Parking For Drive-Thru Service Only 2

Thank You

More Static Semantics

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**? $\leq$?

$$O(id) = T_0$$
$$O \vdash e_1 : T_1$$
$$T_1 \leq T_0$$
$$\frac{}{O \vdash id \leftarrow e_1 : T_1} \text{[Assign]}$$

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

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

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
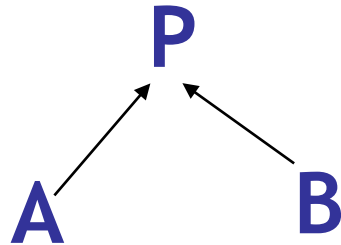

Wayne Hart's 5-day forecast

Today	Tonight	Saturday	Sunday	Monday	Tuesday
Sun & Clouds, Warmer	Partly Cloudy	Partly Cloudy	Partly Sunny, Breezy & Cooler	Sun & Clouds	Partly Cloudy, Warmer
52°	33°	53° 34°	48° 33°	48° 3320°	54° 37°

Watch NEWS 25 for weather changes throughout the day

If-Then-Else example

- Consider the class hierarchy

$$P$$

A   B

- ... 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

$$O \vdash e_0 : \text{Bool}$$
$$O \vdash e_1 : T_1$$
$$O \vdash e_2 : T_2$$

$$O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \textbf{lub}(T_1, T_2)$$

[If-Then-Else]

Case

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

$$O \vdash e_0 : T_0$$

$$O[T_1/x_1] \vdash e_1 : T_1'$$

$$...$$

$$O[T_n/x_n] \vdash e_n : T_n'$$

———————————————————————————————— [Case]

$$O \vdash \text{case } e_0 \text{ of } x_1:T_1 \Rightarrow e_1;$$

$$...; x_n : T_n \Rightarrow e_n; \text{ esac} : lub(T_1',...,T_n')$$

Method Dispatch

- There is a problem with type checking method calls:

$$O \vdash e_0 : T_0$$

$$O \vdash e_1 : T_1$$

$$...$$

$$O \vdash e_n : T_n$$

$$\frac{}{O \vdash e_0.f(e_1,...,e_n) : ?} \text{[Dispatch]}$$

- 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:

$$M(C,f) = (T_1, . . . T_n, T_{ret})$$

means in class C there is a method f

$$f(x_1:T_1, . . ., x_n:T_n): T_{ret}$$

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:

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

  - Only the dispatch rules uses M

The Dispatch Rule Revisited

$$O, M \vdash e_0 : T_0$$

Check receiver object e_0

$$O, M \vdash e_1 : T_1$$

$$...$$

$$O, M \vdash e_n : T_n$$

Check actual arguments

$$M(T_0, f) = (T_1',...,T_n', T_{n+1}')$$

Look up formal argument types T_i'

$$T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n)$$

[Dispatch]

$$O, M \vdash e_0.f(e_1,...,e_n) : T_{n+1}'$$

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.)

$$O, M \vdash e_0 : T_0$$

$$O, M \vdash e_1 : T_1$$

$$\dots$$

$$O, M \vdash e_n : T_n$$

$$T_0 \leq T$$

$$M(T, f) = (T_1',\dots,T_n', T_{n+1}')$$

$$\frac{T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n)}{O, M \vdash e_0@T.f(e_1,\dots,e_n) : T_{n+1}'} \quad \text{[Static Dispatch]}$$

How should we handle 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? ☒

☐ <u>D</u>on't ask me again

[<u>O</u>K] [<u>Y</u>es] [<u>N</u>o] [<u>C</u>ancel]

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

Recall: Soundness

Soundness theorem for the Cool type system:

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

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;
    }
  };
};
```

But there is **disaster lurking** in the type system!

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

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

$$a : Stock \leftarrow (new\ Stock).inc\ ()$$

- 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
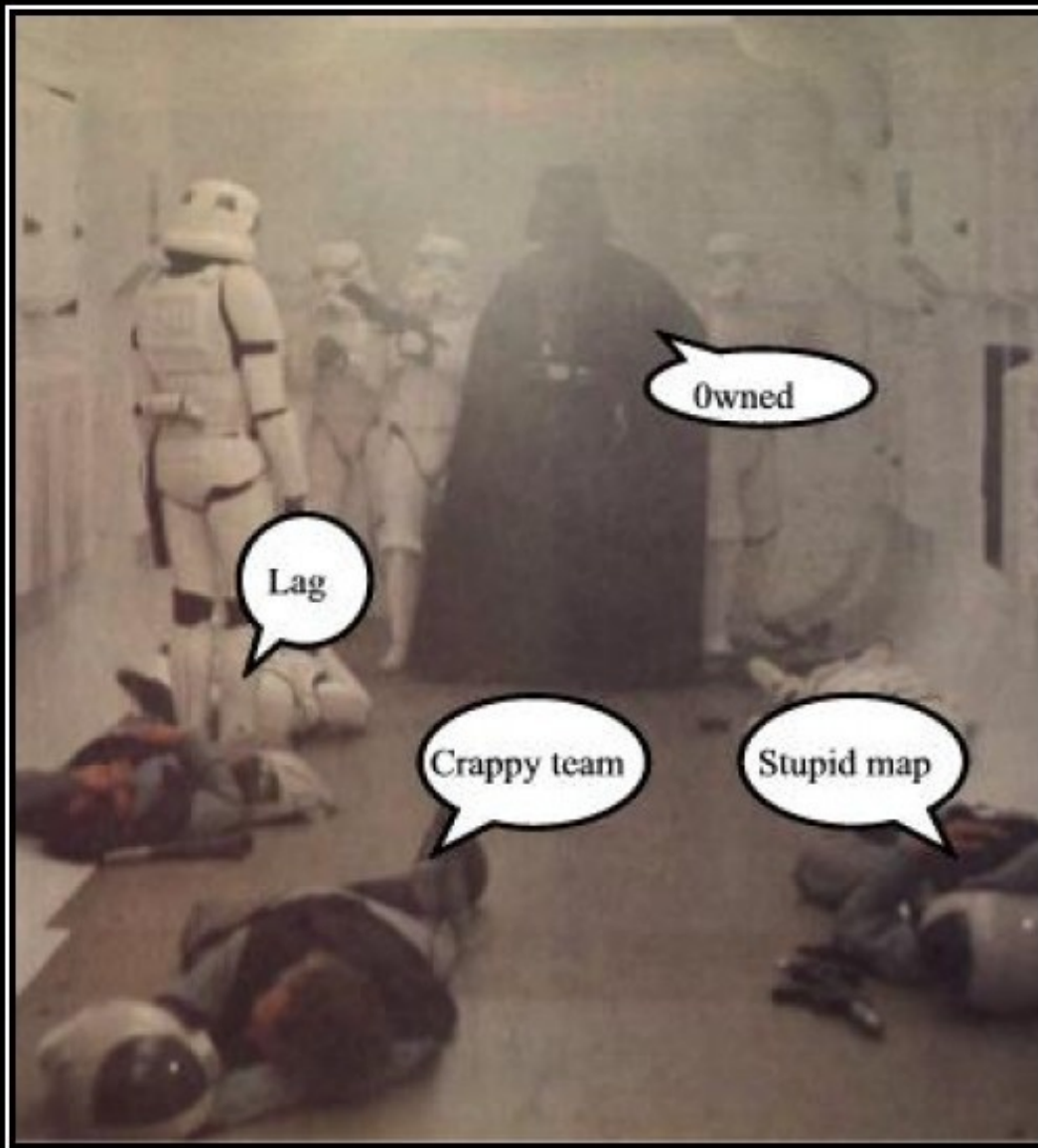
ONLINE GAMING

Get your excuses ready beforehand.
You're going to need them.

#26

I Need A Hero!
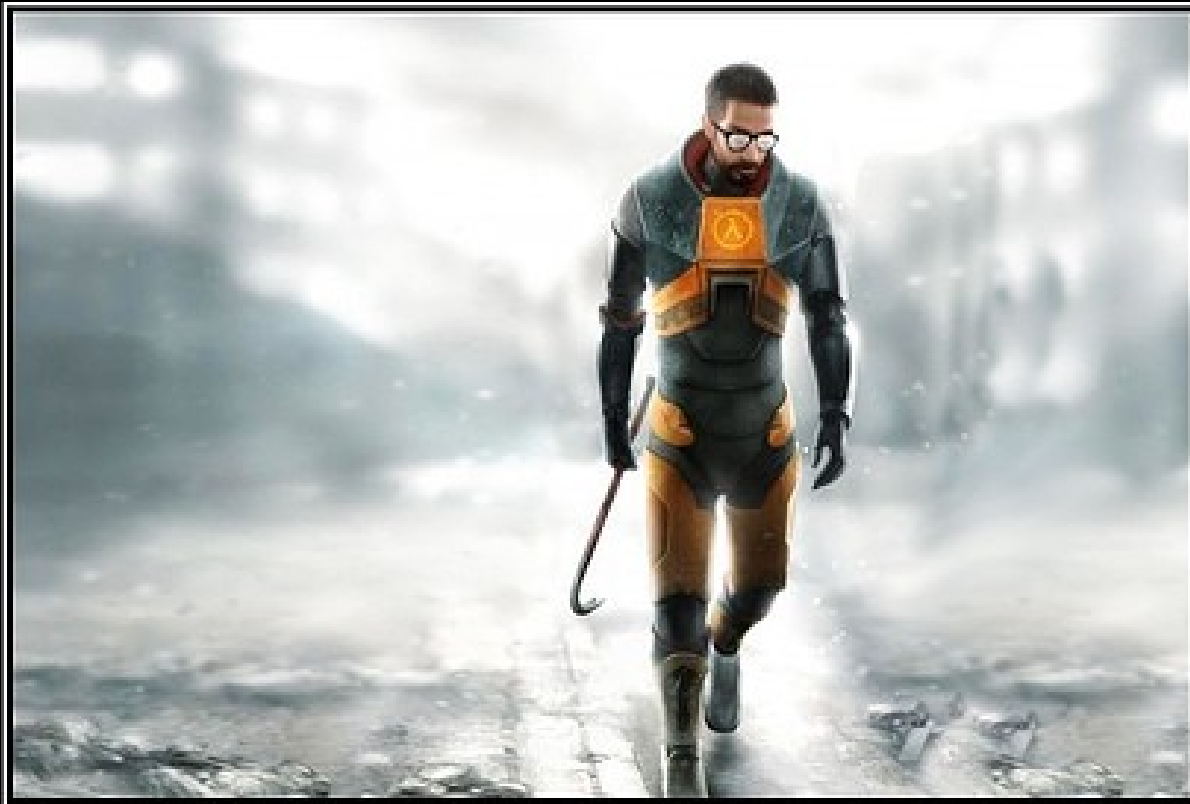
Type Systems

One tool. One million uses.

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 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 as a Tool

- 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:

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

- 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
  - $lub(T_1, T_2)$ the least-upper bound of T_1 and T_2

- We must extend these operations to handle SELF_TYPE
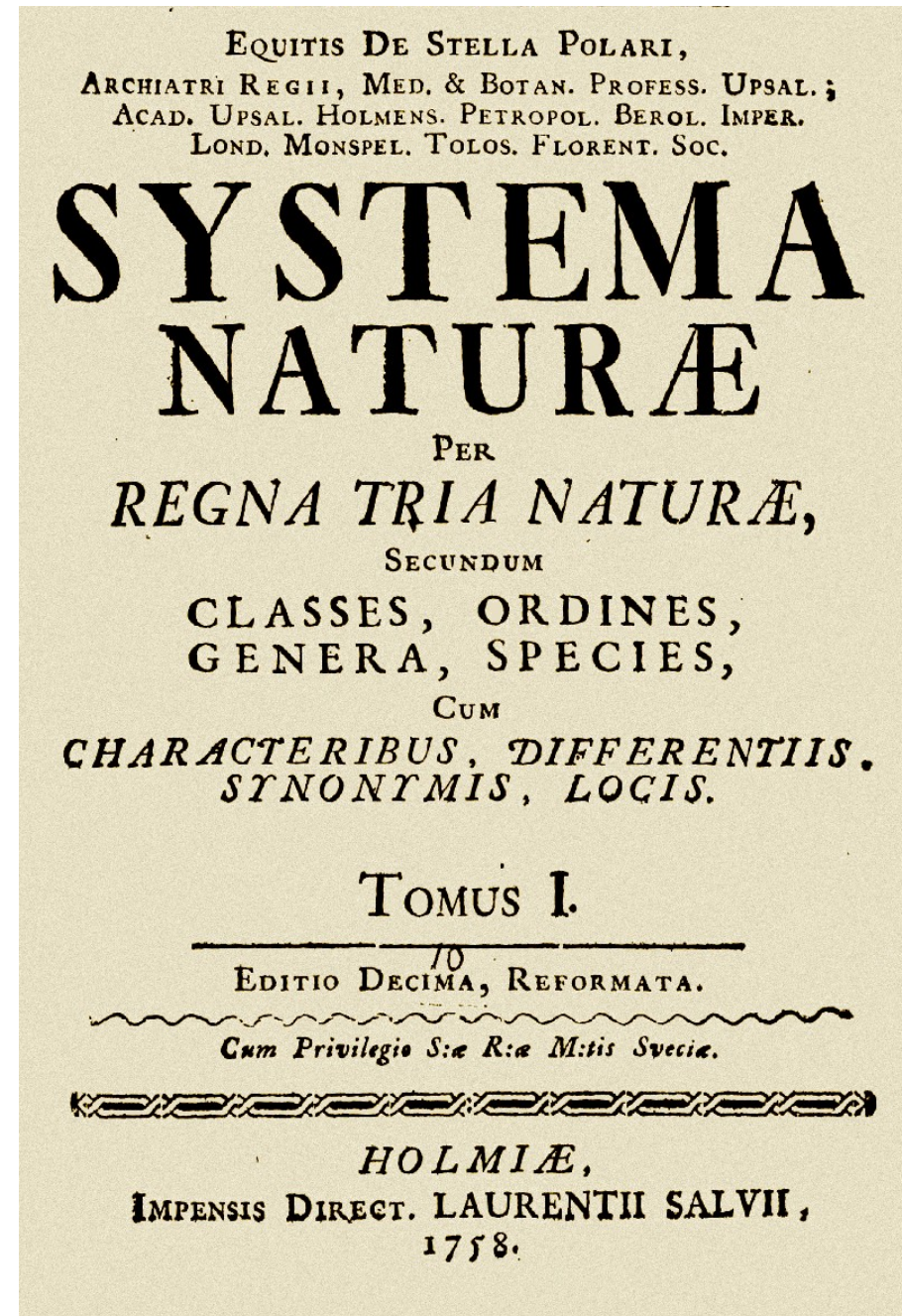
- Might take some time ...
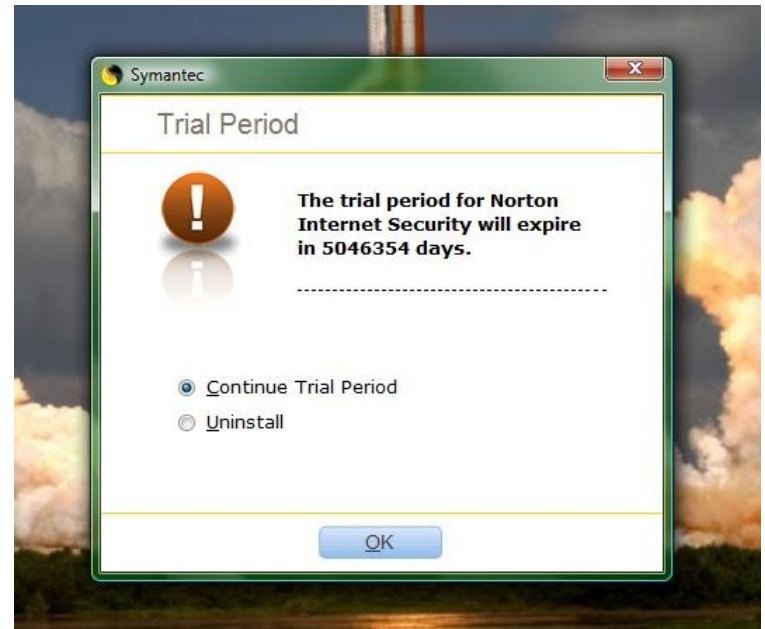
Medieval History

- This collection of verse and prose tales by Geoffrey Chaucer describes the stories told by a group of travelers. The stories present an oblique critique of society and the church. It was influential in promoting the English vernacular (as opposed to the more stylish French or Latin) as a vehicle for literature.

Medical History

- This 18th century Swedish botanist introduced the modern taxonomy used classify plants and animals. His influential *Systema Naturae* spearheaded and popularized the use of "two word" descriptors: a generic name (genus) and a specific name (species).

EQUITIS DE STELLA POLARI,
ARCHIATRI REGII, MED. & BOTAN. PROFESS. UPSAL.;
ACAD. UPSAL. HOLMENS. PETROPOL. BEROL. IMPER.
LOND. MONSPEL. TOLOS. FLORENT. SOC.

SYSTEMA NATURÆ

PER

REGNA TRIA NATURÆ,

SECUNDUM

CLASSES, ORDINES, GENERA, SPECIES,

CUM

CHARACTERIBUS, DIFFERENTIIS, SYNONYMIS, LOCIS.

TOMUS I.

EDITIO DECIMA, REFORMATA.

Cum Privilegio S:æ R:æ M:tis Sveciæ.

HOLMIÆ,
IMPENSIS DIRECT. LAURENTII SALVII,
1758.

- This 1983 adventure game designed by Roberta Williams described Sir Graham's attempts to recover the three magical treasures of Daventry and become the next king. It featured a parser for simple textual commands (e.g., "get carrot") and spawned numerous sequels.

Real-World Languages

- This is the second-largest Slavic language (after Russian but ahead of Ukranian). It features an extended Latin alphabet, high inflection, no articles, free word order, and mostly S-V-O sentences. Stanisław Lem is the most famous science fiction and fantasy writer in this language.

Extending $\leq$

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

- $\text{SELF\_TYPE}_C \leq T$ 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
- $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
  - 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.)

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

  Note: SELF_TYPE_C can denote any subtype of C.

- $T \leq T'$ (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 except SELF_TYPE

Again there are four cases:

- lub(SELF_TYPE$_C$, SELF_TYPE$_C$) = SELF_TYPE$_C$

- lub(SELF_TYPE$_C$, T) = lub(C, T)

  This is the best we can do because SELF_TYPE$_C \leq$ C

- lub(T, SELF_TYPE$_C$) = lub(C, T)

- 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:

- class T inherits T' {…}
  - T, T' cannot be SELF_TYPE
  - Because SELF_TYPE is never a dynamic type

- x : T
  - T can be SELF_TYPE
  - An attribute whose type is $SELF\_TYPE_C$

Where Can SELF_TYPE Appear in COOL?

1. let x : T in E
   - T can be SELF_TYPE
   - x has type $SELF\_TYPE_C$
2. new T
   - T can be SELF_TYPE
   - Creates an object of the same type as self
- $m@T(E_1,...,E_n)$
   - 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)

OMC = "Oh My Cool!" ?

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:

$$O(id) = T_0$$
$$O,M,C \vdash e_1 : T_1$$
$$T_1 \leq T_0$$

$$\overline{O,M,C \vdash id \leftarrow e_1 : T_1}$$

What's Different?

- Recall the old rule for dispatch

$$O,M,C \vdash e_0 : T_0$$

$$\ldots$$

$$O,M,C \vdash e_n : T_n$$

$$M(T_0, f) = (T_1',\ldots,T_n',T_{n+1}')$$

$$\color{red}{T_{n+1}' \neq \textbf{SELF\_TYPE}}$$

$$T_i \leq T_i' \qquad 1 \leq i \leq n$$

$$\overline{}$$

$$O,M,C \vdash e_0.f(e_1,\ldots,e_n) : T_{n+1}'$$

The Big Rule for SELF_TYPE

- If the return type of the method is SELF_TYPE then the type of the dispatch is the type of the dispatch expression:

$$O,M,C \vdash e_0 : T_0$$

$$\ldots$$

$$O,M,C \vdash e_n : T_n$$

$$M(T_0, f) = (T_1', \ldots, T_n', \textbf{SELF\_TYPE})$$

$$\frac{T_i \leq T_i' \qquad 1 \leq i \leq n}{O,M,C \vdash e_0.f(e_1, \ldots, e_n) : T_0}$$

What's Different?

- Note this rule handles the Stock example

- Formal parameters cannot be SELF_TYPE

- Actual arguments can be SELF_TYPE

  – The extended $\leq$ 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

$$O,M,C \vdash e_0 : T_0$$

$$\ldots$$

$$O,M,C \vdash e_n : T_n$$

$$T_0 \leq T$$

$$M(T, f) = (T_1',\ldots,T_n',T_{n+1}')$$

$$\color{red}{T_{n+1}' \neq \text{SELF\_TYPE}}$$

$$T_i \leq T_i' \qquad 1 \leq i \leq n$$

$$\overline{O,M,C \vdash e_0@T.f(e_1,\ldots,e_n) : T_{n+1}'}$$

Static Dispatch

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

$$O,M,C \vdash e_0 : T_0$$

$$\dots$$

$$O,M,C \vdash e_n : T_n$$

$$T_0 \leq T$$

$$M(T, f) = (T_1',\dots,T_n',\textbf{SELF\_TYPE})$$

$$T_i \leq T_i' \qquad 1 \leq i \leq n$$

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

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 body appears
  - *Not* the class in which the call appears!

- The static dispatch class cannot be SELF_TYPE

New Rules

- There are two new rules using SELF_TYPE

$$\frac{}{\textbf{O,M,C} \vdash \textbf{self : SELF\_TYPE}_\text{C}}$$

$$\frac{}{\textbf{O,M,C} \vdash \textbf{new SELF\_TYPE : SELF\_TYPE}_\text{C}}$$

- 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 $\leq$ 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
  - Without allowing in any "bad" programs
- 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

- PA4c Checkpoint Due
- RS4 Recommended Next