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
  - "static" because no knowledge of dynamic behavior (run-time behavior) of program

- But some correct programs are disallowed
  - In fact, any static type system that does the correct thing is going to have to disallow some correctly typed programs, because it can't reason precisely at compile time about everything that could happen as the program runs.

Expressiveness of Static Type Systems

- Static type systems detect common errors
- But some correct programs are disallowed
  - So some argue for dynamic type checking instead
    - At run time we check whether operations being performed are appropriate for the actual data
  - Others argue for more expressive static type checking
- Been a good amount of work on both, with good progress
  - But more expressive type systems are more complex (so some ideas not yet in real languages)

Dynamic And Static Types

- The <u>dynamic type</u> of an object is the class *C* that is used in the "new *C*" expression that created it (may be different than declared type of object)
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type
  - dynamic type can vary during execution

- The <u>static type</u> of an expression captures all dynamic types the expression could have
  - A compile-time notion
  - static type is invariant

Dynamic and Static Types

- There must be some relationship between the static and dynamic types of an expression if the static type checker is to be correct

- This relationship can be formalized via some theorem such as…

Dynamic and Static Types. (Cont.)

- Soundness theorem: for all expressions E

  dynamic_type(E) = static_type(E)

  (in all executions, E evaluates to values of the type determined by the static type checker in the compiler)

- In early type systems the set of static types corresponded directly with the dynamic types
  - So we had situation above

- This gets more complicated in advanced type systems (such as for COOL)

Dynamic and Static Types in COOL

```
class A { … }
class B inherits A {…}
class Main {
   x:A ← 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 ≤ *A*

Dynamic and Static Types

Soundness theorem for the Cool type system:

$$\forall E.\quad \text{dynamic\_type}(E) \leq \text{static\_type}(E)$$

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 <u>only add</u> attributes or methods
- Methods can be redefined but with same type!

Bottom Line

- Type systems are becoming more expressive, but while progress has been made in both static and dynamic typing, many of the results of this progress have not yet found their way into current programming languages
  - Düg?

- And now, on to the much mentioned coverage of SELF_TYPE
  - We start with a motivating example…

An Example

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

- Class Count just increments a counter
- The inc method works for any subclass

An Example

- Count could be thought of as a base class that provides counter functionality…
- …In which case, whenever someone needs a counter for a specific purpose, they could subclass Count…
- Said subclass would automatically inherit the inc() method…
- So that they could have a counter without having to reimplement code
  - Of course here the code is small and simple, but you can imagine similar situation with more complex code

An Example

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

- Class Count just increments a counter
- The inc method works for any subclass
- 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
};
```

- E.g., Implementing a warehouse accounting program and want to keep track of number of number of items of different kinds that are in stock

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 ();
  …  a.name …
};
```

Type checking error!
Why?
But note that it would work fine!

What Went Wrong?

- (new Stock).inc() has dynamic type Stock
  - starts with a new Stock object, increments its i instance variable, and returns self (a Stock object)
- So it is legitimate to write

  ```
  Stock a ← (new Stock).inc ()
  ```

- But this is not well-typed
  - (new Stock).inc() has static type Count, which is correct, but not useful in this context. And since Count is not a subtype of Stock, type error!
- 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

- To solve this problem, 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 a new 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
  - i.e., allows us to reason about how the actual return type of inc() changes dynamically when inc() is inherited

- Modify the declaration of inc to read

  inc() : SELF_TYPE { … }

- The type checker can now prove:

  $C,M \vdash$ (new Count).inc() : Count
  $C,M \vdash$ (new Stock).inc() : Stock

- The program from before is now well typed
  - Remember what self is: the type of the expression on which the method was dispatched

Notes About SELF_TYPE

- SELF_TYPE is not a dynamic type (nor is it a class name – the only type that is not a class)
  - 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 dynamic 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 in the code of class C as the declared type of some expression E then

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

  Why? Because SELF_TYPE is the type of self, which is defined to be the type of the object on which a method is dispatched. And any object that is "dispatching" a method of class C must either be of class C or a subclass of C

SELF_TYPE and Dynamic Types (Example)

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

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

- Significant: The meaning of SELF_TYPE depends on the context (where it appears)
  - We write SELF_TYPE_C to refer to an occurrence of SELF_TYPE in the body of C
    - This is done to remind ourselves what class we're talking about
- This suggests a simple typing rule:

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

SELF_TYPE and Dynamic Types (Example)

- So: $\text{SELF\_TYPE}_C \leq C$ (*)

This rule, while simple, and somewhat obvious, is also important, because it gives us some idea of what SELF_TYPE really is: The best way to think of SELF_TYPE is as a type variable (i.e., a variable whose value is a type) that ranges over all the subclasses of the class in which it appears.

Because it's a variable, it doesn't have a fixed type, but is guaranteed to be some type bounded by C -- it is some type that inherits directly or indirectly from C

Type Checking

- Rule (*) has an important consequence:
  - In type checking it is always safe to replace SELF_TYPE_C by C
    - Think about why this is: when a method returns self, then since self "is a" C, changing the return type to C doesn't violate type rules (you are returning a C)

- This suggests one way to handle SELF_TYPE :
  - Replace all occurrences of SELF_TYPE_C by C

- This would be correct but not very useful – it's like not having SELF_TYPE at all

Operations on SELF_TYPE

- So to do better than just throwing away all the SELF_TYPEs, we need to incorporate them into the type system.

- Recall the only two 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

Extending ≤

Let T and T' be any types except SELF_TYPE

There are four cases in the definition of ≤

1. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
   - That this rule is true is easily seen by considering the notion of SELF_TYPE as a variable. So, SELF_TYPE can be any subtype of C. But just like in 8th grade algebra, once you give a value to a variable, you have to give that same value to ALL occurrences of that variable. So here, if SELF_TYPE has value C', a subtype of C, then what this rule is asserting is that $C' \leq C'$

Extending ≤

Let T and T' be any types except SELF_TYPE
There are four cases in the definition of ≤

1. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
   - It's also reasonable to ask what happens if you are comparing SELF_TYPES from different classes, say C and D.
   - Answer: In Cool we never need to compare SELF_TYPEs coming from different classes (so no need to have, say, C and D here). Though we haven't shown it yet, the type rules of COOL are written in such a way that this just will never happen.

Extending $\leq$

Let T and T' be any types except SELF_TYPE

There are four cases in the definition of $\leq$

1. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$

2. $\text{SELF\_TYPE}_C \leq T$ if $C \leq T$
   - SELF_TYPE_C can be any subtype of C
   - This includes C itself
   - Thus this is the most flexible rule we can allow

Extending $\leq$ (Cont.)

3. $T \leq \text{SELF\_TYPE}_C$ always false

   That is, a regular class type is NEVER a subtype of SELF_TYPE_C

   To see this, think about the possibilities: where can C and T be in the type hierarchy?

   It T and C are unrelated (both inherit from Object but otherwise have nothing to do with each other), then clearly T can't be a subtype of SELF_TYPE_C

Extending ≤ (Cont.)

3. $T \leq \text{SELF\_TYPE}_C$ always false

   That is, a regular class type is NEVER a subtype of SELF_TYPE_C

   To see this, think about the possibilities: where can C and T be in the type hierarchy?

   If they are related, then you might think that if T is a subtype of C, this could work out. But we can't allow it even in this case: Think about a hierarchy where T has some strict subtype A. The because SELF_TYPE_C ranges over all subtypes of C, SELF_TYPE_C could assume value A, in which case relationship above is in wrong order. Since it can't work for all possible subtypes of C, have to have it be false.

Extending $\leq$ (Cont.)

3. $T \leq SELF\_TYPE_C$ always false
   - That is, a regular class type is NEVER a subtype of $SELF\_TYPE_C$
   - To see this, think about the possibilities: where can C and T be in the type hierarchy?
   - But, there is one very special case where one could argue that we should allow this to be true: the case where T is the only leaf of the descendants of C (T is the *unique minimal type*) in the class hierarchy, in which case T truly would be a subtype of all possible values of $SELF\_TYPE_C$
   - (For this to happen need type hierarchy below C to be a single chain down to T)

Extending ≤ (Cont.)

3. $T \le SELF\_TYPE_C$ always false

That is, a regular class type is NEVER a subtype of $SELF\_TYPE_C$

To see this, think about the possibilities: where can C and T be in the type hierarchy?

The problem with this special case is that it is extremely fragile: a programmer might come along and add a subclass of C that is unrelated to T (not an ancestor of T) and this would no longer work. In this case you would suddenly be getting type errors in code that previously type checked fine, and hadn't been changed at all. Not a good language design. So no regular class type is ever a subtype of $SELF\_TYPE_C$

Extending ≤ (Cont.)

4. T ≤ T' (just use rules from before we added SELF_TYPE, since neither type here involves SELF_TYPE)

Based on these rules we can extend lub …

Summary $\leq$

Let T and T' be any types except SELF_TYPE
There are four cases in the definition of $\leq$

1. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
2. $\text{SELF\_TYPE}_C \leq T$ if $C \leq T$
3. $T \leq \text{SELF\_TYPE}_C$ always false
4. $T \leq T'$

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 this is too permissive

- But SELF_TYPE is not allowed everywhere a type can appear:

1. class T inherits T' {…}
   - T, T' cannot be SELF_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 dynamic type as self

5. $m@T(E_1,\ldots,E_n)$
   - T cannot be SELF_TYPE
     - Because in dynamic dispatch T must be an actual class name

Where Can SELF_TYPE Not Appear in COOL?

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

What could go wrong if T were SELF_TYPE?

Consider the method call e.m(e'), where e' has type T_0. According to our rule for method types, T_0 must be a subtype of the type of the formal parameter. If, however, the formal parameter has type SELF_TYPE, then must have $T_0 \leq$ SELF_TYPE, which is a no-no

Where Can SELF_TYPE Not Appear in COOL?

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

What else could go wrong if T were SELF_TYPE?

Where Can SELF_TYPE Not Appear in COOL?

presumably compares input parameter with self

```
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); …
…
```

static type A
dynamic type B
the key here!

note use of attribute b in overriding method

Where Can SELF_TYPE Not Appear in COOL?

presumably compares input parameter with self

```
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); …
…
```

statically type checks fine: x has static type A, and in code for A, SELF_TYPE is A, so passing in a new A is fine. But dynamic type of x is B, so when x.comp() call is executed, it is executing the b version method, but with an object of dynamic type A! Result is a runtime crash!

Another Extension

- So now, let’s extend our type rules by integrating the rules for SELF_TYPE.

Typing Rules for SELF_TYPE

- Since occurrences of SELF_TYPE depend on the enclosing class we need to carry more context during type checking (i.e., another environment)
  - We need to always know the class to which the rule is applying, which is why there is a C below
- 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 look the same except that the augmented $\leq$ and lub are used
- Example:

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

What's Different?

- Some rules do have to change: Recall the old rule for dispatch

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

implicit in our previous version of this rule, but made explicit here

But being able to return SELF_TYPE is exactly where the use of SELF_TYPE buys us something!

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:

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

note!

What's Different?

- Note 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

What's Different?

- 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

$$\frac{O,M,C \vdash e_0 : \text{SELF\_TYPE}_C \qquad M(C, f) = (\ldots)}{O,M,C \vdash e_0.f(e_1)}$$

because e_0 occurs in class C, use C in the lookup of type signature

Static Dispatch

- Recall the original rule for static dispatch

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

Static Dispatch

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

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

note return type is T_0, not T. Why?

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
  - SELF_TYPE is the type of the self parameter. So even though we are dispatching to a method of class T, the self parameter still has type To.

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

Note these are rules where we need to know the class *C*

Summary of SELF_TYPE

- The extended ≤ and lub operations can do a lot of the work.
- 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 of 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 itself is 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

Error Recovery

- As with parsing, it is important to recover from type errors
- Detecting where errors occur is easier than in parsing
  - Because we already have the AST
  - So there is no reason to skip over portions of code
- The Problem:
  - What type is assigned to an expression with no legitimate type?
    - Type checker works by structural induction, and it can't just get stuck
    - Need to assign some type to something like this, because…
  - This type will influence the typing of the enclosing expression

Error Recovery Attempt

assume bug in code and x is undefined

- Assign type Object to ill-typed expressions

let y : Int ← x + 2 in y + 3

- We'll walk down the AST. When we get to the leaf for x, we'll see that x is undefined, which will generate an error message
- In order to recover, since x is undeclared we'll assume its type is Object
- But now we move up the AST, and attempt to type the + operation, in which we have Object + Int

Error Recovery Attempt

assume bug in code and x is undefined

- Assign type Object to ill-typed expressions

  let y : Int ← x + 2 in y + 3

  - This will generate another typing error, something like "+ applied to an object"
  - Since we can't type Object + Int, our recovery strategy says we give it type Object, so Object + Int = Object

Error Recovery Attempt

- But the next AST node will be the initialization, in which we are assigning something of type Object to a variable declared as type Int, which will generate yet another type error.
- Bottom line: this strategy is workable (we don't get stuck), but a single error can lead to a whole cascade of errors
- Part of the reason for cascade: very few operations defined for type Object

Better Error Recovery

- We can introduce a new type called No_type for use with ill-typed expressions
  - Not available to the programmer – just there for the use of the compiler

- Special property: Define No_type ≤ *C* for all types *C*
  - Subtype of every type
  - Note this is opposite of Object

- Every operation is defined for No_type
  - And all return a No_type result

- Only one typing error ("x is undefined") 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 (it's a DAG with No_type at the bottom)
    - So tree algorithms can no longer be applied
- The Object solution is fine in the class project
  - Because the above issue is just too much of a hassle to deal with at our level
  - We just live with the cascading errors