Type Checking in COOL (II)

Lecture 10

Lecture Outline

• Type systems and their expressiveness
• Type checking with SELF_TYPE in COOL
• Error recovery in semantic analysis

Expressiveness of Static Type Systems

• Static type systems detect common errors
• But 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 more complex

Dynamic And Static Types

• The dynamic type of an object is the class $C$ that is used in the “new $C$” expression that created it
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type
• The static type of an expression captures all dynamic types the expression could have
  - 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

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

• A variable of static type $A$ can hold values of static type $B$, if $B \preceq A$

Here, $x$'s value has dynamic type $A$

Here, $x$'s value has dynamic type $B$
Dynamic and Static Types

Soundness theorem for the Cool type system:
\[ \forall E. \ dynamic\_type(E) \leq \ static\_type(E) \]

Why is this Ok?
- All operations that can be used on an object of type \( C \) can also be used on an object of type \( C' \leq C \)
  - Such as fetching the value of an attribute
  - Or invoking a method on the object
- Subclasses only add attributes or methods
- Methods can be redefined but with same type!

An Example

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

• Class Count incorporates a counter
• The inc method works for any subclass
• But there is disaster lurking in the type system

An Example (Cont.)

```java
class Stock inherits Count {
    name : String;  -- name of item
};
```

• Consider a subclass Stock of Count
• And the following use of Stock:
```java
class Main {
    Stock a ← (new Stock).inc ();
    ... a.name ...
};
```

• But this is not well-typed
  - (new Stock).inc() has static type Count
• The type checker “loses” type information
  - This makes inheriting inc useless
  - So, we must redefine inc for each of the subclasses, with a specialized return type

What Went Wrong?

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

SELF_TYPE to the Rescue

• We will extend the type system

  Insight:
  - inc returns “self”
  - Therefore the return value has same type as “self”
  - Which could be Count or any subtype of Count!

  - Introduce the keyword SELF_TYPE to use for the return value of such functions
  - We will also need to modify the typing rules to handle SELF_TYPE

SELF_TYPE to the Rescue (Cont.)

• SELF_TYPE allows the return type of inc to change when inc is inherited
• Modify the declaration of inc to read
  ```java
  inc() : SELF_TYPE {
  ```
• The type checker can now prove:
  ```java
  C,M ⊢ (new Count).inc() : Count
  C,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

SELF_TYPE and Dynamic Types (Example)

- What can be the dynamic type of the object returned by inc?
  - Answer: whatever could be the type of “self”
    ```
    class A inherits Count { };
    class B inherits Count { };
    class C inherits Count { };
    (inc could be invoked through any of these classes)
    ```
  - Answer: Count or any subtype of Count

Type Checking

- 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
  - \( T_1 \leq T_2 \) if \( T_1 \) 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

Extending \( \leq \)

Let \( T \) and \( T' \) be any types but SELF_TYPE
There are four cases in the definition of \( \leq \)

1. \( \text{SELF_TYPE}_C \leq \text{SELF_TYPE}_C \)
   - In Cool we never need to compare SELF_TYPEs coming from different classes
2. \( \text{SELF_TYPE}_C \leq T \) if \( C \leq T \)
   - \( \text{SELF_TYPE}_C \) can be any subtype of \( C \)
   - This includes \( C \) itself
   - Thus this is the most flexible rule we can allow
Extending ≤ (Cont.)

3. \( T \leq \text{SELF\_TYPE} \) always false
   Note: \( \text{SELF\_TYPE} \) can denote any subtype of \( C \).

4. \( T \leq T' \) (according to the rules from before)

Based on these rules we can extend \( \text{lub} \) ...

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

Let \( T \) and \( T' \) be any types but \( \text{SELF\_TYPE} \)
Again there are four cases:
1. \( \text{lub} (\text{SELF\_TYPE} \_ C, \text{SELF\_TYPE} \_ C) = \text{SELF\_TYPE} \_ C \)
2. \( \text{lub} (\text{SELF\_TYPE} \_ C, T) = \text{lub} (C, T) \)
   This is the best we can do because \( \text{SELF\_TYPE} \_ C \leq C \)
3. \( \text{lub} (T, \text{SELF\_TYPE} \_ C) = \text{lub} (C, T) \)
4. \( \text{lub} (T, T') \) defined as before

Where Can \( \text{SELF\_TYPE} \) Appear in COOL?

- The parser checks that \( \text{SELF\_TYPE} \) appears only where a type is expected
- But \( \text{SELF\_TYPE} \) is not allowed everywhere a type can appear:
  - class \( T \) inherits \( T' \) { … }
    - \( T, T' \) cannot be \( \text{SELF\_TYPE} \)
  - \( x : T \)
    - \( T \) can be \( \text{SELF\_TYPE} \)
    - An attribute whose type is \( \leq \text{SELF\_TYPE} \_ C \)

Where Can \( \text{SELF\_TYPE} \) Not Appear in COOL?

- \( m(x : T) : T' \) { … }
  - Only \( T' \) can be \( \text{SELF\_TYPE} \)

What could go wrong if \( T \) were \( \text{SELF\_TYPE} \)?

- class \( A \) { comp(x : \text{SELF\_TYPE}) : \text{Bool} { … } ; }
- class \( B \) inherits \( A \) { 
  \( b : \text{int} ; \)
  comp(x : \text{SELF\_TYPE}) : \text{Bool} { … x.b … } ; }
- …
- let \( x : A \leftarrow \text{new} B \) in … x.comp(new A) ; …

Typing Rules for \( \text{SELF\_TYPE} \)

- Since occurrences of \( \text{SELF\_TYPE} \) depend on the enclosing class we need to carry more context during type checking
- New form of the typing judgment:
  \( O, M, C \vdash e : T \)
  (An expression \( e \) occurring in the body of \( C \) has static type \( T \) given a variable type environment \( O \) and method signatures \( M \) )
Type Checking Rules

• The next step is to design type rules using `SELF_TYPE` for each language construct.

• Most of the rules remain the same except that `≤` and `lub` are the new ones.

• Example:

  \[
  \begin{align*}
  T_0 & \triangleq \text{Id} \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{O,M,C} \vdash \text{e}_0 : T_0 \\
  \text{O,M,C} \vdash \text{e}_1 : T_1 \\
  \text{O,M,C} \vdash \text{Id} \leftarrow \text{e}_1 : T_1 \\
  \end{align*}
  \]

What’s Different?

• Recall the old rule for dispatch

  \[
  \begin{align*}
  \text{O,M,C} \vdash \text{e}_0 : T_0 \\
  \vdots \\
  \text{O,M,C} \vdash \text{e}_n : T_n \\
  \text{M}(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \\
  T_{n+1}' = \text{SELF_TYPE} \\
  T_i \leq T_i' \quad 1 \leq i \leq n \\
  \text{O,M,C} \vdash \text{e}_0.f(\text{e}_1, \ldots, \text{e}_n) : T_{n+1} \\
  \end{align*}
  \]

What’s Different?

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

  \[
  \begin{align*}
  \text{O,M,C} \vdash \text{e}_0 : T_0 \\
  \vdots \\
  \text{O,M,C} \vdash \text{e}_n : T_n \\
  \text{M}(T_0, f) = (T_1', \ldots, T_n', \text{SELF_TYPE}) \\
  T_i \leq T_i' \quad 1 \leq i \leq n \\
  \text{O,M,C} \vdash \text{e}_0.f(\text{e}_1, \ldots, \text{e}_n) : T_0 \\
  \end{align*}
  \]

Static Dispatch

• Recall the original rule for static dispatch

  \[
  \begin{align*}
  \text{O,M,C} \vdash \text{e}_0 : T_0 \\
  \vdots \\
  \text{O,M,C} \vdash \text{e}_n : T_n \\
  T_0 \leq T \\
  \text{M}(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\
  T_{n+1}' = \text{SELF_TYPE} \\
  T_i \leq T_i' \quad 1 \leq i \leq n \\
  \text{O,M,C} \vdash \text{e}_0@T.f(\text{e}_1, \ldots, \text{e}_n) : T_{n+1} \\
  \end{align*}
  \]

Static Dispatch

• If the return type of the method is `SELF_TYPE` we have:

  \[
  \begin{align*}
  \text{O,M,C} \vdash \text{e}_0 : T_0 \\
  \vdots \\
  \text{O,M,C} \vdash \text{e}_n : T_n \\
  T_0 \leq T \\
  \text{M}(T, f) = (T_1', \ldots, T_n', \text{SELF_TYPE}) \\
  T_i \leq T_i' \quad 1 \leq i \leq n \\
  \text{O,M,C} \vdash \text{e}_0@T.f(\text{e}_1, \ldots, \text{e}_n) : T_0 \\
  \end{align*}
  \]
Static Dispatch

• Why is this rule correct?

• If we dispatch a method returning \( \text{SELF\_TYPE} \) in class \( T \), don’t we get back a \( T \)?

• No. \( \text{SELF\_TYPE} \) is the type of the self parameter, which may be a subtype of the class in which the method appears.

New Rules

• There are two new rules using \( \text{SELF\_TYPE} \)

\[
O, M, C \vdash \text{self} : \text{SELF\_TYPE}_C
\]

\[
O, M, C \vdash \text{new SELF\_TYPE} : \text{SELF\_TYPE}_C
\]

• There are a number of other places where \( \text{SELF\_TYPE} \) is used.

Summary of \( \text{SELF\_TYPE} \)

• The extended \( = \) and \( \sqcup \) operations can do a lot of the work.

• \( \text{SELF\_TYPE} \) can be used only in a few places. Be sure it isn’t used anywhere else.

• A use of \( \text{SELF\_TYPE} \) always refers to any subtype of the current class.
  - The exception is the type checking of dispatch. The method return type of \( \text{SELF\_TYPE} \) might have nothing to do with the current class.

Why Cover \( \text{SELF\_TYPE} \)?

• \( \text{SELF\_TYPE} \) is a research idea
  - It adds more expressiveness to the type system.

• \( \text{SELF\_TYPE} \) is itself not so important
  - except for the project.

• Rather, \( \text{SELF\_TYPE} \) 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.

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 \( \text{Object} \) to ill-typed expressions

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

• Since \( x \) is undeclared its type is \( \text{Object} \).

• But now we have \( \text{Object} + \text{Int} \).

• This will generate another typing error.

• We then say that that \( \text{Object} + \text{Int} = \text{Object} \).

• Then the initializer’s type will not be \( \text{Int} \).

⇒ 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 ≤ 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 ← 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
- The `Object` solution is fine in the class project