Type Checking
Lecture 11

Happy Valentine’s Day
Semantic Analysis

Two broad categories

1. Everything except expressions
   - No multiply defined classes
   - No inheritance cycles
   - No redefining attributes in child classes
   - No overriding methods with different signatures
   - ... a few others

2. Expressions (annotate the AST)
   - **Annotate nodes** of the AST with **Type** information
Semantic Analysis

- Build Class map and Implementation map to perform non-expression analyses
- Build Parent map from class map to detect inheritance cycles
- Detect duplicates in class map to know about multiply defined classes
- et cetera...
Expression Type Checking

- Annotate AST with typing information

```
Plus : Int
  Int: 5
  Minus
    Int: 6
    Int: 3
```

```
Plus : Int
  Int: 5
  Minus: Int
    Int: 6
    Int: 3
```
Inference Rules

- Formal way to infer program properties from syntax
  - More broadly, a formal representation of proofs
  - You can automate it!

\[ \vdash \textit{Hypotheses...} \]
\[ \vdash \textit{Conclusion...} \]

- Inference rules serve as a basis for inferring types of AST nodes!
Why Inference Rules?

Many things can be traced back to formal logic. Classic examples:

- Entscheidungsproblem.
  - Is there an algorithm for determining whether a statement is provable?
- Hilbert’s tenth problem
  - Can we determine integer roots of arbitrary polynomials?

Broader examples:

- Modern criminal law
  - If we prove you did $x$, you are guilty of $y$.
- Modern advertising!
  - Aristotle’s *Rhetoric* relates to formal logic work!
Formal Proofs

*110·632. \( \vdash \mu \in NC. \mathfrak{C}. \mu +_e 1 = \hat{\xi} \{(A_y). y \in \xi. \xi - \xi' y \in sm``\mu\}\)

\textit{Dem.}

\[ \vdash *110·631. *51·211·22. \mathfrak{C} \]

\[ \vdash H_p. \mathfrak{C}. \mu +_e 1 = \hat{\xi} \{(A_y, y). y \in \xi. y \in \xi. y = \xi - \xi' y\} \]

\[ [*13·195] = \hat{\xi} \{(A_y). y \in \xi. \xi - \xi' y \in sm``\mu\}\] : \( \mathfrak{C} \vdash \text{Prop} \)

*110·64. \( \vdash 0 +_e 0 = 0 \) \[ [*110·62] \]

*110·641. \( \vdash 1 +_e 0 = 0 +_e 1 = 1 \) \[ [*110·51·61. *101·2] \]

*110·642. \( \vdash 2 +_e 0 = 0 +_e 2 = 2 \) \[ [*110·51·61. *101·31] \]

*110·643. \( \vdash 1 +_e 1 = 2 \)

\textit{Dem.}

\[ \vdash *110·632. *101·21·28. \mathfrak{C} \]

\[ \vdash 1 +_e 1 = \hat{\xi} \{(A_y). y \in \xi. \xi - \xi' y \in 1\} \]

\[ [*54·3] = 2. \mathfrak{C} \vdash \text{Prop} \]

The above proposition is \textbf{occasionally useful}. It is used at least three times, in *113·66 and *120·123·472.
Inference Rules, Type Checking

\[ \vdash 1 : \text{Int} \quad \vdash 2 : \text{Int} \]
\[ \vdash 1 + 2 : \text{Int} \]

- Type checking proves facts \( e : T \)
  - One type rule is used for each kind of expression

- In the type rule used for a node \( e \)
  - The hypotheses are proofs of types of \( e \)'s subexpressions
  - The conclusion is the proof of the type of \( e \) itself
Soundness and Completeness

- Type system should
  - accept all valid programs
  - reject all invalid programs
- In practice, we must compromise...
  - Cool’s type system rejects all invalid programs
    - But also rejects some valid programs :(

Compiler Construction
Soundness and Completeness

- **Soundness** — We reject all invalid programs
  - A sound jury will convict a guilty person

- **Completeness** — We *never* reject valid programs
  - A complete jury will never convict an innocent person

- Cool’s type system is *sound*, but *incomplete*. 
Soundness and Completeness

```java
1 class Main {
2   a : Int;
3   main () : Object {
4       a <-
5           if ( 1 < 0 ) then
6               "Hello"
7           else
8               5
9           fi
10   };  
11 }
```

Humans can see a will only ever be assigned Int: 5.

► Our sound but incomplete type check will reject this program!
Soundness and Completeness

- We like sound inference rules

\[(i \text{ is an integer constant}) \quad \vdash i : \text{Object}\]

- But we also want precise rules

\[(i \text{ is an integer constant}) \quad \vdash i : \text{Int}\]
Typing: Example

Consider while not false loop 1 + 2 * 3 pool

While
Typing: Example

Consider while not false loop 1 + 2 * 3 pool
Typing: Example

Consider while not false loop 1 + 2 * 3 pool

Diagram:

```
  While
    not
      false
```
Typing: Example

Consider while not false loop 1 + 2 * 3 pool

While

not

false : Bool
Typing: Example

Consider while not false loop 1 + 2 * 3 pool
Typing: Example

Consider while not false loop 1 + 2 * 3 pool

```
While
  not : Bool
  false : Bool
  Plus
```

Compiler Construction
Typing: Example

Consider while not false loop 1 + 2 * 3 pool
Typing: Example

Consider \texttt{while not false loop 1 + 2 * 3 pool}
Typing: Example

Consider \texttt{while not false loop 1 + 2 * 3 pool}

![Diagram]

- \texttt{While}
  - \texttt{not : Bool}
    - True
  - \texttt{Plus}
    - \texttt{1 : Int}
Typing: Example

Consider `while not false loop 1 + 2 * 3 pool`
Typing: Example
Consider while not false loop 1 + 2 * 3 pool
Typing: Example

Consider while not false loop 1 + 2 * 3 pool
Typing: Example

Consider while not false loop 1 + 2 * 3 pool

```
While
  not : Bool
    false : Bool
  Plus
    1 : Int
  Multiply
    2 : Int
    3
```
Typing: Example

Consider \texttt{while not false loop 1 + 2 * 3 pool}

![Diagram of the expression](image)
Typing: Example

Consider while not false loop 1 + 2 * 3 pool

```
While
  not : Bool
    false : Bool
  Plus
    1 : Int
    Multiply : Int
      2 : Int
      3 : Int
```

Compiler Construction
Typing: Example

Consider `while not false loop 1 + 2 * 3 pool`
Typing: Example

Consider \texttt{while not false loop 1 + 2 \ast 3 pool}

\begin{itemize}
  \item \texttt{While : Object}
  \item \texttt{not : Bool}
    \item \texttt{false : Bool}
  \item \texttt{Plus : Int}
    \item \texttt{1 : Int}
    \item \texttt{Multiply : Int}
      \item \texttt{2 : Int}
      \item \texttt{3 : Int}
\end{itemize}
Typing Derivations

The typing reasoning can be expressed with a recursive application of inference rules!

\[
\begin{align*}
\vdash \text{false} : \text{Bool} & \quad \vdash 2 : \text{Int} \quad \vdash 3 : \text{Int} \\
\vdash \text{not false} : \text{Bool} & \quad \vdash 1 : \text{Int} \quad \vdash 2 \times 3 : \text{Int} \\
\vdash \text{while not false loop} 1 + 2 \times 3 : \text{Object} & \\
\end{align*}
\]

- The **root** of this tree is a whole expression
- Each node is an **instance** of a typing rule
- **Leaves** are the rules without hypotheses
Next up: \( O, M, C \)

- How to annotate variable references?

\[ \Gamma \vdash x : ? \text{Var} \quad (x \text{ is an identifier}) \]

The local structural rule does not carry enough information to give \( x \) a type. Fail.
Type Environments

A type environment gives types for free variables

- A type environment maps Identifiers to Types
- a variable is free in an expression if
  - The expression contains an occurrence of the variable that refers to the declaration outside the expression

- In “x”, the variable “x” is free
- in “let x : Int in x + y”, only “y” is free
- in “x + let x : Int in x + y” both “x”, “y” are free
Type Environments

We use $O$ as a function from Identifiers to Types

The sentence $O \vdash e : T$

means: Assuming that variables have types given by $O$, it is provable that the expression $e$ has type $T$. 
Inference rules

Now we can include more information in inference rules:

\[ O \vdash i : \text{Int} \quad [\text{Int}] \quad (i \text{ is an integer}) \]

\[ O \vdash e_1 : \text{Int} \quad O \vdash e_2 : \text{Int} \]

\[ O \vdash e_1 + e_2 : \text{Int} \quad [\text{Add}] \]
Using $O$ in Let

We can create a new rule for Identifiers

$$
O(x) = T \\
O \vdash x : T \quad \text{[Var]}
$$

For our Let inference rule:

$$
O[T_0/x] \vdash e_1 : T_1 \\
O \vdash \text{let } x : T_0 \text{ in } e_1 : T_1 \quad \text{[Let-no-init]}
$$

Here, $O[T_0/x]$ means “$O$ modified to map $x$ to type $T_0$, but otherwise unchanged.”

▶ This is our formal notation for pushing a new entry in the symbol table
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

\(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)
Example Let Typing

Consider:
let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)

$E_{x,y}, F_{x,y}$ are expressions that use $x$ and $y$
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

\(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)

\[O \vdash \text{let } x : T_0 \text{ in}
\]

\[O[T_0/x] \vdash \text{Plus}
\]

\[O[T_0/x] \vdash \text{let } x : T_2 \text{ in}
\]

\[E_{x,y}
\]

\[F_{x,y}
\]

\[x
\]
Example Let Typing

Consider:
let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)

- $E_{x,y}, F_{x,y}$ are expressions that use x and y

$$\text{let } x : T_0 \text{ in } \text{Plus}$$

$$\text{let } y : T_1 \text{ in } E_{x,y}$$

$$\text{let } x : T_2 \text{ in } F_{x,y}$$

$$O \vdash$$

$$O[T_0/x] \vdash$$

$$O[T_0/x] \vdash$$

Compiler Construction
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

\(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)

\(O \vdash E_{x,y}, y, F_{x,y}\) are expressions that use \(x\) and \(y\)

\(O[T_0/x] \vdash \text{let } x : T_0 \text{ in }\)

\( O[T_0/x] \vdash \text{let } y : T_1 \text{ in }\)

\( O[T_0/x] \vdash \text{let } x : T_2 \text{ in }\)

\((O[T_0/x])[T_1/y] \vdash\)

\[X\]
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

- \(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)

\[O \vdash \]

\[O[T_0/x] \vdash \]

\[O[T_0/x] \vdash \]

\[O[T_0/x] \vdash \]

\[(O[T_0/x])[T_1/y] \vdash \]

\[(O[T_0/x])[T_1/y] \vdash \]

\[E_{x,y} \]

\[F_{x,y} \]
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

\[\vdash E_{x,y}, F_{x,y} \text{ are expressions that use } x \text{ and } y\]

\[
O \vdash \text{let } x : T_0 \text{ in }
\]

\[
O[T_0/x] \vdash
\]

\[
O[T_0/x] \vdash \text{let } y : T_1 \text{ in }
\]

\[
O[T_0/x] \vdash \text{let } x : T_2 \text{ in }
\]

\[
(O[T_0/x])[T_1/y] \vdash E_{x,y}
\]

\[
(O[T_0/x])[T_1/y] \vdash F_{x,y}
\]

\[
(O[T_0/x])[T_2/x] \vdash
\]

\[
\text{Compiler Construction}
\]
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

- \(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)

\[O \vdash \]

\[O[T_0/x] \vdash \]

\[O[T_1/x] \vdash \]

\[O[T_2/x] \vdash \]

\[(O[T_0/x])[T_1/y] \vdash \]

\[(O[T_0/x])[T_2/x] \vdash \]

\[x : T_0 \]

\[E_{x,y} \]

\[F_{x,y} \]

Compiler Construction
Example Let Typing

Consider:
\[ \text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y}) \]

\[ E_{x,y}, F_{x,y} \text{ are expressions that use } x \text{ and } y \]

\[ O \vdash \text{let } x : T_0 \text{ in} \]

\[ O[T_0/x] \vdash \text{let } y : T_1 \text{ in} \]

\[ O[T_0/x] \vdash \text{let } x : T_2 \text{ in} \]

\[ (O[T_0/x])[T_1/y] \vdash E_{x,y} : \text{Int} \]

\[ (O[T_0/x])[T_2/x] \vdash F_{x,y} \]

\[ (O[T_0/x])[T_1/y] \vdash x : T_0 \]
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

- \(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)

\[
\O \vdash \text{let } x : T_0 \text{ in }
\]

\[
\O[T_0/x] \vdash \text{let } x : T_0 \text{ in }
\]

\[
\O[T_0/x] \vdash \text{let } y : T_1 \text{ in } : \text{Int}
\]

\[
\O[T_0/x] \vdash \text{let } x : T_2 \text{ in }
\]

\[
\O[T_0/x] \vdash E_{x,y} : \text{Int}
\]

\[
\O[T_0/x] \vdash F_{x,y}
\]

\[
\O[T_0/x][T_1/y] \vdash x : T_0
\]

\[
\O[T_0/x][T_1/y] \vdash E_{x,y} : \text{Int}
\]

\[
\O[T_0/x][T_2/x] \vdash F_{x,y}
\]
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

- \(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)

\[
O \vdash \text{let } x : T_0 \text{ in } \text{let } y : T_1 \text{ in } E_{x,y}, y : \text{Int}
\]

\[
O \vdash \text{let } x : T_2 \text{ in } F_{x,y}
\]

\[
O[T_0/x] \vdash \text{let } y : T_1 \text{ in } \text{Int}
\]

\[
(O[T_0/x])[T_1/y] \vdash E_{x,y} : \text{Int}
\]

\[
(O[T_0/x])[T_2/x] \vdash F_{x,y}
\]
Example Let Typing

Consider:

let x : $T_0$ in (let y : $T_1$ in $E_{x,y}$) + (let x : $T_2$ in $F_{x,y}$)

$E_{x,y}, F_{x,y}$ are expressions that use x and y
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

- \(E_{x,y}, F_{x,y}\) are expressions that use \(x\) and \(y\)

\[
\begin{align*}
O \vdash & \quad \text{let } x : T_0 \text{ in } \\
O[T_0/x] \vdash & \quad \text{Plus} \\
O[T_0/x] \vdash & \quad \text{let } y : T_1 \text{ in } : \text{Int} \\
O[T_0/x] \vdash & \quad \text{let } x : T_2 \text{ in } : \text{Int} \\
(O[T_0/x])[T_1/y] \vdash & \quad E_{x,y} : \text{Int} \\
(O[T_0/x])[T_2/x] \vdash & \quad F_{x,y} : \text{Int}
\end{align*}
\]
Example Let Typing

Consider:

\[ \text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y}) \]

\[ \quad \begin{align*}
E_{x,y}, F_{x,y} & \text{ are expressions that use } x \text{ and } y \\
O \vdash & \text{let } x : T_0 \text{ in } \\
O[T_0/x] \vdash & \text{Plus : Int} \\
O[T_0/x] \vdash & \text{let } y : T_1 \text{ in : Int} \\
O[T_0/x] \vdash & \text{let } x : T_2 \text{ in : Int} \\
(O[T_0/x])[T_1/y] \vdash & E_{x,y} : \text{Int} \\
(O[T_0/x])[T_2/x] \vdash & F_{x,y} : \text{Int} \\
(x : T_0) & \text{Compiler Construction}
\end{align*} \]
Example Let Typing

Consider:

\[
\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})
\]

\( E_{x,y}, F_{x,y} \) are expressions that use \( x \) and \( y \)

\[
O \vdash \text{let } x : T_0 \text{ in : Int}
\]

\[
O[T_0/x] \vdash \text{let } y : T_1 \text{ in : Int}
\]

\[
O[T_0/x] \vdash \text{let } x : T_2 \text{ in : Int}
\]

\[
O[T_0/x] \vdash E_{x,y} : \text{Int}
\]

\[
O[T_0/x] \vdash F_{x,y} : \text{Int}
\]

Compiler Construction
Remarks

- The type environment $O$ gives types to the free identifiers in the current scope
  - We implemented this with the symbol table in the example code

- The type environment is passed down the AST from the root towards the leaves
  - We saw the symbol table passed recursively through the type checking function

- The types are computed up the AST from the leaves towards the root!
Let with Initialization

\[
O \vdash e_0 : T_0 \quad O[T_0/x] \vdash e_1 : T_1
\]

\[
O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1
\]

What if we had “let x : P <- new C in...”?

▶ This rule is *too weak* or *incomplete*

Recall

\[
(i \text{ is an integer})
\]

\[
O \vdash i : \text{Object}
\]
Subtyping

We use the relation $X \leq Y$ to say:

- An object of type $X$ could be used when one of type $Y$ is acceptable
- $X$ conforms with $Y$
- $X$ is a *subclass* of $Y$

Definitions:

$X \leq X$

$X \leq Y$ if $X$ inherits from $Y$

$X \leq Z$ if $X \leq Y$ and $Y \leq Z$
Let with Initialization

\[
O \vdash e_0 : T \quad T \leq T_0 \quad O[T_0/x] \vdash e_1 : T_1
\]

\[
O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1
\]

- Both rules are sound
- But this one is more complete (we reject fewer correct programs)
- Flexible rules do not constrain programming
- Restrictive rules ensure safety of execution
- Expressive type systems are more complex
Static and Dynamic Types

- The **dynamic type** of an object is the class $C$ that is used in “new $C$” expression that creates the object
  - Dynamic type is a **runtime notion**

- The **static type** of an expression is a notation that captures all possible dynamic types the expression could take
  - A **compile time notion**
A variable of static type \( A \) can hold values of static type \( B \) if \( B \leq A \)
Soundness in Cool

Soundness theorem:

\[ \forall E. \text{dynamic	extunderscore type}(E) \leq \text{static	extunderscore type}(E) \]

- For \(E\), the compiler uses \text{static	extunderscore type}(E)
- All operations that can be used on an object of type \(C\) can also be used on an object of type \(C' \leq C\).
  - Fetching the value of an attribute
  - Invoking a method on the object
- Subclass can only add attributes or methods
- Methods can be redefined, but only with the same types
Subtyping in Inference Rules

Recall

\[
\text{O} \vdash e_0 : T \quad T \leq T_0 \quad \text{O}[T_0/x] \vdash e_1 : T_1
\]

\[
\text{O} \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1
\]

Consider: Why is the rule below wrong?

\[
\text{O} \vdash e_0 : T \quad T_0 \leq T \quad \text{O}[T_0/x] \vdash e_1 : T_1
\]

\[
\text{O} \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1
\]
Typing Rule Notation

- Typing rules are carefully constructed
- Slight changes in a rule make the type system:
  - unsound
    - (bad programs are accepted as well-typed)
  - incomplete (i.e., less usable)
    - (good programs are rejected as badly-typed)
- N.B. Some good programs will be rejected anyway
  - “Good program” is undecidable
If Rule

Consider “if $e_0$ then $e_1$ else $e_2$ fi”

- The *dynamic* type is either $e_1$’s or $e_2$’s type

The best we can do *statically* is the smallest supertype larger than the type of $e_1$ and $e_2$

```java
1 class P {
2 class A inherits P{}; class B inherits P{};
3 class Main {
4     main () : Object{
5         if ... then
6             new A
7         else
8             new B
9         fi
10     }
11 }
```

We need to allow for the dynamic type of the if-expression to be $A$ or $B$.

- Small supertype is $P$
Least Upper Bounds

We use $lub(X, Y)$ to be the least upper bound of $X$ and $Y$. $lub(X, Y) = Z$ if:

- $X \leq Z \land Y \leq Z$
  - $Z$ is an upper bound
- $X \leq Z' \land Y \leq Z' \Rightarrow Z \leq Z'$
  - $Z$ is least among the upper bounds

In Cool, the least upper bound of two types is their least common ancestor in the inheritance tree.
If-Then-Else Rule

\[
\begin{align*}
O \vdash e_0 &: Bool & O \vdash e_1 &: T_1 & O \vdash e_2 &: T_2 \\
O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} &: \text{lub}(T_1, T_2)
\end{align*}
\]

1 a : A
2 method () : Object {
3 a <- if e0 then e1 else e2 fi
4 }

You can use the rule to annotate the if expression, then check whether it conforms to the type of a!
Summary so far

- Type rules are based on rules of inference
- We implement the type rules to annotate a program’s AST
  - We reject programs whose types violate our type rules.
- The type system ideally would accept all valid programs and reject all invalid programs (sound and complete)
  - Reminder: it’s not ideal.
- We use subtyping relations ($\leq$) and least-upper-bounds ($lub$) as part of implementing our type-checking rules.
Recall: Dispatch means method calls

Much like introducing $O$ for identifiers, we have a similar problem for method calls.

\[
\begin{align*}
O & \vdash e_0 : T_0 \\
O & \vdash e_1 : T_1 \\
& \vdots \\
O & \vdash e_n : T_n \\
O & \vdash e_0.f(e_1, \ldots, e_n) : ???
\end{align*}
\]

We need information about the formal parameters and return type of $f$. 
Method environment $M$

- Recall you can name an attribute the same as a method in a class.
  - A method `foo` and an object `foo` can coexist in the same scope, but different name spaces.
- The typing rules for Cool use the $M$ function as a separate mapping from $O$

\[ M(C, f) = (T_1, ..., T_n, T_{ret}) \]
means in Class $C$ there is a method $f$

\[ f(x_1 : T_1, ..., x_n : T_n) : T_{ret} \]
Extended Typing Judgments

Now we have two environments: $O$ and $M$.

$$O, M \vdash e : T$$

is read as “assuming that object identifiers have types given by $O$, and method identifiers have signatures as given by $M$, the expression $e$ has type $T$.”
Method Environment

- Method environment $M$ must be added to the rules
- Most of the time, $M$ is not really used

\[
\frac{O, M \vdash e_1 : T_1 \quad O, M \vdash e_2 : T_2}{O, M \vdash e_1 + e_2 : \text{Int}}
\]

Hint: only dispatch uses $M$
Dispatch Rule

\[ O, M \vdash e_0 : T_0 \quad \text{Check receiver object } e_0 \]

\[ O, M \vdash e_1 : T_1 \quad \text{Check passed arguments} \]

\[ \ldots \]

\[ O, M \vdash e_n : T_n \quad \text{Check passed arguments} \]

\[ M(T_0, f) = (T'_1, \ldots, T'_n, T'_{n+1}) \]

\[ T_i \leq T'_{i} \quad \text{(for } 1 \leq i \leq n) \]

\[ O, M \vdash e_0.f(e_1, \ldots e_n) : T'_{n+1} \quad \text{Look up types from function definition} \]
Static Dispatch

Just a special case of dispatch. The developer gives us an explicitly-named class (i.e., we do not check it via $e_0$)

- The inferred type of the dispatch must conform to the specified type
Static Dispatch

\[ O, M \vdash e_0 : T_0 \]
\[ O, M \vdash e_1 : T_1 \]

...\[
\]
\[ O, M \vdash e_n : T_n \]
\[ T_0 \leq T \]
\[ M(T, f) = (T_1', ..., T_i', T_{n+1}') \]
\[ T_i \leq T_i' \quad \text{(for } 1 \leq i \leq n) \]

\[ O, M \vdash e_0@T.f(e_1, ... e_n) : T_{n+1}' \]
SELF_TYPE

We want the type system to:

- Give flexibility to the programmer
- Prevent the programmer from “going wrong”
Example

class Count {
  i : Int <- 0
  inc () : Count {
    i <- i + 1;
    self;
  }
}

class Stock inherits Count {
  name() : String { ... } ; -- something
}

class Main {
  a : Stock <- (new Stock).inc();
  ... a.name() ...        
}

Type checking error!
Example Post-Mortem

- (new Stock).inc() has **dynamic** type Stock
  - Should be legitimate to use
    
    ```
    a : Stock <- (new Stock).inc()
    ```

- Not well-typed because (new Stock).inc() has **static** type **Count**

- We have “lost” type information

- We would have to redefine the inc method for every subclass with a specialized return type
Enter SELF_TYPE

- We extend the type system
- Insights
  - inc returns “self”
  - Therefore the return value should be the same type as “self”
  - Which could be Count or any subtype of Count
  - e.g., (new Stock).inc() should give type Stock
- SELF_TYPE is a keywork we use for the return value of such methods
  - We must edit the typing rules
SELF_TYPE

Modify our inc method to read:
inc() : SELF_TYPE ...

➤ Now, our type checker can prove

\[ O, M \vdash (\text{new Count}).\text{inc}() : \text{Count} \]
\[ O, M \vdash (\text{new Stock}).\text{inc}() : \text{Stock} \]

➤ SELF_TYPE is a static type, not a dynamic type
Notes on SELF_TYPE

- SELF_TYPE refers to the dynamic type of the “self” expression.

- N.B. The meaning of SELF_TYPE depends on where it appears
  - We use SELF_TYPE_C to refer to an occurrence of SELF_TYPE in the body of class C.
Type Checking SELF_TYPE

- While $\text{SELF\_TYPE}_C \leq C$, it is incorrect to replace $\text{SELF\_TYPE}_C$ with $C$
  - That would be like not having SELF_TYPE in the first place
Extending $\leq$ for SELF_TYPE

We extend $\leq$ to handle SELF_TYPE

- $\text{SELF_TYPE}_C \leq T$ if $C \leq T$
  - $\text{SELF_TYPE}_C$ can be any subtype of $C$, including $C$

- $\text{SELF_TYPE}_C \leq \text{SELF_TYPE}_C$
  - It is the type of the “self” expression
  - We never need to compare SELF_TYPE$_A$ to SELF_TYPE$_B$

- $T \leq \text{SELF_TYPE}_C$ always false

We can also extend $lub$. 
lub with SELF_TYPE

\[ \text{lub}(\text{SELF\_TYPE}_C, \text{SELF\_TYPE}_C) = \text{SELF\_TYPE}_C \]

\[ \text{lub}(\text{SELF\_TYPE}_C, T) = \text{lub}(C, T) \]

\[ \text{lub}(T, \text{SELF\_TYPE}_C) = \text{lub}(C, T) \]
Inference Rules with SELF_TYPE

Because occurrences of SELF_TYPE depend on the enclosing class, we must carry more information during type checking.

We introduce \( C \) to our typing judgments:

\[
O, M, C \vdash e : T
\]

An expression \( e \) occurring in the body of \( C \) has static type \( T \) given variable environment \( O \) and method signatures \( M \).
Rules for SELF_TYPE

\[ O, M, C \vdash \text{self: SELF_TYPE}_c \]

\[ O, M, C \vdash \text{new SELF_TYPE : SELF_TYPE}_C \]
Remarks

- Read the CRM for restrictions on where SELF_TYPE can be used
- Think about how you would modify the dispatch rules shown previously to handle SELF_TYPE
- SELF_TYPE is a static type we use to represent a dynamic type during execution
  - This feature of our type system increases the type system’s expressiveness (but adds complexity)
- There is a tradeoff between flexibility and safety