

# Semantic Analysis and Type checking

- A compiler has to do semantic checks in addition to syntactic checks.
- Semantic checks
  - Static – done during compilation
  - Dynamic – done during run-time
- *Type checking* is one of these static checking operations.
  - we may not do all type checking at compile-time.
  - Some systems also use dynamic type checking too.



# 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**

# 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 . . . cool 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 misused**

**And others ...**
- **The requirements depend on the language**

# Scope

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

```
fn main() {  
  // Parent scope  
  let x = 1; {  
    // `x` in this nested scope shadows `x` in the parent scope.  
    let x = "Hello, world";  
    assert_eq!(x, 1);  
  }  
}
```

# 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 runtime behavior
  - Cool has static scope
- A few languages are **dynamically** scoped
  - Lisp, Perl
  - Lisp has changed to mostly static scoping
  - Scope depends on execution of the program

# Static Scope

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

Uses of **x** refer to **closest enclosing definition**



# Dynamic Scope

- A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program

## Example

`bar(x) = x + a`

`foo(y) = let a ← 4 in bar(3);`

`baz(z) = let a ← 5 in bar(3)`

# Scope in Cool

- **Cool identifier names are introduced by**
  1. **Class declarations (introduce class names)**
  2. **Method definitions (introduce method names)**
  3. **Let expressions (introduce object id's)**
  4. **Formal parameters (introduce object id's)**
  5. **Attribute definitions in a class (introduce object id's)**
  6. **Case expressions (introduce object id's)**

# 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**

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

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

# More More Scope in Cool

- 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)

# SYMBOL TABLE AND SCOPE

- Symbol tables typically need to support multiple declarations of the same identifier within a program.
- We shall implement scopes by setting up a separate symbol table for each scope.

# Who Creates Symbol Table??

- Identifiers and attributes are entered by the analysis phases when processing a definition (declaration) of an identifier
- In simple languages with only global variables and implicit declarations:
  - ✓ The scanner can enter an identifier into a symbol table if it is not already there
- In block-structured languages with scopes and explicit declarations:
  - ✓ The parser and/or semantic analyzer enter identifiers and corresponding attributes

# USE OF SYMBOL TABLE

- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined(declared)
- To verify that expressions and assignments are semantically correct ——type checking
- To generate intermediate or target code

# IMPLEMENTATION OF SYMBOL TABLE

- Each entry in the symbol table can be implemented as a record consisting of several fields.
- These fields are dependent on the information to be saved about the name
- But since the information about a name depends on the usage of the name, the entries in the symbol table records will not be uniform.
- Hence to keep the symbol tables records uniform some information are kept outside the symbol table and a pointer to this information is stored in the symbol table record.



# SYMBOL TABLE ORGANIZATION

**Int** x,y;

Procedure P:

**Bool** x, a ;

Procedure Q:

**Real** x,y,z ;

    begin

    .....

    end

begin

end

Top →

<b>z</b>	<b>Real</b>
<b>y</b>	<b>Real</b>
<b>x</b>	<b>Real</b>

Symbol table  
for Q

<b>Q</b>	<b>Proc</b>
<b>x</b>	<b>Bool</b>
<b>a</b>	<b>Bool</b>

Symbol table  
for P

<b>P</b>	<b>Proc</b>
<b>x</b>	<b>Int</b>
<b>y</b>	<b>Int</b>

Symbol table  
for global

# SYMBOL TABLE DATA STRUCTURES

## **Issues to consider : Operations required**

- Insert :Add symbol to symbol table
- Look UP: Find symbol in the symbol table (and get its attributes)
- Insertion is done only once
- Look Up is done many times
- Need Fast Look Up
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

# A Fancier Symbol Table

1. **enter\_scope()** start a new nested scope
2. **find\_symbol(x)** finds current x (or null)
3. **add\_symbol(x)** add a symbol x to the table
4. **check\_scope(x)** true if x defined in current scope
5. **exit\_scope()** exit current scope

# Scopes - Summary

- **Scoping rules match uses of identifiers with their declarations**
  - **Static scoping is the most common form**
- **Scoping rules can be implemented using symbol tables**
  - **In one or more passes over the AST**

# Class Definitions

- **Class names can be used before being defined**
- **We cannot check class names**
  - 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**

# 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!

E.g.

**add \$r1, \$r2, \$r3**

# 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



# What Can Types do For Us?

**Can detect certain kinds of errors**

- **Memory errors (Rust):**

- ✓ Data races
- ✓ Dereferencing a null/dangling raw pointer
- ✓ Reads of undef (uninitialized) memory
- ✓ 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 !  
}
```

# Type Checking Overview

Three kinds of languages:

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

```
>>> x = 1
>>> def f():
    print(x)
    x(1)
>>> f()
1
TypeError: 'int' object is not callable
```

# 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
- 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, unsafe code in Rust
  - Some **dynamically typed languages** support “**pragmas**” or “**advice**”
    - type declarations

# Type Checking in Cool

- 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
  - Base classes: object, IO, Int, String, Bool
  - SELF\_TYPE
  - Note: there are no base types (as int in C)
- **The user declares types for all identifiers**
- **The compiler infers types for expressions**
  - Infers a type for every sub-expression

# Type Checking and Type Inference

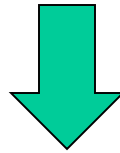
- **Type Checking** is the process of verifying fully typed programs
- **Type Inference** is the process of filling in missing type information
- The two are different, but the terms are often used interchangeably
- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions
  - Context-free grammars
- The appropriate formalism for type checking is **logical rules of inference**

# Why Rules of Inference?

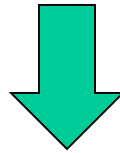
- Inference rules have the form If Hypothesis is true, then Conclusion is true
- Type checking computes via reasoning
  - If  $E_1$  and  $E_2$  have certain types, then  $E_3$  has a certain type
- Rules of inference are a compact notation for “If-Then” statements
- Start with a simplified system and gradually add features by type inference
- **Building blocks**
  - Symbol  $\wedge$  is “and”,  $\vee$  is “or”,
  - Symbol  $\Rightarrow$  is “if-then”
  - $x:T$  is “x has type T”

# From English to an Inference Rule

If  $e_1$  has type **Int** and  $e_2$  has type **Int**,  
then  $e_1 + e_2$  has type **Int**



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



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

**General inference rule:**

**$\text{Hypothesis}_1 \wedge \text{Hypothesis}_2 \wedge \dots \wedge \text{Hypothesis}_n \Rightarrow \text{Conclusion}$**



# Notation for Inference Rules

- General inference rule:

**Hypothesis<sub>1</sub> ∧ Hypothesis<sub>2</sub> ∧ ... ∧ Hypothesis<sub>n</sub> → Conclusion**

- By tradition inference rules are written

**| - Hypothesis<sub>1</sub> ... | - Hypothesis<sub>n</sub>**

---

**| - Conclusion**

- | - means “we can prove that...”

# Basic Rules

---

**$\vdash i:\text{Int}$**       **[int]**

**$\vdash e_1:\text{Int}$**

**$\vdash e_2:\text{Int}$**

---

**[Add]**

**$\vdash e_1+e_2:\text{Int}$**

---

**$\vdash 1:\text{Int}$**

---

**$\vdash 2:\text{Int}$**

---

**$\vdash 1+2:\text{Int}$**

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete types for expressions
- Example:  $1+2$

# 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

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

**In Cool, class Int inherits from Object**

# Soundness and Completeness

- A type-system is **sound** implies that all of type-checked programs are correct (in the other words, all of the incorrect program can't be type checked), i.e. there won't be any *false negative*, but may have **false positive**
- A type-system is **complete** implies that all of the correct program can be accepted by the type checker, i.e. there won't be any *false positive*, but may have **false negative**

# Type Checking Proofs

- Type checking proves facts  $e:T$ 
  - Proof is on the structure of the AST  $e$
  - Proof has the shape of the AST
  - One type rule is used for each AST node (sub-expressive)
- 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

# Rules for Constants

————— [False]  
|- false:Bool

————— [True]  
|- True:Bool

————— [String]  
|- s:String

————— [Int]  
|- i:Int

# Rule for New

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

————— [New]  
|- new T: T

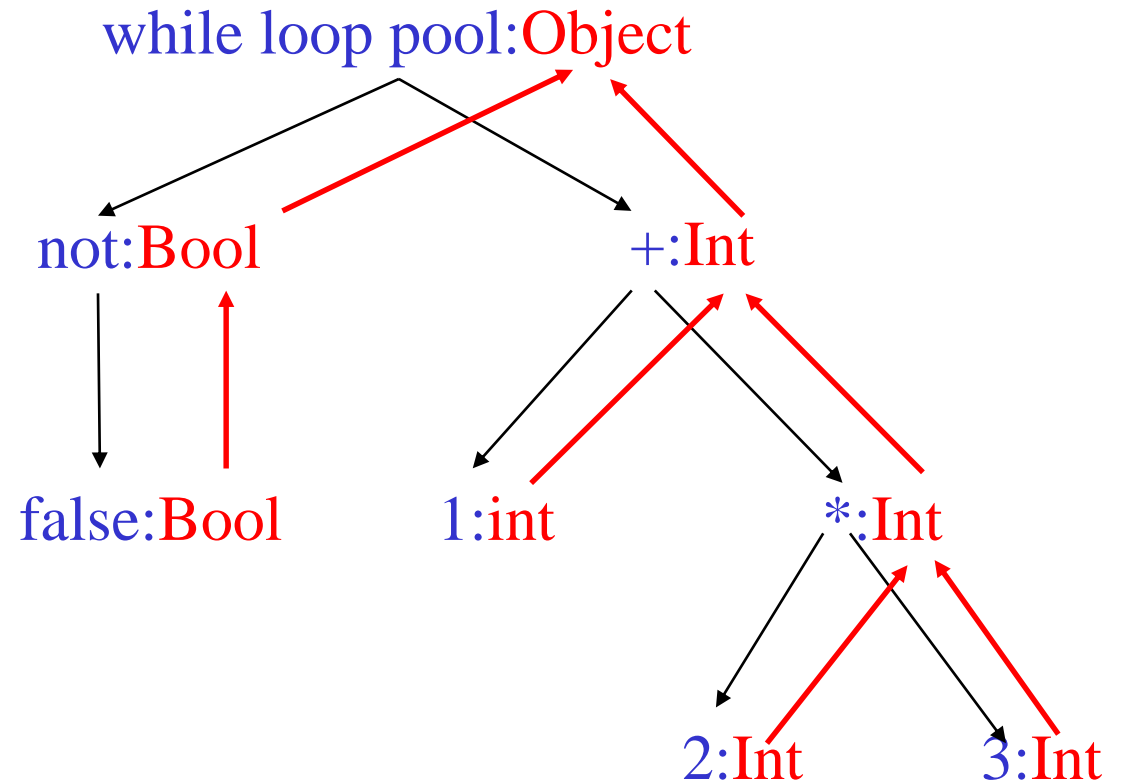
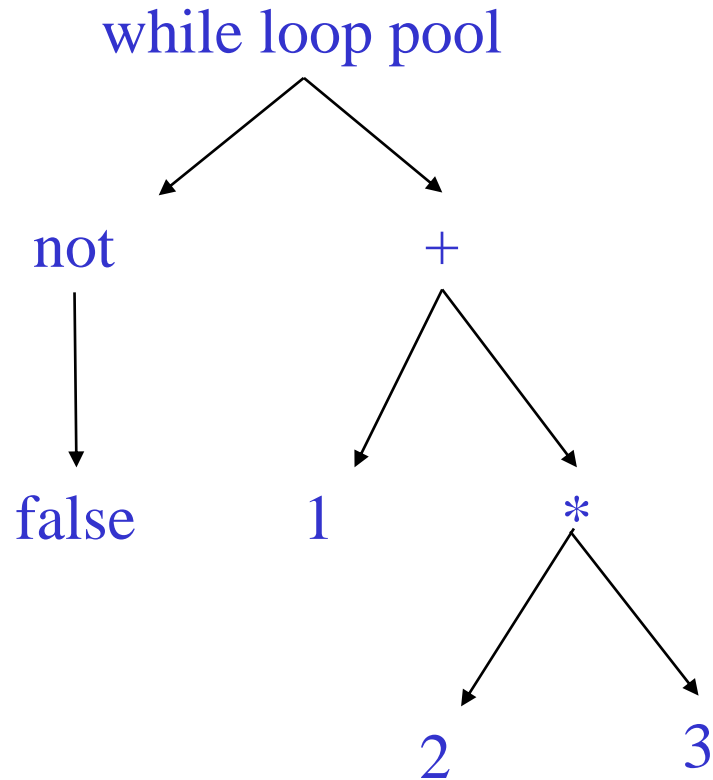
# Two More Rules

$$\frac{|- e:\text{Bool}}{|- \text{not } e: \text{Bool}} \quad [\text{Not}]$$
$$\frac{\begin{array}{l} |- e_1:\text{Bool} \\ |- e_2:T \end{array}}{|- \text{while } e_1 \text{ loop } e_2 \text{ pool: Object}} \quad [\text{While}]$$



# Typing: Example

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



# Typing Derivations

- The typing reasoning can be expressed as an inverted tree:
  - 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

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

# A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?} \text{ [Var]}$$

- The local, structural rule does not carry enough information to give a type.
- We need a hypothesis of the form “we are in the scope of a declaration of  $x$  with type  $T$ ”)

# A Solution: Put more information in the rules!

- A **type environment** gives types for **free variables**
- A variable is **free** in an expression  
if it is **not** defined within the expression
- A **type environment** is a function

**O: ObjectIds  $\rightarrow$  Types**

E.g.:

1. **x** and **y** are free in the expression **x \* y**
2. **x** is **not** free, **y** is free in **let x: Int x + y**
3. **x** and **y** are free in the expression **x + let x: Int in x + y**

# Type Environments

- Let **O** be a **type environment** function **O: ObjectIds  $\rightarrow$  Types**

The sentence **O|-e:T**

is read: Under the type environment **O**, it is provable that the expression **e** has the type **T**

# Modified Rules for Constants

$$\frac{}{O \vdash \text{false}:\text{Bool}} \quad [\text{False}]$$
$$\frac{}{O \vdash \text{True}:\text{Bool}} \quad [\text{True}]$$
$$\frac{}{O \vdash s:\text{String}} \quad [\text{String}]$$
$$\frac{}{O \vdash i:\text{Int}} \quad [\text{Int}]$$
$$\frac{\begin{array}{l} O \vdash e_1:\text{Int} \\ O \vdash e_2:\text{Int} \end{array}}{O \vdash e_1 + e_2:\text{Int}} \quad [\text{Add}]$$
$$\frac{\begin{array}{l} O \vdash e_1:\text{Bool} \\ O \vdash e_2:T \end{array}}{O \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool}:\text{Object}} \quad [\text{While}]$$

# New Rules

- And we can write new rules:

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

# Rules

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

$$\frac{}{O \vdash \text{True}:\text{Bool}} \quad [\text{True}]$$

$$\frac{}{O \vdash s:\text{String}} \quad [\text{String}]$$

$$\frac{}{O \vdash i:\text{Int}} \quad [\text{Int}]$$

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

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

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



# Let Rule

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

$O(T_0/x)$  is a new environment obtained from  $O$  by assigning  $T_0$  to  $x$

$$\begin{aligned} O(T_0/x)(x) &= T_0 \\ O(T_0/x)(y) &= O(y) \text{ if } x \neq y \end{aligned}$$

Note that the let-rule enforces variable scope

# Let Example

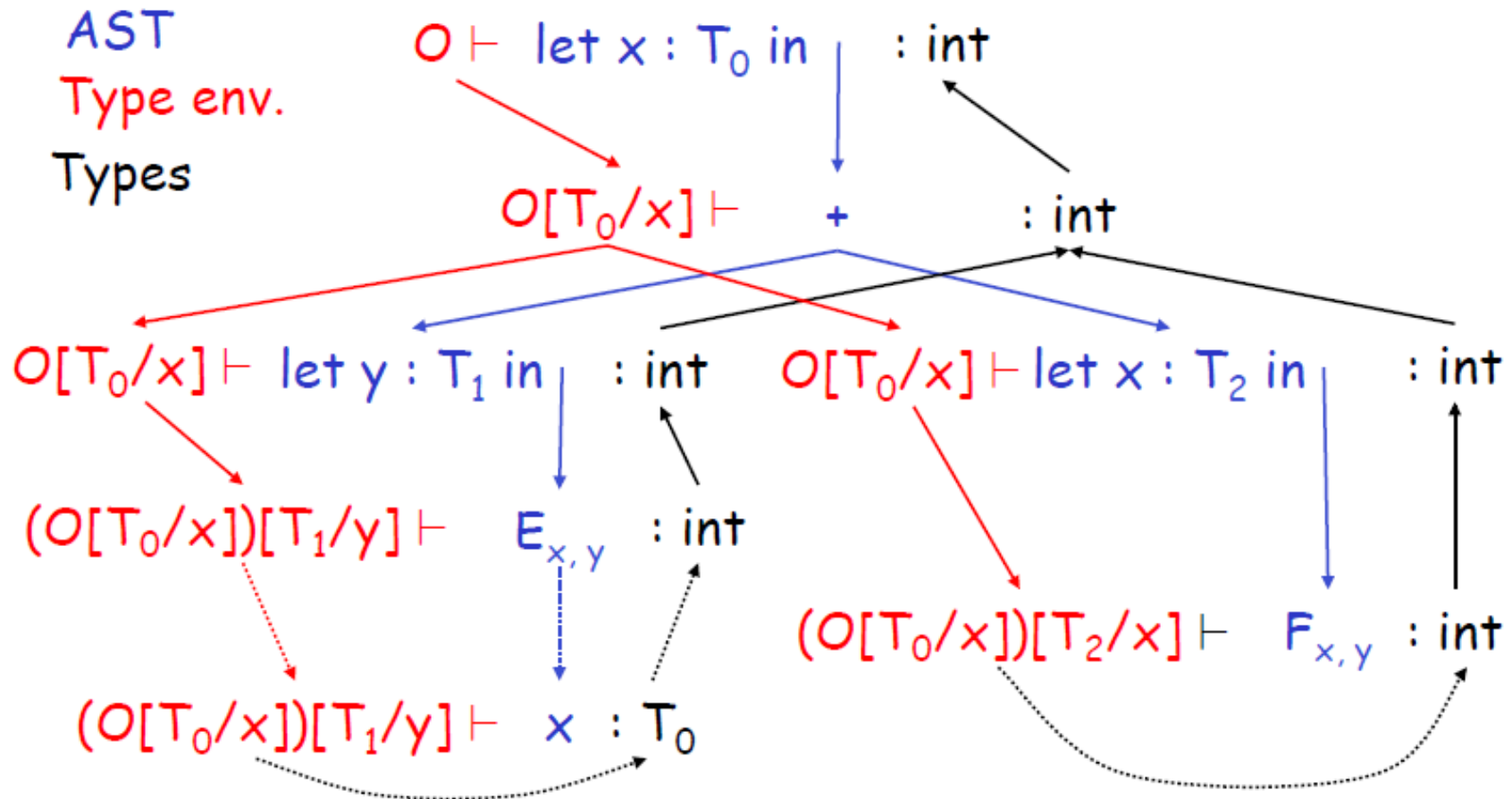
- Consider the Cool expression

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

- 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 let-rule

# Let Example

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



# Review

- Semantic analysis
  - Scope vs. Symbol Table
- Type systems
  - Let **O** be a **type environment** function

**$O: \text{ObjectIds} \rightarrow \text{Types}$**

The sentence  **$O|-e:T$**

is read: Under the type environment **O**, it is provable that the expression **e** has the type **T**

- Typing algorithm: depth-first search

# 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:

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

This rule is weak.

```
class C inherits P { ... }  
...  
let x : P ← new C in ...  
...
```

**The previous let rule does not allow this code**

# Subtyping

- Define a relation  $X \leq Y$  on classes (types) 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 (reflexive transitive closure)
  1.  $X \leq X$
  2.  $X \leq Y$  if  $X$  inherits from  $Y$
  3.  $X \leq Z$  if  $X \leq Y$  and  $Y \leq Z$

# Let with Initialization

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

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

class C inherits P { ... }  
 ...  
 let x : P  $\leftarrow$  new C in ...  
 ...

- Both rules for let are sound
  - Flexible rules that do not constrain programming
  - Restrictive rules that ensure safety of execution
- But more programs type check with the latter



# Quiz

```
class A {...}  
class B inherits A {...}
```

```
let x : A ← new B in 1+2
```

Draw Type Derivation Tree

## Answer

$$\begin{array}{c}
 \frac{}{\mathbf{O(A/x) \vdash 1: Int}} \quad \frac{}{\mathbf{O(A/x) \vdash 2: Int}} \\
 \hline
 \mathbf{O(A/x) \vdash 1+2: Int} \qquad \frac{}{\mathbf{O \vdash \text{new } B: B}} \quad \mathbf{B \leq A} \\
 \hline
 \mathbf{O \vdash \text{let } x : A \leftarrow \text{new } B \text{ in } 1+2: \underline{Int}}
 \end{array}$$

# 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
- There usually does not exist a sound and complete type systems for programming languages

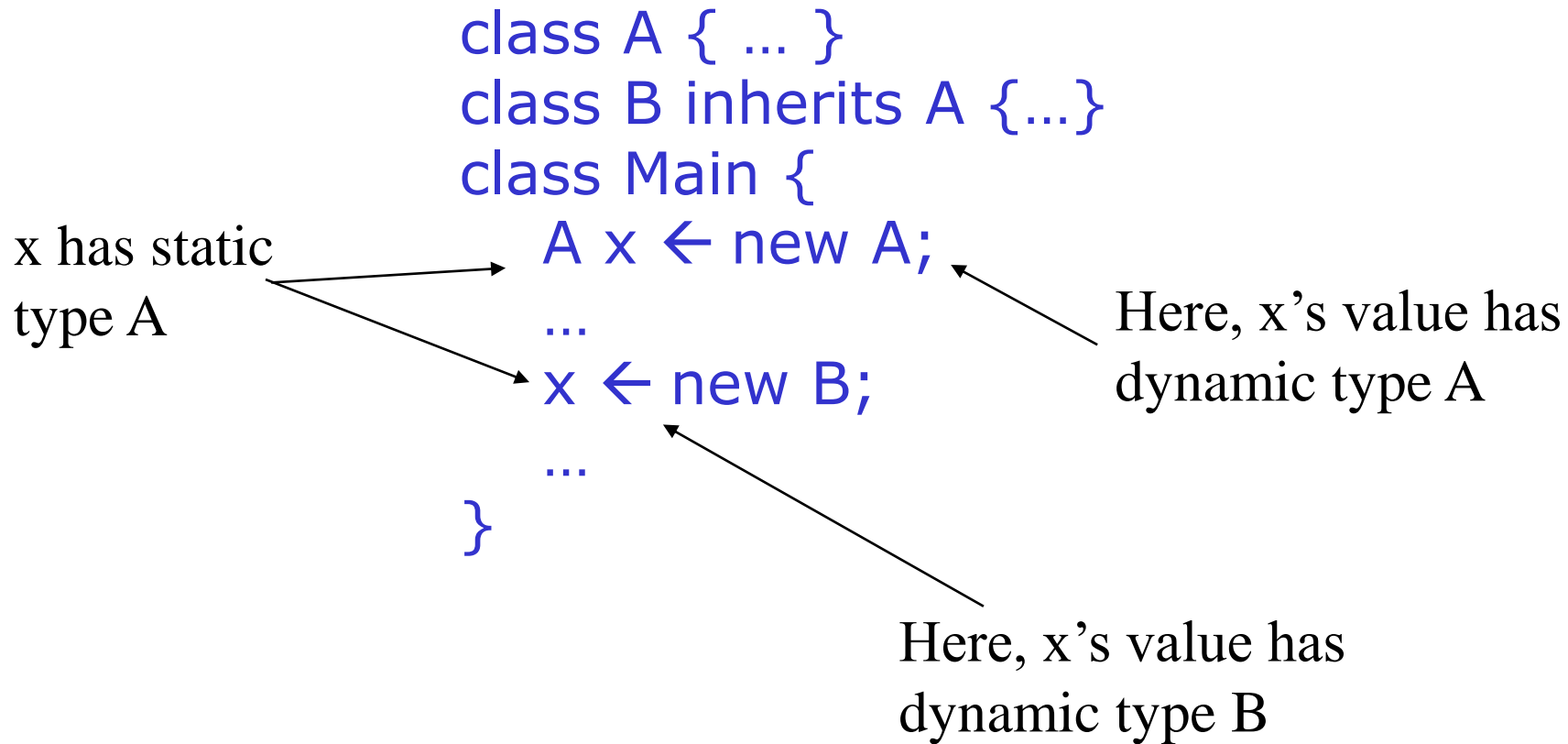
# Dynamic and Static Types

- 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

# 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

# Dynamic and Static Types in COOL



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

# 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  $\text{static\_type}(E)$  (call it  $C$ )
- 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 !

# Let Example

- Consider the following Cool class definitions

```
Class A { a() : Int { 0 }; }
```

```
Class B inherits A { b() : Int { 1 }; }
```

- An instance of **A** has method “a”
- An instance of **B** has methods “a” and “b”
  - A type error occurs if we try to invoke method “b” on an instance of **A**
  - It is OK to invoke method “a” on an instance of **B**

```
Let mya: A ← new B (OK)
```

```
Let myb: B ← new A (error)
```

```
mya.b() (error)
```

```
Let myb: B ← new B (OK)
```

```
myb.a() (OK)
```



# Review

- Type systems
  - Let **O** be a **type environment** function

$$\mathbf{O}: \mathbf{ObjectIds} \rightarrow \mathbf{Types}$$

The sentence  $\mathbf{O}|-e:T$

is read: Under the type environment **O**, it is provable that the expression **e** has the type **T**

**Soundness theorem** for the Cool type system:

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

# Review

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

$$\frac{}{O \vdash \text{True}:\text{Bool}} \quad [\text{True}]$$

$$\frac{}{O \vdash s:\text{String}} \quad [\text{String}]$$

$$\frac{}{O \vdash i:\text{Int}} \quad [\text{Int}]$$

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

$$O \vdash e_1:\text{Bool} \\ O \vdash e_2:T$$

$$\frac{}{O \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool}:\text{Object}} \quad [\text{While}]$$

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

$$O(T_0/x) \vdash e : T_1$$

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

$$O \vdash e_0:T \\ T \leq T_0 \\ O(T_0/x) \vdash e_1:T_1$$

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

# Let Example

Any error?

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

# Let Example

Any error?

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

# 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, although sound  
(good programs are rejected)
- But some good programs will be rejected anyway
  - The notion of a good program is undecidable

# Example

Is there any type error?

```
class A {a():Int {0}; }  
class B inherits A {b():Int {1};}  
class Main {  
  A x ← new A;  
  ...  
  x ← new B;  
  y: Int ← x.b ()  
  ...  
}
```

x has static  
type A

Here, x's value has  
dynamic type A

Here, x's value has  
dynamic type B

# Assignment

- More uses of subtyping:

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

# 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

$$\frac{\begin{array}{l} O_C(x)=T_0 \\ T_1 \leq T_0 \\ O_C \vdash e_1:T_1 \end{array}}{O_C \vdash x:T_0 \leftarrow e_1;} \quad [\text{Attr-Init}] \quad \frac{O_C(x)=T_0}{O_C \vdash x:T_0} \quad [\text{Attr-No-Init}]$$



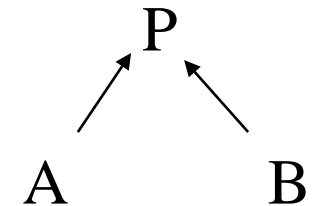
# If-Then-Else

- 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  $e_2$

- Consider the class hierarchy



if  $e_0$  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 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

$O \vdash e_0 : \text{Bool}$

$O \vdash e_1 : T_1$

$O \vdash e_2 : T_2$

---

$O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \text{lub}(T_1, T_2)$  **[If-Then-Else]**

# Case

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

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

# Method Dispatch

- There is a problem with type checking method calls:

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

- 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

$$M(C, f) = (T_1, \dots, T_n, T)$$

means in class **C** there is a method **f**:  $f(x_1:T_1, \dots, x_n:T_n): T$

- 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
  - Examples of a rule that does not use **M**:

$\frac{}{O, M \vdash \text{false}:\text{Bool}} \quad [\text{False}]$	$\frac{}{O, M \vdash 1:\text{Int}} \quad [\text{Int}]$
$\frac{O, M \vdash e_1:\text{Int} \quad O, M \vdash e_2:\text{Int}}{O, M \vdash e_1 + e_2:\text{Int}} \quad [\text{Add}]$	$\frac{O, M \vdash e_1:\text{Bool} \quad O, M \vdash e_2:T}{O, M \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool}:\text{Object}} \quad [\text{While}]$

- Only the dispatch rules use **M**

# Method Dispatch

- Method dispatch:

$$\begin{array}{c} \mathbf{O, M \vdash e_0 : T_0} \\ \mathbf{O, M \vdash e_1 : T_1} \\ \mathbf{\dots\dots\dots} \\ \mathbf{O, M \vdash e_n : T_n} \\ \mathbf{M(T_0, f) = (T_1', \dots, T_n', T)} \\ \mathbf{T_i \leq T_i' \text{ for all } 1 \leq i \leq n} \\ \hline \mathbf{O, M \vdash e_0.f(e_1, \dots, e_n) : T} \end{array} \quad \mathbf{[Dispatch]}$$

# 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

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



# Handling the SELF\_TYPE

- 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

# An Example

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

Any error?

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

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

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

# SELF\_TYPE to the Rescue

- Insight:
  - `inc` returns “`self`”
  - Therefore the return value has same type as “`self`”
  - Which could be `Count` or any subtype of `Count` at runtime
  - 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
- `SELF_TYPE` allows the return type of `inc` to change when `inc` is inherited
- Modify the declaration of `inc` to read
$$\text{inc}() : \text{SELF\_TYPE} \{ \dots \}$$
- The type checker can now prove:
$$\begin{aligned} \text{O, M} \vdash (\text{new Count}).\text{inc}() &: \text{Count} \\ \text{O, M} \vdash (\text{new Stock}).\text{inc}() &: \text{Stock} \end{aligned}$$
- The program from before is now well typed

# 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
- What can be the dynamic type of the object returned by **inc**?
  - Answer: whatever could be the type of “**self**” at runtime

```
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**

# SELF\_TYPE and Dynamic Types

- 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:

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

- Note: The meaning of **SELF\_TYPE** depends on where it appears
  - We write **SELF\_TYPE<sub>C</sub>** to refer to an occurrence of **SELF\_TYPE** in the body of **C**

$$\text{SELF\_TYPE}_C \leq C$$

# 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

# 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 not SELF\_TYPE
- Four cases:
  1.  $\text{SELF\_TYPE}_C \leq T$  if  $C \leq T$ 
    - $\text{SELF\_TYPE}_C$  can be any subtype of  $C$  including  $C$  itself
    - Thus this is the most flexible rule we can allow
  2.  $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
  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)

# 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 SELF\_TYPE Appear 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. **m@T(E<sub>1</sub>,...,E<sub>n</sub>) // static dispatch**

- T cannot be SELF\_TYPE

3. **x : T.**

- T can be SELF\_TYPE, an attribute whose type is SELF\_TYPE<sub>C</sub>

4. **let x : T in E**

- T can be SELF\_TYPE, x has type SELF\_TYPE<sub>C</sub>

5. **new T**

- T can be SELF\_TYPE, creates an object of the same type as self

6. **m(x : T) : T' { ... } Only T' can be SELF\_TYPE !**

# Typing Rules for SELF\_TYPE

- Since occurrences of **SELF\_TYPE** depend on the enclosing class, we need to know the class in which **an expression** occurs.
- We need to carry more context during type checking
- New form of the typing judgment:

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

- A mapping **O** giving types to object id's
- A mapping **M** giving types to methods
- The current class **C** where **e** occurs

# New Rules

- 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 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  $\leq$  and **lub** are the new ones
- Example:

$$\frac{\begin{array}{l} \text{O,M,C} \vdash e_1:\text{Int} \\ \text{O,M,C} \vdash e_2:\text{Int} \end{array}}{\text{O,M,C} \vdash e_1+e_2:\text{Int}} \text{ [Add]} \quad \frac{\begin{array}{l} \text{O(x)} = \text{T}_0 \\ \text{O,M,C} \vdash e_1:\text{T}_1 \\ \text{T}_1 \leq \text{T}_0 \end{array}}{\text{O,M,C} \vdash x \leftarrow e_1:\text{T}_1} \text{ [Assign]}$$

# What's Different?

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

---

$O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_{n+1}'$

$O, M, C \vdash e_0 : T_0$   
 $O, M, C \vdash e_1 : T_1$   
 .....  
 $O, M, C \vdash e_n : T_n$   
 $M(T_0, f) = (T_1', \dots, T_n', \text{SELF\_TYPE})$   
 $T_i \leq T_i'$  for all  $1 \leq i \leq n$

---

$O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_0$

If the return type of the method is **SELF\_TYPE** then the type of the dispatch is the type of the dispatch expression

# An Example

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

```
class Stock inherits Count {  
  name() : String { ... };  
};  
  
class Main {  
  a : Stock ← (new Stock).inc ();  
  ... a.name() ...  
};
```

- (new Stock).inc() has dynamic type Stock,
- (new Stock).inc() has static type Stock,
- So this is well-typed

# Static Dispatch

- Revisit the original rule for static dispatch

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


---

$$O, M, C \vdash e_0 @ \mathbf{T.f}(e_1, \dots, e_n) : \mathbf{T_{n+1}'}$$

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


---

$$O, M, C \vdash e_0 @ \mathbf{T.f}(e_1, \dots, e_n) : \mathbf{T_0}$$

# Attributes and Methods

$$O_C(x)=T_0 \quad T_1 \leq T_0$$

$$O_C, M, C \vdash e_1 : T_1$$


---

[Attr-Init]

$$O_C, M, C \vdash x : T_0 \leftarrow e_1;$$

$$O_C(x)=T_0$$


---

[Attr-No-Init]

$$O_C, M, C \vdash x : T_0;$$

$$M(C, f) = (T_1, \dots, T_n, \mathbf{T_0}) \quad \mathbf{T_0 \neq SELF\_TYPE} \quad \mathbf{T_0' \leq T_0}$$

$$O_C[\mathbf{SELF\_TYPE}_C/\mathbf{self}][T_1/x_1] \dots [T_n/x_n], M, C \vdash e : \mathbf{T_0'}$$


---

[Method]

$$O_C, M, C \vdash f(x_1 : T_1, \dots, x_n : T_n) : \mathbf{T_0} \{ e \}$$

$$M(C, f) = (T_1, \dots, T_n, \mathbf{SELF\_TYPE}) \quad \mathbf{T_0' \leq SELF\_TYPE}_C$$

$$O_C[\mathbf{SELF\_TYPE}_C/\mathbf{self}][T_1/x_1] \dots [T_n/x_n], M, C \vdash e : \mathbf{T_0'}$$


---

[Method]

$$O_C, M, C \vdash f(x_1 : T_1, \dots, x_n : T_n) : \mathbf{SELF\_TYPE} \{ e \}$$



# Summary of SELF\_TYPE

- The extended  $\leq$  and **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

# 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
- 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

# One-Pass Type Checking

- 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

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

```
TypeCheck(Environment, n) // node n denotes the expression e1 + e2
{
    T1 = TypeCheck(Environment, n.leftchild);
    T2 = TypeCheck(Environment, n.rightchild);
    Check T1 == T2 == Int;
    return Int;
}
```

# Error Recovery

- As with parsing, it is important to recover from type errors
- Detecting where errors occur is easier than in parsing
  - There is no reason to skip over portions of code
- The Problem:
  - What type is assigned to an expression with no legitimate type?
  - This type will influence the typing of the enclosing expression

# Error Recovery Attempt

- Assign type **Object** to ill-typed expressions

$\text{let } y : \text{Int} \leftarrow x + 2 \text{ in } y + 3$

- Since **x** is undeclared its type is **Object**
- But now we have **Object + Int**
- This will generate another typing error
- We then say that **Object + Int = Object**
- Then the initializer's type will not be **Int**

$\Rightarrow$  a workable solution but with cascading errors

# Better Error Recovery

- We can introduce a new type called `No_type` for use with ill-typed expressions
- Define `No_type`  $\leq$  `C` for all types `C`
- Every operation is defined for `No_type`
  - With a `No_type` result
- Only one typing error for:

`let y : Int  $\leftarrow$  x + 2 in y + 3`

# Notes

- A “real” compiler would use something like  
`No_type`
- However, there are some implementation issues
  - The class hierarchy is not a tree anymore



# Quiz

Write the typing derivation for

i: Int;

x: Int;

i  $\leftarrow$  2;

x  $\leftarrow$  1;

while ~ i < 0 loop{

    i  $\leftarrow$  i - 1;

    let y: Int  $\leftarrow$  2 in x  $\leftarrow$  x \* y;

}

pool;