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 in COOL

```java
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 $B \leq A$
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 \( C' \leq C \)
  - 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 \( \text{Count} \) incorporates a counter
- The \( \text{inc} \) method works for any subclass
- But there is disaster lurking in the type system

An Example (Cont.)

- Consider a subclass \( \text{Stock} \) of \( \text{Count} \)
  ```java
class Stock inherits Count {
    name : String; -- name of item
  };
```
- And the following use of \( \text{Stock} \):
  ```java
class Main {
  Stock a ← (new Stock).inc ();  // Type checking error!
  ... a.name ...
};
```

What Went Wrong?

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

SELF_TYPE to the Rescue

- We will extend the type system
- Insight:
  - \( \text{inc} \) returns “self”
  - Therefore the return value has same type as “self”
  - Which could be \( \text{Count} \) or any subtype of \( \text{Count} \)
- Introduce the keyword \( \text{SELF\_TYPE} \) to use for the return value of such functions
  - We will also need to modify the typing rules to handle \( \text{SELF\_TYPE} \)

SELF_TYPE to the Rescue (Cont.)

- \( \text{SELF\_TYPE} \) allows the return type of \( \text{inc} \) to change when \( \text{inc} \) is inherited
- Modify the declaration of \( \text{inc} \) 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 <= T_2, 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 <=

Let T and T' be any types but SELF_TYPE
There are four cases in the definition of T <=

1. SELF_TYPE_C <= SELF_TYPE_C
   - In Cool we never need to compare SELF_TYPE Cs coming from different classes
2. SELF_TYPE_C <= T if C <= 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 ≤ (Cont.)

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

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

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

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:
  1. class \( T \) inherits \( T' \) { ... }
     • \( T, T' \) cannot be \( \text{SELF\_TYPE} \)
  2. \( x : T \)
     • \( T \) 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 \( \text{SELF\_TYPE} \)?

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

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

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

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

What's Different?

- Recall the old rule for dispatch

  \[ O,M,C \vdash e_0 : T_0 \]
  \[ \vdash \]
  \[ O,M,C \vdash e_n : T_n \]
  \[ M(T_0, f) = (T_1', ..., T_n', T_{n+1}) \]
  \[ T_{n+1} = \text{SELF_TYPE} \]
  \[ T_i \leq T_i' \quad 1 \leq i \leq n \]
  \[ O,M,C \vdash e_0.f(e_1, ..., e_n) : T_{n+1} \]

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:

  \[ O,M,C \vdash e_0 : T_0 \]
  \[ \vdash \]
  \[ O,M,C \vdash e_n : T_n \]
  \[ M(T_0, f) = (T_1', ..., T_n', \text{SELF_TYPE}) \]
  \[ T_i \leq T_i' \quad 1 \leq i \leq n \]
  \[ O,M,C \vdash e_0.f(e_1, ..., e_n) : T_0 \]

What's Different?

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

  \[ O,M,C \vdash e_0 : T_0 \]
  \[ \vdash \]
  \[ O,M,C \vdash e_n : T_n \]
  \[ M(T_0, f) = (T_1', ..., T_n', \text{SELF_TYPE}) \]
  \[ T_i \leq T_i' \quad 1 \leq i \leq n \]
  \[ O,M,C \vdash e_0.f(e_1, ..., e_n) : T_0 \]
**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 \(\text{SELF\_TYPE}\)

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

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

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

**Summary of SELF\_TYPE**

- The extended \(\leq\) and \(\text{ lub}\) 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 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 \(\text{Object + Int}\)
- This will generate another typing error
- We then say that that \(\text{Object + Int} = \text{Object}\)
- Then the initializer’s type will not be \(\text{Int}\)
  
\[
\Rightarrow \text{a workable solution but with cascading errors}
\]
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:
  ```plaintext
  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