# 12 Type Rules

This section formally defines the type rules of Cool. The type rules define the type of every Cool expression in a given context. The context is the *type environment*, which describes the type of every unbound identifier appearing in an expression. The type environment is described in Section 12.1. Section 12.2 gives the type rules.

## 12.1 Type Environments

To a first approximation, type checking in Cool can be thought of as a bottom-up algorithm: the type of an expression $e$ is computed from the (previously computed) types of $e$'s subexpressions. For example, an integer `1` has type `Int`; there are no subexpressions in this case. As another example, if $\mathtt{e_n}$ has type `X`, then the expression $\{\ \mathtt{e_1};\ldots;\mathtt{e_n};\ \}$ has type `X`.

A complication arises in the case of an expression `v`, where `v` is an object identifier. It is not possible to say what the type of `v` is in a strictly bottom-up algorithm; we need to know the type declared for `v` in the larger expression. Such a declaration must exist for every object identifier in valid Cool programs.

To capture information about the types of identifiers, we use a *type environment*. The environment consists of three parts: a method environment $M$, an object environment $O$, and the name of the current class in which the expression appears. The method environment and object environment are both functions (also called *mappings*). The object environment is a function of the form

$$O(v) = T$$

which assigns the type $T$ to object identifier $v$. The method environment is more complex; it is a function of the form

$$M(C, f) = (T_1, \ldots, T_{n-1}, T_n)$$

where $C$ is a class name (a type), $f$ is a method name, and $t_1, \ldots, t_n$ are types. The tuple of types is the *signature* of the method. The interpretation of signatures is that in class $C$ the method $f$ has formal parameters of types $(t_1, \ldots, t_{n-1})$—in that order—and a return type $t_n$.

Two mappings are required instead of one because object names and method names do not clash—i.e., there may be a method and an object identifier of the same name.

The third component of the type environment is the name of the current class, which is needed for type rules involving `SELF_TYPE`.

Every expression $e$ is type checked in a type environment; the subexpressions of $e$ may be type checked in the same environment or, if $e$ introduces a new object identifier, in a modified environment. For example, consider the expression

```
let c : Int <- 33 in
  ...
```

The `let` expression introduces a new variable `c` with type `Int`. Let $O$ be the object component of the type environment for the `let`. Then the body of the `let` is type checked in the object type environment

$$O[Int/c]$$

where the notation $O[T/c]$ is defined as follows:

$$\begin{array}{rcl} O[T/c](c) & = & T \\ O[T/c](d) & = & O(d) \;\; \text{if } d \neq c \end{array}$$

## 12.2 Type Checking Rules

The general form a type checking rule is:

$$\frac{\vdots}{O, M, C \vdash e : T}$$

The rule should be read: In the type environment for objects $O$, methods $M$, and containing class $C$, the expression $e$ has type $T$. The dots above the horizontal bar stand for other statements about the types of sub-expressions of $e$. These other statements are hypotheses of the rule; if the hypotheses are satisfied, then the statement below the bar is true. In the conclusion, the "turnstyle" ("$\vdash$") separates context $(O, M, C)$ from statement $(e : T)$.

The rule for object identifiers is simply that if the environment assigns an identifier $Id$ type $T$, then $Id$ has type $T$.

$$\frac{O(Id) = T}{O, M, C \vdash Id : T} \qquad \text{[Var]}$$

The rule for assignment to a variable is more complex:

$$\frac{\begin{array}{l} O(Id) = T \\ O, M, C \vdash e_1 : T' \\ T' \leq T \end{array}}{O, M, C \vdash Id \leftarrow e_1 : T'} \qquad \text{[ASSIGN]}$$

Note that this type rule—as well as others—use the conformance relation $\leq$ (see Section 3.2). The rule says that the assigned expression $e_1$ must have a type $T'$ that conforms to the type $T$ of the identifier $Id$ in the type environment. The type of the whole expression is $T'$.

The type rules for constants are all easy:

$$\frac{}{O, M, C \vdash true : Bool} \quad \text{[True]}$$

$$\frac{}{O, M, C \vdash false : Bool} \quad \text{[False]}$$

$$\frac{i \text{ is an integer constant}}{O, M, C \vdash i : Int} \quad \text{[Int]}$$

$$\frac{s \text{ is a string constant}}{O, M, C \vdash s : String} \quad \text{[String]}$$

There are two cases for `new`, one for `new SELF_TYPE` and one for any other form:

$$\frac{T' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T = \texttt{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}}{O, M, C \vdash new\ T : T'} \quad \text{[New]}$$

Dispatch expressions are the most complex to type check.

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0' = \begin{cases} C & \text{if } T_0 = \texttt{SELF\_TYPE}_C \\ T_0 & \text{otherwise} \end{cases} \\ M(T_0', f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[Dispatch]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[StaticDispatch]}$$

To type check a dispatch, each of the subexpressions must first be type checked. The type $T_0$ of $e_0$ determines which declaration of the method $f$ is used. The argument types of the dispatch must conform to the declared argument types. Note that the type of the result of the dispatch is either the declared return type or $T_0$ in the case that the declared return type is SELF_TYPE. The only difference in type checking a static dispatch is that the class $T$ of the method $f$ is given in the dispatch, and the type $T_0$ must conform to $T$.

The type checking rules for `if` and {-} expressions are straightforward. See Section 7.5 for the definition of the $\sqcup$ operation.

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \\ O, M, C \vdash e_3 : T_3 \end{array}}{O, M, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : T_2 \sqcup T_3} \quad \text{[If]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ \vdots \\ O, M, C \vdash e_n : T_n \end{array}}{O, M, C \vdash \{\ e_1; e_2; \ldots e_n;\ \} : T_n} \quad \text{[Sequence]}$$

The `let` rule has some interesting aspects.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0' \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \quad \text{[Let-Init]}$$

First, the initialization $e_1$ is type checked in an environment without a new definition for $x$. Thus, the variable $x$ cannot be used in $e_1$ unless it already has a definition in an outer scope. Second, the body $e_2$ is type checked in the environment $O$ extended with the typing $x : T_0'$. Third, note that the type of $x$ may be SELF_TYPE.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O[T_0'/x], M, C \vdash e_1 : T_1 \end{array}}{O, M, C \vdash \text{let } x : T_0 \text{ in } e_1 : T_1} \quad \text{[Let-No-Init]}$$

The rule for `let` with no initialization simply omits the conformance requirement. We give type rules only for a `let` with a single variable. Typing a multiple `let`

$$\mathrm{let\ x_1 : T_1\ [\leftarrow e_1],\ x_2 : T_2\ [\leftarrow e2], \ldots,\ x_n : T_n\ [\leftarrow e_n]\ in\ e}$$

is defined to be the same as typing

$$\mathrm{let\ x_1 : T_1\ [\leftarrow e_1]\ in\ (let\ x_2 : T_2\ [\leftarrow e_2], \ldots,\ x_n : T_n\ [\leftarrow e_n]\ in\ e\ )}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O[T_1/x_1], M, C \vdash e_1 : T_1' \\ \vdots \\ O[T_n/x_n], M, C \vdash e_n : T_n' \end{array}}{O, M, C \vdash \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \ldots x_n : T_n \Rightarrow e_n; \text{ esac} : \bigsqcup_{1 \leq i \leq n} T_i'} \quad \text{[Case]}$$

Each branch of a `case` is type checked in an environment where variable $x_i$ has type $T_i$. The type of the entire `case` is the join of the types of its branches. The variables declared on each branch of a `case` must all have distinct types.

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : Object} \quad \text{[Loop]}$$

The predicate of a loop must have type *Bool*; the type of the entire loop is always *Object*. An `isvoid` test has type *Bool*:

$$\frac{O, M, C \vdash e_1 : T_1}{O, M, C \vdash \text{isvoid } e_1 : Bool} \quad \text{[Isvoid]}$$

With the exception of the rule for equality, the type checking rules for the primitive logical, comparison, and arithmetic operations are easy.

$$\frac{O, M, C \vdash e_1 : Bool}{O, M, C \vdash \neg e_1 : Bool} \quad \text{[Not]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{<, \leq\} \end{array}}{O, M, C \vdash e_1 \ op \ e_2 : Bool} \quad \text{[Compare]}$$

$$\frac{O, M, C \vdash e_1 : Int}{O, M, C \vdash \tilde{\ } e_1 : Int} \quad \text{[Neg]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{*, +, -, /\} \end{array}}{O, M, C \vdash e_1 \ op \ e_2 : Int} \quad \text{[Arith]}$$

The wrinkle in the rule for equality is that any types may be freely compared except `Int, String` and `Bool`, which may only be compared with objects of the same type.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, String, Bool\} \vee T_2 \in \{Int, String, Bool\} \Rightarrow T_1 = T_2 \end{array}}{O, M, C \vdash e_1 = e_2 : Bool} \quad \text{[Equal]}$$

The final cases are type checking rules for attributes and methods. For a class $C$, let the object environment $O_C$ give the types of all attributes of $C$ (including any inherited attributes). More formally, if $x$ is an attribute (inherited or not) of $C$, and the declaration of $x$ is $x : T$, then

$$O_C(x) = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T = \texttt{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}$$

The method environment $M$ is global to the entire program and defines for every class $C$ the signatures of all of the methods of $C$ (including any inherited methods).

The two rules for type checking attribute defininitions are similar the rules for `let`. The essential difference is that attributes are visible within their initialization expressions. Note that `self` is bound in the initialization.

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C[\texttt{SELF\_TYPE}_C/self], M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C, M, C \vdash x : T_0 \leftarrow e_1;} \quad \text{[Attr-Init]}$$

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T;} \quad \text{[Attr-No-Init]}$$

The rule for typing methods checks the body of the method in an environment where $O_C$ is extended with bindings for the formal parameters and `self`. The type of the method body must conform to the declared return type.

$$\frac{\begin{array}{l} M(C, f) = (T_1, \ldots, T_n, T_0) \\ O_C[\texttt{SELF\_TYPE}_C/self][T_1/x_1] \ldots [T_n/x_n], M, C \vdash e : T_0' \\ T_0' \leq \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \end{array}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \ \{\ e\ \};} \quad \text{[Method]}$$