Type Checking in Cool (continued)

Lecture 17

Dr. Sean Peisert – ECS 142 – Spring 2009
Midterm

- High = 79%
- Mean = 59%
- Median = 60%
- StdDev = 12%
• Midterm and Parser grades this week...
• 2.5 weeks to go on project 3
• You read Ch. 5 last week....
• Read/skim Ch. 6 by Monday, May 11.
• Read Sec. 7.1 through 7.4 by Monday, May 18
Assignment

Very similar to let:

\[ O(\text{id}) = T_0 \]

\[ O \vdash e_0 : T_1 \]

\[ T_1 \leq T_0 \]

\[ O \vdash \text{id} \leftarrow e_0 : T_1 \]

[Assign]
Initialized Attributes

• Let $O_c(x) = T$ for all attributes $x:T$ in class C

• Attribute initialization is similar to let, except for the scope of names

\[
O_c(id) = T_0
\]

\[
O_c \vdash e_0 : T_1
\]

\[
T_1 \leq T_0
\]

\[
O_c \vdash id \leftarrow e_0 : T_1
\]
If-Then-Else

• Consider:

\[
\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi}
\]

• The result can be either \( e_1 \) or \( e_2 \)

• The type is either \( e_1 \)'s type or \( e_2 \)'s type

• The best we can do 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

- $\text{lub}(X, Y)$, the least upper bound of $X$ 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 least common ancestor in the inheritance tree.
If-Then-Else

\[ \text{\texttt{O \vdash e_0 : Bool}} \]

\[ \text{\texttt{O \vdash e_1 : T_1}} \]

\[ \text{\texttt{O \vdash e_2 : T_2}} \]

\[ \text{\texttt{O \vdash if e_0 \then e_1 \else e_2 fi : lub(T_1, T_2)}} \]
Case

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

\[ \begin{align*}
O & \vdash e_0 : T_0 \\
O[T_1/x_1] & \vdash e_1 : T_1' \\
& \cdots \\
O[T_n/x_n] & \vdash e_n : T_n'
\end{align*} \]

\[ O \vdash \text{case } e_0 \text{ of } x_1 \Rightarrow e_1; \ldots; x_n : T_n \Rightarrow e_n; \text{ esac: } \text{lub}(T_1', \ldots, T_n') \]
Method Dispatch

• There is a problem with type checking method calls:
  \[ O \vdash e_0 : T_0 \]
  \[ O \vdash e_1 : T_1 \]
  ...
  \[ O \vdash e_n : T_n \]

  \[ O \vdash e_0.f(e_1, ..., e_n) : ? \]

• 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 $X$ and an object $X$ 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:
  \[ O, M \vdash e_1 : T_1 \]
  \[ O, M \vdash e_2 : T_2 \]
  \[ O, M \vdash e_1 + e_2 : \text{Int} \]  
  [Add]

- Only the dispatch rules uses M.
The Dispatch Rule Revisited

\[
\begin{align*}
O \vdash e_0 &: T_0 \\
O \vdash e_1 &: T_1 \\
& \ldots \\
O \vdash e_n &: T_n \\
M(T_0, f) &= (T'_1, \ldots, T'_n, T'_{n+1}) \\
T_i &\leq T'_i \quad (\text{for } 1 \leq i \leq n) \\
O, M \vdash e_0.f(e_1, \ldots, e_n) &: T'_{n+1}
\end{align*}
\]
Static Dispatch

- Static dispatch is a variation on normal dispatch
- The method found in the class explicitly named by the programmer
- The inferred type of the dispatch expression must conform to the specified type.
Static Dispatch

\[
\begin{align*}
O & \vdash e_0 : T_0 \\
O & \vdash e_1 : T_1 \\
& \quad \ldots \\
O & \vdash e_n : T_n \\
T_0 & \leq T \\
M(T_0, f) & = (T'_1, \ldots, T'_n, T'_{n+1}) \\
T_i & \leq T'_i \quad (\text{for } 1 \leq i \leq n) \\
O, M & \vdash e_0@T.f(e_1, \ldots, e_n) : T'_{n+1}
\end{align*}
\]
Handling the SELF_TYPE
Recall that type systems have two conflicting goals:

- Give flexibility to the programmer.
- Prevent valid programs to “go wrong”


An active line of research is in the area of inventing more flexible type systems while preserving soundness.
(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.
Dynamic and Static Types. Review.

- The **dynamic type** of an object is the class C that is used in the “new C” expression that creates the object.
- 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.
Dynamic and Static Types Review

• 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 the 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
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 ... 
  };

Wednesday, May 6, 2009
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 ⊢ (new Count).inc : Count`
  - `O, M ⊢ (new Stock).inc : Stock`
- The program from before is now well typed.
Notes about SELF_TYPE

• SELF_TYPE is not a dynamic type
  • It is 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:

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

- Note: the meaning of SELF_TYPE depends on where it appears.
  - We write SELF_TYPE_\text{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 \(\text{SELF\_TYPE}_C\) 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
  - $T_1 \leq T_2$  \hspace{1cm} $T_1$ is a subtype of $T_2$
  - lub($T_1, T_2$) \hspace{1cm} 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. \texttt{SELF\_TYPE}_C $\leq$ $T$ if $C \leq T$
     - \texttt{SELF\_TYPE}_C can be any subtype of $C$
     - This includes $C$ itself
     - Thus this is the most flexible rule we can allow
  2. \texttt{SELF\_TYPE}_C $\leq$ \texttt{SELF\_TYPE}_C
     - \texttt{SELF\_TYPE}_C is the type of the “self” expression
     - In Cool we never need to compare \texttt{SELF\_TYPE}s coming from different classes.
Extending $\leq$

1. $T \leq \text{SELF\_TYPE}_C$
   - Note $\text{SELF\_TYPE}_C$ can denote any subtype of $C$

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

- Let \( T \) and \( T' \) be any types but SELF\_TYPE. Again, there are four cases:
  1. \( \text{lub}(\text{SELF\_TYPE}_C, \text{SELF\_TYPE}_C) = \text{SELF\_TYPE}_C \)
  2. \( \text{lub}(\text{SELF\_TYPE}_C, T) = \text{lub}(C, T) \)
     - This is the best we can do because \( \text{SELF\_TYPE}_C \leq C \)
  3. \( \text{lub}(T, \text{SELF\_TYPE}_C) = \text{lub}(C, T) \)
  4. \( \text{lub}(T, T') \) defined as before
Where Can 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

Wednesday, May 6, 2009
Where Can SELF_TYPE Appear in COOL?

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

4. new T
   - T can be SELF_TYPE
   - Creates an object of the same type as self

5. m@T(E₁,...,Eₙ)
   - 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.

\[
\begin{align*}
O(id) &= T_0 \\
O \vdash e_1 : T_1 \\
T_1 &\leq T_0 \\
\hline
O \vdash id \leftarrow e_1 : T_1
\end{align*}
\]