

Type Checking in COOL (II)

Lecture 10

Instructor: Fredrik Kjolstad

Slide design by Prof. Alex Aiken, with modifications

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

x has static type A

```
class A { ... }  
class B inherits A {...}  
class Main {  
   $x:A \leftarrow \text{new } A;$  ← Here,  $x$ 's value has dynamic type  $A$   
  ...  
   $x \leftarrow \text{new } B;$  ← 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:

$$\forall E. \text{dynamic_type}(E) \leq \text{static_type}(E)$$

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 only add 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
- 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  
};
```

- And the following use of **Stock**:

```
class Main {  
    Stock a ← (new Stock).inc ();  Type checking 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`!
- 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
`inc() : SELF_TYPE { ... }`
- The type checker can now prove:
 $C, M \vdash (\text{new Count}).\text{inc}() : \text{Count}$
 $C, M \vdash (\text{new Stock}).\text{inc}() : \text{Stock}$
- 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`

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:

$$\text{SELF_TYPE}_C \leq C \quad (*)$$

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**

Extending \leq

Let T and T' be any types but $SELF_TYPE$

There are four cases in the definition of \leq

1. $SELF_TYPE_C \leq SELF_TYPE_C$

- In Cool we never need to compare $SELF_TYPE$ s coming from different classes

2. $SELF_TYPE_C \leq T$ 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

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

Let T and T' be any types but 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 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
 2. x : T
 - T can be SELF_TYPE
 - An attribute whose type is $\leq \text{SELF_TYPE}_c$

Where Can SELF_TYPE Appear in COOL?

3. `let x : T in E`

- T can be SELF_TYPE
- x has a type $\leq \text{SELF_TYPE}_c$

4. `new T`

- T can be SELF_TYPE
- Creates an object of the same type as `self`

5. `m@T(E1, ..., En)`

- T cannot be SELF_TYPE

Where Can SELF_TYPE Not Appear in COOL?

6. $m(x : T) : T' \{ \dots \}$

- 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 include that context during type checking

- Recall the form of a 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 \leq and **lub** are the new ones
- Example:

$$O(\text{Id}) = T_0$$

$$O, M, C \vdash e_1 : T_0$$

$$T_1 \leq T_0$$

$$O, M, C \vdash \text{Id} \leftarrow e_1 : T_1$$

What's Different?

- Recall the old rule for dispatch

$$\begin{array}{c} O, M, C \vdash e_0 : T_0 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ M(T_0, f) = (T_1', \dots, T_n', T_{n+1}') \\ T_{n+1}' \neq \text{SELF_TYPE} \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ \hline O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_{n+1}' \end{array}$$

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:

$$\begin{array}{c} O, M, C \vdash e_0 : T_0 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ M(T_0, f) = (T_1', \dots, T_n', \text{SELF_TYPE}) \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ \hline O, M, C \vdash e_0.f(e_1, \dots, e_n) : T_0 \end{array}$$

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

Static Dispatch

- Recall the original rule for static dispatch

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

Static Dispatch

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

$$O, M, C \vdash e_0 : T_0$$
$$\vdots$$
$$O, M, C \vdash e_n : T_n$$
$$T_0 \leq T$$
$$M(T, f) = (T_1', \dots, T_n', \text{SELF_TYPE})$$
$$T_i \leq T_i' \quad 1 \leq i \leq n$$

$$O, M, C \vdash e_0 @ T.f(e_1, \dots, e_n) : T_0$$

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

New Rules

- There are two new rules using **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 **SELF_TYPE** is used

Summary of SELF_TYPE

- The extended \leq 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 is itself 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
 - 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

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

- Since **x** is undeclared its type is **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 **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 ≤ 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