

# Semantic Analysis Typechecking in COOL

Lectures 10-13

Prof. Neula CS 164

1

## Administration

---

- Midterm I
  - Thursday, February 23, in class
  - Be here on time (we start at 12:40)
  - Including parsing, no semantic analysis
  - 1 cheat sheet, front and back, handwritten, by you!

Prof. Neula CS 164

2

## Outline

---

- The role of semantic analysis in a compiler
  - A laundry list of tasks
- Scope
- Types

## The Compiler So Far

---

- Lexical analysis
  - Detects inputs with illegal tokens
- Parsing
  - Detects inputs with ill-formed parse trees
- Semantic analysis
  - Last "front end" phase
  - Catches more errors

## What's Wrong?

---

- Example 1

let y: Int in x + 3

- Example 2

let y: String ← "abc" in y + 3

## Why a Separate Semantic Analysis?

---

- Parsing cannot catch some errors
- Some language constructs are not context-free
  - Example: All used variables must have been declared (i.e. scoping)
  - Example: A method must be invoked with arguments of proper type (i.e. typing)

## What Does Semantic Analysis Do?

---

- Checks of many kinds . . . **coolc** checks:
  1. All identifiers are declared
  2. Types
  3. Inheritance relationships
  4. Classes defined only once
  5. Methods in a class defined only once
  6. Reserved identifiers are not misusedAnd others . . .
- The requirements depend on the language

Prof. Neula CS 164

7

## Scope

---

- Matching identifier declarations with uses
  - Important semantic analysis step in most languages
  - Including *COOL* !

Prof. Neula CS 164

8

## Scope (Cont.)

---

- 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 program
  - Different scopes for same name don't overlap
- An identifier may have restricted scope

## Static vs. Dynamic Scope

---

- Most languages have static scope
  - Scope depends only on the program text, not run-time behavior
  - Cool has static scope
- A few languages are dynamically scoped
  - Lisp, SNOBOL
  - Lisp has changed to mostly static scoping
  - Scope depends on execution of the program

## Static Scoping Example

---

```
let x: Int <- 0 in
{
  x;
  let x: Int <- 1 in
    x;
  x;
}
```

Prof. Necula CS 164

11

## Static Scoping Example (Cont.)

---

```
let x: Int <- 0 in
{
  x;
  let x: Int <- 1 in
    x;
  x;
}
```

Uses of `x` refer to closest enclosing definition

Prof. Necula CS 164

12

## Scope in Cool

---

- Cool identifier bindings are introduced by
  - Class declarations (introduce class names)
  - Method definitions (introduce method names)
  - Let expressions (introduce object id's)
  - Formal parameters (introduce object id's)
  - Attribute definitions in a class (introduce object id's)
  - Case expressions (introduce object id's)

## Implementing the Most-Closely Nested Rule

---

- Much of semantic analysis can be expressed as a recursive descent of an AST
  - Process an AST node  $n$
  - Process the children of  $n$
  - Finish processing the AST node  $n$

## Implementing . . . (Cont.)

---

- Example: the scope of `let` bindings is one subtree

`let x: Int ← 0 in e`

- `x` can be used in subtree `e`

## Symbol Tables

---

- Consider again: `let x: Int ← 0 in e`
- Idea:
  - Before processing `e`, add definition of `x` to current definitions, overriding any other definition of `x`
  - After processing `e`, remove definition of `x` and restore old definition of `x`
- A *symbol table* is a data structure that tracks the current bindings of identifiers
  - We'll give you an implementation for the project

## Scope in Cool (Cont.)

---

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool
  - Cannot be nested
  - Are *globally visible* throughout the program
- In other words, a class name can be used before it is defined

Prof. Necula CS 164

17

## Example: Use Before Definition

---

```
Class Foo {  
  ... let y: Bar in ...  
};
```

```
Class Bar {  
  ...  
};
```

Prof. Necula CS 164

18

## More Scope in Cool

---

Attribute names are global within the class in which they are defined

```
Class Foo {  
  f(): Int { a };  
  a: Int ← 0;  
}
```

## More Scope (Cont.)

---

- Method and attribute names have complex rules
- A method need not be defined in the class in which it is used, but in some parent class
- Methods may also be redefined (overridden)

## Class Definitions

---

- Class names can be used before being defined
- We can't check this property
  - using a symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all class names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two

## Types

---

- What is a type?
  - The notion varies from language to language
- Consensus
  - A set of values
  - A set of operations on those values
- Classes are one instantiation of the modern notion of type

## Why Do We Need Type Systems?

---

Consider the assembly language fragment

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

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

## Types and Operations

---

- Certain operations are legal for values of each type
  - 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!

## Type Systems

---

- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values, because nothing else will!
- Type systems provide a concise formalization of the semantic checking rules

Prof. Necula CS 164

25

## What Can Types do For Us?

---

- Can detect certain kinds of errors
- Memory errors:
  - Reading from an invalid pointer, etc.
- Violation of abstraction boundaries:

```
class FileSystem {
  open(x : String) : File {
    ...
  }
  ...
}

class Client {
  f(fs : FileSystem) {
    File fdesc <- fs.open("foo")
    ...
  } -- f cannot see inside fdesc !
}
```

Prof. Necula CS 164

26

## Type Checking Overview

---

- Three kinds of languages:
  - *Statically typed*: All or almost all checking of types is done as part of compilation (C, Java, Cool)
  - *Dynamically typed*: Almost all checking of types is done as part of program execution (Scheme)
  - *Untyped*: No type checking (machine code)

## The Type Wars

---

- Competing views on static vs. dynamic typing
- Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks
- Dynamic typing proponents say:
  - Static type systems are restrictive
  - Rapid prototyping easier in a dynamic type system

## The Type Wars (Cont.)

---

- In practice, most code is written in statically typed languages with an “escape” mechanism
  - Unsafe casts in C, native methods in Java, unsafe modules in Modula-3

## Type Checking in Cool

## Outline

---

- Type concepts in COOL
- Notation for type rules
  - Logical rules of inference
- COOL type rules
- General properties of type systems

## Cool Types

---

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

## 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

## Rules of Inference

---

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions (for the lexer)
  - Context-free grammars (for the parser)
- The appropriate formalism for type checking is logical rules of inference

## Why Rules of Inference?

---

- Inference rules have the form  
*If Hypothesis is true, then Conclusion is true*
- Type checking computes via reasoning  
*If  $E_1$  and  $E_2$  have certain types, then  $E_3$  has a certain type*
- Rules of inference are a compact notation for "If-Then" statements

Prof. Necula CS 164

35

## From English to an Inference Rule

---

- The notation is easy to read (with practice)
- Start with a simplified system and gradually add features
- Building blocks
  - Symbol  $\wedge$  is "and"
  - Symbol  $\Rightarrow$  is "if-then"
  - $x:T$  is " $x$  has type  $T$ "

Prof. Necula CS 164

36

## From English to an Inference Rule (2)

---

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

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

$(e_1: \text{Int} \wedge e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}$

## From English to an Inference Rule (3)

---

The statement

$(e_1: \text{Int} \wedge e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}$

is a special case of

$(\text{Hypothesis}_1 \wedge \dots \wedge \text{Hypothesis}_n) \Rightarrow \text{Conclusion}$

This is an inference rule

## Notation for Inference Rules

---

- By tradition inference rules are written

$$\frac{\vdash \text{Hypothesis}_1 \quad \dots \quad \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

- Cool type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$  means "we can prove that ..."

## Two Rules

---

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

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

## Two Rules (Cont.)

---

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions
- We can fill the template with ANY expression!

$$\frac{\frac{}{\vdash \text{true} : \text{Int}} \quad \frac{}{\vdash \text{false} : \text{Int}}}{\vdash \text{true} + \text{false} : \text{Int}}$$

Prof. Necula CS 164

41

## Example: 1 + 2

---

$$\frac{\frac{}{\vdash 1 : \text{Int}} \quad \frac{}{\vdash 2 : \text{Int}}}{\vdash 1 + 2 : \text{Int}}$$

Prof. Necula CS 164

42

## Soundness

---

- 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:

$$\frac{}{\vdash i : \text{Object}} \quad (i \text{ is an integer})$$

Prof. Necula CS 164

43

## Type Checking Proofs

---

- 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 the proofs of types of  $e$ 's subexpressions
  - The conclusion is the proof of type of  $e$

Prof. Necula CS 164

44

## Rules for Constants

---

$$\frac{}{\vdash \text{false} : \text{Bool}} \quad [\text{Bool}]$$
$$\frac{}{\vdash s : \text{String}} \quad [\text{String}] \quad (s \text{ is a string constant})$$

## Rule for New

---

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

$$\frac{}{\vdash \text{new } T : T} \quad [\text{New}]$$

## Two More Rules

---

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

$$\frac{\begin{array}{c} \vdash e_1 : \text{Bool} \\ \vdash e_2 : T \end{array}}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}} \quad [\text{Loop}]$$

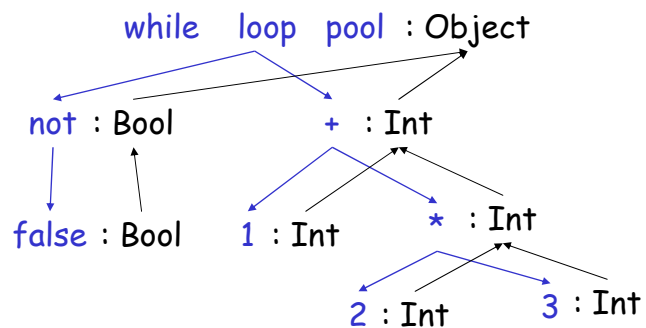
Prof. Neula CS 164

47

## Typing: Example

---

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



Prof. Neula CS 164

48

## Typing Derivations

---

- The typing reasoning can be expressed as a tree:

$$\frac{\frac{\frac{\frac{}{\vdash \text{false} : \text{Bool}}{\vdash \text{not false} : \text{Bool}}}{\vdash 1 : \text{Int}} \quad \frac{\frac{\frac{\frac{}{\vdash 2 : \text{Int}}{\vdash 2 * 3 : \text{Int}}}{\vdash 3 : \text{Int}}}{\vdash 1 + 2 * 3 : \text{Int}}}{\vdash \text{while not false loop } 1 + 2 * 3 : \text{Object}}}$$

- 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

Prof. Necula CS 164

49

## A Problem

---

- What is the type of a variable reference?

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

- The local, structural rule does not carry enough information to give  $x$  a type.

Prof. Necula CS 164

50

## A Solution: Put more information in the rules!

---

- A *type environment* gives types for *free* variables
  - A type environment is a mapping from **ObjectIdentifiers** to **Types**
  - A variable is free in an expression if:
    - The expression contains an occurrence of the variable that refers to a declaration 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
  - E.g. in "**x + let x : Int in x + y**" both "**x**" and "**y**" are free

Prof. Necula CS 164

51

## Type Environments

---

Let  $\mathcal{O}$  be a function from **ObjectIdentifiers** to **Types**

The sentence  $\mathcal{O} \vdash e : T$

is read: Under the assumption that variables have the types given by  $\mathcal{O}$ , it is provable that the expression  $e$  has the type  $T$

Prof. Necula CS 164

52

## Modified Rules

---

The type environment is added to the earlier rules:

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

$$\frac{\begin{array}{l} O \vdash e_1 : \text{Int} \\ O \vdash e_2 : \text{Int} \end{array}}{O \vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

## New Rules

---

And we can write new rules:

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

## Let

---

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

$O[T_0/x]$  means “ $O$  modified to map  $x$  to  $T_0$  and behaving as  $O$  on all other arguments”:

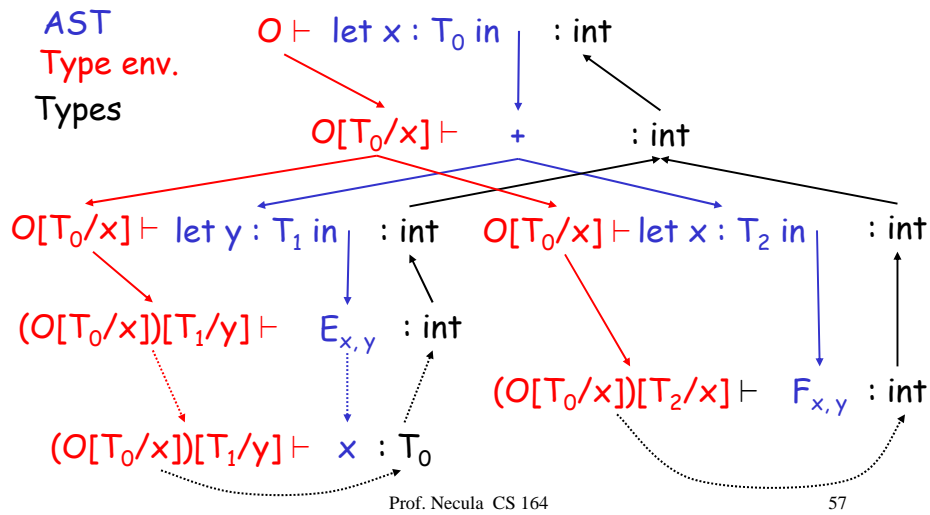
$$\begin{aligned} O[T_0/x](x) &= T_0 \\ O[T_0/x](y) &= O(y) \end{aligned}$$

## Let. Example.

---

- Consider the Cool expression  
 $\text{let } x : T_0 \text{ in } (\text{let } y : T_1 \text{ in } E_{x,y}) + (\text{let } x : T_2 \text{ in } F_{x,y})$   
(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.



## Notes

- 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:

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

This rule is weak. Why?

## Let with Initialization

---

- Consider the example:

```
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  $X \leq Y$  on classes to say that:
  - An object of type  $X$  could be used when one of type  $Y$  is acceptable, or equivalently
  - $X$  conforms with  $Y$
  - In Cool this means that  $X$  is a subclass of  $Y$
- 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  $Y \leq Z$

Prof. Necula CS 164

61

## Let with Initialization (Again)

---

$$\frac{\begin{array}{l} O \vdash e_0 : T \\ T \leq T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \text{[Let-Init]}$$

- Both rules for let are sound
- But more programs type check with the latter

Prof. Necula CS 164

62

## Let with Subtyping. Notes.

---

- There is a tension between
  - Flexible rules that do not constrain programming
  - Restrictive rules that ensure safety of execution

## Expressiveness of Static Type Systems

---

- A static type system enables a compiler to detect many common programming errors
- The cost is that some correct programs are disallowed
  - Some argue for dynamic type checking instead
  - Others argue for more expressive static type checking
- But more expressive type systems are also more complex

## Dynamic And Static 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

Prof. Necula CS 164

65

## Dynamic and Static Types. (Cont.)

---

- In early type systems the set of static types correspond directly with the dynamic types
- Soundness theorem: for all expressions  $E$   
 $\text{dynamic\_type}(E) = \text{static\_type}(E)$   
(in all executions,  $E$  evaluates to values of the type inferred by the compiler)
- This gets more complicated in advanced type systems

Prof. Necula CS 164

66

## Dynamic and Static Types in COOL

---

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

x has static type A →

← Here, x's value has dynamic type A

← Here, x's value has dynamic type B

- A variable of static type  $A$  can hold values of static type  $B$ , if  $B \leq A$

Prof. Necula CS 164

67

## Dynamic and Static Types

---

Soundness theorem for the Cool type system:

$$\forall E. \text{dynamic\_type}(E) \leq \text{static\_type}(E)$$

Why is this Ok?

- For  $E$ , compiler uses  $\text{static\_type}(E)$
- All operations that can be used on an object of type  $C$  can also be used on an object of type  $C' \leq C$ 
  - 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 !

Prof. Necula CS 164

68

## Let. Examples.

---

- 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

Prof. Necula CS 164

69

## Example of Wrong Let Rule (1)

---

- Now consider a hypothetical let rule:

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

- How is it different from the correct rule?
- The following good program does not typecheck  
`let x : Int ← 0 in x + 1`
- Why?

Prof. Necula CS 164

70

## Example of Wrong Let Rule (2)

---

- Now consider another 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  
`let x : B ← new A in x.b()`
- Why is this program bad?

Prof. Necula CS 164

71

## Example of Wrong Let Rule (3)

---

- Now consider another 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  
`let x : A ← new B in {... x ← new A; x.a(); }`
- Why is this program not well typed?

Prof. Necula CS 164

72

## Comments

---

- 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)
  - Or, makes the type system less usable  
(good programs are rejected)
- But some good programs will be rejected anyway
  - The notion of a good program is undecidable

Prof. Necula CS 164

73

## Assignment

---

More uses of subtyping:

$$\frac{\begin{array}{l} O(\text{id}) = T_0 \\ O \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O \vdash \text{id} \leftarrow e_1 : T_1} \quad [\text{Assign}]$$

Prof. Necula CS 164

74

## Initialized Attributes

---

- Let  $O_c(x) = T$  for all attributes  $x:T$  in class  $C$ 
  - $O_c$  represents the class-wide scope
- Attribute initialization is similar to **let**, except for the scope of names

$$\frac{O_c(\text{id}) = T_0 \quad O_c \vdash e_1 : T_1 \quad T_1 \leq T_0}{O_c \vdash \text{id} : T_0 \leftarrow e_1 ;} \quad [\text{Attr-Init}]$$

Prof. Necula CS 164

75

## If-Then-Else

---

- Consider:  
if  $e_0$  then  $e_1$  else  $e_2$  fi
- The result can be either  $e_1$  or  $e_2$
- 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  $e_2$

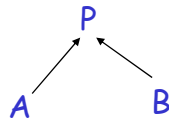
Prof. Necula CS 164

76

## If-Then-Else example

---

- Consider the class hierarchy



- ... 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

Prof. Necula CS 164

77

## Least Upper Bounds

---

- $\text{lub}(X, Y)$ , the least upper bound of X and Y, is Z if
  - $X \leq Z \wedge Y \leq Z$   
Z is an upper bound
  - $X \leq Z' \wedge Y \leq Z' \Rightarrow Z \leq Z'$   
Z is least among upper bounds
- In COOL, the least upper bound of two types is their least common ancestor in the inheritance tree

Prof. Necula CS 164

78

## If-Then-Else Revisited

---

$$\frac{\begin{array}{l} O \vdash e_0 : \text{Bool} \\ O \vdash e_1 : T_1 \\ O \vdash e_2 : T_2 \end{array}}{O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \text{lub}(T_1, T_2)} \quad [\text{If-Then-Else}]$$

Prof. Necula CS 164

79

## Case

---

- The rule for **case** expressions takes a lub over all branches

$$\frac{\begin{array}{l} O \vdash e_0 : T_0 \\ O[T_1/x_1] \vdash e_1 : T_1' \\ \dots \\ O[T_n/x_n] \vdash e_n : T_n' \end{array}}{O \vdash \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \dots; x_n : T_n \Rightarrow e_n \text{ esac} : \text{lub}(T_1', \dots, T_n')} \quad [\text{Case}]$$

Prof. Necula CS 164

80

## Next

---

- Type checking method dispatch
- Type checking with SELF\_TYPE in COOL

## Method Dispatch

---

- There is a problem with type checking method calls:

$$\begin{array}{c} O \vdash e_0 : T_0 \\ O \vdash e_1 : T_1 \\ \dots \\ O \vdash e_n : T_n \end{array} \quad \text{[Dispatch]} \quad \frac{}{O \vdash e_0.f(e_1, \dots, e_n) : ?}$$

- We need information about the formal parameters and return type of  $f$

## Notes on 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

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

means in class  $C$  there is a method  $f$

$$f(x_1:T_1, \dots, x_n:T_n): T_{n+1}$$

## An Extended Typing Judgment

---

- Now we have two environments  $O$  and  $M$
- The form of the typing judgment is

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

read as: "with the assumption that the object identifiers have types as given by  $O$  and the method identifiers have signatures as given by  $M$ , the expression  $e$  has type  $T$ "

## The Method Environment

---

- 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  $M$ :

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

- Only the dispatch rules uses  $M$

Prof. Necula CS 164

85

## The Dispatch Rule Revisited

---

$$\frac{\begin{array}{l} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \dots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T'_1, \dots, T'_n, T'_{n+1}) \\ T_i \leq T'_i \quad (\text{for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0.f(e_1, \dots, e_n) : T'_{n+1}} \quad [\text{Dispatch}]$$

Prof. Necula CS 164

86

## Static Dispatch

---

- Static dispatch is a variation on normal dispatch
- The method is found in the class explicitly named by the programmer
- The inferred type of the dispatch expression must conform to the specified type

Prof. Neula CS 164

87

## Static Dispatch (Cont.)

---

$$\begin{array}{l} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \dots \\ O, M \vdash e_n : T_n \\ T_0 \leq T \quad \text{[StaticDispatch]} \\ M(T, f) = (T'_1, \dots, T'_n, T'_{n+1}) \\ T_i \leq T'_i \quad (\text{for } 1 \leq i \leq n) \\ \hline O, M \vdash e_0 @ T.f(e_1, \dots, e_n) : T'_{n+1} \end{array}$$

Prof. Neula CS 164

88

## Handling the SELF\_TYPE

Prof. Neula CS 164

89

### Flexibility vs. Soundness

---

- Recall that type systems have two conflicting 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

Prof. Neula CS 164

90

## Dynamic And Static Types. Review.

---

- The dynamic type of an object is the class  $C$  that is used in the "new  $C$ " expression that created it
  - 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

Prof. Necula CS 164

91

## Dynamic and Static Types. Review

---

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 !

Prof. Necula CS 164

92

## An Example

---

```
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

Prof. Necula CS 164

93

## An Example (Cont.)

---

- Consider a subclass `Stock` of `Count`

```
class Stock inherits Count {  
  name() : String { ... }; -- name of item  
};
```

- And the following use of `Stock`:

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

Prof. Necula CS 164

94

## What Went Wrong?

---

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

Prof. Necula CS 164

95

## SELF\_TYPE to the Rescue

---

- 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`

Prof. Necula CS 164

96

## SELF\_TYPE to the Rescue (Cont.)

---

- SELF\_TYPE allows the return type of `inc` to change when `inc` is inherited
- Modify the declaration of `inc` to read  
`inc() : SELF_TYPE { ... }`
- 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

Prof. Necula CS 164

97

## Notes About SELF\_TYPE

---

- SELF\_TYPE is not a dynamic type
- It is a static type
- It helps the type checker to keep better track of types
- It enables the type checker to accept more correct programs
- In short, having SELF\_TYPE increases the expressive power of the type system

Prof. Necula CS 164

98

## SELF\_TYPE and Dynamic Types (Example)

---

- What can be the dynamic type of the object returned by `inc`?

- Answer: whatever could be the type of "self"

```
class A inherits Count { } ;  
class B inherits Count { } ;  
class C inherits Count { } ;
```

(`inc` could be invoked through any of these classes)

- Answer: `Count` or any subtype of `Count`

Prof. Neula CS 164

99

## SELF\_TYPE and Dynamic Types (Example)

---

- In general, if `SELF_TYPE` appears textually in the class `C` as the declared type of `E` then it denotes the dynamic type of the "self" expression:

$\text{dynamic\_type}(E) = \text{dynamic\_type}(\text{self}) \leq C$

- Note: The meaning of `SELF_TYPE` depends on where it appears
  - We write `SELF_TYPEC` to refer to an occurrence of `SELF_TYPE` in the body of `C`

Prof. Neula CS 164

100

## Type Checking

---

- 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

Prof. Necula CS 164

101

## Operations on $\text{SELF\_TYPE}$

---

- 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  $\text{SELF\_TYPE}$

Prof. Necula CS 164

102

## Extending $\leq$

---

Let  $T$  and  $T'$  be any types but  $\text{SELF\_TYPE}$

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  $\text{SELF\_TYPE}$ s coming from different classes

Prof. Necula CS 164

103

## Extending $\leq$ (Cont.)

---

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** ...

Prof. Necula CS 164

104

## Extending $\text{lub}(T, T')$

---

Let  $T$  and  $T'$  be any types but  $\text{SELF\_TYPE}$

Again there are four cases:

1.  $\text{lub}(\text{SELF\_TYPE}_C, \text{SELF\_TYPE}_C) = \text{SELF\_TYPE}_C$
2.  $\text{lub}(\text{SELF\_TYPE}_C, T) = \text{lub}(C, T)$   
This is the best we can do because  $\text{SELF\_TYPE}_C \leq C$
3.  $\text{lub}(T, \text{SELF\_TYPE}_C) = \text{lub}(C, T)$
4.  $\text{lub}(T, T')$  defined as before

## Where Can $\text{SELF\_TYPE}$ Appear in COOL?

---

- The parser checks that  $\text{SELF\_TYPE}$  appears only where a type is expected
- But  $\text{SELF\_TYPE}$  is not allowed everywhere a type can appear:
  1.  $\text{class } T \text{ inherits } T' \{ \dots \}$ 
    - $T, T'$  cannot be  $\text{SELF\_TYPE}$
    - Because  $\text{SELF\_TYPE}$  is never a dynamic type
  2.  $x : T$ 
    - $T$  can be  $\text{SELF\_TYPE}$
    - An attribute whose type is  $\text{SELF\_TYPE}_C$

## Where Can SELF\_TYPE Appear in COOL?

---

3. `let x : T in E`
  - T can be SELF\_TYPE
  - x has type SELF\_TYPE<sub>c</sub>
4. `new T`
  - T can be SELF\_TYPE
  - Creates an object of the same type as `self`
5. `m@T(E1,...,En)`
  - T cannot be SELF\_TYPE

## Typing Rules for 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

---

- 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:

$$\frac{\begin{array}{c} O(\text{id}) = T_0 \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O, M, C \vdash \text{id} \leftarrow e_1 : T_1}$$

Prof. Necula CS 164

109

## What's Different?

---

- Recall the old rule for dispatch

$$\frac{\begin{array}{c} O, M, C \vdash e_0 : T_0 \\ \dots \\ O, M, C \vdash e_n : T_n \\ M(T_0, f) = (T_1', \dots, T_n', T_{n+1}') \\ T_{n+1}' \neq \text{SELF\_TYPE} \\ T_i \leq T_i' \quad 1 \leq i \leq n \end{array}}{O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_{n+1}'}$$

Prof. Necula CS 164

110

## What's Different?

---

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

$$\begin{array}{c} O, M, C \vdash e_0 : T_0 \\ \dots \\ O, M, C \vdash e_n : T_n \\ M(T_0, f) = (T_1', \dots, T_n', \text{SELF\_TYPE}) \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ \hline O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_0 \end{array}$$

Prof. Neula CS 164

111

## What's Different?

---

- 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

Prof. Neula CS 164

112

## Static Dispatch

---

- Recall the original rule for static dispatch

$$\begin{array}{c}
 O, M, C \vdash e_0 : T_0 \\
 \dots \\
 O, M, C \vdash e_n : T_n \\
 T_0 \leq T \\
 M(T, f) = (T_1', \dots, T_n', T_{n+1}') \\
 T_{n+1}' \neq \text{SELF\_TYPE} \\
 \frac{T_i \leq T_i' \quad 1 \leq i \leq n}{O, M, C \vdash e_0 @ T.f(e_1, \dots, e_n) : T_{n+1}'}
 \end{array}$$

Prof. Necula CS 164

113

## Static Dispatch

---

- If the return type of the method is **SELF\_TYPE** we have:

$$\begin{array}{c}
 O, M, C \vdash e_0 : T_0 \\
 \dots \\
 O, M, C \vdash e_n : T_n \\
 T_0 \leq T \\
 M(T, f) = (T_1', \dots, T_n', \text{SELF\_TYPE}) \\
 \frac{T_i \leq T_i' \quad 1 \leq i \leq n}{O, M, C \vdash e_0 @ T.f(e_1, \dots, e_n) : T_0}
 \end{array}$$

Prof. Necula CS 164

114

## Static Dispatch

---

- Why is this rule correct?
- If we dispatch a method returning `SELF_TYPE` in class `T`, don't we get back a `T`?
- No. `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 `SELF_TYPE`

Prof. Necula CS 164

115

## New Rules

---

- There are two new rules using `SELF_TYPE`

$$\frac{}{O, M, C \vdash \text{self} : \text{SELF\_TYPE}_C}$$
$$\frac{}{O, M, C \vdash \text{new SELF\_TYPE} : \text{SELF\_TYPE}_C}$$

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

Prof. Necula CS 164

116

## Where SELF\_TYPE Cannot Appear in COOL?

---

$m(x : T) : T \{ \dots \}$

- Only  $T$  can be SELF\_TYPE !

What could go wrong if  $T$  were SELF\_TYPE?

```
class A { comp(x : SELF_TYPE) : Bool { ... }; }  
class B inherits A {  
  b() : int { ... };  
  comp(y : SELF_TYPE) : Bool { ... y.b() ... }; }  
...  
let x : A ← new B in ... x.comp(new A); ...  
...
```

Prof. Necula CS 164

117

## Summary of SELF\_TYPE

---

- The extended  $\leq$  and  $\text{lub}$  operations can do a lot of the work. Implement them to handle SELF\_TYPE
- SELF\_TYPE can be used only in a few places. Be sure it isn't used anywhere else.
- A use of SELF\_TYPE always refers to any subtype in the current class
  - The exception is the type checking of dispatch.
  - SELF\_TYPE as the return type in an invoked method might have nothing to do with the current class

Prof. Necula CS 164

118

## Why Cover SELF\_TYPE ?

---

- SELF\_TYPE is a research idea
  - It adds more expressiveness to the type system
- SELF\_TYPE is itself not so important
  - except for the project
- Rather, SELF\_TYPE is meant to illustrate that type checking can be quite subtle
- In practice, there should be a balance between the complexity of the type system and its expressiveness

## 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