More Static Semantics

Martin Kellogg

Agenda

- Finish discussion of subtyping/let from last class
- More type rules
- Method dispatch rules
  - static
  - dynamic
- SELF_TYPE

Course Announcements

- Don’t ignore PA2!
    - Listen to the TAs when they tell you to start ASAP

Agenda

- **Finish discussion of subtyping/let from last class**
- More type rules
- Method dispatch rules
  - static
  - dynamic
- SELF_TYPE

Semi-review: Assignment Rule

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma(id) = T_0 \qquad T_1 \leq T_0}{\Gamma \vdash id \texttt{ <- } e_1 : T_1}\ \textbf{[Assign]}$$

Semi-review: Assignment Rule

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma(id) = T_0 \qquad T_1 \leq T_0}{\Gamma \vdash id \texttt{ <- } e_1 : T_1} \text{[Assign]}$$

- How do I read this rule?

Semi-review: Assignment Rule

$$\frac{\Gamma \vdash \mathbf{e_1} : \mathbf{T_1} \qquad \Gamma(\mathbf{id}) = \mathbf{T_0} \qquad \mathbf{T_1} \leq \mathbf{T_0}}{\Gamma \vdash \mathbf{id} \texttt{ <- } \mathbf{e_1} : \mathbf{T_1}} \; \textbf{[Assign]}$$

- How do I read this rule?
- What is "Γ"? "⊢"? "≤"?

Examples of Wrong Let Rule (1)

Examples of Wrong Let Rule (1)

- Now consider a hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \text{ <- } e_0 \textbf{ in } e_1 : T_1} \; \textbf{[Let-Init]}$$

- How is it different from the correct rule?

Examples of Wrong Let Rule (1)

- Now consider a hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash \mathbf{e_0} : \mathbf{T} \qquad \mathbf{T} \leq \mathbf{T_0} \qquad \Gamma \vdash \mathbf{e_1} : \mathbf{T_1}}{\Gamma \vdash \mathbf{let}\ \mathbf{x} : \mathbf{T_0} \ \mathbf{<\!-}\ \mathbf{e_0}\ \mathbf{in}\ \mathbf{e_1} : \mathbf{T_1}} \ \textbf{[Let-Init]}$$

- How is it different from the correct rule?
- The following program does *not* typecheck:
  let x : Int <- 0 in x + 1
- Why not?

Examples of Wrong Let Rule (1)

- Now consider a hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{[Let-Init]}$$

- How is it different from the correct rule?
- The following program does ***not*** typecheck:

  let x : Int <- 0 in x + 1
- Why not? **Typing environment** hasn't been updated!

Examples of Wrong Let Rule (2)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T_0 \leq T \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let}\ x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \ \textbf{[Let-Init]}$$

- How is this one different from the correct rule?

Examples of Wrong Let Rule (2)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T_0 \leq T \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \; \textbf{[Let-Init]}$$

- How is this one different from the correct rule?
- The following ***bad program (!)*** is well-typed:

  let x : B <- new A in x.b()

- Why is this program bad?

Examples of Wrong Let Rule (3)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma[T/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \quad \textbf{[Let-Init]}$$

- How is this one different from the correct rule?

Examples of Wrong Let Rule (3)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma[T/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \; \textbf{[Let-Init]}$$

- How is this one different from the correct rule?
- This "good" program is not well-typed:

  let x : A <- new B in { … x <- new A ; x.a(); }
- Why isn't this program well-typed?

Type Rule Notation

- The type rules use **very concise** notation

Type Rule Notation

- The type rules use **very concise** notation
    - and they are very carefully constructed

Type Rule Notation

- The type rules use **very concise** notation
    - and they are very carefully constructed
- Virtually **any change** in a rule either:

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:
  - Makes the type system **unsound**
    - i.e., bad programs are well-typed

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:
  - Makes the type system **unsound**
    - i.e., bad programs are well-typed
  - Or, makes the type system less usable (more **incomplete**)
    - i.e., good programs are rejected

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:
  - Makes the type system **unsound**
    - i.e., bad programs are well-typed
  - Or, makes the type system less usable (more **incomplete**)
    - i.e., good programs are rejected
- But no matter how well we choose the type rules, ***some good programs will be rejected anyway***

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:
  - Makes the type system **unsound**
    - i.e., bad programs are well-typed
  - Or, makes the type system less usable (more **incomplete**)
    - i.e., good programs are rejected
- But no matter how well we choose the type rules, ***some good programs will be rejected anyway***
  - Rice's Theorem strikes again: typechecking is **undecidable**

Agenda

- Finish discussion of subtyping/let from last class
- **More type rules**
- Method dispatch rules
    - static
    - dynamic
- SELF_TYPE

Attribute Initialization

Attribute Initialization

- Let $\Gamma_C(x) = T$ for each attribute $x : T$ in class C
  - Γ_C represents the class-wide scope

Attribute Initialization

- Let $\Gamma_C(x) = T$ for each attribute $x : T$ in class C
  - Γ_C represents the class-wide scope
    - we “preload” the environment Γ with all attributes

Attribute Initialization

- Let $\Gamma_C(x) = T$ for each attribute $x : T$ in class C
    - Γ_C represents the class-wide scope
        - we "preload" the environment Γ with all attributes
- Attribute initialization is like **let**, except for the scope of names:

Attribute Initialization

- Let $\Gamma_C(\mathbf{x}) = \mathbf{T}$ for each attribute $\mathbf{x} : \mathbf{T}$ in class $\mathbf{C}$
  - Γ_C represents the class-wide scope
    - we "preload" the environment Γ with all attributes
- Attribute initialization is like **let**, except for the scope of names:

$$\frac{\Gamma_C(\mathbf{id}) = \mathbf{T_0} \qquad \Gamma_C \vdash \mathbf{e_1} : \mathbf{T_1} \qquad \mathbf{T_1} \leq \mathbf{T_0}}{\Gamma_C \vdash \mathbf{id} \texttt{ <- } \mathbf{e_1} : \mathbf{T_0}\,;} \quad \textbf{[Attr-Init]}$$

If-Then-Else

- Consider how to type the expression: **if** e_0 **then** e_1 **else** e_2 **fi**

If-Then-Else

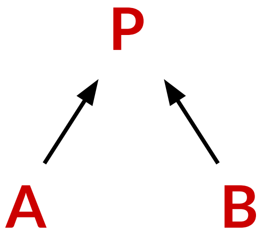
- Consider how to type the expression: **if** e_0 **then** e_1 **else** e_2 **fi**
- The result can be either e_1 or e_2
  - i.e., the dynamic type is either e_1's type or e_2's type

If-Then-Else

- Consider how to type the expression: **if e_0 then e_1 else e_2 fi**
- The result can be either e_1 or e_2
    - i.e., the dynamic type is either e_1's type or e_2's type
- The best we can do statically is the **closest supertype** of e_1's type and e_2's type

If-Then-Else

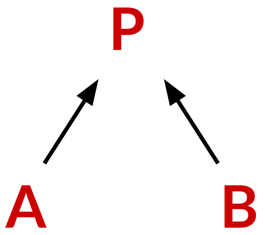
- Consider how to type the expression: **if e_0 then e_1 else e_2 fi**
- The result can be either e_1 or e_2
    - i.e., the dynamic type is either e_1's type or e_2's type
- The best we can do statically is the **closest supertype** of e_1's type and e_2's type
- E.g., consider the class hierarchy:



(note the arrows here mean inheritance)

If-Then-Else

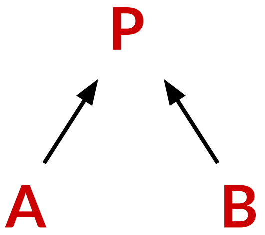
- Consider how to type the expression: **if $\mathbf{e_0}$ then $\mathbf{e_1}$ else $\mathbf{e_2}$ fi**
- The result can be either $\mathbf{e_1}$ or $\mathbf{e_2}$
  - i.e., the dynamic type is either $\mathbf{e_1}$'s type or $\mathbf{e_2}$'s type
- The best we can do statically is the **closest supertype** of $\mathbf{e_1}$'s type and $\mathbf{e_2}$'s type
- E.g., consider the class hierarchy:
  - if **A** is the type of $\mathbf{e_1}$ and **B** is the type of $\mathbf{e_2}$...



(note the arrows here mean inheritance)

If-Then-Else

- Consider how to type the expression: **if e_0 then e_1 else e_2 fi**
- The result can be either e_1 or e_2
  - i.e., the dynamic type is either e_1's type or e_2's type
- The best we can do statically is the **closest supertype** of e_1's type and e_2's type
- E.g., consider the class hierarchy:
  - if **A** is the type of e_1 and **B** is the type of e_2...
  - ...then we want to type the whole expression as **P**



(note the arrows here mean inheritance)

Least Upper Bounds

Definition: the ***least upper bound*** or ***lub*** over some relation ≤ of two elements of ≤'s domain X and Y is Z if:

Least Upper Bounds

Definition: the ***least upper bound*** or ***lub*** over some relation ≤ of two elements of ≤'s domain X and Y is Z if:

- $X \leq Z \wedge Y \leq Z$
  - "Z is **an** upper bound"

Least Upper Bounds

Definition: the ***least upper bound*** or ***lub*** over some relation ≤ of two elements of ≤'s domain X and 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"

Least Upper Bounds

Definition: the ***least upper bound*** or ***lub*** over some relation ≤ of two elements of ≤'s domain X and Y is Z if:

- $X \leq Z \land Y \leq Z$
  - "Z is **an** upper bound"
- $X \leq Z' \land Y \leq Z' \Rightarrow Z \leq Z'$
  - "Z is **least** among upper bounds"

- In Cool, the least upper bound of two types is their **closest common ancestor** in the inheritance tree (= type hierarchy)

If-Then-Else Rule

[If-Then-Else]

If-Then-Else Rule

$$\frac{\Gamma \vdash e_0 : \mathbf{Bool} \quad \Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \mathbf{fi} : \mathit{lub}(T_1, T_2)} \ \text{[If-Then-Else]}$$

Case

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

Case

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

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma[T_1/x_1] \vdash e_1 : T_1' \qquad \ldots \qquad \Gamma[T_n/x_n] \vdash e_n : T_n'}{\Gamma \vdash \textbf{case } e_0 \textbf{ of } x_1 : T_1 \texttt{ => } e_1 \,;\, \ldots \,;\, x_n : T_n \texttt{ => } e_n \,;\, \textbf{esac}\textbf{:}\ \mathit{lub}(T_1', \ldots, T_n')}\ \textbf{[Case]}$$

Method Dispatch

- There is a problem with typechecking method calls

Method Dispatch

- There is a problem with typechecking method calls
- Naïvely, we might want a rule like this:

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma \vdash e_1 : T_1 \qquad \dots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash e_0.f(e_1, \dots, e_n) : ?}\ \textbf{[Dispatch]}$$

Method Dispatch

- There is a problem with typechecking method calls
- Naïvely, we might want a rule like this:

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash e_0.f(e_1, \ldots, e_n) : ?}\ \textbf{[Dispatch]}$$

- We need information about the **formal parameter types** and **return type** of **f**, but Γ doesn't contain that information!

Notes on Dispatch

Notes on Dispatch

- In Cool, method and object identifiers live in different ***name spaces***
    - A method `foo` and a variable `foo` can coexist in the same scope without any problems

Notes on Dispatch

- In Cool, method and object identifiers live in different ***name spaces***
  - A method `foo` and a variable `foo` can coexist in the same scope without any problems
- In the type rules, this needs to be reflected by **separating** the type environments that contain information about methods and variables

Notes on Dispatch

- In Cool, method and object identifiers live in different ***name spaces***
  - A method `foo` and a variable `foo` can coexist in the same scope without any problems
- In the type rules, this needs to be reflected by **separating** the type environments that contain information about methods and variables
  - Add a ***second type environment*** for methods $\mathbf{M}$!
    - $\mathbf{M}$ maps a (class, method) tuple to a method signature

Notes on Dispatch

- In Cool, method and object identifiers live in different ***name spaces***
  - A method `foo` and a variable `foo` can coexist in the same scope without any problems
- In the type rules, this needs to be reflected by **separating** the type environments that contain information about methods and variables
  - Add a ***second type environment*** for methods $\mathbf{M}$!
    - $\mathbf{M}$ maps a (class, method) tuple to a method signature
    - e.g., $\mathbf{M}(C, f) = (T_1, \ldots, T_n, T_{ret})$ means that there is a method in class **C** with the signature $\mathbf{f}(x_1 : T_1, \ldots, x_n : T_n) : T_{ret}$

An Extended Typing Judgment

- Now we have ***two*** type environments: Γ and $\mathbf{M}$

An Extended Typing Judgment

- Now we have ***two*** type environments: Γ and $\mathbf{M}$
- The form of the typing judgment then becomes:

$$\Gamma, \mathbf{M} \vdash \mathsf{e} : \mathsf{T}$$

An Extended Typing Judgment

- Now we have ***two*** type environments: Γ and $\mathbf{M}$
- The form of the typing judgment then becomes:

$$\Gamma, \mathbf{M} \vdash \mathbf{e} : \mathrm{T}$$

- We read this as "with the assumptions that the free object identifiers in e have the types given by Γ *and the method identifiers in e have the signatures given by* $\mathbf{M}$, the expression e has type T."

The Method Environment

- The method environment needs to be added to all of our rules

The Method Environment

- The method environment needs to be added to all of our rules
- In most cases, $\mathbf{M}$ is passed down but not actually used

The Method Environment

- The method environment needs to be added to all of our rules
- In most cases, $\mathbf{M}$ is passed down but not actually used
  - for example, the **Add** rule does not use $\mathbf{M}$:

$$\frac{\Gamma, \mathbf{M} \vdash e_1 : \text{Int} \qquad \Gamma, \mathbf{M} \vdash e_2 : \text{Int}}{\Gamma, \mathbf{M} \vdash e_1 + e_2 : \text{Int}} \; \textbf{[Add]}$$

The Method Environment

- The method environment needs to be added to all of our rules
- In most cases, $\mathbf{M}$ is passed down but not actually used
  - for example, the **Add** rule does not use $\mathbf{M}$:

$$\frac{\Gamma, \mathbf{M} \vdash \mathbf{e_1} : \mathbf{Int} \qquad \Gamma, \mathbf{M} \vdash \mathbf{e_2} : \mathbf{Int}}{\Gamma, \mathbf{M} \vdash \mathbf{e_1} + \mathbf{e_2} : \mathbf{Int}} \; \textbf{[Add]}$$

- Only the **Dispatch** rule actually uses $\mathbf{M}$.

Method Dispatch Revisited

Method Dispatch Revisited

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

Method Dispatch Revisited

first check receiver object e_0

$$\frac{\boxed{\Gamma, M \vdash e_0 : T_0} \quad \Gamma, M \vdash e_1 : T_1 \quad \ldots \quad \Gamma, M \vdash e_n : T_n \qquad M(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \qquad \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i'}{\Gamma, M \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'}\ [\text{Dispatch}]$$

Method Dispatch Revisited

then check actual arguments are well-typed

$$\dfrac{\begin{array}{l}\Gamma, \mathbf{M} \vdash e_0 : T_0 \\ \boxed{\begin{array}{c}\Gamma, \mathbf{M} \vdash e_1 : T_1 \\ \ldots \\ \Gamma, \mathbf{M} \vdash e_n : T_n\end{array}}\end{array} \quad \begin{array}{l}\mathbf{M}(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \forall\ i \text{ in } (1\ldots n),\ T_i \leq T_i'\end{array}}{\Gamma, \mathbf{M} \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{ [Dispatch]}$$

Method Dispatch Revisited

next, look up method signature...

$$\dfrac{\begin{array}{ll} \Gamma, \mathbf{M} \vdash e_0 : T_0 & \\ \Gamma, \mathbf{M} \vdash e_1 : T_1 & \boxed{\mathbf{M}(T_0, f) = (T_1', \ldots, T_n', T_{n+1}')} \\ \ldots & \\ \Gamma, \mathbf{M} \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, \mathbf{M} \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{ [Dispatch]}$$

Method Dispatch Revisited

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

...and check that each argument is a subtype of the corresponding formal parameter

Method Dispatch Revisited

four steps!

1. first check receiver object e_0

2. then check actual arguments are well-typed

3. next, look up method signature...

4. ...and check that each argument is a subtype of the corresponding formal parameter

$$\frac{\begin{array}{ll} \Gamma, \mathbf{M} \vdash e_0 : T_0 & \\ \Gamma, \mathbf{M} \vdash e_1 : T_1 & \mathbf{M}(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, \mathbf{M} \vdash e_n : T_n & \forall\, i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, \mathbf{M} \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{[Dispatch]}$$

Static Dispatch

- The rule one previous slide is for ***dynamic dispatch***

Static Dispatch

- The rule one previous slide is for ***dynamic dispatch***
  - i.e., the method that gets called is based on the dynamic type of the receiver

Static Dispatch

- The rule one previous slide is for ***dynamic dispatch***
    - i.e., the method that gets called is based on the dynamic type of the receiver
- Cool also supports ***static dispatch***

Static Dispatch

- The rule one previous slide is for ***dynamic dispatch***
  - i.e., the method that gets called is based on the dynamic type of the receiver
- Cool also supports ***static dispatch***
  - i.e., where the method to be called is chosen based on a class **explicitly specified** by the programmer

Static Dispatch

- The rule one previous slide is for ***dynamic dispatch***
  - i.e., the method that gets called is based on the dynamic type of the receiver
- Cool also supports ***static dispatch***
  - i.e., where the method to be called is chosen based on a class **explicitly specified** by the programmer
- The static dispatch rule is similar to the dynamic dispatch rule, but the inferred type of the receiver must conform to the type that the programmer specifies

(new additions/changes from the regular Dispatch rule in purple)

Static Dispatch Rule

$$\frac{\begin{array}{ll} \Gamma, \mathbf{M} \vdash e_0 : T_0 & T_0 \leq T \\ \Gamma, \mathbf{M} \vdash e_1 : T_1 & \mathbf{M}(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \quad \ldots & \\ \Gamma, \mathbf{M} \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, \mathbf{M} \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}'} \text{ [Static Dispatch]}$$

Agenda

- Finish discussion of subtyping/let from last class
- More type rules
- Method dispatch rules
    - static
    - dynamic
- **SELF_TYPE**

Flexibility vs Soundness

- Recall that type systems have two conflicting goals:

Flexibility vs Soundness

- Recall that type systems have two conflicting goals:
  - give **flexibility** to the programmer
    - "allow more good programs"
    - **completeness**

Flexibility vs Soundness

- Recall that type systems have two conflicting goals:
  - give **flexibility** to the programmer
    - "allow more good programs"
    - **completeness**
  - prevent **incorrect** programs from being compiled
    - "don't allow bad programs"
    - **soundness**

Flexibility vs Soundness

- Recall that type systems have two conflicting goals:
    - give **flexibility** to the programmer
        - "allow more good programs"
        - **completeness**
    - prevent **incorrect** programs from being compiled
        - "don't allow bad programs"
        - **soundness**
- An active line of research: inventing more flexible type systems while preserving soundness

Flexibility vs Soundness

- Recall that type systems have two conflicting goals:
  - give **flexibility** to the programmer
    - "allow more good programs"
    - **completeness**
  - prevent **incorrect** programs from being compiled
    - "don't allow bad programs"
    - **soundness**
- An active line of research: inventing more flexible type systems while preserving soundness
  - SELF_TYPE is an "advanced" feature, to give you a taste of this

Review: Dynamic and Static Types

- Define the ***dynamic type*** of an object as ???

- Define the ***static type*** of an expression as ???

Review: Dynamic and Static Types

- Define the *dynamic type* of an object as the class C that is used in the “new C” expression that creates the object in some execution
    - run-time notion, present even in languages without static types
- Define the ***static type*** of an expression as the *least upper bound* of the dynamic types that the expression can take on, in some execution
    - cf. static vs dynamic semantics

Review: Cool Soundness

- Soundness theorem for the Cool type system:

Review: Cool Soundness

- Soundness theorem for the Cool type system:

for all expressions E, dynamic_type(E) ≤ static_type(E)

Review: Cool Soundness

- Soundness theorem for the Cool type system:

  for all expressions E, dynamic_type(E) ≤ static_type(E)

- Why is this ok?

Review: Cool Soundness

- Soundness theorem for the Cool type system:

  for all expressions E, dynamic_type(E) ≤ static_type(E)

- Why is this ok?
  - For all E, the compiler allows only operations that **static_type(E)** permits

Review: Cool Soundness

- Soundness theorem for the Cool type system:

  for all expressions E, dynamic_type(E) ≤ static_type(E)

- Why is this ok?
  - For all E, the compiler allows only operations that **static_type(E)** permits
  - Liskov substitutability guarantees that any operation available on a supertype is also available on its subtypes

Review: Cool Soundness

- Soundness theorem for the Cool type system:

  for all expressions E, dynamic_type(E) ≤ static_type(E)

- Why is this ok?
  - For all E, the compiler allows only operations that **static_type(E)** permits
  - Liskov substitutability guarantees that any operation available on a supertype is also available on its subtypes
    - subclasses can **only add** attributes or methods

Review: Cool Soundness

- Soundness theorem for the Cool type system:

  for all expressions E, dynamic_type(E) ≤ static_type(E)

- Why is this ok?
  - For all E, the compiler allows only operations that **static_type(E)** permits
  - Liskov substitutability guarantees that any operation available on a supertype is also available on its subtypes
    - subclasses can **only add** attributes or methods
    - methods can be redefined, but **only with the same types**

An Example

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

An Example

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

- This **Count** class implements a simple counter

An Example

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

- This **Count** class implements a simple counter
- The **inc** method works for any subclass…

An Example

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

- This **Count** class implements a simple counter
- The **inc** method works for any subclass...
  - ...or does it?

An Example

```
class Count {
  i : Int <- 0;
  inc() : Count {
    {
      i <- i + 1;
      self;
    }
  };
};
```

- This **Count** class implements a simple counter
- The **inc** method works for any subclass…
  - …or does it?
- There is a **problem lurking** here!

Co(u)ntinuing the Example

- Consider a subclass **Stock** of **Count**:

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

Co(u)ntinuing the 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() …
};
```

Co(u)ntinuing the 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() …
};
```

our current rules will cause a typechecking error here, because inc() returns a Count (not a Stock)

Postmortem

Postmortem

- `(new Stock).inc()` has *dynamic* type **Stock**

Postmortem

- `(new Stock).inc()` has *dynamic* type Stock
  - So, it's legitimate to write:

```
a : Stock <- (new Stock).inc()
```

Postmortem

- `(new Stock).inc()` has *dynamic* type **Stock**
  - So, it's legitimate to write:

```
a : Stock <- (new Stock).inc()
```

- But this is not well-typed, because `(new Stock).inc()` has *static* type **Count**

Postmortem

- `(new Stock).inc()` has *dynamic* type **Stock**
  - So, it's legitimate to write:

```
a : Stock <- (new Stock).inc()
```

- But this is not well-typed, because `(new Stock).inc()` has *static* type **Count**
- The typechecker has "lost" information

Trivia Break: History

This city on the Danube river famously was the home of a number of influential figures of the 20th century for a short time in 1913, including Leon Trotsky, Joseph Stalin, Adolf Hitler, Sigmund Freud, and Josip Broz Tito. It was the seat of the Holy Roman Emperors of the Habsburg dynasty from the 16th-century until the empire's dissolution in 1806 (with only brief interruptions). Afterward, it was the seat of Austria-Hungary until the dissolution of that empire following the first World War. It is now the capital of Austria.

Trivia Break: Computer Science

This American computer scientist and mathematician was the recipient of the 1974 Turing Award. He has been called the "father of the analysis of algorithms". He is the author of the multi-volume work *The Art of Computer Programming*. In addition to his work in theoretical computer science, he is the creator of the TeX computer typesetting system, the related METAFONT font definition language and rendering system, and the Computer Modern family of typefaces.

SELF_TYPE to the Rescue

- We will **extend** the type system

SELF_TYPE to the Rescue

- We will **extend** the type system
  - That is, make it **more expressive**

SELF_TYPE to the Rescue

- We will **extend** the type system
  - That is, make it **more expressive**
- Insight:

SELF_TYPE to the Rescue

- We will **extend** the type system
  - That is, make it **more expressive**
- Insight:
  - inc returns “self”

SELF_TYPE to the Rescue

- We will **extend** the type system
  - That is, make it **more expressive**
- Insight:
  - inc returns “self”
  - therefore the return value will be the same type as “self”

SELF_TYPE to the Rescue

- We will **extend** the type system
  - That is, make it **more expressive**
- Insight:
  - inc returns “self”
  - therefore the return value will be the same type as “self”
  - which could be **Count** *or any subtype* of **Count**

SELF_TYPE to the Rescue

- We will **extend** the type system
  - That is, make it **more expressive**
- Insight:
  - inc returns "self"
  - therefore the return value will be the same type as "self"
  - which could be **Count** *or any subtype* of **Count**
  - In the case of `(new Stock).inc()`, the type is **Stock**

SELF_TYPE to the Rescue

- We will extend the type system
    - That is, make it more expressive
- Insight:
    - inc returns “self”
    - therefore the return value will be the 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

SELF_TYPE to the Rescue

- We will extend the type system
  - That is, make it **more expressive**
- Insight:
  - inc returns "self"
  - therefore the return value will be the 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 need to modify the type 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

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 { ... }
```

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 typechecker can now prove:

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 typechecker can now prove:

$$\Gamma, \mathbf{M} \vdash \texttt{(new Count).inc()} : \textbf{Count}$$

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 typechecker can now prove:

$$\Gamma, \mathbf{M} \vdash \texttt{(new Count).inc()} : \textbf{Count}$$

$$\Gamma, \mathbf{M} \vdash \texttt{(new Stock).inc()} : \textbf{Stock}$$

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 typechecker can now prove:

$$\Gamma, \mathbf{M} \vdash \texttt{(new Count).inc()} : \textbf{Count}$$

$$\Gamma, \mathbf{M} \vdash \texttt{(new Stock).inc()} : \textbf{Stock}$$

- The program from before is now well typed

What does SELF_TYPE do?

- SELF_TYPE is **not** a dynamic type

What does SELF_TYPE do?

- SELF_TYPE is **not** a dynamic type
    - SELF_TYPE is a static type

What does SELF_TYPE do?

- SELF_TYPE is **not** a dynamic type
    - SELF_TYPE is a static type
- It helps the typechecker to keep better track of types

What does SELF_TYPE do?

- SELF_TYPE is **not** a dynamic type
    - SELF_TYPE is a static type
- It helps the typechecker to keep better track of types
- It enables the typechecker to accept more correct programs

What does SELF_TYPE do?

- SELF_TYPE is **not** a dynamic type
  - SELF_TYPE is a static type
- It helps the typechecker to keep better track of types
- It enables the typechecker 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?

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”!

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"!
  - Equally-valid answer: **Count** or ***any subclass of Count***!

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”!
  - Equally-valid answer: **Count** or ***any subclass of Count***!
- 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:

  dynamic_type(E) = dynamic_type(self) ≤ C

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”!
  - Equally-valid answer: **Count** or ***any subclass of Count***!
- 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:

  dynamic_type(E) = dynamic_type(self) ≤ 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 some specific class C

Typechecking SELF_TYPE

- The example on the previous slide suggests a type rule for **SELF_TYPE**:

Typechecking SELF_TYPE

- The example on the previous slide suggests a type rule for **SELF_TYPE**:

$$\mathbf{SELF\_TYPE_C \leq C}$$

Typechecking SELF_TYPE

- The example on the previous slide suggests a type rule for **SELF_TYPE**:

$$\mathbf{SELF\_TYPE_C \leq C}$$

- This rule has an important consequence:
  - In typechecking, it is always safe to replace **SELF_TYPE**$_C$ by C

Typechecking SELF_TYPE

- The example on the previous slide suggests a type rule for **SELF_TYPE**:

$$\mathbf{SELF\_TYPE_C \leq C}$$

- This rule has an important consequence:
  - In typechecking, it is always safe to replace **SELF_TYPE**$_C$ by C
- This suggests one way to handle **SELF_TYPE** in a typechecker implementation:

Typechecking SELF_TYPE

- The example on the previous slide suggests a type rule for **SELF_TYPE**:

$$\mathbf{SELF\_TYPE_C \leq C}$$

- This rule has an important consequence:
  - In typechecking, it is always safe to replace **SELF_TYPE**$_C$ by C
- This suggests one way to handle **SELF_TYPE** in a typechecker implementation:
  - replace all occurrences of **SELF_TYPE**$_C$ with C

Typechecking SELF_TYPE

- The example on the previous slide suggests a type rule for **SELF_TYPE**:

$$\mathbf{SELF\_TYPE_C \leq C}$$

- This rule has an important consequence:
  - In typechecking, it is always safe to replace **SELF_TYPE**$_C$ by C
- This suggests one way to handle **SELF_TYPE** in a typechecker implementation:
  - replace all occurrences of **SELF_TYPE**$_C$ with C
- What's **wrong** with this?

Typechecking SELF_TYPE

- The example on the previous slide suggests a type rule for **SELF_TYPE**:

$$\mathbf{SELF\_TYPE_C \leq C}$$

- This rule has an important consequence:
  - In typechecking, it is always safe to replace **SELF_TYPE**$_C$ by C
- This suggests one way to handle **SELF_TYPE** in a typechecker implementation:
  - replace all occurrences of **SELF_TYPE**$_C$ with C
- What's **wrong** with this?
  - It's sound, but it's like not having **SELF_TYPE** at all (oops)

Typechecking SELF_TYPE (properly)

Typechecking SELF_TYPE (properly)

- Recall the operations that we've defined over types:

Typechecking SELF_TYPE (properly)

- Recall the operations that we've defined over types:
  - subtyping: $T_1 \leq T_2$
  - least upper bound: $lub(T_1, T_2)$

Typechecking SELF_TYPE (properly)

- Recall the operations that we've defined over types:
  - subtyping: $T_1 \leq T_2$
  - least upper bound: $\mathit{lub}(T_1, T_2)$
- To handle **SELF_TYPE** properly, we need to **extend** these operations to handle it
  - doing so is surprisingly involved…

Extending ≤

Extending ≤

- Let T and T’ be any types except **SELF_TYPE**

Extending ≤

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

Extending ≤

- Let T and T' be any types except **SELF_TYPE**
- There are four cases in the definition of ≤:
  1. $\mathbf{SELF\_TYPE}_C \leq T$ if $\mathbf{C} \leq T$

Extending ≤

- Let T and T’ be any types except **SELF_TYPE**
- There are four cases in the definition of ≤:

1. **SELF_TYPE$_C$** ≤ T if **C** ≤ T
    - **SELF_TYPE$_C$** can be any subtype of **C,** including **C** itself

Extending ≤

- Let T and T' be any types except **SELF_TYPE**
- There are four cases in the definition of ≤:
  1. $\textbf{SELF\_TYPE}_C \leq T$ if $\textbf{C} \leq T$
     - $\textbf{SELF\_TYPE}_C$ can be any subtype of **C,** including **C** itself
     - Thus this is the most flexible rule we can allow

Extending ≤

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

1. $\mathbf{SELF\_TYPE_C} \le T$ if $\mathbf{C} \le T$
    - $\mathbf{SELF\_TYPE_C}$ can be any subtype of **C,** including **C** itself
    - Thus this is the most flexible rule we can allow
2. $\mathbf{SELF\_TYPE_C} \le \mathbf{SELF\_TYPE_C}$
    - recall that $\mathbf{SELF\_TYPE_C}$ is the type of the “self” expression

Extending $\leq$

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

1. $\text{SELF\_TYPE}_C \leq T$ if $C \leq T$
    - SELF_TYPE_C can be any subtype of **C**, including **C** itself
    - Thus this is the most flexible rule we can allow
2. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
    - recall that SELF_TYPE_C is the type of the “self” expression
    - In Cool we *never* need to compare **SELF_TYPE**s coming from different classes (why? left as an exercise…)

Extending ≤ (continued)

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

  3.

Extending ≤ (continued)

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

3. **T** ≤ **SELF_TYPE**$_C$ is always false
    - Why?

Extending ≤ (continued)

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

3. T ≤ **SELF_TYPE**$_C$ is always false
   - Why? **SELF_TYPE**$_C$ could be any subtype of C…

Extending ≤ (continued)

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

3. T ≤ **SELF_TYPE**$_C$ is always false
    - Why? **SELF_TYPE**$_C$ could be any subtype of C…
4. T ≤ T'

Extending ≤ (continued)

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

3. **T** ≤ **SELF_TYPE**$_C$ is always false
    - Why? **SELF_TYPE**$_C$ could be any subtype of C…
4. **T** ≤ **T'**
    - according to the rules from before

Extending ≤ (continued)

- Let T and T’ be any types except **SELF_TYPE**
- There are four cases in the definition of ≤:

3. T ≤ **SELF_TYPE**$_C$ is always false
    - Why? **SELF_TYPE**$_C$ could be any subtype of C…
4. T ≤ T’
    - according to the rules from before

- Note these rules covered every combination + order of **T**s and **SELF_TYPE**$_C$s

Extending ≤ (continued)

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

3. **T** ≤ **SELF_TYPE$_C$** is always false
   - Why? **SELF_TYPE$_C$** could be any subtype of C…
4. **T** ≤ **T'**
   - according to the rules from before

- Note these rules covered every combination + order of **T**s and **SELF_TYPE$_C$**s
- Using these rules, we can extend ***lub*** too…

Extending *lub*

- Again, let **T** and **T'** be any types except **SELF_TYPE**

Extending *lub*

- Again, let T and T' be any types except **SELF_TYPE**
- Again, there are four cases:

Extending *lub*

- Again, let **T** and **T'** be any types except **SELF_TYPE**
- Again, there are four cases:
  1. *lub*($\mathbf{SELF\_TYPE}_C$, $\mathbf{SELF\_TYPE}_C$) = $\mathbf{SELF\_TYPE}_C$

Extending *lub*

- Again, let T and T' be any types except **SELF_TYPE**
- Again, there are four cases:
  1. *lub*($\mathbf{SELF\_TYPE_C}$, $\mathbf{SELF\_TYPE_C}$) = $\mathbf{SELF\_TYPE_C}$
  2. *lub*($\mathbf{SELF\_TYPE_C}$, T) = *lub*(**C**, T)
     - this is the best we can do because $\mathbf{SELF\_TYPE_C} \leq \mathbf{C}$

Extending *lub*

- Again, let T and T' be any types except **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)

Extending *lub*

- Again, let T and T' be any types except **SELF_TYPE**
- Again, there are four cases:
  1. *lub*($\textbf{SELF\_TYPE}_C$, $\textbf{SELF\_TYPE}_C$) = $\textbf{SELF\_TYPE}_C$
  2. *lub*($\textbf{SELF\_TYPE}_C$, T) = *lub*(C, T)
     - this is the best we can do because $\textbf{SELF\_TYPE}_C \leq \textbf{C}$
  3. *lub*(T, $\textbf{SELF\_TYPE}_C$) = *lub*(C, T)
     - bonus question: why is this the same as case 2?

Extending *lub*

- Again, let T and T' be any types except **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)
     - bonus question: why is this the same as case 2?
  4. *lub*(T, T') defined the same as before

Where can SELF_TYPE Appear in Cool?

Where can SELF_TYPE Appear in Cool?

- The parser checks that **SELF_TYPE** only appears in locations where a type is permitted

Where can SELF_TYPE Appear in Cool?

- The parser checks that **SELF_TYPE** only appears in locations where a type is permitted
  - But **SELF_TYPE** isn’t allowed everywhere that a type is!

Where can SELF_TYPE Appear in Cool?

- The parser checks that **SELF_TYPE** only appears in locations where a type is permitted
  - But **SELF_TYPE** isn't allowed everywhere that a type is!
  - For example, in `class T inherits T' {...}`:
    - neither `T` nor `T'` can be **SELF_TYPE**
      - because **SELF_TYPE** is never a dynamic type

Where can SELF_TYPE Appear in Cool?

- The parser checks that **SELF_TYPE** only appears in locations where a type is permitted
  - But **SELF_TYPE** isn't allowed everywhere that a type is!
  - For example, in `class T inherits T' {...}`:
    - neither `T` nor `T'` can be **SELF_TYPE**
      - because **SELF_TYPE** is never a dynamic type
- On the other hand, in an attribute declaration `x : T`,
  - `T` *can* be **SELF_TYPE**

Where can SELF_TYPE Appear in Cool?

- The parser checks that **SELF_TYPE** only appears in locations where a type is permitted
  - But **SELF_TYPE** isn't allowed everywhere that a type is!
  - For example, in `class T inherits T' {...}`:
    - neither `T` nor `T'` can be **SELF_TYPE**
      - because **SELF_TYPE** is never a dynamic type
- On the other hand, in an attribute declaration `x : T`,
  - `T` *can* be **SELF_TYPE**
    - it means the attribute's type is $\mathbf{SELF\_TYPE}_C$

Where can SELF_TYPE Appear in Cool?

- What about `let` expressions? Can the variable be **SELF_TYPE**?

Where can SELF_TYPE Appear in Cool?

- What about `let` expressions? Can the variable be **SELF_TYPE**?
    - Yes: in `let x : T in E`, if `T` is **SELF_TYPE**, then `x` has the type $\textbf{SELF\_TYPE}_C$

Where can SELF_TYPE Appear in Cool?

- What about `let` expressions? Can the variable be **SELF_TYPE**?
  - Yes: in `let x : T in E`, if `T` is **SELF_TYPE**, then `x` has the type **SELF_TYPE**$_C$
- What about `new` expressions?

Where can SELF_TYPE Appear in Cool?

- What about `let` expressions? Can the variable be **SELF_TYPE**?
  - Yes: in `let x : T in E`, if `T` is **SELF_TYPE**, then `x` has the type **SELF_TYPE**$_C$
- What about `new` expressions?
  - Yes: in `new T`, if `T` is **SELF_TYPE**, then the expression evaluates to a value of type **SELF_TYPE**$_C$

Where can SELF_TYPE Appear in Cool?

- What about `let` expressions? Can the variable be **SELF_TYPE**?
  - Yes: in `let x : T in E`, if `T` is **SELF_TYPE**, then `x` has the type **SELF_TYPE**$_C$
- What about `new` expressions?
  - Yes: in `new T`, if `T` is **SELF_TYPE**, then the expression evaluates to a value of type **SELF_TYPE**$_C$
- What about static dispatch? E.g., in $\texttt{m@T(E}_1\texttt{, . . . ,E}_n\texttt{)}$, can `T` be **SELF_TYPE**?

Where can SELF_TYPE Appear in Cool?

- What about `let` expressions? Can the variable be **SELF_TYPE**?
    - Yes: in `let x : T in E`, if `T` is **SELF_TYPE**, then `x` has the type **$\mathbf{SELF\_TYPE}_C$**
- What about `new` expressions?
    - Yes: in `new T`, if `T` is **SELF_TYPE**, then the expression evaluates to a value of type **$\mathbf{SELF\_TYPE}_C$**
- What about static dispatch? E.g., in $\texttt{m@T(}E_1\texttt{, ..., }E_n\texttt{)}$, can `T` be **SELF_TYPE**?
    - No: the `T` in static dispatch needs to refer to a *specific, dynamic type*

Type Rules for SELF_TYPE

Type Rules for SELF_TYPE

- Since occurrences of **SELF_TYPE** depend on the **enclosing class**, we need to carry **more context** during typechecking
  - In particular, we need to add the enclosing class!

Type Rules for SELF_TYPE

- Since occurrences of **SELF_TYPE** depend on the **enclosing class**, we need to carry **more context** during typechecking
  - In particular, we need to add the enclosing class!
- This leads to a new typing judgment form:

$$\Gamma, \mathbf{M}, C \vdash e : T$$

Type Rules for SELF_TYPE

- Since occurrences of **SELF_TYPE** depend on the **enclosing class**, we need to carry **more context** during typechecking
    - In particular, we need to add the enclosing class!
- This leads to a new typing judgment form:

$$\Gamma, \mathbf{M}, \mathsf{C} \vdash \mathsf{e} : \mathsf{T}$$

- Read as "An expression **e** occurring in the body of C has static type **T** given a variable type environment Γ and method signatures **M**"

Changing the Type Rules for SELF_TYPE

- The next step is to design type rules that account for **SELF_TYPE** for each language construct

Changing the Type Rules for SELF_TYPE

- The next step is to design type rules that account for **SELF_TYPE** for each language construct
- Most of these rules are the same as the rules without **SELF_TYPE**, except that ≤ and ***lub*** are the new versions with **SELF_TYPE** support; only change is to pass through the enclosing class

Changing the Type Rules for SELF_TYPE

- The next step is to design type rules that account for **SELF_TYPE** for each language construct
- Most of these rules are the same as the rules without **SELF_TYPE**, except that ≤ and *lub* are the new versions with **SELF_TYPE** support; only change is to pass through the enclosing class
- E.g.,:

$$\frac{\Gamma, M, C \vdash e_1 : T_1 \quad \Gamma(id) = T_0 \quad T_1 \leq T_0}{\Gamma, M, C \vdash id \texttt{ <- } e_1 : T_1} \text{ [Assign]}$$

Changes to Dispatch Rules

- The rules for dispatch need to change. We modify the old dispatch rule:

Changes to Dispatch Rules

- The rules for dispatch need to change. We modify the old dispatch rule:

$$\frac{\Gamma, \mathbf{M}, \mathbf{C} \vdash e_0 : T_0 \quad \Gamma, \mathbf{M}, \mathbf{C} \vdash e_1 : T_1 \quad \ldots \quad \Gamma, \mathbf{M}, \mathbf{C} \vdash e_n : T_n \qquad \mathbf{M}(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \qquad \forall \text{ i in } (1 \ldots n), T_i \leq T_i'}{\Gamma, \mathbf{M}, \mathbf{C} \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{[Dispatch]}$$

Changes to Dispatch Rules

- The rules for dispatch need to change. We modify the old dispatch rule:

$$\frac{\begin{array}{ll} \Gamma, \mathbf{M}, \mathbf{C} \vdash e_0 : T_0 \quad \Gamma, & T_{n+1}' \neq \textbf{SELF\_TYPE} \\ \mathbf{M}, \mathbf{C} \vdash e_1 : T_1 & \mathbf{M}(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, \mathbf{M}, \mathbf{C} \vdash e_n : T_n & \forall \text{ i in } (1 \ldots n), T_i \leq T_i' \end{array}}{\Gamma, \mathbf{M}, \mathbf{C} \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{[Dispatch]}$$

Changes to Dispatch Rules

- Then, we add a **new rule** for the **SELF_TYPE** case:

Changes to Dispatch Rules

- Then, we add a **new rule** for the **SELF_TYPE** case:
  - (changes in **pink**)

$$\frac{\Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, M, C \vdash e_1 : T_1 \quad \ldots \quad \Gamma, M, C \vdash e_n : T_n \qquad M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \qquad \forall\, i \text{ in } (1 \ldots n),\ T_i \leq T_i'}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0}\ [\text{Dispatch-Self}]$$

What's different about this rule?

$$
\frac{
\begin{array}{ll}
\Gamma, \mathbf{M}, \mathbf{C} \vdash e_0 : T_0 \quad \Gamma, & \\
\mathbf{M}, \mathbf{C} \vdash e_1 : T_1 & \mathbf{M}(T_0, f) = (T_1', \ldots, T_n', \mathrm{SELF\_TYPE}) \\
\ldots & \\
\Gamma, \mathbf{M}, \mathbf{C} \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i'
\end{array}
}{
\Gamma, \mathbf{M}, \mathbf{C} \vdash e_0.f(e_1, \ldots, e_n) : T_0
}\ \text{[Dispatch-Self]}
$$

What's different about this rule?

- It handles the **Stock** example

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, M, C \vdash e_1 : T_1 & \\ & M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\, i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{[Dispatch-Self]}$$

What's different about this rule?

- It handles the **Stock** example
- Formal parameters **can't** be SELF_TYPE

$$
\frac{
\begin{array}{ll}
\Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, & \\
M, C \vdash e_1 : T_1 & M(T_0, f) = (T_1', \ldots, T_n', \mathrm{SELF\_TYPE}) \\
\ldots & \\
\Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n),\ T_i \leq T_i'
\end{array}
}{
\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0
}
\quad \textbf{[Dispatch-Self]}
$$

What's different about this rule?

- It handles the **Stock** example
- Formal parameters **can't** be SELF_TYPE
- Actual arguments **can** be SELF_TYPE
  - extended ≤ handles this case

$$\frac{\Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, M, C \vdash e_1 : T_1 \quad \ldots \quad \Gamma, M, C \vdash e_n : T_n \qquad M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \qquad \forall\, i \text{ in } (1 \ldots n),\ T_i \leq T_i'}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{[Dispatch-Self]}$$

What's different about this rule?

- It handles the **Stock** example
- Formal parameters **can't** be SELF_TYPE
- Actual arguments **can** be SELF_TYPE
  - extended ≤ handles this case
- The type T_0 of the dispatch expression *could* be SELF_TYPE

$$\frac{\Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, M, C \vdash e_1 : T_1 \quad \ldots \quad \Gamma, M, C \vdash e_n : T_n \qquad M(T_0, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \qquad \forall\, i \text{ in } (1 \ldots n),\ T_i \le T_i'}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{[Dispatch-Self]}$$

Changes to Dispatch Rules

- What about **static dispatch**? Does it need changes?

Changes to Dispatch Rules

- What about **static dispatch**? Does it need changes? Yes…

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 & \Gamma, T_0 \leq T \\ M, C \vdash e_1 : T_1 & \\ & M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n), T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}'}\ \text{[Static Dispatch]}$$

Changes to Dispatch Rules

- What about static dispatch? Does it need changes? Yes...

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, T_0 \leq T & T_{n+1}' \neq \text{SELF\_TYPE} \\ M, C \vdash e_1 : T_1 & M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\ i \text{ in } (1 \ldots n), T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}'} \text{[Static Dispatch]}$$

Changes to Dispatch Rules

- And again we need a special rule for when the method's return type is **SELF_TYPE**:

Changes to Dispatch Rules

- And again we need a special rule for when the method's return type is **SELF_TYPE**: (changes again in **pink**)

$$\frac{\begin{array}{ll} \Gamma, M, C \vdash e_0 : T_0 \quad \Gamma, & T_0 \leq T \\ M, C \vdash e_1 : T_1 & M(T, f) = (T_1', \ldots, T_n', \text{SELF\_TYPE}) \\ \ldots & \\ \Gamma, M, C \vdash e_n : T_n & \forall\, i \text{ in } (1 \ldots n),\ T_i \leq T_i' \end{array}}{\Gamma, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_0} \text{[St.-Dispatch-Self]}$$

Static Dispatch Notes

Static Dispatch Notes

- Why is the rule on the previous slide correct?
  - If we dispatch a method returning **SELF_TYPE** in some class T, don’t we get back a T?

Static Dispatch Notes

- Why is the rule on the previous slide correct?
    - If we dispatch a method returning **SELF_TYPE** in some class **T**, don't we get back a **T**?
- **No**. SELF_TYPE is the type of "self", which may be a **subclass** of the class in which the **method body** appears

Static Dispatch Notes

- Why is the rule on the previous slide correct?
    - If we dispatch a method returning **SELF_TYPE** in some class **T**, don't we get back a **T**?
- **No**. SELF_TYPE is the type of "self", which may be a **subclass** of the class in which the **method body** appears
    - Note: ***not*** the class in which the **call site** appears!

Static Dispatch Notes

- Why is the rule on the previous slide correct?
  - If we dispatch a method returning **SELF_TYPE** in some class **T**, don't we get back a **T**?
- **No**. SELF_TYPE is the type of "self", which may be a **subclass** of the class in which the **method body** appears
  - Note: ***not*** the class in which the **call site** appears!
- The static dispatch class cannot be SELF_TYPE

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

$$\frac{}{\Gamma, M, C \vdash \textbf{self} : \textbf{SELF\_TYPE}_C}\ \textbf{[Self]}$$

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

$$\frac{}{\Gamma, M, C \vdash \textbf{self} : \textbf{SELF\_TYPE}_C}\ \textbf{[Self]}$$

$$\frac{}{\Gamma, M, C \vdash \textbf{new SELF\_TYPE} : \textbf{SELF\_TYPE}_C}\ \textbf{[New-Self]}$$

New SELF_TYPE Rules

- There are also two other new rules specifically for SELF_TYPE:

$$\frac{}{\Gamma, M, C \vdash \textbf{self} : \textbf{SELF\_TYPE}_C} \quad \textbf{[Self]}$$

$$\frac{}{\Gamma, M, C \vdash \textbf{new SELF\_TYPE} : \textbf{SELF\_TYPE}_C} \quad \textbf{[New-Self]}$$

- There are a number of other places in the rules where SELF_TYPE appears - read the CRM carefully

Where is SELF_TYPE illegal in Cool?

Where is SELF_TYPE illegal in Cool?

- `m(x : T) : T’ { … }`
    - only `T’` (not `T`) can be SELF_TYPE!

Where is SELF_TYPE illegal in Cool?

- `m(x : T) : T' { … }`
  - only `T'` (not `T`) can be SELF_TYPE!
  - What would go wrong if `T` were SELF_TYPE?

Where is SELF_TYPE illegal in Cool?

- `m(x : T) : T' { … }`
  - only T' (not T) can be SELF_TYPE!
  - What would 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

Summary of SELF_TYPE

- The extended ≤ and ***lub*** operations can do a lot of the work.
    - Implement them to handle SELF_TYPE

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.***

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

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 typechecking of **dispatch**.

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 typechecking of **dispatch**.
  - SELF_TYPE as the return type in an invoked method might have ***nothing to do*** with the current class

Why Do We Cover SELF_TYPE?

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
  - it adds **expressiveness** to the type system without allowing any "bad" programs

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
  - it adds **expressiveness** to the type system without allowing any "bad" programs
  - but at the cost of **additional complexity**

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
  - it adds **expressiveness** to the type system without allowing any "bad" programs
  - but at the cost of **additional complexity**
- SELF_TYPE itself isn't that important
  - although you have to get it right for PA2…

Why Do We Cover SELF_TYPE?

- SELF_TYPE is an example of a **research idea**
  - it adds **expressiveness** to the type system without allowing any "bad" programs
  - but at the cost of **additional complexity**
- SELF_TYPE itself isn't that important
  - although you have to get it right for PA2...
- But it is **illustrative** of a class of ideas that trade-off expressiveness for complexity
  - and gives you a taste of how this works in practice!

Type Systems

- The rules in these lectures were **Cool-specific**

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions
  - Types of variables are modeled by a **type environment**

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions
  - Types of variables are modeled by a **type environment**
  - There is a tradeoff between **safety** and **flexibility**

Type Systems

- The rules in these lectures were **Cool-specific**
  - Other languages have (very!) different rules
  - We'll survey some other type systems later in the course
- **General themes** of type systems (that aren't Cool-specific):
  - Type rules are defined on the **structure** of expressions
  - Types of variables are modeled by a **type environment**
  - There is a tradeoff between **safety** and **flexibility**
  - There is another tradeoff between **expressiveness** and **complexity**