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
- The user declares types for identifiers
- The compiler infers types for expressions
- 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
  - Context-free grammars
- The appropriate formalism for type checking is logical rules of inference
  *If E₁ and E₂ have certain types, then E₃ has a certain type*

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 \( \land \) is "and"
  - Symbol \( \Rightarrow \) is "if-then"
  - \( x : T \) is "\( x \) has type \( T \)"

From English to an Inference Rule (2)

If \( e₁ \) has type \( \text{Int} \) and \( e₂ \) has type \( \text{Int} \), then \( e₁ + e₂ \) has type \( \text{Int} \)

\[ (e₁ \text{ has type } \text{Int} \land e₂ \text{ has type } \text{Int}) \Rightarrow e₁ + e₂ \text{ has type } \text{Int} \]

\( (e₁: \text{Int} \land e₂: \text{Int}) \Rightarrow e₁ + e₂: \text{Int} \)
From English to an Inference Rule (3)

The statement
\[(e_1: \text{Int} \land e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}\]
is a special case of
\[\text{Hypothesis}_1 \land \ldots \land \text{Hypothesis}_n \Rightarrow \text{Conclusion}\]

This is an inference rule.

Notation for Inference Rules

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

- Cool type rules have hypotheses and conclusions
  
  \[e: T\]
  
  means "it is provable that . . ."

Two Rules

\[
\begin{align*}
\text{i is an integer} & \quad \text{i: int} & \quad \text{[Int]} \\
\text{e}_1: \text{int} & \quad \text{e}_2: \text{int} & \quad \text{[Add]} \\
\text{e}_1 + \text{e}_2: \text{int} & \quad \text{[Add]} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{1 is an integer} & \quad \text{2 is an integer} & \quad \text{[Int]} \\
\text{1: Int} & \quad \text{2: Int} & \quad \text{[Add]} \\
\text{1+2: Int} & \quad \text{[Add]} \\
\end{align*}
\]

Soundness

- A type system is **sound** if
  
  - Whenever \( e: T \)
  
  - Then \( e \) evaluates to a value of type \( T \)

- We only want sound rules.

\[
\begin{align*}
\text{i is an integer} & \quad \text{i: Object} \\
\end{align*}
\]

Type Checking Proofs

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

- In the type rule used for a node \( e \):
  
  - Hypotheses are the proofs of types of \( e \)'s sub-expressions
  
  - Conclusion is the type of \( e \)

- Types are computed in a **bottom-up** pass over the AST
Rules for Constants

- false: Bool
  [Bool]

s is a string constant
s: String
  [String]

Rule for New

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

new T: T
  [New]

Two More Rules

- e: Bool
  [Not]

- e: Bool
  [Loop]

- e: Bool
  e; T
  while e1 loop e2 pool: Object
  [Loop]

A Problem

- What is the type of a variable reference?
  
  x is an identifier
  x: ?
  [Var]

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

A Solution

- Put more information in the rules!

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

Type Environments

Let O be a function from ObjectIdentifiers to Types

The sentence

O e: T

is read: Under the assumption that variables have the types given by O, it is provable that the expression e has the type T
**Modified Rules**

The type environment is added to the earlier rules:

- **i** is an integer
  
  \[
  \frac{\text{[Int]}}{O \ i : \text{int}}
  \]

- \[O e_1 : \text{int}\]

- \[O e_2 : \text{int}\]

- \[O e_1 + e_2 : \text{int}\]

**New Rules**

And we can write new rules:

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

**Let**

\[
O[T_0/x] e_1 : T_1 \quad [\text{Let-Init}]
\]

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

\[O[T/y]\] means \(O\) modified to return \(T\) on argument \(y\)

Note that the let rule enforces variable scope

**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:

\[
O e_0 : T_0 \quad [\text{Let-Init}]
\]

\[
O[T_0/x] e_1 : T_1 \quad [\text{Let-Init}]
\]

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

This rule is weak. Why?

**Subtyping**

- Define a relation \(\leq\) on classes
  - \(X \leq X\)
  - \(X \leq Y\) if \(X\) inherits from \(Y\)
  - \(X \leq Z\) if \(X \leq Y\) and \(Y \leq Z\)
- An improvement
  \[
  O e_0 : T \quad T \leq T_0
  \]

  \[
  O[T_0/x] e_1 : T_1 \quad [\text{Let-Init}]
  \]

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

- Both let rules are correct, but more programs typecheck with the second one.
- More uses of subtyping:

\[
\begin{align*}
O(\text{Id}) &= T_0 \\
O e_1 : T_1 \\
T_i &\leq T_0 \\
O \text{ Id} &\leftarrow e_1 : T_i
\end{align*}
\]

[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:

\[
\begin{align*}
O_c(\text{Id}) &= T_0 \\
O_c e_1 : T_1 \\
T_i &\leq T_0 \\
O_c \text{ Id} &\leftarrow e_1;
\end{align*}
\]

[Attr-Init]

If-Then-Else

- Consider:

\[
\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi}
\]
- The result can be either \( e_1 \) or \( e_2 \).
- The type is either \( e_1 \)'s type or \( e_2 \)'s type.
- The best we can do is the smallest supertype larger than the type of \( e_1 \) or \( e_2 \).

Least Upper Bounds

- \( \text{lub}(X,Y) \), the least upper bound of \( X \) and \( Y \), is \( Z \) if:
  - \( X \leq Z \land Y \leq Z \)
  - \( Z \) is an upper bound.
  - \( X \leq Z \land Y \leq Z \Rightarrow Z \leq Z' \)
  - \( Z \) is least among upper bounds.
- Digression (Examples with ordered sets):
  - Least common multiple, Greatest common divisor
  - Set Union, Set Intersection
  - Maximum, Minimum

If-Then-Else Revisited

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

\[
\begin{align*}
O e_1 : \text{Bool} \\
O e_2 : T_2 \\
O e_3 : T_3 \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : \text{lub}(T_2, T_3)
\end{align*}
\]

[If-Then-Else]

Case

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

\[
\begin{align*}
O e_0 : T_0 \\
O[T_1/X_1] e_1 : T_1' \\
\vdots \\
O[T_n/X_n] e_n : T_n' \\
O \text{ case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \cdots; x_n : T_n \Rightarrow e_n; \text{ esac} : \text{lub}(T_1', \cdots, T_n')
\end{align*}
\]
Method Dispatch

- There is a problem with type checking method calls:
  \[ O, e_0 : T_0 \]
  \[ O, e_1 : T_1 \]
  \[ \vdots \]
  \[ O, e_n : T_n \]
  \[ O, e_0.f(e_1, \ldots, e_n) : ? \]

- 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 \( \text{foo} \) and an object \( \text{foo} \) can coexist in the same scope

- In the type rules, this is reflected by a separate mapping \( M \) for method signatures

\[ M(C, f) = (T_1, \ldots, T_n, T_{n+1}) \]

means in class \( C \) there is a method \( f \)

\[ f(x_1 : T_1, \ldots, x_n : T_n) : T_{n+1} \]

The Dispatch Rule Revisited

\[ O, M e_0 : T_0 \]
\[ O, M e_1 : T_1 \]
\[ \vdots \]
\[ O, M e_n : T_n \]
\[ M(T_n, f) = (T'_1, \ldots, T'_n, T'_n) \]
\[ T_i \leq T'_i \text{ for } 1 \leq i \leq n \]
\[ O, M e_0.f(e_1, \ldots, e_n) : T_{n+1} \]

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

Static Dispatch (Cont.)

\[ O, M e_0 : T_0 \]
\[ O, M e_1 : T_1 \]
\[ \vdots \]
\[ O, M e_n : T_n \]
\[ T_0 \leq T \]
\[ M(T, f) = (T'_1, \ldots, T'_n, T'_n) \]
\[ T_i \leq T'_i \text{ for } 1 \leq i \leq n \]
\[ O, M e_0@T.f(e_1, \ldots, e_n) : T_{n+1} \]

The Method Environment

- The method environment must be added to all rules

- In most cases, \( M \) is passed down but not actually used
  - Only the dispatch rules use \( M \)

\[ O, M e_1 : \text{int} \]
\[ \vdots \]
\[ O, M e_2 : \text{int} \]
\[ O, M e_1 + e_2 : \text{int} \]
More Environments

- For some cases involving SELF_TYPE, we need to know the class in which an expression appears.
- The full type environment for COOL:
  - A mapping O giving types to object id's
  - A mapping M giving types to methods
  - The current class C

Sentences

The form of a sentence in the logic is

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

Example:

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

Type Systems

- The rules in this lecture are COOL specific
  - More info on rules for self next time
  - 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
- Warning: Type rules are very compact!

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

Implementing Type Systems

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

\[
\text{TypeCheck}(\text{Environment}, e_1 + e_2) = \{
\text{T}_1 = \text{TypeCheck}(\text{Environment}, e_1);
\text{T}_2 = \text{TypeCheck}(\text{Environment}, e_2);
\text{Check } \text{T}_1 == \text{T}_2 == \text{Int};
\text{return Int;}\}
\]

Soundness: Java Digression
Kinds of Conversion

- Identity Conversions
- String Conversions
- Widening Primitive Conversions (19)
  - byte ⇒ short ⇒ int ⇒ long ⇒ float ⇒ double
  - char ⇒ int ⇒ long ⇒ float ⇒ double
  - int ⇒ float may lose precision
- Narrowing Primitive Conversions (19+4)
  - char ⇒ short, char ⇒ byte,
  - byte ⇒ char, short ⇒ char
  - may lose sign and magnitude information

- Widening Reference Conversions
  - null type ⇒ any reference type
  - any reference type ⇒ class Object
  - class S ⇒ class T, if S extends T
  - class S ⇒ interface K, if S implements K
  - interface J ⇒ interface K, if J extends K
  - array SC [] ⇒ array TC [], if SC ⇒ TC, and both SC and TC are reference types.
  - array T ⇒ interface Cloneable
  - array T ⇒ interface Serializable

Motivation for Primitive Array Type Constraints

```java
int i = 10;
long l = i;
byte b = (byte) i;
int[] iA = new int[5];
long[] iA = (long[]) iA; //error
byte[] bA = (byte[]) iA; // error
b = bA[2]; // reason
```

Motivation for Reference Array Type Constraints

```java
Point i = new Point();
Object i = i;
ColoredPoint b = (ColoredPoint) i;
// throws exception
Point[] iA = new Point[5];
Object[] iA = (Object[]) iA;
iA[2] = l; // no exception
b = iA[2]; // *error*
ColoredPoint[] bA = (ColoredPoint[]) iA;
// throws exception
```
### Another Example: Primitive Array Type

```java
void f(int[] a) {
    int i = a[0];
    a[0] = i;
}
long[] la = new long[8];
short[] sa = new short[8];
f(la);  // banned: compile-time error
f(sa);  // banned: compile-time error
```

### Another Example: Reference Array Type

```java
void f(Point[] pa) {
    pa[0] = new Point();
}
Object[] oa = new Object[8];
f(oa);   // compile-time error
f(new Point[] oa);  // throws ClassCastException
ColoredPoint[] cpa = new ColoredPoint[8];
f(cpa);  // array store exception in f
```

### Class S ⇒ Interface K

```
class S;
interface K;
class C extends S
    implements K;
C c = new C();
S s = c;
K k = (C) s;
s = (C) k;
```