Part III

A Functional Language

Chapter 8

MinML, A Minimal Functional Language

The language MinML will serve as the jumping-off point for much of our study of programming language concepts. MinML is a call-by-value, effect-free language with integers, booleans, and a (partial) function type.

8.1 Abstract Syntax

The first-order abstract syntax of MinML is divided into three main syntactic categories, *types*, *expressions*, and *programs*. Their definition involves some auxiliary syntactic categories, namely *variables*, *numbers*, and *operators*.

These categories are defined by the following grammar:

Variables	x	$::=$	$\ldots$
Numerals	n	$::=$	$\ldots$
Operators	o	$::=$	`+` | `*` | `-` | `=` | `<`
Types	τ	$::=$	`int` | `bool` | $\tau_1 \rightarrow \tau_2$
Expressions	e	$::=$	x | n | $o(e_1 \ldots, e_n)$ | `true` | `false` | `if` e `then` e_1 `else` e_2 `fi` | `fun` f `(`x`:`τ_1`):`τ_2 `is` e `end` | `apply(`e_1, e_2`)`
Programs	p	$::=$	e

We do not specify precisely the sets of numbers or variables. We generally write x, y, *etc.* for variables, and we write numbers in ordinary decimal notation.

To specify the higher-order abstract syntax of MinML we need only define specify which expression-forming operators bind variables and, for those that do, specify the range of significance, or scope, of each bound variable. There is only one variable binding operator in MinML, the function expression `fun` f `(`x`:`τ_1`):`τ_2 `is` e `end`. In such an expression the variables f and x are both bound within the *body* of the function, e. Using the notation of higher-order terms, this would be written as `fun`$(\tau_1)\tau_2 f, x.e$. But we will stick to the more readable concrete syntax given above.

8.2 Static Semantics

Not all expressions in MinML are well-formed. For example, the expression `if` 3 `then` 1 `else` 0 `fi` is not well-formed because 3 is an integer, whereas the conditional test expects a boolean. In other words, this expression is *ill-typed* because the expected constraint is not met. Expressions which do satisfy these constraints are said to be *well-typed*.

Typing is clearly context-sensitive. The expression $x + 3$ may or may not be well-typed, according to the type we assume for the variable x. That is, it depends on the surrounding context whether this sub-expression is well-typed or not.

The definition of well-typed expressions is given by a three-place *typing relation*, or *typing judgement*, written $\Gamma \vdash e : \tau$, where Γ is a partial function with finite domain mapping variables to types, and $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$. This relation may be read as "the expression e has type τ, under the assumption that its free variables have the types given by Γ." The function Γ may be thought of as a "symbol table" recording the types of the free variables of the expression e; the type τ is the type of e under the assumption that its free variables have the types assigned by Γ. When e is closed (has no free variables), we write simply $e : \tau$ instead of the more unwieldy $\emptyset \vdash e : \tau$.

We write $\Gamma(x)$ for the unique type τ (if any) assigned to x by Γ. The function $\Gamma[x{:}\tau]$, where $x \notin \mathrm{dom}(\Gamma)$, is defined by the following equation

$$\Gamma[x{:}\tau](y) = \begin{cases} \tau & \textit{if } y = x \\ \Gamma(y) & \textit{otherwise} \end{cases}$$

The typing relation is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \tag{8.1}$$

Here it is understood that if $\Gamma(x)$ is undefined, then no type for x is derivable from assumptions Γ.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \tag{8.2}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \tag{8.3}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \tag{8.4}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{+}(e_1, e_2) : \texttt{int}} \tag{8.5}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{*}(e_1, e_2) : \texttt{int}} \tag{8.6}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{-}(e_1, e_2) : \texttt{int}} \tag{8.7}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{=}(e_1, e_2) : \texttt{bool}} \tag{8.8}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{<}(e_1, e_2) : \texttt{bool}} \tag{8.9}$$

The typing rules for the arithmetic and boolean primitive operators are as expected.

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau} \tag{8.10}$$

Notice that the "then" and the "else" clauses must have the same type!

$$\frac{\Gamma[f{:}\tau_1{\to}\tau_2][x{:}\tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\, \texttt{is}\, e\, \texttt{end} : \tau_1{\to}\tau_2} \tag{8.11}$$

Here we require that the variables f and x be chosen (by suitable renaming of the function expression) so that $\{ f, x \} \cap \mathrm{dom}(\Gamma) = \emptyset$.

$$\frac{\Gamma \vdash e_1 : \tau_2{\to}\tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply(}e_1, e_2\texttt{)} : \tau} \tag{8.12}$$

8.3 Properties of Typing

It is useful at this stage to catalogue some properties of the typing relation. We will make use of the principle of *induction on typing derivations*, or *induction on the typing rules*.

A key observation about the typing rules is that there is exactly one rule for each form of expression — that is, there is one rule for the each of the boolean constants, one rule for functions, *etc.*. The typing relation is therefore said to be *syntax-directed*; the form of the expression determines the typing rule to be applied. While this may seem inevitable at this stage, we will later encounter type systems for which this is not the case.

A simple — but important — consequence of syntax-directedness are the following *inversion principles* for typing. The typing rules define *sufficient* conditions for typing. For example, to show that

$$\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau,$$

it suffices to show that $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$, and $\Gamma \vdash e_2 : \tau$, because of Rule 8.10. Since there is exactly one typing rule for each expression, the typing rules also express *necessary* conditions for typing. For example, if $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, then $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. That is, we can "invert" each rule to obtain a necessary typing condition. This is the content of the following theorem.

Theorem 5 (Inversion)

1. *If* $\Gamma \vdash x : \tau$, *then* $\Gamma(x) = \tau$.
2. *If* $\Gamma \vdash n : \tau$, *then* $\tau = \texttt{int}$.
3. *If* $\Gamma \vdash \texttt{true} : \tau$, *then* $\tau = \texttt{bool}$, *and similarly for* $\texttt{false}$.
4. *If* $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, *then* $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ *and* $\Gamma \vdash e_2 : \tau$.
5. *If* $\Gamma \vdash \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\, \texttt{is}\, e\, \texttt{end} : \tau$, *then* $\Gamma[f{:}\tau_1{\to}\tau_2][x{:}\tau_1] \vdash e : \tau_2$ *and* $\tau = \tau_1{\to}\tau_2$.
6. *If* $\Gamma \vdash \texttt{apply(}e_1, e_2\texttt{)} : \tau$, *then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \tau_2{\to}\tau$ *and* $\Gamma \vdash e_2 : \tau_2$.

Proof: Each case is proved by induction on typing. In each case exactly one rule applies, from which the result is obvious. ■

Lemma 6

1. *Typing is not affected by "junk" in the symbol table. If* $\Gamma \vdash e : \tau$ *and* $\Gamma' \supseteq \Gamma$*, then* $\Gamma' \vdash e : \tau$*.*
2. *Substitution for a variable with type* τ *by an expression of the same type doesn't affect typing. If* $\Gamma[x{:}\tau] \vdash e' : \tau'$*, and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash \{e/x\}e' : \tau'$*.*

Proof:

1. By induction on the typing rules. For example, consider the typing rule for applications. Inductively we may assume that if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_1 : \tau_2{\to}\tau$ and if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_2 : \tau_2$. Consequently, if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash \texttt{apply(}e_1, e_2\texttt{)} : \tau$, as required. The other cases follow a similar pattern.
2. By induction on the derivation of the typing $\Gamma[x{:}\tau] \vdash e' : \tau'$. We will consider several rules to illustrate the idea.

   (Rule 8.1) We have that e' is a variable, say y, and $\tau' = \Gamma[x{:}\tau](y)$. If $y \neq x$, then $\{e/x\}y = y$ and $\Gamma[x{:}\tau](y) = \Gamma(y)$, hence $\Gamma \vdash y : \Gamma(y)$, as required. If $x = y$, then $\tau' = \Gamma[x{:}\tau](x) = \tau$, and $\{e/x\}x = e$. By assumption $\Gamma \vdash e : \tau$, as required.

   (Rule 8.11) We have that $e' = \texttt{fun}\, f\, \texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e_2\, \texttt{end}$ and $\tau' = \tau_1{\to}\tau_2$. We may assume that f and y are chosen so that

   $$\{\, f, y \,\} \cap (\mathrm{FV}(e) \cup \{\, x \,\} \cup \mathrm{dom}(\Gamma)) = \emptyset.$$

   By definition of substitution,

   $$\{e/x\}e' = \texttt{fun}\, f\, \texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, \{e/x\}e_2\, \texttt{end}.$$

   Applying the inductive hypothesis to the premise of Rule 8.11,

   $$\Gamma[x{:}\tau][f{:}\tau_1{\to}\tau_2][y{:}\tau_1] \vdash e_2 : \tau_2,$$

   it follows that

   $$\Gamma[f{:}\tau_1{\to}\tau_2][y{:}\tau_1] \vdash \{e/x\}e_2 : \tau_2.$$

   Hence

   $$\Gamma \vdash \texttt{fun}\, f\, \texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, \{e/x\}e_2\, \texttt{end} : \tau_1{\to}\tau_2,$$

   as required.

■

8.4 Dynamic Semantics

The dynamic semantics of MinML is given by an inductive definition of the *one-step evaluation* relation, $e \mapsto e'$, between *closed* expressions. Recall that we are modelling computation in MinML as a form of "in place" calculation; the relation $e \mapsto e'$ means that e' is the result of performing a single step of computation starting with e. To calculate the value of an expression e, we repeatedly perform single calculation steps until we reach a *value*, v, which is either a number, a boolean constant, or a function.

The rules defining the dynamic semantics of MinML may be classified into two categories: rules defining the fundamental computation steps (or, *instructions*) of the language, and rules for determining where the next instruction is to be executed. The purpose of the search rules is to ensure that the dynamic semantics is *deterministic*, which means that for any expression there is at most one "next instruction" to be executed.[1]

First the instructions governing the primitive operations. We assume that each primitive operation o defines a total function — given values v_1, $\ldots$, v_n of appropriate type for the arguments, there is a unique value v that is the result of performing operation o on $v_1, \ldots, v_n$. For example, for addition we have the following primitive instruction:

$$\frac{}{\texttt{+}(m,n) \mapsto m+n} \tag{8.13}$$

The other primitive operations are defined similarly.

The primitive instructions for conditional expressions are as follows:

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi} \mapsto e_1} \tag{8.14}$$

$$\frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi} \mapsto e_2} \tag{8.15}$$

The primitive instruction for application is as follows:

$$\frac{(v = \texttt{fun}\ f\ \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\ \texttt{is}\ e\ \texttt{end})}{\texttt{apply(}v, v_1\texttt{)} \mapsto \{v, v_1/f, x\}e} \tag{8.16}$$

To apply the function $v = \texttt{fun}\ f\ \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\ \texttt{is}\ e\ \texttt{end}$ to an argument v_1 (which must be a value!), we substitute the function itself, v, for f, and the

[1] Some languages are, by contrast, *non-determinstic*, notably those involving concurrent interaction. We'll come back to those later.

argument value, v_1, for x in the body, e, of the function. By substituting v for f we are "unrolling" the recursive function as we go along.

This completes the primitive instructions of MinML. The "search" rules, which determine which instruction to execute next, follow.

For the primitive operations, we specify a left-to-right evaluation order. For example, we have the following two rules for addition:

$$\frac{e_1 \mapsto e_1'}{\texttt{+}(e_1, e_2) \mapsto \texttt{+}(e_1', e_2)} \tag{8.17}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{+}(v_1, e_2) \mapsto \texttt{+}(v_1, e_2')} \tag{8.18}$$

The other primitive operations are handled similarly.

For the conditional, we evaluate the test expression.

$$\frac{e \mapsto e'}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \mapsto \texttt{if}\, e'\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}} \tag{8.19}$$

For applications, we first evaluate the function position; once that is complete, we evaluate the argument position.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply(}e_1, e_2\texttt{)} \mapsto \texttt{apply(}e_1', e_2\texttt{)}} \tag{8.20}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{apply(}v_1, e_2\texttt{)} \mapsto \texttt{apply(}v_1, e_2'\texttt{)}} \tag{8.21}$$

This completes the definition of the MinML one-step evaluation relation.

The *multi-step evaluation relation*, $e \mapsto^* e'$, is inductively defined by the following rules:

$$\frac{}{e \mapsto^* e} \tag{8.22}$$

$$\frac{e \mapsto e' \quad e' \mapsto^* e''}{e \mapsto^* e''} \tag{8.23}$$

In words: $e \mapsto^* e'$ iff performing zero or more steps of evaluation starting from the expression e yields the expression e'. The relation $\mapsto^*$ is sometimes called the *Kleene closure*, or *reflexive-transitive closure*, of the relation $\mapsto$.

8.5 Properties of the Dynamic Semantics

Let us demonstrate that the dynamic semantics of MinML is well-defined in the sense that it assigns at most one value to each expression. (We should be suspicious if this weren't true of the semantics, for it would mean that programs have no definite meaning.)

First, observe that if v is a value, then there is no e (value or otherwise) such that $v \mapsto e$. Second, observe that the evaluation rules are arranged so that at most one rule applies to any given form of expression, *even though* there are, for example, $n+1$ rules governing each n-argument primitive operation. These two observations are summarized in the following lemma.

Lemma 7
For every closed expression e, there exists at most one e' such that $e \mapsto e'$. In other words, the relation $\mapsto$ is a partial function.

Proof: By induction on the structure of e. We leave the proof as an exercise to the reader. Be sure to consider *all* rules that apply to a given expression e! ■

It follows that evaluation to a value is deterministic:

Lemma 8
For every closed expression e, there exists at most one value v such that $e \mapsto^* v$.

Proof: Follows immediately from the preceding lemma, together with the observation that there is no transition from a value. ■

8.6 Exercises

1. Can you think of a type system for a variant of MinML in which inversion fails? What form would such a type system have to take? *Hint:* think about overloading arithmetic operations.

2. Prove by induction on the structure of e that for every e and every Γ there exists at most one τ such that $\Gamma \vdash e : \tau$. *Hint:* use rule induction for the rules defining the abstract syntax of expressions.

Chapter 9

Type Safety for MinML

Programming languages such as ML and Java are said to be "safe" (or, "type safe", or "strongly typed"). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, it will never arise that an integer is to be applied to an argument, nor that two functions could be added to each other. The goal of this section is to make this informal notion precise. What is remarkable is that we will be able to clarify the idea of type safety without making reference to an implementation. Consequently, the notion of type safety is extremely robust — it is shared by *all* correct implementations of the language.

9.1 Defining Type Safety

Type safety is a *relation* between the static and dynamic semantics. It tells us something about the execution of well-typed programs; it says nothing about the execution of ill-typed programs. In implementation terms, we expect ill-typed programs to be rejected by the compiler, so that nothing need be said about their execution behavior (just as syntactically incorrect programs are rejected, and nothing is said about what such a program might mean).

In the framework we are developing, type safety amounts to the following two conditions:

1. **Preservation**. If e is a well-typed program, and $e \mapsto e'$, then e' is also a well-typed program.
2. **Progress**. If e is a well-typed program, then either e is a value, or there exists e' such that $e \mapsto e'$.

Preservation tells us that the dynamic semantics doesn't "run wild". If we start with a well-typed program, then each step of evaluation will necessarily lead to a well-typed program. We can never find ourselves lost in the tall weeds. Progress tells us that evaluation never "gets stuck", unless the computation is complete (*i.e.*, the expression is a value). An example of "getting stuck" is provided by the expression `apply(`$3, 4$`)` — it is easy to check that no transition rule applies. Fortunately, this expression is also ill-typed! Progress tells us that this will always be the case.

Neither preservation nor progress can be expected to hold without some assumptions about the primitive operations. For preservation, we must assume that if the result of applying operation o to arguments $v_1, \ldots, v_n$ is v, and $o(v_1, \ldots, v_n) : \tau$, then $v : \tau$. For progress, we must assume that if $o(v_1, \ldots, v_n)$ is well-typed, then there exists a value v such that v is the result of applying o to the arguments $v_1, \ldots, v_n$. For the primitive operations we're considering, these assumptions make sense, but they do preclude introducing "partial" operations, such as quotient, that are undefined for some arguments. We'll come back to this shortly.

9.2 Type Safety of MinML

Theorem 9 (Preservation)
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Proof: Note that we are proving not only that e' is well-typed, but that it has the same type as e. The proof is by induction on the rules defining one-step evaluation. We will consider each rule in turn.

(Rule 8.13) Here $e =$ `+`(m, n), $\tau =$ `int`, and $e' = m + n$. Clearly $e' :$ `int`, as required. The other primitive operations are handled similarly.

(Rule 8.14) Here $e =$ `if true then` e_1 `else` e_2 `fi` and $e' = e_1$. Since $e : \tau$, by inversion $e_1 : \tau$, as required.

(Rule 8.15) Here $e =$ `if false then` e_1 `else` e_2 `fi` and $e' = e_2$. Since $e : \tau$, by inversion $e_2 : \tau$, as required.

(Rule 8.16) Here $e =$ `apply(`v_1, v_2`)`, where $v_1 =$ `fun` f `(`x`:`τ_2`):`τ `is` e_2 `end`, and $e' = \{v_1, v_2/f, x\}e_2$. By inversion applied to e, we have $v_1 : \tau_2 \to \tau$ and

$v_2 : \tau_2$. By inversion applied to v_1, we have $[f{:}\tau_2{\to}\tau][x{:}\tau_2] \vdash e_2 : \tau$. Therefore, by substitution we have $\{v_1, v_2/f, x\}e_2 : \tau$, as required.

(Rule 8.17) Here $e = \texttt{+}(e_1, e_2)$, $e' = \texttt{+}(e_1', e_2)$, and $e_1 \mapsto e_1'$. By inversion $e_1 : \texttt{int}$, so that by induction $e_1' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

(Rule 8.18) Here $e = \texttt{+}(v_1, e_2)$, $e' = \texttt{+}(v_1, e_2')$, and $e_2 \mapsto e_2'$. By inversion $e_2 : \texttt{int}$, so that by induction $e_2' : \texttt{int}$, and hence $e' : \texttt{int}$, as required.

The other primitive operations are handled similarly.

(Rule 8.19) Here $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}$ and $e' = \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}$. By inversion we have that $e_1 : \texttt{bool}$, $e_2 : \tau$ and $e_3 : \tau$. By inductive hypothesis $e_1' : \texttt{bool}$, and hence $e' : \tau$.

(Rule 8.20) Here $e = \texttt{apply(}e_1, e_2\texttt{)}$ and $e' = \texttt{apply(}e_1', e_2\texttt{)}$. By inversion $e_1 : \tau_2{\to}\tau$ and $e_2 : \tau_2$, for some type τ_2. By induction $e_1' : \tau_2{\to}\tau$, and hence $e' : \tau$.

(Rule 8.21) Here $e = \texttt{apply(}v_1, e_2\texttt{)}$ and $e' = \texttt{apply(}v_1, e_2'\texttt{)}$. By inversion, $v_1 : \tau_2{\to}\tau$ and $e_2 : \tau_2$, for some type τ_2. By induction $e_2' : \tau_2$, and hence $e' : \tau$.

■

The type of a closed value "predicts" its form.

Lemma 10 (Canonical Forms)
Suppose that $v : \tau$ is a closed, well-formed value.

1. *If $\tau = \texttt{bool}$, then either $v = \texttt{true}$ or $v = \texttt{false}$.*
2. *If $\tau = \texttt{int}$, then $v = n$ for some n.*
3. *If $\tau = \tau_1{\to}\tau_2$, then $v = \texttt{fun}\ f\ \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\ \texttt{is}\ e\ \texttt{end}$ for some f, x, and e.*

Proof: By induction on the typing rules, using the fact that v is a value. ■

Exercise 11
Give a proof of the canonical forms lemma.

Theorem 12 (Progress)
If $e : \tau$, then either e is a value, or there exists e' such that $e \mapsto e'$.

Proof: The proof is by induction on the typing rules.

(Rule 8.1) Cannot occur, since e is closed.

(Rules 8.2, 8.3, 8.4, 8.11) In each case e is a value, which completes the proof.

(Rule 8.5) Here $e = \texttt{+}(e_1, e_2)$ and $\tau = \texttt{int}$, with $e_1 : \texttt{int}$ and $e_2 : \texttt{int}$. By induction we have either e_1 is a value, or there exists e_1' such that $e_1 \mapsto e_1'$ for some expression e_1'. In the latter case it follows that $e \mapsto e'$, where $e' = \texttt{+}(e_1', e_2)$. In the former case, we note that by the canonical forms lemma $e_1 = n_1$ for some n_1, and we consider e_2. By induction either e_2 is a value, or $e_2 \mapsto e_2'$ for some expression e_2'. If e_2 is a value, then by the canonical forms lemma $e_2 = n_2$ for some n_2, and we note that $e \mapsto e'$, where $e' = n_1 + n_2$. Otherwise, $e \mapsto e'$, where $e' = \texttt{+}(v_1, e_2')$, as desired.

(Rule 8.10) Here $e = \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi}$, with $e_1 : \texttt{bool}$, $e_2 : \tau$, and $e_3 : \tau$. By the first inductive hypothesis, either e_1 is a value, or there exists e_1' such that $e_1 \mapsto e_1'$. If e_1 is a value, then we have by the Canonical Forms Lemma, either $e_1 = \texttt{true}$ or $e_1 = \texttt{false}$. In the former case $e \mapsto e_2$, and in the latter $e \mapsto e_3$, as required. If e_1 is not a value, then $e \mapsto e'$, where $e' = \texttt{if}\, e_1'\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi}$, by Rule 8.19.

(Rule 8.12) Here $e = \texttt{apply(}e_1, e_2\texttt{)}$, with $e_1 : \tau_2 \to \tau$ and $e_2 : \tau_2$. By the first inductive hypothesis, either e_1 is a value, or there exists e_1' such that $e_1 \mapsto e_1'$. If e_1 is not a value, then $e \mapsto \texttt{apply(}e_1', e_2\texttt{)}$ by Rule 8.20, as required. By the second inductive hypothesis, either e_2 is a value, or there exists e_2' such that $e_2 \mapsto e_2'$. If e_2 is not a value, then $e \mapsto e'$, where $e' = \texttt{apply(}e_1, e_2'\texttt{)}$, as required. Finally, if both e_1 and e_2 are values, then by the Canonical Forms Lemma, $e_1 = \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_2\texttt{):}\tau\, \texttt{is}\, e''\, \texttt{end}$, and $e \mapsto e'$, where $e' = \{e_1, e_2/f, x\}e''$, by Rule 8.16.

■

Theorem 13 (Safety)
If e is closed and well-typed, then evaluation of e can only terminate with a value of the same type. In particular, evaluation cannot "get stuck" in an

ill-defined state.

9.3 Run-Time Errors and Safety

Stuck states correspond to ill-defined programs that attempt to, say, treat an integer as a pointer to a function, or that move a pointer beyond the limits of a region of memory. In an *unsafe* language there are no stuck states — every program will do *something* — but it may be impossible to predict how the program will behave in certain situations. It may "dump core", or it may allow the programmer to access private data, or it may compute a "random" result.

The best-known example of an unsafe language is C. It's lack of safety manifests itself in numerous ways, notably in that computer viruses nearly always rely on overrunning a region of memory as a critical step in an attack. Another symptom is lack of portability: an unsafe program may execute sensibly on one platform, but behave entirely differently on another. To avoid this behavior, standards bodies have defined portable subsets of C that are guaranteed to have predictable behavior on all platforms. But there is no good way to ensure that a programmer, whether through malice or neglect, adheres to this subset.[1]

Safe languages, in contrast, avoid ill-defined states entirely, by imposing typing restrictions that ensure that well-typed programs have well-defined behavior. MinML is a good example of a safe language. It is inherently portable, because its dynamic semantics is specified in an implementation-independent manner, and because its static semantics ensures that well-typed programs never "get stuck". Stated contrapositively, the type safety theorem for MinML assures us that stuck states are ill-typed.

But suppose that we add to MinML a primitive operation, such as quotient, that is undefined for certain arguments. An expression such as $3/0$ would most-assuredly be "stuck", yet would be well-typed, at least if we take the natural typing rule for it:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ / } e_2 : \texttt{int}}$$

What are we to make of this? Is the extension of MinML with quotient unsafe?

To recover safety, we have two options:

[1] It should be easy to convince yourself that it is undecidable whether a given C program can reach an implementation-dependent state.

1. *Enhance the type system* so that no well-typed program can ever divide by zero.

2. Modify the dynamic semantics so that division by zero is not "stuck", but rather incurs a *run-time error*.

The first option amounts to requiring that the type checker *prove* that the denominator of a quotient is non-zero in order for it to be well-typed. But this means that the type system would, in general, be undecidable, for we can easily arrange for the denominator of some expression to be non-zero exactly when some Turing machine halts on blank tape. It is the subject of ongoing research to devise conservative type checkers that are sufficiently expressive to be useful in practice, but we shall not pursue this approach any further here.

The second option is widely used. It is based on distinguishing *checked* from *unchecked* errors. A *checked* error is one that is detected at execution time by an explicit test for ill-defined situations. For example, the quotient operation tests whether its denominator is zero, incurring an error if so. An *unchecked* error is one that is not detected at execution time, but rather is regarded as "stuck" or "ill-defined". Type errors in MinML are unchecked errors, precisely because the static semantics ensures that they can never occur.

The point of introducing checked errors is that they ensure *well-defined* behavior even for *ill-defined* programs. Thus $3/0$ evaluates to `error`, rather than simply "getting stuck" or behaving unpredictably. The essence of type safety is that well-typed programs should have well-defined behavior, even if that behavior is to signal an error. That way we can predict how the program will behave simply by looking at the program itself, without regard to the implementation or platform. In this sense safe languages are inherently portable, which explains the recent resurgence in interest in them.

How might checked errors be added to MinML? The main idea is to add to MinML a special expression, `error`, that designates a run-time fault in an expression. Its typing rule is as follows:

$$\frac{}{\Gamma \vdash \texttt{error} : \tau} \tag{9.1}$$

Note that a run-time error can have *any* type at all. The reasons for this will become clear once we re-state the safety theorem.

The dynamic semantics is augmented in two ways. First, we add new transitions for the checked errors. For example, the following rule checks

for a zero denominator in a quotient:

$$\overline{v_1 \texttt{ / } 0 \mapsto \texttt{error}} \tag{9.2}$$

Second, we add rules to *propagate* errors; once an error has arisen, it aborts the rest of the computation. Here are two representative error propagation rules:

$$\overline{\texttt{error(}v_2\texttt{)} \mapsto \texttt{error}} \tag{9.3}$$

$$\overline{v_1\texttt{(error)} \mapsto \texttt{error}} \tag{9.4}$$

These rule state that if the function or argument position of an application incur an error, then so does the entire application.

With these changes, the type safety theorem may be stated as follows:

Theorem 14 (Safety With Errors)
If an expression is well-typed, it can only evaluate to a value or evaluate to `error`*. It cannot "get stuck" in an ill-defined state.*

As before, safety follows from preservation and progress. The preservation theorem states that types are preserved by evaluation. We have already proved this for MinML; we need only consider error transitions. But for these preservation is trivial, since `error` has any type whatsoever. The canonical forms lemma carries over without change. The progress theorem is proved as before, relying on checked errors to ensure that progress can be made, even in ill-defined states such as division by zero.