

## Type Checking in Cool I

Adapted from Lectures by  
Prof. Alex Aiken and George Necula (UCB)

## 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  $E_1$  and  $E_2$  have certain types, then  $E_3$  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  $\wedge$  is "and"
  - Symbol  $\Rightarrow$  is "if-then"
  - $x:T$  is " $x$  has type  $T$ "

## From English to an Inference Rule (2)

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

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

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

## From English to an Inference Rule (3)

---

The statement

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

is a special case of

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

This is an inference rule

## Notation for Inference Rules

---

- By tradition inference rules are written

$$\frac{\square \text{Hypothesis}_1 \dots \square \text{Hypothesis}_n}{\square \text{Conclusion}}$$

- Cool type rules have hypotheses and conclusions

$$\square e:T$$

- $\square$  means "it is provable that ..."

## Two Rules

---

$$\frac{i \text{ is an integer}}{\square i : \text{int}} \quad [\text{Int}]$$

$$\frac{\square e_1 : \text{int} \quad \square e_2 : \text{int}}{\square e_1 + e_2 : \text{int}} \quad [\text{Add}]$$

## Two Rules (Cont.)

---

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions

$$\frac{\frac{1 \text{ is an integer}}{\square 1 : \text{Int}} \quad \frac{2 \text{ is an integer}}{\square 2 : \text{Int}}}{\square 1+2 : \text{Int}}$$

## Soundness

---

- A type system is *sound* if
  - Whenever  $\square e:T$
  - Then  $e$  evaluates to a value of type  $T$
- We only want sound rules.

$$\frac{i \text{ is an integer}}{\square i : \text{Object}}$$

## 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

---

$\frac{}{\square \text{ false: Bool}}$  [Bool]

$\frac{s \text{ is a string constant}}{\square s: \text{String}}$  [String]

## Rule for New

---

$\text{new } T$  produces an object of type  $T$   
- Ignore  $\text{SELF\_TYPE}$  for now ...

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

## Two More Rules

---

$\frac{}{\square e: \text{Bool}}$   
 $\frac{}{\square \neg e: \text{Bool}}$  [Not]

$\frac{}{\square e_1: \text{Bool}}$   
 $\frac{}{\square e_2: T}$   
 $\frac{}{\square \text{ while } e_1 \text{ loop } e_2 \text{ pool: Object}}$  [Loop]

## A Problem

---

- What is the type of a variable reference?

$\frac{x \text{ is an identifier}}{\square x: ?}$  [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

$O \square e: T$

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:

$$\frac{i \text{ is an integer}}{O \sqcap i: \text{int}} \quad [\text{Int}]$$

$$O \sqcap e_1: \text{int}$$

$$\frac{O \sqcap e_2: \text{int}}{O \sqcap e_1 + e_2: \text{int}} \quad [\text{Add}]$$

## New Rules

And we can write new rules:

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

## Let

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

$O[T/y]$  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:

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

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

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

## Assignment

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

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

## 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

$$\frac{\begin{array}{l} O_C(\text{Id}) = T_0 \\ O_C \sqsubseteq e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C \sqsubseteq \text{Id} : T_0 \leftarrow e_1} \quad [\text{Attr-Init}]$$

## 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 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
  - $X \leq Z \wedge Y \leq Z$   
 $Z$  is an upper bound
  - $X \leq Z' \wedge Y \leq Z' \Rightarrow Z \leq Z'$   
 $Z$  is least among upper bounds
- 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.

$$\frac{\begin{array}{l} O \sqsubseteq e_1 : \text{Bool} \\ O \sqsubseteq e_2 : T_2 \\ O \sqsubseteq e_3 : T_3 \end{array}}{O \sqsubseteq \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : \text{lub}(T_2, T_3)} \quad [\text{If-Then-Else}]$$

## Case

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

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

## Method Dispatch

- There is a problem with type checking method calls:

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

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

## Notes on Dispatch

- In Cool, method and object identifiers live in different name spaces
  - A method `foo` and an object `foo` can coexist in the same scope
- In the type rules, this is reflected by a separate mapping  $M$  for method signatures
 
$$M(C, f) = (T_1, \dots, T_n, T_{n+1})$$
 means in class  $C$  there is a method  $f$ 

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

## The Dispatch Rule Revisited

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

## 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.)

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

## 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  $M$

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

## 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

$$O, M, C \sqsubseteq e : T$$

Example:

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

## 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

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

```
TypeCheck(Environment, e1 + e2) = {  
  T1 = TypeCheck(Environment, e1);  
  T2 = TypeCheck(Environment, e2);  
  Check T1 == T2 == Int;  
  return Int; }
```

Soundness : Java Digression

## Kinds of Conversion

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

## • Widening Reference Conversions

- `null` type  $\Rightarrow$  any reference type
- any reference type  $\Rightarrow$  `class Object`
- `class S`  $\Rightarrow$  `class T`, if `S extends T`
- `class S`  $\Rightarrow$  `interface K`, if `S implements K`
- `interface J`  $\Rightarrow$  `interface K`, if `J extends K`
  
- `array SC []`  $\Rightarrow$  `array TC []`, if `SC`  $\Rightarrow$  `TC`, and both `SC` and `TC` are reference types.
  
- `array T`  $\Rightarrow$  `interface Cloneable`
  
- `array T`  $\Rightarrow$  `interface Serializable`

## • Narrowing Reference Conversions

- Permitted among reference types that can *potentially share common instances*.
- These conversions require *run-time test* to determine if the actual reference is a legitimate value of the new type.
  - Observe the interpretation of *final class*.
  - Observe that every *non-final class* can potentially be *extended to implement an interface*.

## Narrowing Reference Conversions

- `class Object`  $\Rightarrow$  any reference type
- `class S`  $\Rightarrow$  `class T`, if `T extends S`
- `class S`  $\Rightarrow$  `interface K`, if `S` is not *final* and `S` does not *implement K*
- `interface K`  $\Rightarrow$  `class T`, if `T` is not *final*
- `interface K`  $\Rightarrow$  `class T`, if `T` is *final* and `T implements K`
- `interface J`  $\Rightarrow$  `interface K`, if `K` does not *extend J* and there is no common method with same signature but different return types.
  
- `array SC []`  $\Rightarrow$  `array TC []`, if `SC`  $\Rightarrow$  `TC`, and both `SC` and `TC` are reference types.

## Motivation for Primitive Array Type Constraints

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

## Motivation for Reference Array Type Constraints

```
Point i = new Point();
Object l = i;
ColoredPoint b = (ColoredPoint) i;
// throws exception
Point[] iA = new Point[5];
Object[] lA = (Object[]) iA;
lA[2] = l; // no exception
b = lA[2]; // *error*
l = lA[2];
ColoredPoint[] bA = (ColoredPoint[]) iA;
// throws exception
```

### Another Example: Primitive Array Type

```
void f(int[] a) {
    int i = a[0];
    a[0] = i;
}
```

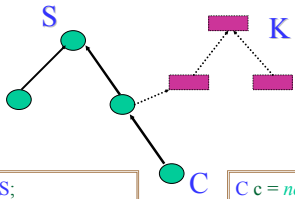
```
long[] la = new long[8];
short[] sa = new short[8];
f(la); // banned : compile-time error
f(sa); // banned : compile-time error
```

### Another Example: Reference Array Type

```
void f(Point[] pa) {
    pa[0] = new Point();
}
```

```
Object[] oa = new Object[8];
f(oa); // compile-time error
f((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;
```