Lecture 10. Types and Type Checking

Wei Le

2015.9
Type

- Type is a property of program constructs such as expressions.
- It defines a set of values (range of variables) and a set of operations on those values.
- Classes are one instantiation of the modern notion of the type:
  - fields and methods of a Java class are meant to correspond to values and operations.
A type system is a collection of rules that assign types to program constructs (more constraints added to checking the validity of the programs, violation of such constraints indicate errors).

A language's type system specifies which operations are valid for which types.

Type systems provide a concise formalization of the semantic checking rules.

Type rules are defined on the structure of expressions.

Type rules are language specific.
Why do we need type systems

Consider the assembly language fragment

```
addi $r1, $r2, $r3
```

What are the types of $r1, $r2, $r3?
Why do we need type systems

Consider the assembly language fragment

```
addi $r1, $r2, $r3
```

What are the types of $r1, $r2, $r3?

- Assembly language is untyped (MIPS assembly)
Why do we need type systems

Consider the assembly language fragment

\texttt{addi \$r1, \$r2, \$r3}

What are the types of $r1$, $r2$, $r3$?

- Assembly language is untyped (MIPS assembly)
- This instruction allows you to add the contents of a register to an immediate value (a constant) and store the result in a (possibly) another register.
Why do we need type systems

- It doesn't make sense to add a function pointer and an integer in C
- It does make sense to add two integers
- But both have the same assembly language implementation!
Use of Types

- Detect errors:
  - Memory errors, such as attempting to use an integer as a pointer.
  - Violations of abstraction boundaries, such as using a private field from outside a class.

- Help compilation:
  - When Python sees \( x+y \), its type systems tells it almost nothing about types of \( x \) and \( y \), so code must be general.
  - In C, C++, Java, code sequences for \( x+y \) are smaller and faster, because representations are known.
Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The two are different, but are often used interchangeably
Inference Rule

- Types of expressions and parameters need not be explicit to have static typing. With the right rules, might infer their types.
- The appropriate formalism for type checking is logical rules of inference having the form
  
  If Hypothesis is true, then Conclusion is true

- For type checking, this might become:
  
  If $E_1$ and $E_2$ have certain types, then $E_3$ has a certain type.

- Given proper notation, easy to read (with practice), so easy to check that the rules are accurate.

- Can even be mechanically translated into programs.
From English to an Inference Rule

- Rules of inference are a compact notation of *if-then* statements
- Symbol $\land$ is "and"
- Symbol $\Rightarrow$ is "if-then"
- $x : T$ is "$x$ has type $T$"
From English to an Inference Rule

If $e_1$ has type $\text{Int}$ and $e_2$ has type $\text{Int}$, then $e_1 + e_2$ has type $\text{Int}$

$(e_1 \text{ has type } \text{Int} \land e_2 \text{ has type } \text{Int}) \Rightarrow e_1 + e_2 \text{ has type } \text{Int}$

$(e_1 : \text{Int} \land e_2 : \text{Int}) \Rightarrow e_1 + e_2 : \text{Int}$
From English to an Inference Rule

The statement

\[(e_1: \text{Int} \land e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}\]

is a special case of

\[(\text{Hypothesis}_1 \land \ldots \land \text{Hypothesis}_n) \Rightarrow \text{Conclusion}\]

This is an *inference rule*
From English to an Inference Rule

- By tradition inference rules are written
  \[ \vdash \text{Hypothesis}_1 \ldots \vdash \text{Hypothesis}_n \]
  \[ \vdash \text{Conclusion} \]

- Cool type rules have hypotheses and conclusions of the form:
  \[ \vdash e : T \]
- \( \vdash \) means “it is provable that . . .”
Example

\[
\frac{i \text{ is an integer}}{\vdash i : \text{Int}} \quad [\text{Int}]
\]

\[
\vdash e_1 : \text{Int} \\
\vdash e_2 : \text{Int} \\
\vdash e_1 + e_2 : \text{Int} \quad [\text{Add}]
\]
Two Rules

- These rules give templates describing how to type integers and + expressions.

- By filling in the templates, we can produce complete typings for expressions.
Example

\[
\begin{align*}
\text{1 is an integer} & \quad \vdash 1 : \text{Int} \\
\text{2 is an integer} & \quad \vdash 2 : \text{Int} \\
\hline
\vdash 1 + 2 : \text{Int}
\end{align*}
\]
Static and Dynamic Typed Languages

- Statically typed languages: all or almost all type checking occurs at compilation time. (C, Java)
- Dynamically typed languages: almost all checking of types is done as part of program execution (Scheme)
- Untyped languages: no type checking (assembly, machine code)
Static and Dynamic Types

- The **dynamic type** of an object is the class \( C \) that is used in the “new \( C \)” expression that creates the object
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type

- The **static type** of an expression is a notation that captures all possible dynamic types the expression could take
  - A compile-time notion
Relations of Static and Dynamic Types in Simple Type Systems

Soundness theorem: for all expressions $E$

\[
dynamic\_type(E) = static\_type(E)
\]

(in all executions, $E$ evaluates to values of the type inferred by the compiler)
So far:

A set of basic concepts:

- type
- type systems
- type checking
- type inference
- inference rules
- static and dynamic typed languages
- static and dynamic types
Semantic Analysis Related to Types

Goals:
- What is the type of the expression – type inference (what is the value the expression potentially produces? based on its range, what type it is?)
- Do we have a correct assignment (following type rules) of types on all the expressions in the program? – type checking

Perspectives of studying types:
- Language designers: designing the type systems
- Compilers: type checking programs

Next:
- We do an overview of the two perspectives
Designing a Type System

Two Conflict Goals:

- Give flexibility to the programmer

- Prevent valid programs to “go wrong”
  - Milner, 1981: “Well-typed programs do not go wrong”

• An active line of research is in the area of inventing more flexible type systems while preserving soundness

In another word: There is a tradeoff between

► Flexible rules that do not constrain programming
► Restrictive rules that ensure safety of execution
Soundness

- It is a property of the type system
- Intuitively, a sound type system can correctly predict the type of a variable at runtime
- There can be many sound type rules, we need to use the most precise ones so it can be useful

• A type system is **sound** if
  - Whenever $\vdash e : T$
  - Then $e$ evaluates to a value of type $T$

• We only want sound rules
  - But some sound rules are better than others:

  $\underbrace{i \text{ is an integer}}_{\vdash i : \text{Object}}$
Tradeoffs of Static and Dynamic Type Checking Systems

- static type system does not have knowledge of input values or execution behaviors
- static type system disallows some correct programs, cannot predict precisely all the behaviors (some program runs correctly will be rejected)
- better static type system or dynamic type system

**Static**
- Static checking catches many programming errors at compile time
- Avoids overhead of runtime type checking
- Using various devices to recover the flexibility lost by ”going static:” subtyping, coercions, type parameterization

**Dynamic:**
- Static type systems are restrictive; can require more work to do reasonable things.
- Rapid prototyping easier in a dynamic type system.
Using Subtypes

- In languages such as Java, can define types (classes) either to
  - Implement a type, or
  - Define the operations on a family of types without (completely) implementing them
- Hence, relaxes static typing a bit: we may know that something is a \( Y \) without knowing precisely which subtype it has
Implicit Coercions

• In Java, can write

```java
    int x = 'c';
    float y = x;
```

• But relationship between `char` and `int`, or `int` and `float` not usually called subtyping, but rather conversion (or coercion).

• Such implicit coercions avoid cumbersome casting operations.

• Might cause a change of value or representation,

• But usually, such coercions allowed implicitly only if type coerced to contains all the values of the that coerced from (a widening coercion).

• Inverses of widening coercions, which typically lose information (e.g., `int→char`), are known as narrowing coercions. and typically required to be explicit.

• `int→float` a traditional exception (implicit, but can lose information and is neither a strict widening nor a strict narrowing.)
Coercion Examples

Object x = ...; String y = ...;
int a = ...; short b = 42;
x = y; a = b;  // OK
y = x; b = a;  // ERRORS { x = (Object) y; // {OK
a = (int) b;  // OK
y = (String) x;  // OK but may cause exception
b = (short) a;  // OK but may lose information

Possibility of implicit coercion complicates type-matching rules (see C++).
Type Checking Algorithm

- Type checking proves facts $e : T$
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each kind of AST node
- In the type rule used for a node $e$:
  - The hypotheses are the proofs of types of $e$'s subexpressions
  - The conclusion is the proof of type of $e$
- Types are computed in a bottom-up pass over the AST
One Pass Type Checking for Cool

• COOL type checking can be implemented in a single traversal over the AST

• Type environment is passed down the tree
  - From parent to child

• Types are passed up the tree
  - From child to parent
So far:

- Concepts
- Overview of type system design and type checking algorithms

Next: COOL (Why Cool is a good language to learn when learning basic concepts?)

- Cool type system touches all the key concepts: e.g., subtyping, method dispatch
Cool Types

- Class names
- SELF_TYPE  Note: there are no base types (as in Java int)
- The user declares types for all identifiers
- The compiler infers types for expressions (Infers a type for every expression)
Rules for Constant

\[
\vdash \text{false} : \text{Bool} \quad \text{[Bool]}
\]

\[
\vdash \text{s is a string constant} \quad \text{[String]}
\]

\[
\vdash \text{s} : \text{String}
\]
Rules for New

new $T$ produces an object of type $T$
- Ignore SELF_TYPE for now ...

\[ \vdash \text{new } T : T \quad [\text{New}] \]
Two More Rules

\[ \vdash e : \text{Bool} \]
\[ \vdash \text{not } e : \text{Bool} \quad \text{[Not]} \]

\[ \vdash e_1 : \text{Bool} \]
\[ \vdash e_2 : \text{T} \]
\[ \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object} \quad \text{[Loop]} \]
Type Inference: Determining Types for Every AST Node

- Typing for `while not false loop 1 + 2 * 3 pool`

```
while  loop  pool : Object
     /    \    
not : Bool + : Int
  /     \       |
false : Bool 1 : Int * : Int
     /     |
    2 : Int 3 : Int
```
Type Derivations

- The typing reasoning can be expressed as a tree:

  \[ \vdash \text{false} : \text{Bool} \]
  \[ \vdash \text{not false} : \text{Bool} \]
  \[ \vdash \text{while not false loop } 1 + 2 * 3 : \text{Object} \]

  \[ \vdash 2 : \text{Int} \quad \vdash 3 : \text{Int} \]
  \[ \vdash 1 : \text{Int} \quad \vdash 2 * 3 : \text{Int} \]
  \[ \vdash 1 + 2 * 3 : \text{Int} \]

- The root of the tree is the whole expression
- Each node is an instance of a typing rule
- Leaves are the rules with no hypotheses
Get Into More Complicated Rules

Important ones:
- Let
- If-then-else, case
- Method
- Self_Type

Pay attention to:
- notation and concept development
- typing rule design
Type Rules and Type Environment

- 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.
Type Rules and Type environment

Type rules have general format:

\[
\begin{array}{c}
\vdots \\
O, M, C \vdash e : T
\end{array}
\]

- \(O\) environment for object
- \(M\) environment for methods
- \(C\) containing class
- 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.
Let $O$ be a function from ObjectIdentifiers to Types

The sentence $O \vdash e : T$

is read: Under the assumption that variables have the types given by $O$, it is provable that the expression $e$ has the type $T$
A type environment gives types for free variables

- A type environment is a function from ObjectIdentifiers to Types
- A variable is free in an expression if:
  • It occurs in the expression
  • It is declared outside the expression

- E.g. in the expression “x”, the variable “x” is free
- E.g. in “let x : Int in x + y” only “y” is free
Notation Understanding

- $O$ is a function (implemented in the symbol table: mapping between variables and types)
- $O[T/x]$ is a also function, extend $O$ with a pair of: $x$ of type $T$
- The type environment provides the type of free variables in the current scope
- During type checking, you can look for the type of a particular variable on the AST in the type environment $O$
- $O[T/x] ⊢ e : T$ execute the expression $e$ in this environment, and get the type $T$

Example:

$O[T_0/x](x) = T_0$
$O[T_0/x](y) = O(y)$
An Example

\[
\begin{align*}
\text{i is an integer} & \quad O \vdash i : \text{Int} \\
& \quad [\text{Int}] \\
O \vdash e_1 : \text{Int} & \quad O \vdash e_2 : \text{Int} \\
& \quad O \vdash e_1 + e_2 : \text{Int} \\
& \quad [\text{Add}]
\end{align*}
\]
New Rule

\[
\frac{O(x) = T}{O \vdash x : T} \quad [\text{Var}]
\]
Let: No Initialization

\[
\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \text{ in } e_1 : T_1}
\]  
[Let-No-Init]
Let Example

• Consider the Cool expression
  
  let x : T_0 in (let y : T_1 in E_{x,y}) + (let x : T_2 in F_{x,y})
  
  (where E_{x,y} and F_{x,y} are some Cool expression that contain occurrences of “x” and “y”)

• Scope
  - of “y” is E_{x,y}
  - of outer “x” is E_{x,y}
  - of inner “x” is F_{x,y}

• This is captured precisely in the typing rule
Let Example

AST
Type env.
Types

\( O \vdash \text{let } x : T_0 \text{ in } : \text{int} \)

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

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

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

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

\( (O[T_0/x])[T_2/x] \vdash F(x, y) : \text{int} \)
Type Inference Approach

- The type environment gives types to the free identifiers in the current scope
- The type environment is passed down the AST from the root towards the leaves
- Types are computed up the AST from the leaves towards the root
Let With Initialization

Now consider let with initialization:

\[
O \vdash e_0 : T_0 \\
O[T_0/x] \vdash e_1 : T_1 \\
\overline{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}
\] [Let-Init]

This rule is weak. Why?
Let With Initialization

• Consider the example:

```plaintext
class C inherits P { ... }

... let x : P ← new C in ...

...```

• The previous let rule does not allow this code
  - We say that the rule is too weak
Subtyping

- Define a relation $\leq$ on classes
  
  $X \leq X$
  
  $X \leq Y$ if $X$ inherits from $Y$
  
  $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

- Reflexive
- Transitive
Let with Initialization Modified

\[
\begin{align*}
O \vdash e_0 : T \\
T \leq T_0 \\
O[T_0/x] \vdash e_1 : T_1
\end{align*}
\text{[Let-Init]}
\]

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

- Both rules for let are correct
- But more programs type check with the latter
Let with Initialization – More Examples

How it is different from the previous Let rule?

\[
\frac{O \vdash e_0 : T \quad T \leq T_0 \quad O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}
\]
Examples of Wrong Typing Rules

• The following good program does not typecheck

\[
\text{let } x : \text{Int} \leftarrow 0 \text{ in } x + 1
\]

• Why?
Examples of Wrong Typing Rules

- Consider the following Cool class definitions

  ```
  Class A { a() : int { 0 }; }
  Class B inherits A { b() : int { 1 }; }
  ```

- An instance of B has methods “a” and “b”
- An instance of A has method “a”
  - A type error occurs if we try to invoke method “b” on an instance of A
Examples of Wrong Typing Rules

• Now consider a hypothetical let rule:

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

• How is it different from the correct rule?

• The following bad program is well typed

\[
\text{let } x : B \leftarrow \text{new } A \text{ in } x.b()
\]

• Why is this program bad?
Examples of Wrong Typing Rules

• Now consider a hypothetical let rule:

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

• How is it different from the correct rule?

• The following good program is not well typed

\[
\text{let } x : A \leftarrow \text{new } B \text{ in } \{ \ldots x \leftarrow \text{new } A; x.a(); \} \]

• Why is this program not well typed?
Typing Rules

- The typing rules use very concise notation
- They are very carefully constructed
- Virtually any change in a rule either:
  - Makes the type system unsound (bad programs are accepted as well typed)
  - makes the type system less usable (perfectly good programs are rejected)
Very similar to `let`:

\[
\begin{align*}
O(id) &= T_0 \\
O \vdash e_1 : T_1 \\
T_1 &\leq T_0 \\
\hline
O \vdash id \leftarrow e_1 : T_1
\end{align*}
\]
Initialized Attributes

- Let $O_c(x) = T$ for all attributes $x : T$ in class $C$

- Attribute initialization is similar to let, except for the scope of names

\[
\begin{align*}
O_c(id) & = T_0 \\
O_c \vdash e_1 : T_1 \\
T_1 & \leq T_0 \\
\hline
O_c \vdash id : T_0 \leftarrow e_1 ; & \quad [\text{Attr-Init}]
\end{align*}
\]
If-then-else

- Consider:
  \[
  \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi}
  \]

- The result can be either $e_1$ or $e_2$

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

- The best we can do is the smallest supertype larger than the type of $e_1$ and $e_2$
If-then-else

- Consider the class hierarchy

```
   P
  / \  
 A   B
```

- ... and the expression

```
   if ... then new A else new B fi
```

- Its type should allow for the dynamic type to be both A or B
  - Smallest supertype is P
Least Upper Bound: Operations

- \( \text{lub}(X, Y) \), the least upper bound of \( X \) and \( Y \), is \( Z \) 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 upper bounds

- In COOL, the least upper bound of two types is their least common ancestor in the inheritance tree

  - inheritance tree (rooted at object)
  - class hierarchy descent from the object
  - walk back the tree to find the parent of the two types
If-then-else

\[ O \vdash e_0 : \text{Bool} \]
\[ O \vdash e_1 : T_1 \]
\[ O \vdash e_2 : T_2 \]
\[ O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \text{lub}(T_1, T_2) \]
The rule for case expressions takes a lub over all branches

\[
\begin{align*}
O \vdash e_0 : T_0 \\
O[T_1/x_1] \vdash e_1 : T_1' \quad [\text{Case}] \\
\vdots \\
O[T_n/x_n] \vdash e_n : T_n' \\
\hline
O \vdash \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \ldots; x_n : T_n \Rightarrow e_n; \text{ esac} : \text{lub}(T_1', \ldots, T_n')
\end{align*}
\]
Method Dispatch

- There is a problem with type checking method calls:

\[
O \vdash e_0 : T_0 \\
O \vdash e_1 : T_1 \\
\vdots \\
O \vdash e_n : T_n \\
\hline
O \vdash e_0.f(e_1, \ldots, e_n) : ?
\]

[Dispatch]

- We need information about the formal parameters and return type of \( f \)
Notes on Method Dispatch

• In Cool, method and object identifiers live in different name spaces
  - A method `foo` and an object `foo` can coexist in the same scope

• In the type rules, this is reflected by a separate mapping $M$ for method signatures
  $$M(C, f) = (T_1, \ldots, T_n, T_{n+1})$$
  means in class $C$ there is a method $f$
  $$f(x_1: T_1, \ldots, x_n: T_n): T_{n+1}$$
Type Environment: Method

- The method environment must be added to all rules
- In most cases, $M$ is passed down but not actually used
  - Example of a rule that does not use $M$:
    \[
    O, M \vdash e_1 : T_1 \\
    O, M \vdash e_2 : T_2 \\
    \overline{O, M \vdash e_1 + e_2 : \text{Int}} [\text{Add}]
    \]
  - Only the dispatch rules uses $M$
Discussion

- study type system from programming languages and compiler points of view
- "type rule is weak?" vs "weakly typed programming languages?"
Three types of dispatch in Cool

We first discuss two:

\[
\text{Consider the dispatch } e_0.f(e_1, \ldots, e_n)
\]

\[
e@B.f() \text{ invokes the method } f \text{ in class } B \text{ on the object that is the value of } e.
\]

\[
<\text{expr}>.@<\text{type}>.id(<\text{expr}>, \ldots, <\text{expr}>)
\]
The Dispatch Rule

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

\[ \cdots \]

\[ O, M \vdash e_n : T_n \]

\[ M(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \quad \text{[Dispatch]} \]

\[ T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n) \]

\[ O, M \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}' \]
Static Dispatch

The class $T$ of the method $f$ is given in the dispatch, and the type $T_0$ must conform to $T$.

\[
\begin{align*}
O, M &\vdash e_0 : T_0 \\
O, M &\vdash e_1 : T_1 \\
& \quad \vdots \\
O, M &\vdash e_n : T_n \\
T_0 &\leq T
\end{align*}
\]

\[
M(T, f) = (T_1', \ldots, T_n', T_{n+1}')
\]

\[
T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n)
\]

\[
O, M \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}'
\]
Tradeoffs between complexity and flexibility
Static and Dynamic Types in Cool

```java
class A {
    ...
}

class B inherits A {
    ...
}

class Main {
    A x ← new A;
    ...
    x ← new B;
    ...
}
```

- A variable of static type A can hold values of static type B, if B ≤ A

Here, x’s value has dynamic type A

Here, x’s value has dynamic type B
Static and Dynamic Types in Cool

Soundness theorem for the Cool type system:
\[ \forall E. \; \text{dynamic\_type}(E) \leq \text{static\_type}(E) \]

Why is this Ok?
- All operations that can be used on an object of type \( C \) can also be used on an object of type \( C' \leq C \)
  - Such as fetching the value of an attribute
  - Or invoking a method on the object
- Subclasses can only add attributes or methods
- Methods can be redefined but with same type!
Self-Type: a Motivating Example

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

- **Class Count** incorporates a counter
- The **inc** method works for any subclass
- But there is disaster lurking in the type system
Self-Type: a Motivating Example

- Consider a subclass **Stock** of **Count**

  ```java
  class Stock inherits Count {
      name : String; -- name of item
  }
  ```

- And the following use of **Stock**:

  ```java
  class Main {
      Stock a ← (new Stock).inc ();   Type checking error!
      ... a.name ...
  }
  ```
Self-Type: a Motivating Example

- \((\text{new Stock}).\text{inc}())\) has dynamic type \textit{Stock}
- So it is legitimate to write
  \[
  \text{Stock}\ a \leftarrow (\text{new Stock}).\text{inc} ()
  \]
- But this is not well-typed
  \[(\text{new Stock}).\text{inc}())\ has static type \textit{Count}\]
- The type checker “loses” type information
- This makes inheriting \textit{inc} useless
  - So, we must redefine \textit{inc} for each of the subclasses, with a specialized return type
Self_Type: a Motivating Example

- We will extend the type system
- Insight:
  - `inc` returns "self"
  - Therefore the return value has same type as "self"
  - Which could be `Count` or any subtype of `Count`!
  - In the case of `(new Stock).inc()` the type is `Stock`
- We introduce the keyword `SELF_TYPE` to use for the return value of such functions
  - We will also need to modify the typing rules to handle `SELF_TYPE`
Self-Type: a Motivating Example

- `SELF_TYPE` allows the return type of `inc` to change when `inc` is inherited
- Modify the declaration of `inc` to read
  \[
  \text{inc}() : \text{SELF\_TYPE} \{ \ldots \}
  \]
- The type checker can now prove:
  \[
  O, M \vdash (\text{new Count}).\text{inc}() : \text{Count}
  \]
  \[
  O, M \vdash (\text{new Stock}).\text{inc}() : \text{Stock}
  \]
- The program from before is now well typed
Self_Type

- A special type
- A concept of static type not dynamic type
- Helps with the expressiveness and flexibility (accept more correct programs)
- It is like a "type variable"
- An example to show tradeoffs between complexity vs expressiveness of the type systems

Note: The meaning of SELF_TYPE depends on where it appears

- We write SELF_TYPE_C to refer to an occurrence of SELF_TYPE in the body of C
Important Typing Rule Regarding Self_Type

- This suggests a typing rule:
  \[ \text{SELF\_TYPE}_C \leq C \]

- This rule has an important consequence:
  - In type checking it is always safe to replace \( \text{SELF\_TYPE}_C \) by \( C \)

- This suggests one way to handle \( \text{SELF\_TYPE} \):
  - Replace all occurrences of \( \text{SELF\_TYPE}_C \) by \( C \)

- This would be correct but it is like not having \( \text{SELF\_TYPE} \) at all
Operations on Self_Type

- Recall the operations on types
  - $T_1 \leq T_2$   \( T_1 \) is a subtype of \( T_2 \)
  - \( \text{lub}(T_1, T_2) \)   the least-upper bound of \( T_1 \) and \( T_2 \)

- We must extend these operations to handle SELF_TYPE
Let $T$ and $T'$ be any types but SELF_TYPE.

There are four cases in the definition of $\leq$:

1. $\text{SELF\_TYPE}_C \leq T$ if $C \leq T$
   - $\text{SELF\_TYPE}_C$ can be any subtype of $C$
   - This includes $C$ itself
   - Thus this is the most flexible rule we can allow

2. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
   - $\text{SELF\_TYPE}_C$ is the type of the “self” expression
   - In Cool we never need to compare SELF_TYPEs coming from different classes
Operations on Self_Type

3. $T \leq \text{SELF\_TYPE}_C$ always false
   Note: $\text{SELF\_TYPE}_C$ can denote any subtype of $C$.

4. $T \leq T'$ (according to the rules from before)

Based on these rules we can extend lub ...
Operations on Self_Type

Let $T$ and $T'$ be any types but SELF_TYPE.
Again there are four cases:
1. $\text{lub}(\text{SELF}_{\text{TYPE}}_c, \text{SELF}_{\text{TYPE}}_c) = \text{SELF}_{\text{TYPE}}_c$

2. $\text{lub}(\text{SELF}_{\text{TYPE}}_c, T) = \text{lub}(C, T)$
   This is the best we can do because $\text{SELF}_{\text{TYPE}}_c \leq C$

3. $\text{lub}(T, \text{SELF}_{\text{TYPE}}_c) = \text{lub}(C, T)$

4. $\text{lub}(T, T')$ defined as before
Self-Type in Cool

• The parser checks that SELF_TYPE appears only where a type is expected
• But SELF_TYPE is not allowed everywhere a type can appear:
  1. class T inherits T’ {...}
     • T, T’ cannot be SELF_TYPE
     • Because SELF_TYPE is never a dynamic type
  2. x : T
     • T can be SELF_TYPE
     • An attribute whose type is SELF_TYPE$_c$
Self-Type in Cool

3. let $x : T$ in $E$
   - $T$ can be SELF_TYPE
   - $x$ has type SELF_TYPE

4. new $T$
   - $T$ can be SELF_TYPE
   - Creates an object of the same type as self

5. $m@T(E_1,\ldots,E_n)$
   - $T$ cannot be SELF_TYPE
Type Checking Rules with Self_Type

• Since occurrences of SELF_TYPE depend on the enclosing class we need to carry more context during type checking
• New form of the typing judgment:
  \[ O, M, C \vdash e : T \]

  (An expression e occurring in the body of C has static type T given a variable type environment O and method signatures M)
Type Checking Rules with Self_Type

- The next step is to design type rules using SELF_TYPE for each language construct.
- Most of the rules remain the same except that ≤ and lub are the new ones.
- Example:

\[
O(id) = T_0 \\
O \vdash e_1 : T_1 \\
T_1 \leq T_0 \\
\hline
O \vdash id \leftarrow e_1 : T_1
\]
Type Checking Rules with Self_Type

Old rule for method dispatch:

\[
\begin{align*}
O, M, C & \vdash e_0 : T_0 \\
& \vdots \\
O, M, C & \vdash e_n : T_n \\
M(T_0, f) &= (T_1', \ldots, T_n', T_{n+1}') \\
T_{n+1}' &\neq \text{SELF\_TYPE} \\
T_i &\leq T_i' & 1 \leq i \leq n \\
O, M, C & \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'
\end{align*}
\]
Type Checking Rules with Self_Type

- If the return type of the method is **SELF_TYPE** then the type of the dispatch is the type of the dispatch expression:

  \[
  O, M, C \vdash e_0 : T_0 \\
  \ldots \\
  O, M, C \vdash e_n : T_n \\
  M(T_0, f) = (T_1', \ldots, T_n', \text{SELF_TYPE}) \\
  T_i \leq T_i' \quad 1 \leq i \leq n \\
  O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0
  \]
Type Checking Rules with Self_Type

- Note this rule handles the *Stock* example
- Formal parameters cannot be *SELF_TYPE*
- Actual arguments can be *SELF_TYPE*
  - The extended \( \leq \) relation handles this case
- The type \( T_0 \) of the dispatch expression could be *SELF_TYPE*
  - Which class is used to find the declaration of \( f \)?
  - Answer: it is safe to use the class where the dispatch appears
Type Checking Rules with Self_Type

- Recall the original rule for static dispatch

\[ O, M, C \vdash e_0 : T_0 \]

\[ ... \]

\[ O, M, C \vdash e_n : T_n \]

\[ T_0 \leq T \]

\[ M(T, f) = (T_1', ..., T_n', T_{n+1}') \]

\[ T_{n+1}' \neq \text{SELF\_TYPE} \]

\[ T_i \leq T_i' \quad 1 \leq i \leq n \]

\[ O, M, C \vdash e_0@T.f(e_1, ..., e_n) : T_{n+1}' \]
Type Checking Rules with Self_Type

- If the return type of the method is \texttt{SELF_TYPE} we have:

\[
\begin{align*}
O, M, C &\vdash e_0 : T_0 \\
\vdots &\\
O, M, C &\vdash e_n : T_n \\
T_0 &\leq T \\
M(T, f) &=(T_1', ..., T_n', \texttt{SELF_TYPE}) \\
T_i &\leq T_i' \quad 1 \leq i \leq n \\
\hline
O, M, C &\vdash e_0@T.f(e_1, ..., e_n) : T_0
\end{align*}
\]
Type Checking Rules with Self_Type

- Why is this rule correct?
- If we dispatch a method returning 
  \texttt{SELF\_TYPE} in class \texttt{T}, don’t we get back a \texttt{T}?

- No. \texttt{SELF\_TYPE} is the type of the self parameter, which may be a subtype of the class in which the method appears

- The static dispatch class cannot be \texttt{SELF\_TYPE}
Type Checking Rules with Self_Type

- There are two new rules using `SELF_TYPE`:

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

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

- There are a number of other places where `SELF_TYPE` is used.
Type Systems

• The rules in these lecture were COOL-specific
  - Other languages have very different rules

• General themes
  - Type rules are defined on the structure of expressions
  - Types of variables are modeled by an environment

• Types are a play between flexibility and safety