CS143: Semantic Analysis II

David L. Dill
Stanford University
Semantic Analysis II

- Subtyping
- Recursive Traversal
- Method Context and Dispatch
- SELF_TYPE
Subtyping
Subtyping

Partial order for inheritance

\[ T \leq T \] (reflexive)

\[ T \leq T' \text{ if } T \text{ inherits from } T' \]

\[ T \leq T'' \text{ if } T \leq T' \text{ and } T' \leq T'' \] (transitive)
\[ O \vdash e_0 : T_0 \]
\[ O[T/x] \vdash e_i : T_i \]
\[ T_0 \leq T \]

\[ O \vdash \text{let } x : T \leftarrow e_0 \text{ in } e_i : T_i \]

Allows \( e_0 \) to have any subtype of declared type of \( x \).
Assignment

\[ O(x) = T_0 \]
\[ O \vdash e_1 : T_1 \]
\[ T_1 \leq T_0 \]

\[ O \vdash x \leftarrow e_1 : T_1 \]

\[ ? \]

Allows \( e_1 \) to be any subtype of declared type of \( x \)
Assignment

\[ O(x) = T_0 \]
\[ O \vdash e_1 : T_1 \]
\[ T_1 \leq T_0 \]
\[ O \vdash x \leftarrow e_1 : T_1 \]

← assignment returns value of \( e_1 \), which has type \( T_i \).
Project

$T_1 \leq T_2$ iff $T_2$ is above $T_1$ in inheritance tree

Incomparable

$T_3 \not\leq T_1$

$T_1 \not\leq T_3$
If-then-else

if $e_0$ then $e_1$ else $e_2$

At compile-time, don’t know which of $e_1, e_2$ will be returned.

$e_1 : T_1$, $e_2 : T_2$ — need a type that is $T_1$ or $T_2$
Least Upper Bound

\[ \text{lub} \left( T_1, T_2 \right) = \text{least type } T_3 \text{ such that } T_1 \leq T_3 \text{ and } T_2 \leq T_3 \]

"least": \( \forall T_4 \ (T_1 \leq T_4 \land T_2 \leq T_4 \rightarrow T_3 \leq T_4) \)

Notation \( T_1 \cup T_2 \) ("join" of \( T_1, T_2 \))
Implementation

$T_1 \cup T_2$ is least common ancestor of $T_1, T_2$

Object

$T_3 \leftarrow \text{lowest node that is ancestor of both } T_1, T_2$

$T_3 = T_1 \cup T_2$
Method Context and Dispatch
Method Dispatch

\[ 0 \vdash e_0 : T_0 \]
\[ 0 \vdash e_1 : T_1 \]
\[ \vdots \]
\[ 0 \vdash e_n : T_n \]
\[ 0 \vdash e_0 \cdot f(e_1, \ldots, e_n) : ? \]
Method Dispatch

\[ e_0 \cdot f(e_1, \ldots, e_n) : ? \]

Need a map from class and method name to the type information for the method.

\[ M(C, f) = (T_1, T_2, \ldots, T_n, T_{n+1}) \]

\[ \uparrow \quad \uparrow \]

class method name

\[ \text{types of formals} \quad \text{return type} \]
Method Dispatch

\[ O, M \vdash e_0 : T_0 \]
\[ O, M \vdash e_i : T_i \]
\[ \vdots \]
\[ O, M \vdash e_n : T_n \]

\[ M(T_0, f) = \langle T'_1, T'_2, \ldots, T'_n, T'_{n+1} \rangle \]

\[ T_1 \leq T_i \quad \text{for all } 1 \leq i \leq n \]

\[ O, M \vdash e_0 \cdot f(e_1, \ldots, e_n) : T_{n+1} \]
Static Dispatch

e_0 \oplus T.f(\_\_\_)

↑

call method e_0 in class T

Class of e_0 must inherit from T.
\[ O, M \vdash e_0 : T_0 \]
\[ O, M \vdash e_i : T_i \]
\[ O, M \vdash e_n : T_n \]

\[ T_0 \leq T \quad \text{type of } e_0 \text{ inherits from } T \]

\[ M(T_0, f) = (T'_1, T'_2, \ldots, T'_n, T_{n+1}) \]

\[ T_i \leq T'_i \quad \text{for all } 1 \leq i \leq n \]

\[ O, M \vdash e_0 @ T. f(e_1, \ldots, e_n) : T_{n+1} \]
Rules involving SELF-TYPE require knowing current class.

Additional component of type environment:

- C — the class we are in.
Final form of type rules

\[ O, m, c \vdash e : T \]

E.g.

\[
\frac{O, m, c \vdash e_1 : \text{Int} \quad O, m, c \vdash e_2 : \text{Int}}{O, m, c \vdash e_1 + e_2 : \text{Int}}
\]
Recursive Traversal
Implementation

Environment is passed down AST
Argument to recursive function

Types of expressions are computed bottom-up.
Recursive Type Check Example

\[ tc(\text{env}, e_1 + e_2) : \]
\[ T_1 = tc(\text{env}, e_1) ; \]
\[ T_2 = tc(\text{env}, e_2) ; \]
\[ \text{check } T_1 == T_2 == \text{Int} \]
\[ \text{return } \text{Int} \]
Static and Dynamic Types in Cool

Dynamic type of an object — the class `C` in the new `C` call that created it.

"Run-time type"

Cool has dynamic types, but no run-time type errors

Static type — the type inferred by the compiler
class A  

class B  

class Main  

x: A <- new A;  

...  

x <- new B;  

...  

A variable of static type A can hold a value of dynamic type B iff B <= A.
Soundness

\( \forall E. \ dynamic\text{-}type(E) \leq static\text{-}type(E) \)

- Operations on values of type \( T_1 \) are always defined for \( T_2 \leq T_1 \)
  - method dispatch
  - attribute read/write
- Subclasses never remove features
- Redefined methods in subclasses must have same types.
SELF_TYPE

Allows more accurate static typing.

(static types are closer to dynamic types)

Intuition: SELF_TYPE is the type of "self".
Example:

Object has a generic copy() method.

```javascript
x = Object();
x.copy() // return a copy of x.
```

What is the return type of the copy method?

```javascript
x.copy() // Object (since x.copy returns an object)
```
Copy is inherited by all classes, so we can use it to copy anything.

```java
Class A {
    ...
    3
    y: A <- new A;
    z: A <- y.copy();
}
```

Object

*Problem:
  Result type of "copy" is "too static"
  "copy" is "too static"

`type error.`
```java
class A {
    ...
}
y : A = new A;
z : A = (A) y.copy();
```

Most languages: User would have "cast" to type A ("downcasting")

Problem: User can lie, resulting in run-time error.
Object

copy() : SELF_TYPE

Return type is "type of current class"

Still a static type, but static results are closer to dynamic type.
Copy(): SELF_TYPE

class A { ... }

y: A ← new A;

z: A ← y.copy()  

↑

type of

y is A

↑

so return

type is A, not Object.
Static Method Dispatch

\[ O, M, C \vdash e_0 : T_0 \]
\[ O, M, C \vdash e_i : T_i \]
\[ \vdots \]
\[ O, M, C \vdash e_n : T_n \]

\[ M(T_0, f) = (T_{i_1}', T_{i_2}', \ldots, T_{i_n}') \]

\[ T_i \leq T_{i'} \quad \text{for all} \quad 1 \leq i \leq n \]

\[ O, M, C \vdash e_0 \circ @ T.f(e_1, \ldots, e_n) : T_0 \]

\[ \text{used to be } T_{n+1} \]

\[ \mathbf{SELF-TYPE} \]

\[ \text{f is in } T_i. \hat{\downarrow} \]

\[ \text{Why is this ok?} \]

\[ \text{Because f returns type of self: } T_0. \]
Notation: SELF_TYPE\_c - use of SELF_TYPE is the body of definition of class C.

SELF_TYPE\_c \leq C
Allowed/Disallowed Uses of SELF_TYPE

Class T inherits T'
    ↕
  can't be SELF_TYPE

Let x: SELF_TYPE in E ← OK

new SELF_TYPE ← OK

e_b@ T (e_1, e_2, ..., e_n) ← T cannot be SELF_TYPE
Attribute
class w {
    x: SELF_TYPE;
}

In subclass, x would have the type of the subclass.
Formals Cannot Be SELF_TYPE

\[ e_0 \ (x : T) : T \ ' \ \{ \ ... \ \} \]

\[ \text{no SELF_TYPE} \rightarrow \text{SELF_TYPE OK} \]

Class A \ \{ f\ (x : \text{SELF\_TYPE}) \ ... \ \} \]

Class B \ inherits \ A \ \{ f\ (x : \text{SELF\_TYPE}) \ ... \ \}

let \ y : A \leftarrow \text{new} B \ \text{in} \ y. f\ (\text{new} A) ; \ ... \]

\[ \{ \text{static type} A, y = \text{new} A \ \text{ok} \}

\[ \{ \text{but dynamic type is} \ B, y = \text{new} A \ \not\text{ok} (\text{SELF\_TYPE is} \ B) \}

Violates soundness
Notation: SELF(TYPE)c - use of SELF_TYPE is the body of definition of class C.

SELF_TYPE_c is \not\, the same as C

Behaves differently in inherited methods.

Project note: type implementation just has one SELF_TYPE.
≤ on SELF_TYPE

SELF_TYPE_C ≤ C  \hspace{1cm} \text{SELF_TYPE might be } ≤ C \text{ when inherited method is called}

SELF_TYPE_C ≤ SELF_TYPE_C

\text{In COOL, never compare SELF_TYPES from different classes}

SELF_TYPE_C ≤ T \hspace{1cm} \text{if } C ≤ T

T ≠ SELF_TYPE_C \hspace{1cm} \text{if } T ≠ SELF_TYPE_C

T ≤ T' as before \hspace{1cm} \text{if } T, T' ≠ SELF_TYPE
hub and SELF_TYPE

SELF_TYPEe ⊨ SELF_TYPEe = SELF_TYPEe

SELF_TYPEe ⊨ T = CUT (T ≠ SELF_TYPE)

All we know is SELF_TYPEe ⊨ C

T ⊨ SELF_TYPEe = CUT

□ needs to be commutative

T ⊨ T' as before when T, T' ≠ SELF_TYPE
More Rules

\[ O, M, C \vdash \text{self} : \text{SELF-TYPE}_C \]

\[ O, M, C \vdash \text{new SELF-TYPE} : \text{SELF-TYPE}_C \]
Other rules remain the same

But use extended definitions of \( \leq \) and \( \mathbb{U} \)
Summary of SELF_TYPE

Extended ≤ , U do most of the work

Usage restricted for soundness

SELF_TYPE is always a subtype of current class, C

EXCEPT for return type from dispatch, where type may be unrelated to C.
Error Recovery
Goal: Report errors after the first one

\[ \text{let } y: \text{Int} \leftarrow x + 2 \text{ in } y + 3 \]

Type for undeclared \( x \) ?

If \( x: \text{Object} \), \( x + 2 \) will cause another error.

Reports too many errors.
Another Solution

Compiler stores "No-type" for erroneous expressions

No-type \leq C \ \text{for all types } C

so \ \text{No-type} \cup C \text{ is always } C

All operations/assignments are ok.

let \ y : \text{Int} \leftarrow x + 2 \text{ in } y + 3 \leftarrow \text{only one error}

\text{No-type} \cup \text{Int} \rightarrow \text{Int}
No - type

Types no longer a tree

Not a problem unless you have code that assumes it's a tree

Not required in project