More Static Semantics 2

One-Slide Summary

- **Typing rules** formalize the semantics checks necessary to validate a program. Well-typed programs do not go wrong.
- **Subtyping** relations ($\leq$) and **least-upper-bounds** (lub) are powerful tools for type-checking dynamic dispatch.
- We will use **SELF_TYPE$_C$** for "C or any subtype of C". It will show off the subtlety of type systems and allow us to check methods that return self objects.

Lecture Outline

- Typing Rules
- Dispatch Rules
  - Static
  - Dynamic
- SELF_TYPE

Assignment

What is this thing? What's $\vdash$? O? $\leq$?

$$\frac{\begin{array}{c} O(id) = T_0 \\ O \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O \vdash id \leftarrow e_1 : T_1} \quad [Assign]$$

Initialized Attributes

- Let $O_C(x) = T$ for all attributes $x{:}T$ in class C
  - O_C represents the class-wide scope
    - we "preload" the environment O with all attributes
- Attribute initialization is similar to let, except for the scope of names

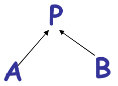
$$\frac{\begin{array}{c} O_C(id) = T_0 \\ O_C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C \vdash id : T_0 \leftarrow e_1 \ ;} \quad [Attr\text{-}Init]$$

If-Then-Else

- Consider:
  if e_0 then e_1 else e_2 fi
- The result can be either e_1 or e_2
- The dynamic type is either e_1's or e_2's type
- The best we can do statically is the **smallest supertype** larger than the type of e_1 and e_2

If-Then-Else example

- Consider the class hierarchy



- … and the expression

  if … then new A else new B fi
- Its type should allow for the dynamic type to be both A or B
  - Smallest supertype is P

Least Upper Bounds

- Define: **lub(X,Y)** to be the **least upper bound** of X and Y. lub(X,Y) is Z if
  - $X \leq Z \wedge Y \leq Z$

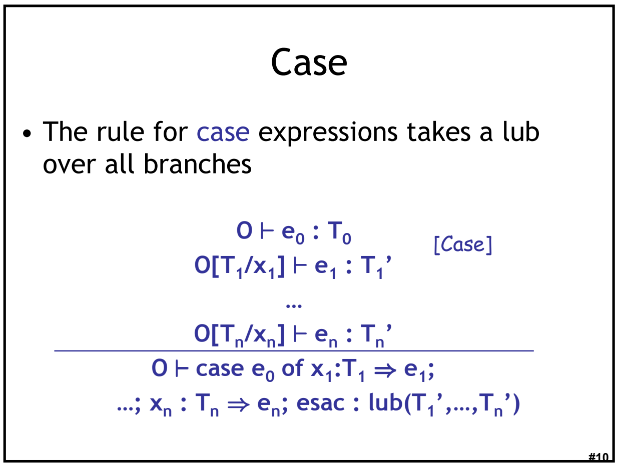
    Z is an upper bound
  - $X \leq Z' \wedge Y \leq Z' \Rightarrow Z \leq Z'$

    Z is least among upper bounds
- In Cool, the least upper bound of two types is their **least common ancestor** in the **inheritance tree**

If-Then-Else Revisited

$$\frac{\begin{array}{c} O \vdash e_0 : \text{Bool} \\ O \vdash e_1 : T_1 \\ O \vdash e_2 : T_2 \end{array}}{O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \text{lub}(T_1, T_2)}$$

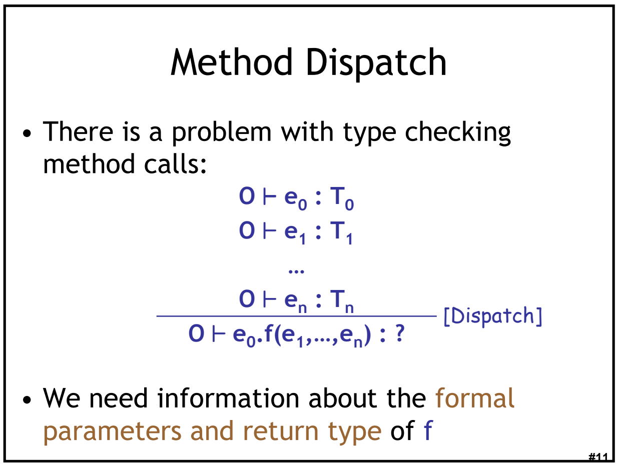
[If-Then-Else]



Case

- The rule for case expressions takes a lub over all branches

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T_1/x_1] \vdash e_1 : T_1' \\ \ldots \\ O[T_n/x_n] \vdash e_n : T_n' \end{array}}{\begin{array}{c} O \vdash \text{case } e_0 \text{ of } x_1{:}T_1 \Rightarrow e_1; \\ \ldots; x_n : T_n \Rightarrow e_n; \text{ esac} : \text{lub}(T_1',\ldots,T_n') \end{array}} \quad [\mathit{Case}]$$

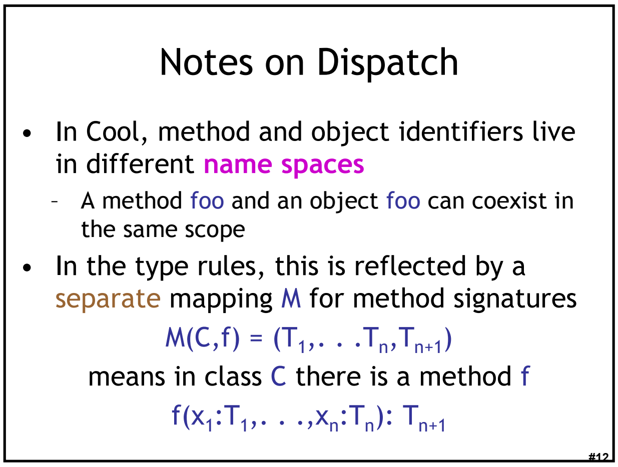


Method Dispatch

- There is a problem with type checking method calls:

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O \vdash e_1 : T_1 \\ \ldots \\ O \vdash e_n : T_n \end{array}}{O \vdash e_0.f(e_1,\ldots,e_n) : ?} \quad [\text{Dispatch}]$$

- We need information about the formal parameters and return type of f



Notes on Dispatch

- In Cool, method and object identifiers live in different **name spaces**
  - A method foo and an object foo can coexist in the same scope
- In the type rules, this is reflected by a separate mapping M for method signatures

$$M(C,f) = (T_1,\ldots T_n,T_{n+1})$$

means in class C there is a method f

$$f(x_1{:}T_1,\ldots,x_n{:}T_n){:}\ T_{n+1}$$

An Extended Typing Judgment

- Now we have *two* environments: O and M
- The form of the typing judgment is

$$O, M \vdash e : T$$

read as: “with the assumption that the object identifiers have types as given by O and the method identifiers have signatures as given by M, the expression e has type T”

The Method Environment

- The method environment must be added to all rules
- In most cases, M is passed down but not actually used
  - Example of a rule that does not use M:

$$\frac{O, M \vdash e_1 : T_1 \qquad O, M \vdash e_2 : T_2}{O, M \vdash e_1 + e_2 : \text{Int}} \text{[Add]}$$

  - Only the dispatch rules uses M

The Dispatch Rule Revisited

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \ldots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T_1',\ldots,T_n', T_{n+1}') \\ T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0.f(e_1,\ldots,e_n) : T_{n+1}'} \text{[Dispatch]}$$

Check receiver object e_0

Check actual arguments

Look up formal argument types T_i'

Static Dispatch

- **Static dispatch** is a variation on normal dispatch
- The method is found in the class explicitly named by the programmer (not via e_0)
- The inferred type of the dispatch expression must conform to the specified type

Static Dispatch (Cont.)

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \ldots \\ O, M \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}'} \ \text{[StaticDispatch]}$$

Handling the SELF_TYPE

Flexibility vs. Soundness

- Recall that type systems have two conflicting goals:
  - Give flexibility to the programmer
  - Prevent valid programs from "going wrong"
    - Milner, 1981: "Well-typed programs do not go wrong"
- An active line of research is in the area of inventing more flexible type systems while preserving soundness

Dynamic And Static Types

- The **dynamic type** of an object is ?
- The **static type** of an expression is ?
- You tell me!

Turn Off Yahoo! Toolbar Smart Tools?

☐ Don't ask me again

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

Soundness

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 can 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

Continuing Example

- Consider a subclass Stock of Count

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

- And the following use of Stock:

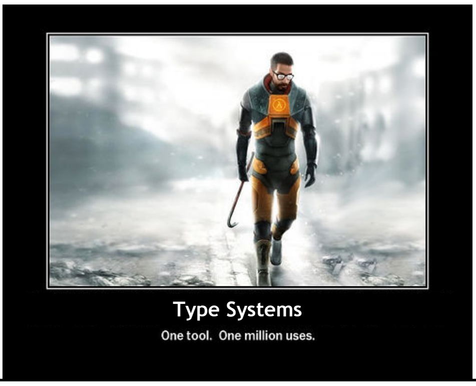
```
class Main {
  a : Stock ← (new Stock).inc ();
  …  a.name() …
};
```

Type checking error !

Post-Mortem

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

I Need A Hero!



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 !
  - In the case of (new Stock).inc () the type is Stock
- We 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 (2)

- 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:

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

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

- The program from before is now well typed

SELF_TYPE: Binford Tools

- SELF_TYPE is not a dynamic type
- SELF_TYPE 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 it denotes the dynamic type of the “self” expression:

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

- 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

Type Checking

- This suggests a typing rule:

$$\text{SELF\_TYPE}_C \leq C$$

- This 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 (whoops!)

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. $\text{SELF\_TYPE}_C \leq T$ 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
2. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
   - SELF_TYPE_C is the type of the “self” expression
   - In Cool we never need to compare SELF_TYPEs coming from different classes

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 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 SELF_TYPE$_C$

Where Can SELF_TYPE Appear in COOL?

3. let x : T in E
   - T can be SELF_TYPE
   - x has type 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,…,E_n)
   - T cannot be 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
- 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 $\leq$ and lub are the new ones
- Example:

$$\frac{\begin{array}{c} O(id) = T_0 \\ O,M,C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O,M,C \vdash id \leftarrow e_1 : T_1}$$

What’s Different?

- Recall the old rule for dispatch

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

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:

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

What’s Different?

- Note this rule handles the Stock example
- Formal parameters cannot be SELF_TYPE
- Actual arguments can be SELF_TYPE
  - The extended $\le$ 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

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

Static Dispatch

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

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

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

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

Where is SELF_TYPE Illegal in COOL?

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

Summary of SELF_TYPE

- The extended ≤ 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.
  - SELF_TYPE as the return type in an invoked method 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

Type Systems

- The rules in these lecture were Cool-specific
  - Other languages have very different rules
  - We'll survey a few more type systems later
- General themes
  - Type rules are defined on the structure of expressions
  - Types of variables are modeled by an environment
- Types are a play between flexibility and safety

Homework

- No WA due this week
- No PA due this week
- For Now: Happy Spring Break!
- For Tue Mar 13: Read Chapters 8.1-8.3
  - Optional Grant & Smith