expression $\mathtt{div}(\mathtt{num}[3]; \mathtt{num}[0])$ 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, practical, but the most common approach is the second. The first requires that the type checker prove that an expression be non-zero before permitting

https://doi.org/10.1017/CBO9781316576892.008 Published online by Cambridge University Press

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. We cannot predict statically whether an expression will be non-zero when evaluated, so the second approach is most often used in practice.

The overall 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 modeling 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 be present in a full inductive definition of this judgment:

$$\frac{e_1 \ \mathsf{val}}{\mathtt{div}(e_1;\mathtt{num}[0])\ \mathsf{err}} \tag{6.1a}$$

$$\frac{e_1 \ \mathsf{err}}{\mathtt{div}(e_1;e_2)\ \mathsf{err}} \tag{6.1b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{err}}{\mathtt{div}(e_1;e_2)\ \mathsf{err}} \tag{6.1c}$$

Rule (6.1a) 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 judgment is available, we may also consider an expression, `error`, which forcibly induces an error, with the following static and dynamic semantics:

$$\frac{}{\Gamma \vdash \mathtt{error} : \tau} \tag{6.2a}$$

$$\frac{}{\mathtt{error}\ \mathsf{err}} \tag{6.2b}$$

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

Theorem 6.5 (Progress With Error)**.** *If $e : \tau$, then either e err, or e val, or there exists e' such that $e \longmapsto e'$.*

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. □

https://doi.org/10.1017/CBO9781316576892.008 Published online by Cambridge University Press

6.4 Notes

The concept of type safety was first formulated by Milner (1978), who invented the slogan "well-typed programs do not go wrong." Whereas Milner relied on an explicit notion of "going wrong" to express the concept of a type error, Wright and Felleisen (1994) observed that we can instead show that ill-defined states cannot arise in a well-typed program, giving rise to the slogan "well-typed programs do not get stuck." However, their formulation relied on an analysis showing that no stuck state is well-typed. The progress theorem, which relies on the characterization of canonical forms in the style of Martin-Löf (1980), eliminates this analysis.

Exercises

6.1. Complete the proof of Theorem 6.2 in full detail.

6.2. Complete the proof of Theorem 6.4 in full detail.

6.3. Give several cases of the proof of Theorem 6.5 to illustrate how checked errors are handled in type safety proofs.

https://doi.org/10.1017/CBO9781316576892.008 Published online by Cambridge University Press