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

$T \le T$ (reflexive)

$T \le T'$ if T inherits from T'

$T \le T''$ if $T \le T'$ and $T' \le T''$ (transitive)

$$\frac{O \vdash e_0 : T_0 \qquad O[T/x] \vdash e_1 : T_1 \qquad T_0 \leq T}{O \vdash \text{let } x : T \leftarrow e_0 \text{ in } e_1 : T_1}$$

Allows e_0 to have any <u>subtype</u> of declared type of x.

Assignment

$$\frac{O(x) = T_0 \qquad O \vdash e_1 : T_1 \qquad T_1 \leq T_0}{O \vdash x \leftarrow e_1 : T_1}$$

Allows e_1 to be any subtype of declared type of x

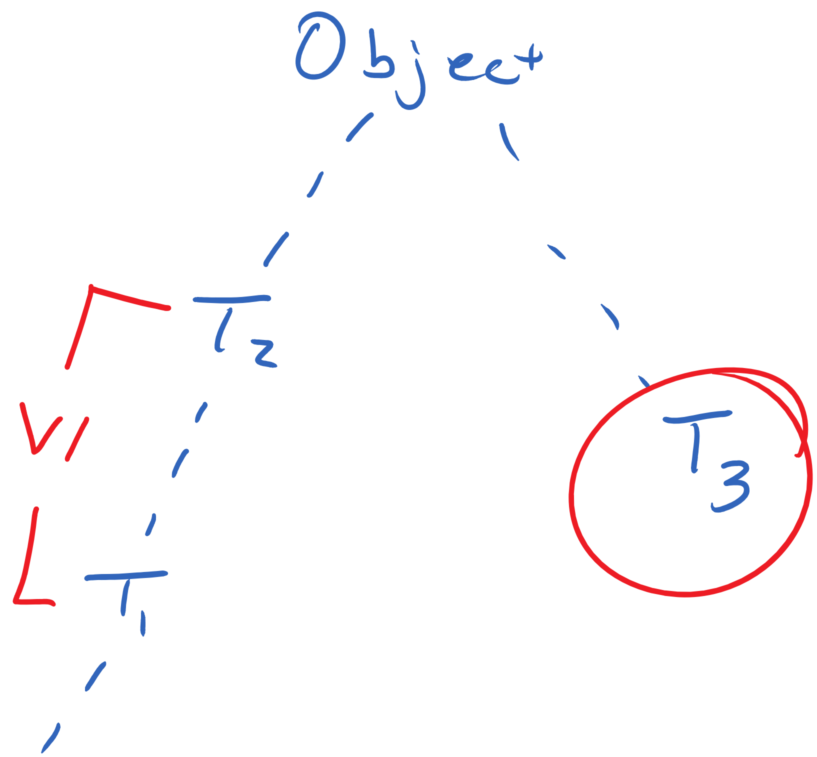
Assignment

$$\frac{O(x) = T_0 \qquad O \vdash e_1 : T_1 \qquad T_1 \leq T_0}{O \vdash x \leftarrow e_1 : T_1}$$

assignment returns value of e_1, which has type T_1.

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

$\text{lub}(T_1, T_2)$ = <u>least</u> type T_3 such that

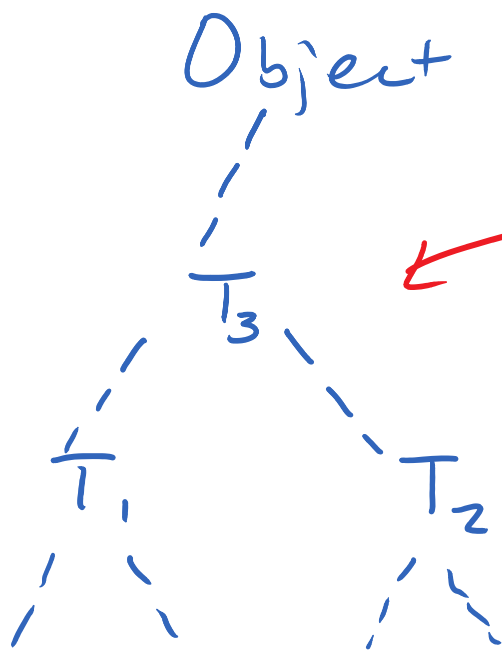
$T_1 \leq T_3$ and $T_2 \leq T_3$

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

Notation $T_1 \sqcup T_2$ ("join" of T_1, T_2)

Implementation

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



lowest node that is ancestor of both T_1, T_2

$T_3 = T_1 \sqcup T_2$

Method Context and Dispatch

Method Dispatch

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O \vdash e_1 : T_1 \\ \vdots \\ O \vdash e_n : T_n \end{array}}{O \vdash e_0.f(e_1, \ldots, e_n) : ?}$$

Method Dispatch

$$O \vdash e_0.f(e_1, \ldots, e_n) : ?$$

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

$$M(\underset{\text{class}}{C}, \underset{\text{method name}}{f}) = (\underbrace{T_1, T_2, \ldots T_n}_{\text{types of formals}}, \underbrace{T_{n+1}}_{\text{return type}})$$

Method Dispatch

$$\dfrac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \vdots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T_1', T_2', \ldots T_n', T_{n+1}) \\ T_i \leq T_i' \text{ for all } 1 \leq i \leq n \end{array}}{O, M \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}}$$

actual types are subtypes of formals.

declared types for formal parameters.

specified return type.

Static Dispatch

$$e_0 \,@\, T.f(\sim\!\sim)$$

call method e_0 in class T

Class of e_0 must inherit from T.

$$\frac{\begin{array}{l} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \vdots \\ O, M \vdash e_n : T_n \\ T_0 \leq T \\ M(T_0, f) = (T_1', T_2', \ldots, T_n', T_{n+1}) \\ T_i \leq T_i' \text{ for all } 1 \leq i \leq n \end{array}}{O, M \vdash e_0 @ T.f(e_1, \ldots, e_n) : T_{n+1}}$$

type of e_0 inherits from T

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.

$$\frac{O, M, C \vdash e_1 : Int \qquad O, M, C \vdash e_2 : Int}{O, M, C \vdash e_1 + e_2 : Int}$$

Recursive Traversal

Implementation

Environment is passed down AST
Argument to recursive function

Types of expressions are computed
bottom-up.

Recursive Type Check Example

```
tc(env, e1 + e2):
    T1 = tc(env, e1);
    T2 = tc(env, e2);
    check T1 == T2 == Int
    return 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 inherits A { ... }
class Main {
    x : A ← new A;
    ...
    x ← new B;
    ...
}
```

← x value has dynamic type A

← x value has dynamic type B.

A variable of static type A can hold a value of dynamic type B <u>iff</u> $B \leq A$

Soundness

$\forall E.\ \text{dynamic\_type}(E) \leq \text{static\_type}(E)$

- 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

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.

x : Object ← new Object;

x.copy() — return a copy of x.

What is the return type of the copy method?

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

```
class A { ... }
y : A ← new A;
z : A ← y.copy()
```

$\underbrace{z : A \leftarrow \underbrace{y.copy()}_{\text{Object}}}_{\text{type error.}}$

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

```
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

$$\frac{\begin{array}{c} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ M(T_0, f) = (T_1', T_2', \ldots T_n', \text{SELF\_TYPE}) \\ T_i \leq T_i' \text{ for all } 1 \leq i \leq n \end{array}}{O, M, C \vdash e_0 @ T.f(e_1, \ldots, e_n) : T_0}$$

used to be T_{n+1}

f is in T.

Why is this ok?

Because f returns type of self: T_0.

Notation: SELF_TYPE_C - use of SELF_TYPE is the body of definition of class C.

$$\text{SELF\_TYPE}_C \leq C$$

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

$e_0 @ T(e_1, e_2, \ldots, e_n)$ ← T cannot be SELF_TYPE

Attribute

```
class ~ {
    x : SELF_TYPE;
}
```

In subclass, x would have the type of the subclass.

Formals Cannot Be SELF_TYPE

$e_0(x : T) : T' \{ \ldots \}$

no SELF_TYPE

SELF_TYPE OK

class A { f(x : SELF_TYPE) ... }
class B inherits A { f(x : SELF_TYPE) ... }
let y : A ← new B in y.f(new A); ...

static type A, y = new A ok

Violates soundness { but dynamic type is B, y = new A not ok (SELF_TYPE is B)

Notation: $SELF\_TYPE_C$ – use of SELF_TYPE in 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.

$\leq$ on SELF_TYPE

$\text{SELF\_TYPE}_C \leq C$

SELF_TYPE might be
$<$ C when inherited
method is called

$\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$

In COOL, never compare SELF_TYPES from
different classes

$\text{SELF\_TYPE}_C \leq T$ if $C \leq T$

$T \not\leq \text{SELF\_TYPE}_C$ if $T \neq \text{SELF\_TYPE}_C$

$T \leq T'$ as before if $T, T' \neq \text{SELF\_TYPE}$

lub and SELF_TYPE

$\text{SELF\_TYPE}_C \sqcup \text{SELF\_TYPE}_C = \text{SELF\_TYPE}_C$

$\text{SELF\_TYPE}_C \sqcup T = C \sqcup T$ ($T \neq \text{SELF\_TYPE}$)

All we know is $\text{SELF\_TYPE}_C \leq C$

$T \sqcup \text{SELF\_TYPE}_C = C \sqcup T$

$\sqcup$ needs to be commutative

$T \sqcup T'$ as before when $T, T' \neq \text{SELF\_TYPE}$

More Rules

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

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

Other rules remain the same

But use extended definitions of

$\leq$ and $\sqcup$

Summary of SELF_TYPE

Extended $\leq$, $\sqcup$ 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

undeclared.

let $y : Int \leftarrow x + 2$ in $y + 3$

Type for undeclared x?

If x : Object, $x+2$ will cause another error.

Reports too many errors.

Another Solution

Compiler stores "No_type" for erroneous expressions

No_type $\leq$ C for all types C

So No_type $\sqcup$ C is always C

All operations / assignments are ok.

let y : Int $\leftarrow \underbrace{x + 2}_{\text{No\_type} \sqcup \text{Int} \rightarrow \text{Int}}$ in y + 3 $\leftarrow$ only one error

No_type

Types no longer a tree

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

Not required in project