Chapter 1

Semantic Analysis

The lexer detects illegal tokens inside the input, while the parser detects ill-formed parse trees inside the input. Semantic analysis is the last "front end" phase of the compiler to catch errors. It is necessary because there are errors that cannot be caught by the parser and lexer, and some language constructs are not context free.

As a typical statically type checked object oriented language, Cool language requires its semantic analyser doing the following checks:

- All identifiers are declared
- Type checking (major function)
- Inheritance relationships
- Classes are defined only once
- Methods in a class are defined only once
- Reserved identifiers are not misused

This is not an exhaustive list.

1.1 Scope

There can be more than one definition of an identifier before it is used. In order to match the correct declaration of an identifier with its uses, we need to understand the conception of scope.

The **scope** of an identifier is the portion of a program in which that identifier is accessible. The same identifier may refer to different things in different parts of the same program. In such case, different scopes of the same identifier should not overlap. Programming languages can have either **static scope** or **dynamic scope**. Today most languages, including Cool, have static scope, which means

scopes of identifiers depend only on the program text, not the runtime behaviors. There are languages that are dynamically scoped, like SNOBOL and ancient Lisp, for which scopes depend on the execution process of the program. Generally speaking, static scoped language follows **the most closely nested rule**, meaning that the variable binds to the definition that is the most closely enclosing it. Dynamically scoped language follows **the most recent binding rule**, meaning that the variable binds to the most recent definition during the execution.

In Cool, identifier bindings are introduced by a lot of mechanisms:

- class definitions (class names)
- method definitions (method names)
- let expressions (object ids)
- formal parameters (object ids)
- attribute definitions (object ids)
- case expressions(object ids)

Not all identifiers in Cool follow the most closely nested rule. For example, class definitions in Cool cannot be nested, and they are globally visible throughout the program, which means that a class can be used before it is defined. Also, attribute names are globally visible within the class in which they are defined. What's more, method names have some complex rules, such as they can be defined in a parent class, and they can be overridden.

1.2 Symbol tables

Much of semantic analysis can be expressed as a recursive descent of an AST. In each step we do the following 3 things:

- Before: Begin processing an AST node n (preprocessing)
- Recurse: Process the children of n
- After: Finish processing node n (post-processing)

When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined. If we divide the processing of the expression $\mathsf{let}\ \mathsf{x}{:}\mathsf{T} \leftarrow e_0\ \mathsf{in}\ e_1$ into the 3 phases listed above, the preprocessing phase will add definition of x to the current definitions and override any previous definition of x, while the post-processing phase will remove definition of x and restore the old definition of x. A **symbol table** is the data structure used to track the current bindings of identifiers.

To implement a simple symbol table, we can use just a stack. It contains three operations:

- add_symbol(x): push symbol x and its associated info on the stack.
- find_symbol(x): search the stack from the top. Returns the first x found or NULL.
- remove_symbol(): pop the stack.

This simple implementation works for let expression because in let expressions, declarations are perfectly nested, and symbols are added to the symbol table one at a time. In other cases, the functionality of this implementation is not sufficient, e.g. in the definition of a method in which more than one symbols can be introduced each time. We need an implementation that covers the following operations:

- enter_scope(): start a new nested scope
- find_symbol(x): find current x (or NULL)
- add_symbol(x): add a symbol x to the table
- check_scope(x): true if x is defined in the current scope
- exit_scope(): exit current scope

Class names should be specially considered here because classes can be used before they are defined. Thus they cannot be checked using a symbol table, neither in one single pass. The solution is to complete two passes: gather all class names in the first one, and do the checking in the second one. In general, semantic analysis requires multiple passes. In the implementation of semantic analysis, a few simple passes is superior to one complex pass.

1.3 Type checking

1.3.1 Types

"What is a type" is a question worthy of asking because type is a notion varying from language to language. The consensus is that a type is a set of values and a set of operations on these values. In OO languages, classes are one instantiation of the modern notion of type, but types do not need to be associated with classes.

The goal of type checking is to ensure that operations are used only with the correct types. It is nonsensical to add a function pointer to an integer in C, but at assembly language level they share the same implementation. Type checking is intended to avoid such errors.

There are 3 kinds of languages:

Statically typed Almost or all type checking is done as part of compilation. e.g. C, java, cool.

Dynamically typed Almost all type checking is done at run time. e.g. Python, Lisp, Perl.

Untyped No type checking. e.g. machine code.

There have always been debates on the merits of static typing v.s. dynamic typing. Static typing proponents assert that static checking catches many errors at compile time, and it avoids overhead of runtime type checks. Dynamic typing proponents argue that static type systems are too restrictive, and it causes difficulty when it comes to rapid prototyping. In the end, we end up with compromises on both sides: static typed languages often provide an "escape" mechanism, e.g. casting in C-like languages; dynamic typed languages are often retrofitted for optimisation and debugging with static typing.

Types in Cool include class names and SELF_TYPE. User is supposed to declare types for identifiers, and the compiler will do the rest of the job: a type will be inferred for every expression.

1.3.2 Logical inference rules

We have seen two formal notations as the specification of parts of a compiler: regular expressions and context free grammars. Logical inference rules are the appropriate formalism for type checking.

Inference rules have the form

If Hypothesis is true, the Conclusion is true.

In the specific case of type checking rules, they often have the form

If E_1 and E_2 have certain types, then E_3 has a certain type.

In order to simplify the notation, we use $\wedge$ to denote "and", $\Rightarrow$ to denote "if-then", and x:T to denote "x has type T". Thus, the rule "if e_1 has type Int, e_2 has type Int, then $e_1 + e_2$ has type Int" is denoted as

$$(\texttt{e}_1 : \texttt{Int} \wedge \texttt{e}_2 : \texttt{Int}) \Rightarrow \texttt{e}_1 + \texttt{e}_2 : \texttt{Int}$$

By convention, inference rules are written in the form

$$\frac{\vdash \text{Hypothesis}_1 \cdots \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

in which $\vdash$ is read "it is provable that". Here we give some rules as examples.

$$\frac{\vdash \texttt{i is an integer literal}}{\vdash \texttt{i} : \texttt{Int}}$$

$$\frac{\vdash \texttt{e}_1 : \texttt{Int} \vdash \texttt{e}_2 : \texttt{Int}}{\vdash \texttt{e}_1 + \texttt{e}_2 : \texttt{Int}}$$

$$\frac{\dfrac{\texttt{1 is an int literal}}{\vdash \texttt{1:Int}} \quad \dfrac{\texttt{2 is an int literal}}{\vdash \texttt{2:Int}}}{\vdash \texttt{1} + \texttt{2} : \texttt{Int}}$$

$$\frac{}{\vdash \texttt{false} : \texttt{Bool}}$$

$$\frac{}{\vdash \texttt{new T} : \texttt{T}}$$

$$\frac{\vdash \texttt{e} : \texttt{Bool}}{\vdash \texttt{!e} : \texttt{Bool}}$$

$$\frac{\vdash \texttt{e}_1 : \texttt{Bool} \quad \vdash \texttt{e}_2 : \texttt{T}}{\vdash \texttt{while e}_1 \texttt{ loop e}_2 \texttt{ pool} : \texttt{Object}}$$

A type system is sound if whenever ⊢e:T, e evaluates to a value of type T. We only want sound rules, but some sound rules are better than others. For example, $\frac{\vdash \texttt{i:Int}}{\vdash \texttt{i:Object}}$ is sound but not helpful at all.

Type check proves facts in the form of e:T. The proof is on the structure of the AST. It actually has the shape of the AST, because one type rule is used for each AST node. In the rule used for an AST node e, Hypotheses are the proofs of the types of e's subexpressions, while the conclusion is the type of e. Types are computed in a bottom-up pass over the AST.

1.3.3 Type environment

For a variable, the local structural rule does not carry enough information to give it a type.

$$\frac{\text{x is a variable}}{\vdash \texttt{x} : ?}$$

More information should be put into the rules in such case. A **type environment** gives types to **free** variables. A variable is free if it is not defined within the expression. A type environment is a function from object identifiers to types. It is implemented by the symbol table.

Let O be a function from ObjectIdentifiers to Types, the sentence O⊢e : T is read: under the assumption that free variables in expression e have the type given by O, it is provable that e has type T. The type environment should be added to the earlier rules. For example now we have

$$\frac{\vdash \text{i is an integer literal}}{\texttt{O} \vdash \texttt{i} : \texttt{Int}}$$

$$\frac{\texttt{O} \vdash \texttt{e}_1 : \texttt{Int} \quad \texttt{O} \vdash \texttt{e}_2 : \texttt{Int}}{\texttt{O} \vdash \texttt{e}_1 + \texttt{e}_2 : \texttt{Int}}$$

And we can now write some new rules:

$$\frac{\texttt{O(x)} = \texttt{T}}{\texttt{O} \vdash \texttt{x} : \texttt{T}}$$

We use O[T/x] to denote the function that returns T for x, and O(y) for whatever y≠x. We can now define the rule for let expression:

$$\texttt{O} \vdash \texttt{e}_0 : \texttt{T}$$

$$\frac{\mathtt{O}[\mathtt{T}/\mathtt{x}] \vdash \mathtt{e}_1 : \mathtt{T}_1}{\mathtt{O} \vdash \mathtt{let\ x : T} \leftarrow \mathtt{e}_0\ \mathtt{in\ e}_1 : \mathtt{T}_1} \tag{1.1}$$

The type environment is passed down the AST from the root to the leaves, while types are computed up the AST from the leaves to the root.

1.3.4 Subtyping

The rule (1.1) is not satisfactory in practice because It is not necessary that x:T. x can actually be of any subtype of T. In order to allow the use of subtypes, we introduce the $\leq$ relationship between classes. Its formal definition is

- $X \leq X$
- $X \leq Y$ if X inherits from Y
- $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

With $\leq$ relationship added, rule (1.1) can be written as

$$\frac{\begin{array}{c}\mathtt{O} \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathtt{O}[\mathtt{T}/\mathtt{x}] \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \mathtt{T}_0 \leq \mathtt{T}\end{array}}{\mathtt{O} \vdash \mathtt{let\ x : T} \leftarrow \mathtt{e}_0\ \mathtt{in\ e}_1 : \mathtt{T}_1}$$

Similarly, the rule of assignment can be written as

$$\frac{\begin{array}{c}\mathtt{O}(\mathtt{x}) = \mathtt{T}_0 \\ \mathtt{O} \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \mathtt{T}_1 \leq \mathtt{T}_0\end{array}}{\mathtt{O} \vdash \mathtt{x} \leftarrow \mathtt{e}_1 : \mathtt{T}_1}$$

Attribute initialization inside a class also uses subtyping. $\mathrm{O}_C(\mathrm{x}) = \mathrm{T}$ means for all attributes x of class C we have x : T.

$$\frac{\begin{array}{c}\mathtt{O_C}(\mathtt{x}) = \mathtt{T}_0 \\ \mathtt{O_C} \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \mathtt{T}_1 \leq \mathtt{T}_0\end{array}}{\mathtt{O_C} \vdash \mathtt{x : T}_0 \leftarrow \mathtt{e}_1 : \mathtt{T}_0}$$

Consider the case of the `if` expression. The type of if e_0 then e_1 else e_2 fi can be either the type of e_1 or that of e_2, depending on whether the else clause or the then clause is executed at runtime. In this case, the best we can do is to use the smallest super type larger than both the types of e_1 and e_2, i.e. **their least upper bound**, which is denoted by $\mathsf{Z} = \mathsf{lub}(\mathsf{X}, \mathsf{Y})$. Its formal definition is

- $X \leq Z \wedge Y \leq Z$

- if $X \leq Z' \wedge Y \leq Z', Z \leq Z'$

In Cool, the least upper bound of two types is their least common ancestor in the inheritance tree. Equipped with the definition of lub(X,Y), we can write the rule of the if expression:

$$\frac{\begin{array}{c}\mathtt{O} \vdash \mathtt{e}_0 : \mathtt{Bool} \\ \mathtt{O} \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \mathtt{O} \vdash \mathtt{e}_2 : \mathtt{T}_2\end{array}}{\mathtt{O} \vdash \mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2\ \mathtt{fi} : \mathtt{lub}(\mathtt{T}_1, \mathtt{T}_2)}$$

The rule of case expression takes a similar but more complex form:

$$\frac{\begin{array}{c}\mathtt{O} \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathtt{O}[\mathtt{T}_1/\mathtt{x}_1] \vdash \mathtt{e}_1 : \mathtt{T}'_1 \\ \ldots \\ \mathtt{O}[\mathtt{T}_n/\mathtt{x}_n] \vdash \mathtt{e}_n : \mathtt{T}'_n\end{array}}{\mathtt{O} \vdash \mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ \mathtt{x}_1 : \mathtt{T}_1 \rightarrow \mathtt{e}_1; \ldots; \mathtt{x}_n : \mathtt{T}_n \rightarrow \mathtt{e}_n : \mathtt{lub}(\mathtt{T}'_1, \ldots, \mathtt{T}'_n)}$$

1.3.5 Methods type environment

In order to check the type of a method call, we need a mechanism similar to type environment O for variables. In Cool, method type rules are put in namespace M different from O, which means that a method and an object can share the same name. A rule

$$\mathtt{M}(\mathtt{C}, \mathtt{f}) = (\mathtt{T}_1, \ldots, \mathtt{T}_n, \mathtt{T}_{n+1})$$

means that in class C, there is a method f with signature $f(x_1 : T_1, \ldots, x_n : T_n) : T_{n+1}$. Now we can write the rule of normal dispatch:

$$\frac{\begin{array}{c}\mathtt{O}, \mathtt{M} \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \ldots \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e}_n : \mathtt{T}_n \\ \mathtt{M}(\mathtt{T}_0, \mathtt{f}) = (\mathtt{T}'_1, \ldots, \mathtt{T}'_n, \mathtt{T}_{n+1}) \\ \mathtt{T}_i \leq \mathtt{T}'_i\ \mathtt{for}\ 1 \leq \mathtt{i} \leq \mathtt{n}\end{array}}{\mathtt{O}, \mathtt{M} \vdash \mathtt{e}_0.\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{T}_{n+1}}$$

Similarly, the rule of static dispatch is

$$\frac{\begin{array}{c}\mathtt{O}, \mathtt{M} \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \ldots \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e}_n : \mathtt{T}_n \\ \mathtt{M}(\mathtt{T}, \mathtt{f}) = (\mathtt{T}'_1, \ldots, \mathtt{T}'_n, \mathtt{T}_{n+1})\end{array}}$$

$$\frac{\begin{array}{c}\mathtt{T}_0 \leq \mathtt{T} \\ \mathtt{T}_i \leq \mathtt{T}'_i \quad \text{for} \quad 1 \leq i \leq n\end{array}}{\mathtt{O}, \mathtt{M} \vdash \mathtt{e}_0@\mathtt{T}.\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{T}_{n+1}}$$

For some cases involving SELF_TYPE, we need to know the class in which an expression appears. Thus the full type environment of Cool contains 3 parts:

- A mapping O giving types to object identifiers.
- A mapping M giving types to methods.
- The current class C.

The whole environment must be added to all rules, although in most cases M and C are passed down but not actually used. For example, the rule for add of int is now

$$\frac{\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_1 : \mathtt{Int} \quad \mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_2 : \mathtt{Int}}{\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_1 + \mathtt{e}_2 : \mathtt{Int}} \tag{1.2}$$

1.3.6 Implementation

Cool type checking can be implemented in a single traversal over the AST. The type environment is passed down the tree from parent to child, while types are passed up the tree from child to parent.

The implementation of a rule is somewhat self-explaining. The addition rule for int (1.2) could be implemented with the following persudo-code.

```
1 TypeCheck(Environment, e1 + e2) {
2   T1 = TypeCheck(Environment, e1);
3   T2 = TypeCheck(Environment, e2);
4   Check T1 == T2 == Int; // report error if false
5   return Int;
6 }
```

The rule of let expression

$$\frac{\begin{array}{c}\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathtt{O}[\mathtt{T}/\mathtt{x}], \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \mathtt{T}_0 \leq \mathtt{T}\end{array}}{\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{let}\ \mathtt{x} : \mathtt{T} \leftarrow \mathtt{e}_0\ \mathtt{in}\ \mathtt{e}_1 : \mathtt{T}_1}$$

can be implemented as

```
1 TypeCheck(Environment, let x:T ← e0 in e1) {
2   TypeCheck(Environment, e0) = T0;
3   TypeCheck(Environment, e1) = T1;
4   Check subtype(T0, T); // report error if false
5   return T1;
6 }
```

1.3.7 Static v.s. dynamic typing

Static type checking systems detect common errors at compile time. Unfortunately, some correct programs from the perspective of runtime are disallowed by the type checker. In order to tackle this problem, some argue for dynamic type checking instead, while others want more expressive, but meantime more complex static type checking.

One of the ideas this discussion suggests is that there are two different notions of type: the dynamic type and the static type. Dynamic type is a runtime notion. The dynamic type of an object C is the class C used in the `new C` expression that created it. Static type is a compile notion. It captures all dynamic types the expression can have. For simple type systems, we have the soundness theorem, which asserts that for all expression E, dynamic_type(E) = static_type(E). However, for complex type systems like that of Cool, this is not always true. For example, if class A inherits class B, then `new A` can be assigned to a variable b that has static type B. The soundness theorem of Cool should be dynamic_type(E) $\leq$ static_type(E), $\forall$E, which implies that subclasses can only add attributes or methods. Methods can be overridden in subclasses, but their types should be observed.

1.4 Self type

1.4.1 Introduction

As an example of "more expressive static typing", we will discuss the type rule of self in Cool.

Consider the following class:

```
1 class Count {
2   i : int ← 0;
3   inc() : Count {{ i ← i + 1; self; }};
4 };
```

This class incorporates a counter and thus can serve as a base class for any class that needs this functionality. Consider one of such classes, the Stock class:

```
1 class Stock inherits Count {
2   name : String;
3 };
4 class Main {
5   main() : Object {
6     Stock a ← (new Stock).inc();
7     a.name = ...
8   };
9 };
```

If the Cool type checker that we have developed is employed on this piece of code, it will actually complain about a type mismatch error: a is expecting a value of type Stock, but `(new Stock).inc()` has static type Count as declared

in class Count. In order to use the functionality of the counter, all subclasses of Count have to do the tedious job of redefining `inc()`.

An extension of the current type system provides an elegant solution to the problem. Instead of specifying the return type of `inc()` as Count or any other subclass of Count, we require that `inc()` should return the type of self, which could be implemented by introducing a new keyword SELF_TYPE:

$$\texttt{inc}() : \texttt{SELF\_TYPE}\{\ldots\}$$

This mechanism allows the return type of `inc()` changing according to the dynamic type of the object calling it. The type checker can now prove

$$\texttt{O}, \texttt{M}, \texttt{C} \vdash (\texttt{newCount}).\texttt{inc}() : \texttt{Count}$$
$$\texttt{O}, \texttt{M}, \texttt{C} \vdash (\texttt{newStock}).\texttt{int}() : \texttt{Stock}$$

1.4.2 Self type operations

In order to make the mechanism of SELF_TYPE work, we need to fully incorporate it into the current type system. We have defined two operations on types: the subtype relationship ($\texttt{T}_1 \leq \texttt{T}_2$) and the least upper bound ($\text{lub}((\texttt{T}_1, \texttt{T}_2))$). They are extended to handle SELF_TYPE as follows.

For subtype relationship:

- $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$ (we never have to compare SELF_TYPE from different classes)
- $\text{SELF\_TYPE}_C \leq \text{T}$ if $\text{C} \leq \text{T}$ (which implies $\text{SELF\_TYPE}_C \leq \text{C}$)
- $\text{T} \leq \text{SELF\_TYPE}_C$ always false

For least upper bound:

- $\text{lub}(\text{SELF\_TYPE}_C, \text{SELF\_TYPE}_C) = \text{SELF\_TYPE}_C$
- $\text{lub}(\text{SELF\_TYPE}_C, \text{T}) = \text{lub}(\text{C}, \text{T})$
- $\text{lub}(\text{T}, \text{SELF\_TYPE}_C) = \text{lub}(\text{C}, \text{T})$

1.4.3 Self type usage

SELF_TYPE is not allowed everywhere a type can appear. Its usage is restricted by the following rules.

1. In an inheritance chain `class T inherits T'`, neither T nor T' can be SELF_TYPE.
2. In an attribute declaration `x:T`, T can be SELF_TYPE.
3. In a let expression `let x:T in E`, T can be SELF_TYPE.

4. In `new T`, T can be SELF_TYPE. It creates an object of the same dynamic type as self.

5. In a static dispatch $\texttt{m@T(E}_1 \ldots \texttt{E}_n\texttt{)}$, T cannot be SELF_TYPE.

6. In a method definition `m(x:T):T'{...}`, only T' can be SELF_TYPE. T cannot be SELF_TYPE because that would result in a $\texttt{T}_0 \leq \texttt{SELF\_TYPE}$ requirement in a dispatch, which is never true. Furthermore, take the following example.

```
1 class A { comp(x : SELF_TYPE) : Bool {...}; };
2 class B inherits A {
3   b : int;
4   comp(x : SELF_TYPE) { x.b <- 0 };
5 }
6 ...
7 let x : A ← new B in x.comp(new A);
8 ...
```

   Here x and new A both have static type A, thus there is no problem during type checking. But at runtime, since x has dynamic type B, it will try to access attribute b of new A when executing comp(), which causes undefined behavior, usually a crash.

1.4.4 Self type checking

A type checking rule $\texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e} : \texttt{T}$ means that an expression e occurring in the body of class C has type T given the variable type environment O and method signatures M. Most type rules using SELF_TYPE remain just the same, except that the $\leq$ and `lub` are the new ones. There are some rules that need to be updated.

The old rule for dispatch requires that the return type is not SELF_TYPE:

$$\frac{\begin{array}{c}\texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_0 : \texttt{T}_0 \\ \texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_1 : \texttt{T}_1 \\ \ldots \\ \texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_n : \texttt{T}_n \\ \texttt{M}(\texttt{T}_0, \texttt{f}) = (\texttt{T}'_1, \ldots, \texttt{T}'_n, \texttt{T}_{n+1}) \\ \texttt{T}_{n+1} \neq \texttt{SELF\_TYPE} \\ \texttt{T}_i \leq \texttt{T}'_i \texttt{ for } 1 \leq \texttt{i} \leq \texttt{n}\end{array}}{\texttt{O}, \texttt{M} \vdash \texttt{e}_0\texttt{.f}(\texttt{e}_1, \ldots, \texttt{e}_n) : \texttt{T}_{n+1}}$$

In case the return type is SELF_TYPE:

$$\begin{array}{c}\texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_0 : \texttt{T}_0 \\ \texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_1 : \texttt{T}_1 \\ \ldots\end{array}$$

$$\frac{\begin{array}{c}\mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_n}:\mathtt{T_n}\\ \mathtt{M}(\mathtt{T_0},\mathtt{f})=(\mathtt{T'_1},\ldots,\mathtt{T'_n},\mathtt{SELF\_TYPE})\\ \mathtt{T_i}\le\mathtt{T'_i}\ \mathtt{for}\ 1\le\mathtt{i}\le\mathtt{n}\end{array}}{\mathtt{O},\mathtt{M}\vdash \mathtt{e_0.f(e_1,\ldots,e_n)}:\mathtt{T_0}}$$

Similarly for static dispatch:

$$\frac{\begin{array}{c}\mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_0}:\mathtt{T_0}\\ \mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_1}:\mathtt{T_1}\\ \ldots\\ \mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_n}:\mathtt{T_n}\\ \mathtt{M}(\mathtt{T},\mathtt{f})=(\mathtt{T'_1},\ldots,\mathtt{T'_n},\mathtt{T_{n+1}})\\ \mathtt{T_0}\le\mathtt{T}\\ \mathtt{T_{n+1}}\neq\mathtt{SELF\_TYPE}\\ \mathtt{T_i}\le\mathtt{T'_i}\quad\mathtt{for}\quad 1\le\mathtt{i}\le\mathtt{n}\end{array}}{\mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_0@T.f(e_1,\ldots,e_n)}:\mathtt{T_{n+1}}}$$

the rule becomes

$$\frac{\begin{array}{c}\mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_0}:\mathtt{T_0}\\ \mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_1}:\mathtt{T_1}\\ \ldots\\ \mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_n}:\mathtt{T_n}\\ \mathtt{M}(\mathtt{T},\mathtt{f})=(\mathtt{T'_1},\ldots,\mathtt{T'_n},\mathtt{SELF\_TYPE})\\ \mathtt{T_0}\le\mathtt{T}\\ \mathtt{T_i}\le\mathtt{T'_i}\quad\mathtt{for}\quad 1\le\mathtt{i}\le\mathtt{n}\end{array}}{\mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{e_0@T.f(e_1,\ldots,e_n)}:\mathtt{T_0}}$$

Note that we are returning type $\mathtt{T_0}$ rather than C, because the type of self (i.e. $\mathtt{T_0}$) can be a subtype of the type in which method f is defined (i.e. C).

There are two new rules due to the introduction of SELF_TYPE:

$$\frac{}{\mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{self}:\mathtt{SELF\_TYPE_C}}$$

$$\frac{}{\mathtt{O},\mathtt{M},\mathtt{C}\vdash \mathtt{newSELF\_TYPE}:\mathtt{SELF\_TYPE_C}}$$

1.4.5 Error recovery

Detecting where errors occur during type checking is easier that during parsing because there is no need to skip over portions of code.

The main problem is what type should be given to an expression with no legitimate type. This type will influence the typing of the enclosing expression.

One choice is to assign type Object to ill-typed expressions. But usually this does not help much because Object does not conform to most of the type rules, thus the error will propagate up the AST, eventually escalating to a series of type errors.

A better approach is to introduce a new type `No_type` for use with ill-typed expressions. It has the property that `No_type`$\leq$`C` for any type C. As a result, every operation is defined for `No_type`. `No_type` will be propagated up the AST just like Object in the previous approach, but the cascading errors disappear. Nonetheless, `No_type` makes the class hierarchy no longer a tree structure. It actually becomes a DAG, which causes implementation difficulty.