Status

- 2 weeks to go on project 3
- Read/skim Ch. 6 by Monday, May 11.
- Read Sec. 7.1 through 7.4 by Monday, May 18
Handling the SELF_TYPE
(Un)-Soundness

- Type preservation
  - Types in programs should remain invariant under evaluation or reduction rules of a language
- Progress
  - Programs should never enter undefined states where no transitions are possible.
- Related to memory safety (copying arbitrary bits between locations)
- Examples:
  - Improper allocation/deallocation of memory
  - Dangling pointers
  - C allows many unchecked conversions
  - Linked libraries can sometimes cause problems even at runtime.
An Example

class Count {
    i : int ← 0;
    inc() : Count {
        i ← i + 1;
        self;
    }
};

• Class Count incorporates a counter
• The inc method works for any subclass
• But there is disaster lurking in the type system
An Example

• Consider a subclass Stock of Count
  class Stock inherits Count {
    name : String; -- name of item
  };

• And the following use of Stock:
  class Main {
    Stock a ← (new Stock).inc();  Type checking error
    ... a.name ...
  };

What Went Wrong?

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

- We will extend the type system
- Insight:
  - `inc()` returns “self”
  - Therefore the return value has same type as “self”
  - Which could be Count or any subtype of Count!
  - 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

- 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:
  - $O, M ⊢ (\text{new Count}).\text{inc} : \text{Count}$
  - $O, M ⊢ (\text{new Stock}).\text{inc} : \text{Stock}$
- The program from before is now well typed.
Notes about SELF_TYPE

- SELF_TYPE is not a dynamic type
- It is only 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 (Examples)

- What can 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:

\[ \text{dynamic} \_\text{type}(E) = \text{dynamic} \_\text{type}(\text{self}) \leq C \]

• 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
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 \( \text{SELF\_TYPE}_C \) by \( C \)
• But that would disallow some programs

This suggests one way to handle \( \text{SELF\_TYPE} \):
• Replace all occurrences of \( \text{SELF\_TYPE}_C \) by \( C \)
• This would be correct but it is like not having \( \text{SELF\_TYPE} \) at all.
Operations on SELF_TYPE

- Recall the operations on types
  - $T_1 \leq T_2 \quad T_1$ is a subtype of $T_2$
  - $\text{lub}(T_1, T_2) \quad$ 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 $C \leq T$
     - $\text{SELF\_TYPE}_C$ can be any subtype of $C$
     - This includes $C$ itself
     - Thus this is the most flexible rule we can allow
  2. $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$
     - $\text{SELF\_TYPE}_C$ is the type of the “self” expression
     - In Cool we never need to compare SELF_TYPEs coming from different classes.
3. $T \leq \text{SELF\_TYPE}_C$
   - Note $\text{SELF\_TYPE}_C$ can denote any subtype of $C$

4. $T \leq T'$ (according to the rules from before)
   - Based on these rules we can extend lub ...
Extending lub(T, T’)

• Let T and T’ be any types but SELF_TYPE. Again, there are four cases:
  1. lub(SELF_TYPEC, SELF_TYPEC) = SELF_TYPEC
  2. lub(SELF_TYPEC, T) = lub(C, T)
     • This is the best we can do because SELF_TYPEC ≤ C
  3. lub(T, SELF_TYPEC) = 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
Where Can SELF_TYPE Appear in COOL?

3. let x : T in E
   - T can be SELF_TYPE
   - x has type SELF_TYPE

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 \texttt{SELF\_TYPE} for each language construct.
- Most of the rules remain the same except that $\leq$ and lub are the new ones.

\[
\begin{align*}
O(id) &= T_0 \\
O &\vdash e_1 : T_1 \\
T_1 &\leq T_0 \\
\hline
O &\vdash id \leftarrow e_1 : T_1
\end{align*}
\]
What’s Different?
Recall the Old Rule for Dispatch

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

\[
O, M, C \vdash e_0.f(e_1,...,e_n) : T_{n+1}'
\]
What’s Different?

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

\[
\begin{align*}
O, M, C & \vdash e_0 : T_0 \\
& \quad \ldots \\
& \quad O, M, C \vdash e_n : T_n \\
M(T_0, f) & = (T_1', \ldots T_n', \text{SELF\_TYPE}) \\
T_i & \leq T_i' \quad (\text{for } 1 \leq i \leq n) \\
\hline
O, M, C & \vdash e_0.f(e_1, \ldots, e_n) : T_0
\end{align*}
\]
What’s Different?

• 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
Static Dispatch

- Recall: `<expr_0>@<type>.id(<expr_1>,...,<expr_n>)`
- Provides a way of accessing parent classes that have been hidden by redefinitions in child classes.
- E.g., Class B has method f(), class C redefines f(), and we’re in class C, but want to call the f() in B, not C
- In normal dispatch, `<expr_0>` would determine the class/method used.
- In static dispatch, `<type>` determines this.
- E.g., `e@B.f()` invokes f() in class B on the object that is the value of e.
- The (static) type of `<expr_0>` must conform to `<type>`.

Friday, May 8, 2009
Static Dispatch
Recall the Old Rule for Static Dispatch

\[
O, M, C \vdash e_0 : T_0
\]

\[
...\]

\[
O, M, C \vdash e_n : T_n
\]

\[
T_0 \leq T
\]

\[
M(T, f) = (T_1',...T_n',T_{n+1}', \text{SELF_TYPE})
\]

\[
T_{n+1}' \neq \text{SELF_TYPE}
\]

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

\[
O, M, C \vdash e_0@T.f(e_1,...,e_n) : T_{n+1}'
\]
Static Dispatch

• If the return type of the method is SELF_TYPE, we have:

\[
\begin{align*}
O, M, C & \vdash e_0 : T_0 \\
& \vdash \ldots \\
O, M, C & \vdash e_n : T_n \\
T_0 & \leq T \\
M(T, f) & = (T_1', \ldots T_n', \text{SELF\_TYPE}) \\
T_i & \leq T_i' \quad (\text{for } 1 \leq i \leq n) \\
O, M, C & \vdash e_0@T.f(e_1, \ldots, e_n) : T_0
\end{align*}
\]
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 method appears.
  - The static dispatch class cannot be SELF_TYPE
New Rules

• There are two new rules using SELF_TYPE

\[
\begin{align*}
O, M, C & \vdash \text{self : SELF\_TYPE}_C
\end{align*}
\]

\[
\begin{align*}
O, M, C & \vdash \text{new SELF\_TYPE : SELF\_TYPE}_C
\end{align*}
\]

• There are a number of other places where SELF\_TYPE is used
Where SELF_TYPE Cannot Appear in COOL?

\[
m(x : T) : T' \{...\}
\]

- Only \(T'\) can be SELF_TYPE
- What could go wrong if \(T\) were SELF_TYPE?

```plaintext
class A { comp(x : SELF_TYPE) : Bool {...}; }
class B inherits A {
    b: int;
    comp(x : SELF_TYPE) : Bool { ... x.b ...}; }
...
let x : A ← new B in ... x.comp(new A); ...
...
```
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 only be used 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 lectures were Cool-specific.

- Other languages have very different rules.

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.
Semantic Analysis

• Semantic analysis (type checking and scoping) check for errors.

• They also determine the behavior of the resulting program
Translate (Very Roughly)

class Main {
    main():Int { 0 };
};

.data
    .align 2
    .globl class_nameTab
    .globl Main_protObj
    .globl Int_protObj

Main_init:
    addiu $sp $sp -12
    sw $fp 12($sp)
    sw $s0 8($sp)
    sw $ra 4($sp)
    addiu $fp $sp 4
    move $s0 $a0
A Huge Step

- General problem is bridging the “semantic gap”
- Cool is high-level
- MIPS is low-level
- Size of gap implies we might want to bridge in smaller steps
- Also, these two operate via different paradigms
  - names vs. memory addresses & registers
  - calls vs. gotos
  - typed vs. untyped
- Need a “model” for bridging this
Bridging Concept Includes

- A call stack for managing call/return
- A call frame for managing local data
- base + offset addressing for variables
  - registers used as temporaries, not long-term storage
- calling conventions for keeping this data straight
- A memory heap for dynamically allocated memory