Type Checking in COOL (II)

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

Lecture Outline

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

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

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 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 corresponded 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?

- All operations that can be used on an object of type *C* can also be used on an object of type D $\leq$ *C*
  - Such as fetching the value of an attribute
  - Or invoking a method on the object
- Subclasses can <u>only add</u> attributes or methods
- Methods can be redefined but with same type !

An Example

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

- Class Count incorporates a counter i
- The inc method works for any subclass
- What is the type of inc?
  - But there is disaster lurking in the type system

An Example (Cont.)

- Consider a subclass Stock of Count

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

- And the following use of Stock:

```
class Main {
  Stock a ← (new Stock).inc ();   Type checker error !
  ...  a.name ...
};
```


What went wrong?

- (new Stock).inc() has dynamic type Stock
- So it is legitimate to write

  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 !
    - Recall that method is type checked once, and is required to guarantee type correctness of all possible/potential invocations.
- Introduce the keyword SELF_TYPE to use for the return value of such functions, and modify 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

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

  C,*M* □ (new Count).inc() : Count

  C,*M* □ (new Stock).inc() : Stock
- The program from before is now well typed

Motivating Examples and Issues

Self

- self names the receiver object inside a method body. In expression languages such as Cool, self can be the "return" expression to conveniently compose a number of side-effect causing operations.
  - Cf. signatures of << and >> in C++

```
class C {      …
        f() : C { {… ; self;} };
        g() : C { {… ; self;} };
        h() : C { {… ; self;} };
};
(new C) . f() . g() . h() ;
```


Method inheritance

```
class C {          id() : C { self };    };
class D inherits C {  };
class Main {
        … D d <- (new D) . id();  …
};
```

- The type of the returned result of id() is C. This is too conservative – the static type checker bans this "reasonable" invocation of id() on a D-object.
- In a dynamically typed language, the type checking is done in the context of a specific use. In a statically typed language, the type checking is done just once for all potential uses.

SELF_TYPE

```
class C {          id() : SELF_TYPE { self };    };
class D inherits C {  };
class Main {
        … D d <- (new D) . id(); …
};
```

- An enhancement to the type language allows "natural" code reuse in a statically typed language.
- The semantics of SELF_TYPE depends on the class in which the method is inherited (actually, it stands for the type of the receiver object).

Unsound Typing

```
class C {
        id() : SELF_TYPE {  new C };
};
class D inherits C {  };
class Main {
      … D d <- (new D) . id(); …
};
```

- Allowing this program causes a type error at run-time. An instance of *super*-class C is assigned to a variable of *sub*-type D.
  - Cf. id() : C { self };

Legal uses of SELF_TYPE in Cool

```
class C {
        s : SELF_TYPE <- new SELF_TYPE;
        id() : SELF_TYPE {
            let
              x: SELF_TYPE <- self,
            in  x
        };
};
```

- Instead of freezing the type, the use of SELF_TYPE permits carrying around a *type equality constraint*.

Expressions with type SELF_TYPE

```
class C {
        f() : SELF_TYPE { self }
        g(C c1, C c2) : SELF_TYPE { self }
        h() : SELF_TYPE { g(new C, self) }
};
class D inherits C {
        i() : SELF_TYPE { g(new D, new D) }
};
```


Expressions in Java

```
class C {
        C f() { return this; }
        C g(C c1, C c2)  { return this; }
        C h() { return g(new C(), this); }
};
class D extends C {
        C g(D d1, D d2)  { return h(); }
        C i() { return g(new D(), new D()) }
};
```



```
class C {
      String g(C c1, C c2)  {  return "\t ---> g(C,C) run\n";   }
};
class D extends C {
      String g(D d1, D d2)  {  return "\t ---> g(D,D) run\n";      }
};
class ThatOverload {
   public static void main(String[] args) {
         C c = new C();      D d = new D();
         System.out.print("\t g(C,C) called" + d.g(c,c));
         System.out.print("\t g(D,C) called" + d.g(d,c));
         System.out.print("\t g(C,D) called" + d.g(c,d));
         System.out.print("\t g(D,D) called" + d.g(d,d));
   }
}
```


Java specifics

```
class C {
        int e(D d)  { return 0; }
        C j() {   return new C();  }
};
class D extends C {
        int k() {return this.e(new D());
                // return e(new D());
                // ambiguous in Java to coerce or to inherit
              // return super.e(new D()); // inherited
              // no obvious way to invoke local e
        }
        int e(C c)  { return 1 + super.e(this) ; }
};
```


Without SELF_TYPE

```
class C {
        id(s : C) : C { C x <- s };
};
class D inherits C {   };
class Main {
      … C c <- (new C) . id(new D); …
      … D d <- (new D) . id(new C); …
};
```

- A subclass instance can be coerced to a class instance.

Illegal Use of SELF_TYPE

```
class C {
     id(s : SELF_TYPE) : SELF_TYPE { SELF_TYPE x <- s };
};
class D inherits C {   };
class Main {
      … C c <- (new D);  …
      … c . id(new C); …
};
```

- If permitted, causes type violations at run-time as follows: D s <- new C; …; D x <- s;

Open-ended issues

Coercion and SELF_TYPE: *Illegal Cool*

```
class C {
    id(s, t : SELF_TYPE) : bool { s == t };
};
class D inherits C {  };
class Main {
    … C c <- (new C);  D d <- (new D); …
    … c . id(c, d); …
};
```

- If formals can have SELF_TYPE, should this type check?

Notes on extensions

- The use of SELF_TYPE can be extended (such as for a formal) in a strongly typed language by performing run-time type checks and throwing exceptions to signal errors.
  - Check that the run-time type of the actual argument is compatible with that of the receiver.
  - Cf. Java's approach to strong typing
- Why is SELF_TYPE not permitted in a case construct in Cool?

- Require that the actual arguments corresponding to the formal parameters of a method with type SELF_TYPE be of type SELF_TYPE.
  - Is this typing rule sound?
  - Is this extension natural and worthwhile?
- If the functions can return l-values, is the type system sound?
  - Check assignments

Interaction between Overloading and Overriding

To match the message with a method, C++ searches the ancestor classes of an object. The class that defines the message name is further searched to "match" the entire signature. Even if a better match is inheritable, it will not be considered. Thus C++ differs from Java.

```
class A {
 public:
  void test(double d){}
}
class B : public A {
 public:
   void test(int i) {}
}
```

Error:

```
  B b;  b.test(2.5);
```

Corrected:

```
 void B::test(double d)
       {  A::test(d); }
```


Back to Type system with SELF_TYPE

Notes About SELF_TYPE

- SELF_TYPE is a static type, not dynamic type
  - Cf. Eiffel's approach
- 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

SELF_TYPE and Dynamic Types (Example)

- In general, if SELF_TYPE appears textually in the class C as the declared type of E then

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

- 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
- This suggests a typing rule:

$$\text{SELF\_TYPE}_C \leq C \qquad (*)$$

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
  - $T_1 \leq T_2$ 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
  - Typically, type restrictions are motivated by assignment-like operations in the context of a class hierarchy.

Extending ≤

Let T and T' be any types but SELF_TYPE.

There are four cases in the definition of ≤

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 $C \leq T$
   - SELF_TYPE_C can be any subtype of C, including C
   - Thus this is the most flexible rule we can allow

Extending ≤ (Cont.)

3. $T \leq SELF\_TYPE_C$ is always false
   Note: $SELF\_TYPE_C$ can denote *any* subtype of C.

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

Based on these rules we can extend lub …

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 $SELF\_TYPE_C \leq C$
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. x : T
   - T can be SELF_TYPE
   - An attribute whose type is $\leq SELF\_TYPE_C$

Where can SELF_TYPE appear in COOL?

3. let x : T in E
   - T can be SELF_TYPE
   - x has a type $\leq SELF\_TYPE_C$
4. new T
   - T can be SELF_TYPE
   - Creates an object of the same type as self
5. $m@T(E_1,\ldots,E_n)$
   - T cannot be SELF_TYPE

Where SELF_TYPE cannot appear in COOL?

6. m(x : T) : T' { … }
   - Only T' can be SELF_TYPE !

What could go wrong if T were SELF_TYPE?

```
class A {  comp(x : SELF_TYPE) : Bool  {…};  };
class B inherits A {
    b : int;
    comp(x : SELF_TYPE) : Bool { … x.b …};  };
…
  let x : A ← new B in  … x.comp(new A); …
…
```


Typing Rules for SELF_TYPE

- Since occurrences of 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 □ 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 $\leq$ and lub are the new ones
- Example:

$$\frac{O(Id) = T_0 \quad O,M,C \vdash e_1 : T_1 \quad T_1 \leq T_0}{O,M,C \vdash Id \leftarrow e_1 : T_1}$$

What's Different?

- Recall the old rule for dispatch

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

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:

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

What's Different?

- Note that this rule handles the Stock example
- Formal parameters cannot be SELF_TYPE
- Actual arguments can be SELF_TYPE
  - The extended $\leq$ relation handles this case
- The type T_0 of the dispatch expression could be SELF_TYPE
  - Which class is used to find the declaration of f?
  - Answer: it is safe to use the class where the dispatch appears

Static Dispatch

- Recall the original rule for static dispatch

$$\frac{\begin{array}{l} O,M,C \vdash e_0 : T_0 \\ \vdots \\ O,M,C \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T_1',\ldots,T_n',T_{n+1}') \\ T_{n+1}' \neq \text{SELF\_TYPE} \\ T_i \leq T_i' \qquad 1 \leq i \leq n \end{array}}{O,M,C \vdash e_0@T.f(e_1,\ldots,e_n) : T_{n+1}'}$$

Static Dispatch

- If the return type of the method is SELF_TYPE we have:

$$\frac{\begin{array}{l} O,M,C \vdash e_0 : T_0 \\ \vdots \\ O,M,C \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T_1',\ldots,T_n',\text{SELF\_TYPE}) \\ T_i \leq T_i' \qquad 1 \leq i \leq n \end{array}}{O,M,C \vdash e_0@T.f(e_1,\ldots,e_n) : T_0}$$

Static Dispatch

- Why is this rule correct?
- If we dispatch a method returning SELF_TYPE in class T, don't we get back a T?
- No. SELF_TYPE is the type of the *self* parameter, which may be a subtype of the class in which the method appears
- The static dispatch class cannot be SELF_TYPE
  - That's dynamic dispatch!

New Rules

- There are two new rules using SELF_TYPE

$$\frac{}{O,M,C \vdash self : SELF\_TYPE_C}$$

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

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

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. The method return type of SELF_TYPE 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

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

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

- Since x is undeclared its type is Object
- But now we have Object + Int
- This will generate another typing error
- We then say that 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 ← x + 2  in  y + 3
```


Notes

- A "real" compiler would use something like No_type
  - Cf. $\perp$ in Denotational Semantics
- However, there are some implementation issues
  - The class hierarchy is not a tree anymore
- The Object solution is adequate for the project