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

Errors

let y: Int in x + 3

Error?

let y: String ← “abc” in y + 3

Error?

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 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

z	**Real**
y	**Real**
x	**Real**

Symbol table for Q

Symbol table for P

Q	**Proc**
x	**Bool**
a	**Bool**

Symbol table for global

P	**Proc**
x	**Int**
y	**Int**

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

We will supply a symbol table manager for your project ,e.g., symtab.h, a list of scope, a scope is a list of entries <id, data>

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 can't 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,)
- 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 ➔ is "if-then"**
  - **x:T is "x has type T"**

From English to an Inference Rule

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

(e_1 has type Int $\wedge$ e_2 has type Int) ➔ ($e_1 + e_2$ has type Int)

$(e_1 : Int \wedge e_2 : Int) \rightarrow (e_1 + e_2 : Int)$

General inference rule:

$Hypothesis_1 \wedge Hypothesis_2 \wedge \ldots \wedge Hypothesis_n \rightarrow Conclusion$

Notation for Inference Rules

- **General inference rule:**
  $\mathbf{Hypothesis_1 \wedge Hypothesis_2 \wedge \ldots \wedge Hypothesis_n \Rightarrow Conclusion}$

- **By tradition inference rules are written**

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

- **|- means "we can prove that…"**

Basic Rules

$$\frac{}{\vdash i:Int}\ [int]$$

$$\frac{\vdash 1:Int \qquad \vdash 2:Int}{\vdash 1+2:Int}$$

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

- 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

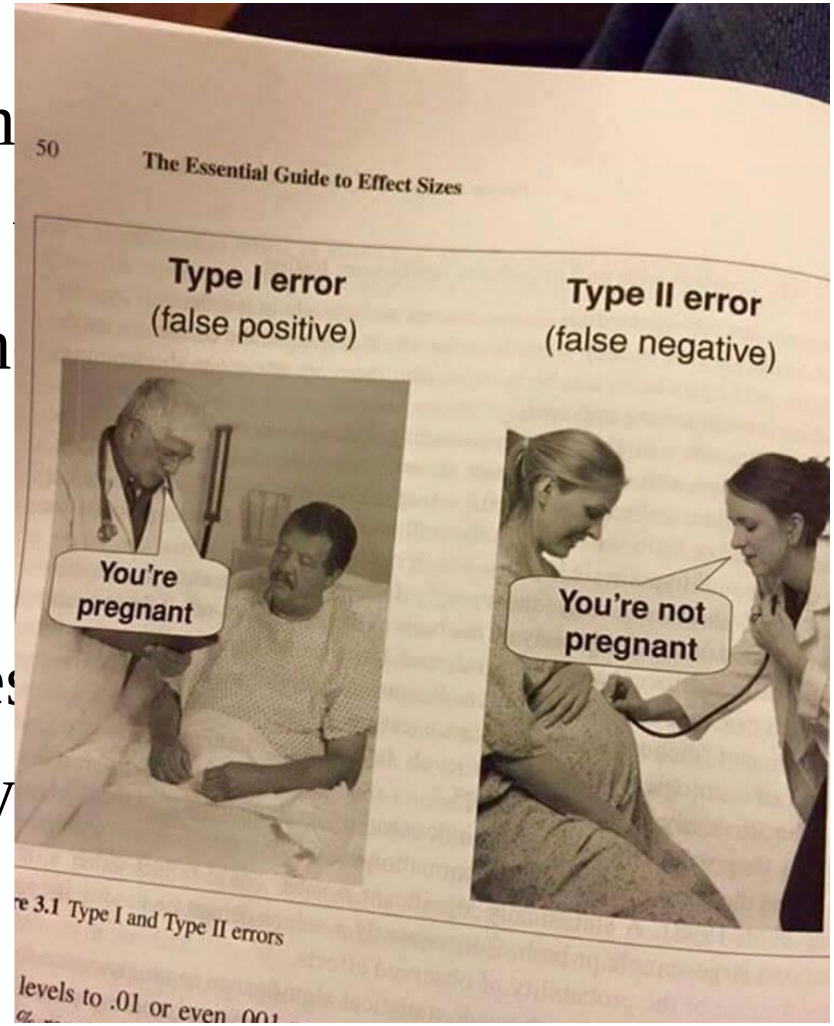
$$\frac{}{\vdash i : \mathrm{Object}} \quad \textbf{(i is an integer constant)}$$

In Cool, class Int inherits from Object

Soundness and Completeness

- A type-system is **sound** implies th
  programs are correct (in the other
  incorrect program can't be type ch
  be any *false positive*.

- A type-system is **complete** implies
  program can be accepted by the ty
  won't be any *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

$$\frac{}{\vdash \text{false:Bool}}\ [\text{False}] \qquad \frac{}{\vdash \text{True:Bool}}\ [\text{True}]$$

$$\frac{}{\vdash \text{s:String}}\ [\text{String}] \qquad \frac{}{\vdash \text{i:Int}}\ [\text{Int}]$$

Rule for New

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

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

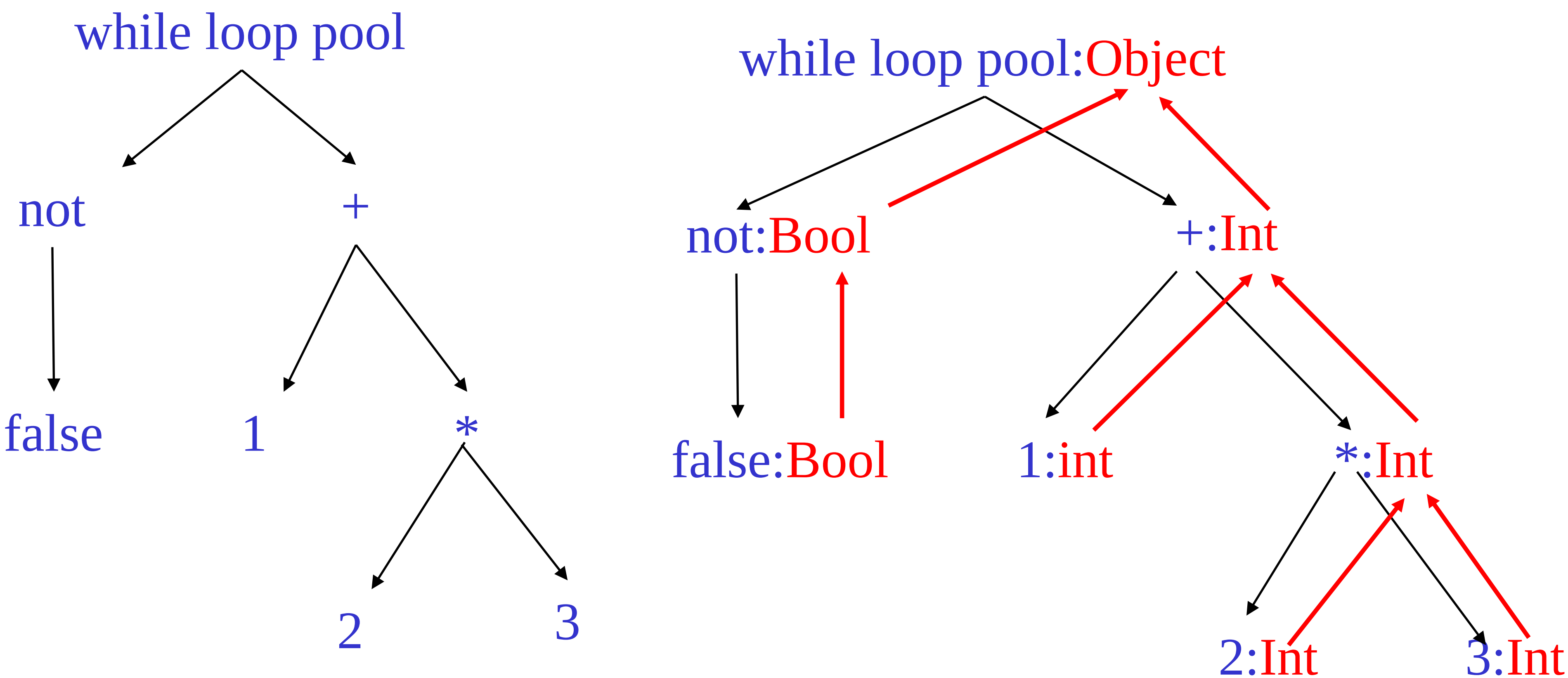
Two More Rules

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

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

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

$$\dfrac{\dfrac{\vdash \textbf{false:Bool}}{\vdash \textbf{not false: Bool}} \qquad \dfrac{\vdash \textbf{1:Int} \quad \dfrac{\vdash \textbf{2:Int} \qquad \vdash \textbf{3:Int}}{\vdash \textbf{2*3: Int}}}{\vdash \textbf{1+2*3: Int}}}{\vdash \textbf{while not false loop 1 + 2 * 3 pool:Object}}$$

A Problem

- What is the type of a variable reference?

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

- 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 → 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 → 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

$$\frac{}{O \vdash false:Bool}\ [False] \qquad \frac{}{O \vdash True:Bool}\ [True]$$

$$\frac{}{O \vdash s:String}\ [String] \qquad \frac{}{O \vdash i:Int}\ [Int]$$

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

New Rules

- And we can write new rules:

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

Review

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

  O: ObjectIds → 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

$$\frac{}{O \vdash false:Bool}\ [False] \qquad \frac{}{O \vdash True:Bool}\ [True]$$

$$\frac{}{O \vdash s:String}\ [String] \qquad \frac{}{O \vdash i:Int}\ [Int]$$

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

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

Let Rule

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

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

$$O(T_0/x)(x)=T_0$$
$$O(T_0/x)(y)=O(y) \text{ if } x!=y$$

Note that the let-rule enforces variable scope

Let Example

- Consider the Cool expression

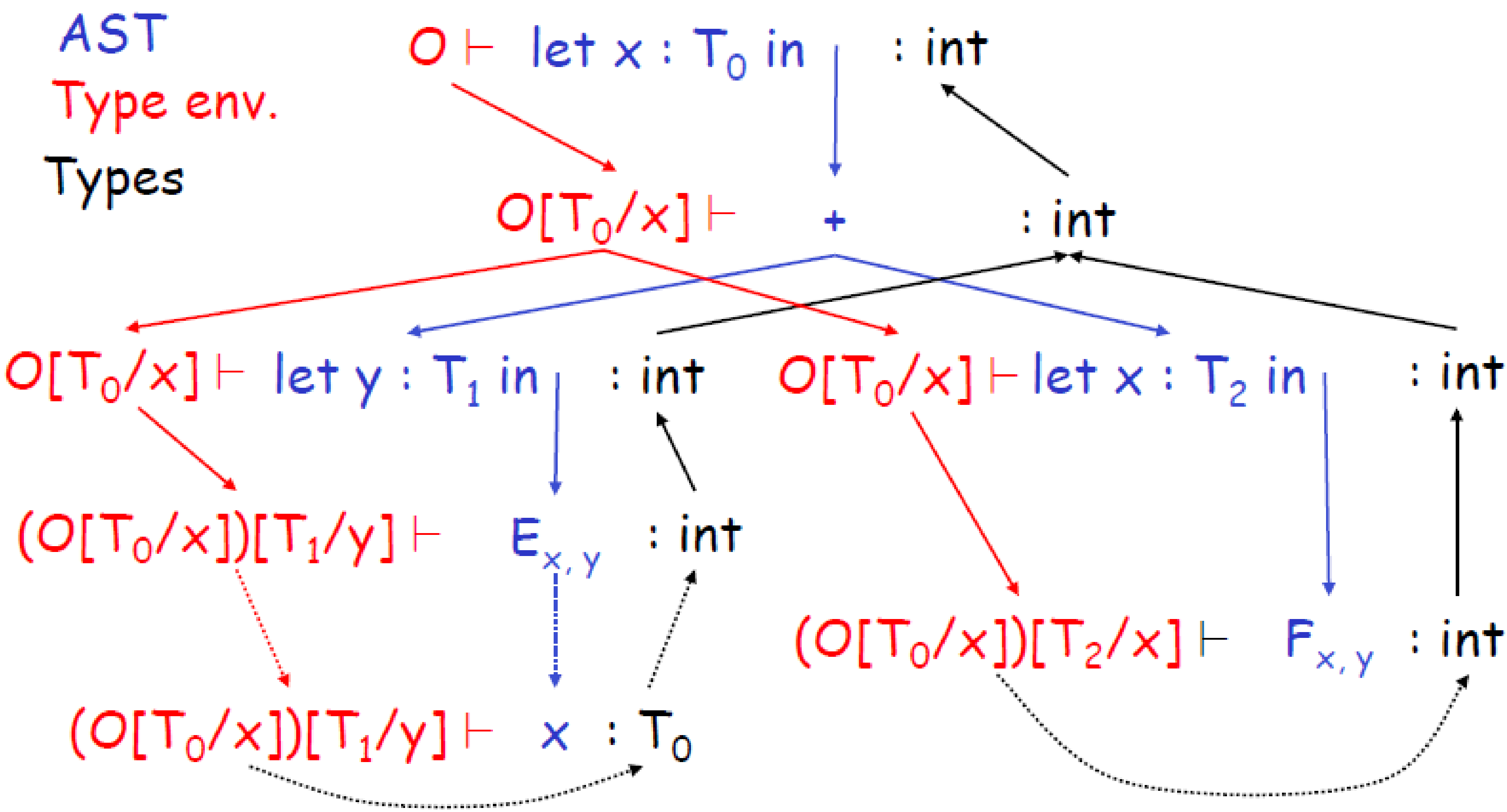
$$\textbf{let } x : T_0 \textbf{ in } (\textbf{let } y : T_1 \textbf{ in } E_{x,y}) + (\textbf{let } x : T_2 \textbf{ in } F_{x,y})$$

- 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

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



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:

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

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

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

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

```
class C inherits P { … }
…
let x : P ← 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

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

  dynamic_type(E) = 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

```
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 $B \leq A$

Dynamic and Static Types

Soundness theorem for the Cool type system:

$\forall$ E. dynamic_type(E) $\leq$ static_type(E)

Why is this Ok?

- For E, compiler uses 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)

Let Example

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

Any error?

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

Any error?

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

Assignment

- More uses of subtyping:

$$\frac{O(x)=T_0 \quad T_1 \le T_0 \quad O \vdash e_1 : T_1}{O \vdash x \leftarrow e_1 : T_1} \quad [Assign]$$

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

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

$$\frac{\begin{array}{c} O_C(x)=T_0 \\ T_1 \leq T_0 \\ O_C \vdash e_1:T_1 \end{array}}{O_C \vdash x:T_0 \leftarrow e_1;} \quad \text{[Attr-Init]}$$

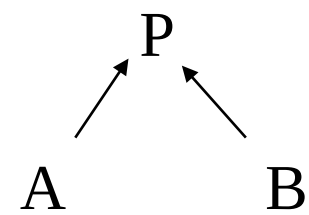
$$\frac{O_C(x)=T_0}{O_C \vdash x:T_0} \quad \text{[Attr-No-Init]}$$

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

- 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

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

Case

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

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

Method Dispatch

- There is a problem with type checking method calls:

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

- 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

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

means in class C there is a method f: $f(x_1{:}T_1, \ldots, x_n{:}T_n){:}\ T_n$

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

$$\mathbf{O,M|\text{-}\ e:T}$$

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:

$$\frac{}{O,M \vdash \text{false} : \text{Bool}}\ [\text{False}] \qquad \frac{}{O,M \vdash 1 : \text{Int}}\ [\text{Int}]$$

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

  - Only the dispatch rules use M

Method Dispatch

- There is a problem with type checking method calls:

$$\frac{\begin{array}{c} O,M \vdash e_0 : T_0 \\ O,M \vdash e_1 : T_1 \\ \ldots\ldots \\ O,M \vdash e_n : T_n \\ M(T_0, f) = (T_1`, \ldots, T_n`, T) \\ T_i \le T_i` \text{ for all } 1 \le i \le n \end{array}}{O,M \vdash e_0.f(e_1, \ldots, e_n) : T} \quad \text{[Dispatch]}$$

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

$$\frac{\begin{array}{c} O,M \vdash e_0 : T_0 \\ O,M \vdash e_1 : T_1 \\ \ldots\ldots \\ O,M \vdash e_n : T_n \\ M(T, f) = (T_1`, \ldots, T_n`, T_{n+1}) \\ T_i \leq T_i` \text{ for all } 1 \leq i \leq n \\ T_0 \leq T \end{array}}{O,M \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}} \quad [\text{StaticDispatch}]$$

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 !
  - 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

  inc() : SELF_TYPE { … }
- The type checker can now prove:

  O, M |- (new Count).inc() : Count

  O, M |- (new Stock).inc() : Stock
- 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"

    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:

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

- 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

$$\mathbf{SELF\_TYPE_C \leq C}$$

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 SELF_TYPE_C by C
- This suggests one way to handle SELF_TYPE :
  - Replace all occurrences of SELF_TYPE_C by C
- This would be correct but it is like not having 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
  - $lub(T_1,T_2)$ 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
- Four cases:
  1. $SELF\_TYPE_C \leq T$ if $C \leq T$
     - $SELF\_TYPE_C$ can be any subtype of C including C itself
     - Thus this is the most flexible rule we can allow
  2. $SELF\_TYPE_C \leq SELF\_TYPE_C$
  3. $T \leq SELF\_TYPE_C$ always false
     - Note: $SELF\_TYPE_C$ can denote any subtype of C.
  4. $T \leq T'$ (according to the rules from before)

Extending lub(T,T')

- Let T and T' be any types but SELF_TYPE
- Again there are four cases:

1. lub(SELF_TYPE_C, SELF_TYPE_C) = SELF_TYPE_C
2. lub(SELF_TYPE_C, T) = lub(C, T)

   This is the best we can do because $\text{SELF\_TYPE}_C \leq C$
3. lub(T, SELF_TYPE_C) = lub(C, T)
4. 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(E1,…,En)

- T **cannot** be SELF_TYPE

3. x : T.

- T can be SELF_TYPE, an attribute whose type is $SELF\_TYPE_C$

4. let x : T in E

- T can be SELF_TYPE, x has type $SELF\_TYPE_C$

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 |- 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

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

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

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

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:

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

$$\frac{O(x) = T_0 \quad O,M,C \vdash e_1 : T_1 \quad T_1 \leq T_0}{O,M,C \vdash x \leftarrow e_1 : T_1} \text{[Assign]}$$

What's Different?

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

$$\frac{\begin{array}{c} O,M,C \vdash e_0:T_0 \\ O,M,C \vdash e_1:T_1 \\ \ldots\ldots \\ O,M,C \vdash e_n:T_n \\ M(T_0, f)=(T_1`,\ldots,T_n`, \text{SELF\_TYPE}) \\ T_i \leq T_i` \text{ for all } 1 \leq i \leq n \end{array}}{O,M,C \vdash e_0.f(e_1,\ldots,e_n):T_0}$$

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

- Recall the original rule for static dispatch

$$\frac{\begin{array}{c} O,M,C \vdash e_0 : T_0 \\ O,M,C \vdash e_1 : T_1 \\ \ldots\ldots \\ O,M,C \vdash e_n : T_n \\ M(T, f) = (T_1`, \ldots, T_n`, T_{n+1}`) \\ T_i \le T_i` \text{ for all } 1 \le i \le n \\ T_0 \le T \\ T_{n=1}` \ne SELF\_TYPE \end{array}}{O,M,C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}`}$$

$$\frac{\begin{array}{c} O,M,C \vdash e_0 : T_0 \\ O,M,C \vdash e_1 : T_1 \\ \ldots\ldots \\ O,M,C \vdash e_n : T_n \\ M(T, f) = (T_1`, \ldots, T_n`, SELF\_TYPE) \\ T_i \le T_i` \text{ for all } 1 \le i \le n \\ T_0 \le T \end{array}}{O,M,C \vdash e_0@T.f(e_1, \ldots, e_n) : T_0}$$

Attributes and Methods

$$\frac{O_C(x)=T_0 \quad T_1 \le T_0 \quad O_C,M,C \vdash e_1:T_1}{O_C,M,C \vdash x:T_0 \leftarrow e_1;}\ \text{[Attr-Init]}$$

$$\frac{O_C(x)=T_0}{O_C,M,C \vdash x:T_0;}\ \text{[Attr-No-Init]}$$

$$\frac{M(C,f)=(T_1,\ldots,T_n,T_0) \quad T_0 \ne SELF\_TYPE \quad T_0' \le T_0 \quad O_C[SELF\_TYPE_C/self][T_1/x_1]\ldots[T_n/x_n], M, C \vdash e: T_0'}{O_C,M,C \vdash f(x_1:T_1,\ldots,x_n:T_n):T_0 \{ e \}}\ \text{[Method]}$$

$$\frac{M(C,f)=(T_1,\ldots,T_n, SELF\_TYPE) \quad T_0' \le SLEF\_TYPE_C \quad O_C[SELF\_TYPE_C/self][T_1/x_1]\ldots[T_n/x_n], M, C \vdash e: T_0'}{O_C,M,C \vdash f(x_1:T_1,\ldots,x_n:T_n): SELF\_TYPE \{ e \}}\ \text{[Method]}$$

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
  - 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

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**

$$\frac{O,M,C \vdash e_1 : Int \qquad O,M,C \vdash e_2 : Int}{O,M,C \vdash e_1 + e_2 : Int}\ [Add]$$

```
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:

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

Notes

- A “real” compiler would use something like

  No_type

- However, there are some implementation issues
  - The class hierarchy is not a tree anymore
- The Object solution is fine in the class project

Quiz

Write the typing derivation for

```
i: Int;
x: Int;
i ← 2;
x← 1;
while ~ i< 0 loop{
        i ← i – 1;
        let y: Int ←2 in x← x * y;
}
pool;
```