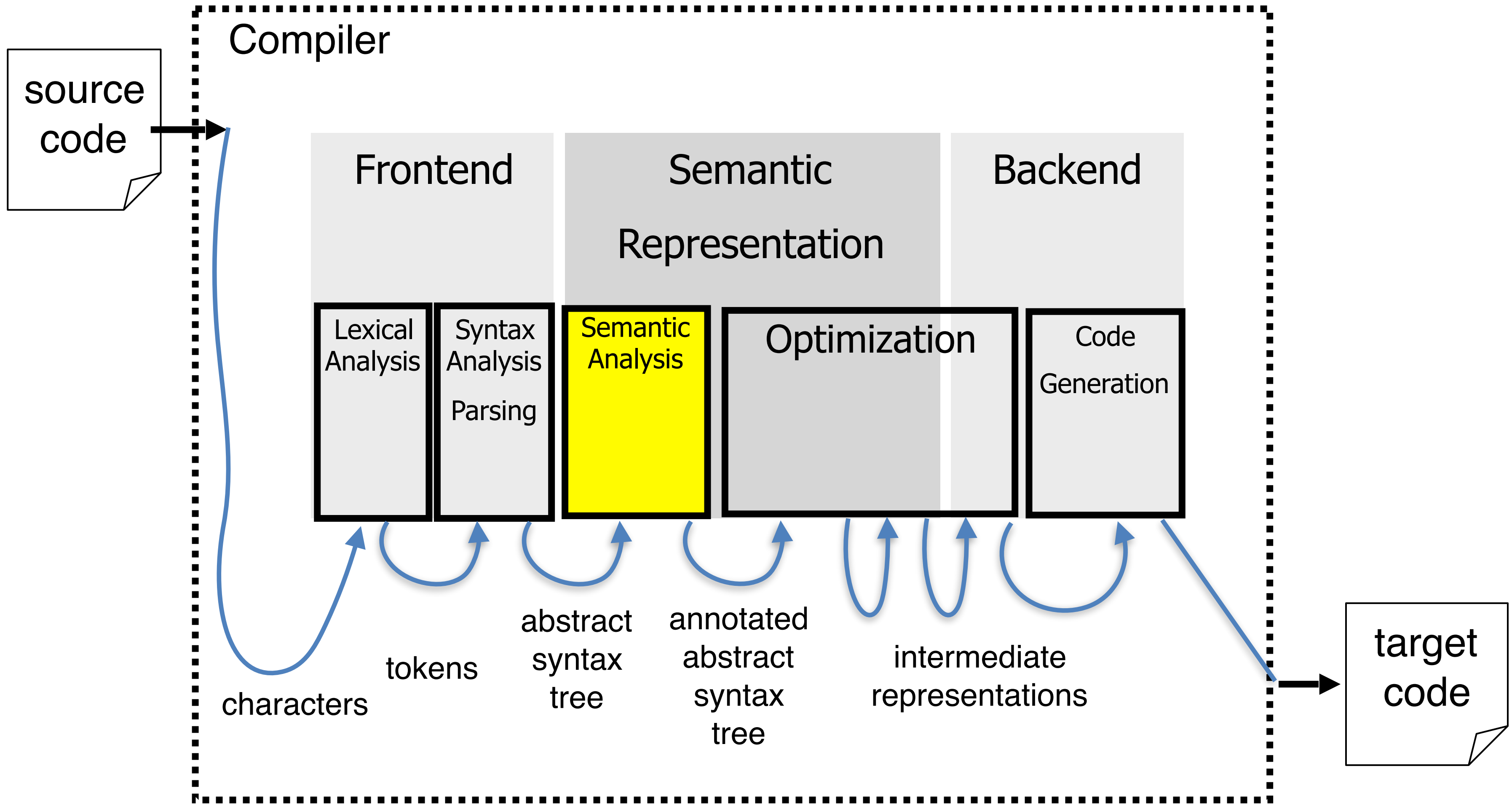
Cool Type Checking

Greta Yorsh



source code
Compiler
Frontend
Semantic Representation
Backend
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Optimization
Code Generation
characters
tokens
abstract syntax tree
annotated abstract syntax tree
intermediate representations
target code

Recap: semantic analysis

- Scope rules
- Symbol tables
- Inheritance graph
- Type is a set of values
- Type equivalence: nominal vs structural
- Expressions: locations (l-values) and values (r-values)
- Type coercions
- Types: strong vs weak types
- Type checking: dynamic vs static
- Type checking vs type inference
- Type rules
- Subtyping relation and least upper bounds

Cool type environment

$$\underbrace{O,M,C}_{\textbf{type environment}} \vdash e: T$$

- **O** mapping Object Id's to types
  - symbol table for the current scope
  - $O(x) = T$
- **M** mapping methods to method signatures
  - $M(K, f) = (A, B, D)$
    means there is a method $f(a{:}A,\ b{:}B){:}\ D$ defined in class K (or its ancestor)
- **C** the class in which expression e appears
  - used when SELF_TYPE is involved

Soundness of type rules

- For every expression **e**, for every value **v** of **e** at runtime **v** $\in$ **values_of(static_type(e))**
  - values_of(T) is the set of values represented by type T
  - static_type(e) is T when O,M,C $\vdash$ e: T
  - static_type(e) may describe more values than e can have in any run

- Static typing can reject correct programs
- More complicated with subtyping (inheritance)

Static type checking: pros and cons

- Catches many programming errors
- Proves properties of your code
- Avoids the overhead of runtime type checks
- Restrictive: may reject correct programs
- Rapid prototyping is difficult
- Complicates the programming language and the compiler
- In practice, most code is written in statically typed languages with escape mechanisms
    - Unsafe casts in C, Java
    - union in C

Types in practice

- Type checking
  - Static: C, Java, Cool, ML
  - Dynamic: machine code, scripting languages (python, ruby)
  - JavaScript is untyped
- Strong vs weak types (coercion)
  - Python is strongly typed
  - Perl is weakly typed

Plan

- Cool type rules
- Implementing type checking for Cool

Type rules

- Rules are **schemas** for inferring types of expressions

$$\frac{O, M, C \vdash e1 : Int \quad O, M, C \vdash e2 : Int}{O, M, C \vdash e1 + e2 : Int}$$

$$\frac{}{O,M,C \vdash int\_const : Int}$$

$$\frac{O(id) = T}{O,M,C \vdash id : T}$$

- Infer types by instantiating the schemas

$$\frac{O, M, C \vdash 1 : Int \quad O, M, C \vdash 2 : Int}{O, M, C \vdash 1 + 2 : Int}$$

$$\frac{}{O,M,C \vdash 1 : Int}$$

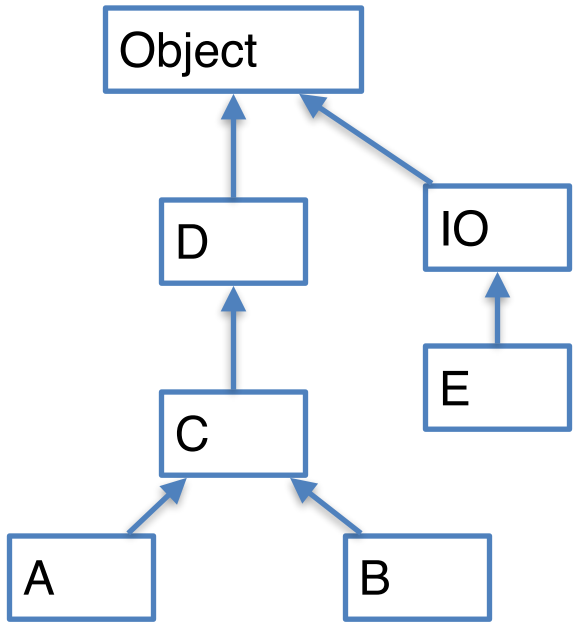
$$\frac{O(y) = Int}{O,M,C \vdash y : Int}$$

$$\frac{O, M, C \vdash y : Int \quad O, M, C \vdash (1 + 2) : Int}{O, M, C \vdash y + (1 + 2) : Int}$$

$$\frac{}{O,M,C \vdash 2 : Int}$$

Subtyping

- Define a relation ≤ on classes
  - X ≤ X
  - X ≤ Y if X inherits from Y
  - X ≤ Z if X ≤ Y and Y ≤ Z
- Example
  - A ≤ C
  - B ≤ Object
  - E ≰ D and D ≰ E



Least upper bounds

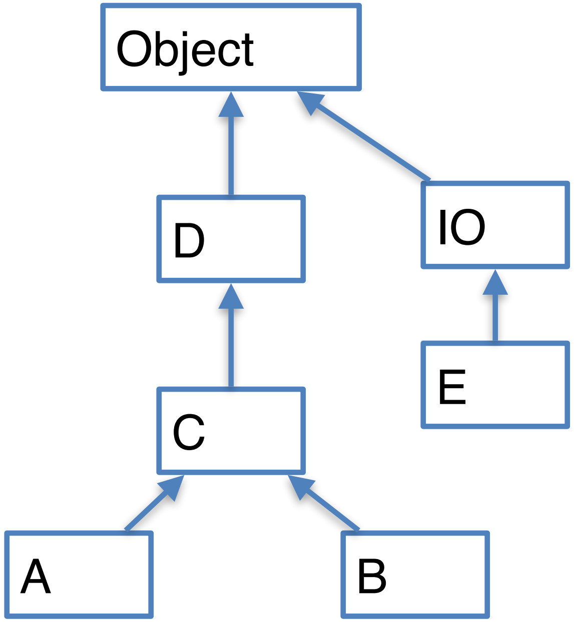
- Z is the least upper bound of X and Y
- lub(X, Y)=Z
  - $X \le Z$ and $Y \le Z$
    Z is **upper** bound
  - $X \le Z'$ and $Y \le Z' \Rightarrow Z \le Z'$

    Z is the **least** upper bound

Least upper bounds

- In Cool, the least upper bound of two types is their **least common ancestor** in the inheritance tree

- Example
  - lub(A,B) = C
  - lub(C,D) = D
  - lub(C,E) = Object



Type rules: Assign

$$\frac{O(x) = T0 \qquad O,M,K \vdash e1 : T1 \qquad T1 \leq T0}{O,M,K \vdash x \leftarrow e1 : T1} \text{[Assign]}$$

```
class A {
        foo() : A  { … }
};
class B inherits A {  };
...
let x:B in x ←(new B).foo();
let x:A in x  ←(new B).foo();
let x:Object in x ←(new B).foo();
```

$\overbrace{\texttt{(new B).foo()}}^{A}$

ERROR
OK
OK

Example: dynamic vs static types

```
class A { ... }
class B inherits A { ... }
class Main (
    x: A ← new A ;
    ...
    x ← new B ;
    }
```

static type of x is A

dynamic type of x is A

dynamic type of x is B

- A variable of static type A
  can hold the value of static type B   if B ≤ A

Types: dynamic vs static

- The **dynamic** type of an object is the class that is used in the new expression
  - a runtime notion
  - **even languages that are statically typed have dynamic types**

- The **static** type of an expression captures all the dynamic types that the expression could have
  - a compile-time notion

Soundness with subtyping

- A type system is **sound** if
  for all expressions e
  dynamic_type(e) ≤ static_type(e)

- If the inferred type of e is T
  then in **all executions** of the program,
  e evaluates to a value of type ≤ T

- We only want sound rules
- But some sound rules are better than others

Let rule with initialization

$$\frac{O,M,K \vdash e_0 : T \qquad O(\mathbf{T}/x) \vdash e_1 : T_1}{O,M,K \vdash \text{let } x : T \leftarrow e_0 \text{ in } e_1 : T_1} \text{[Let Weak Rule]}$$

```
class A {
        foo():C  { … }
};
class B inherits A {  };
...
let  x:A ← new B in x.foo();
```

Let rule with initialization

$$\frac{\begin{array}{l} O,M,K \vdash e_0 : T \\ O(\mathbf{T}/x) \vdash e_1 : T_1 \end{array}}{O,M,K \vdash \text{let } x{:}\ T \leftarrow e_0 \text{ in } e_1 : T_1} \quad \text{[Let Weak Rule]}$$

$$\frac{\begin{array}{l} O,M,K \vdash e_0 : T \\ O(\mathbf{T_0}/x) \vdash e_1 : T_1 \\ T \leq T_0 \end{array}}{O,M,K \vdash \text{let } x{:}\ T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \quad \text{[Let]}$$

```
class A {
        foo():C  { … }
};
class B inherits A {  };
...
let  x:A ← new B in x.foo();
```

- Both rules are sound but the second one type checks more programs (using subtyping)

Conditional

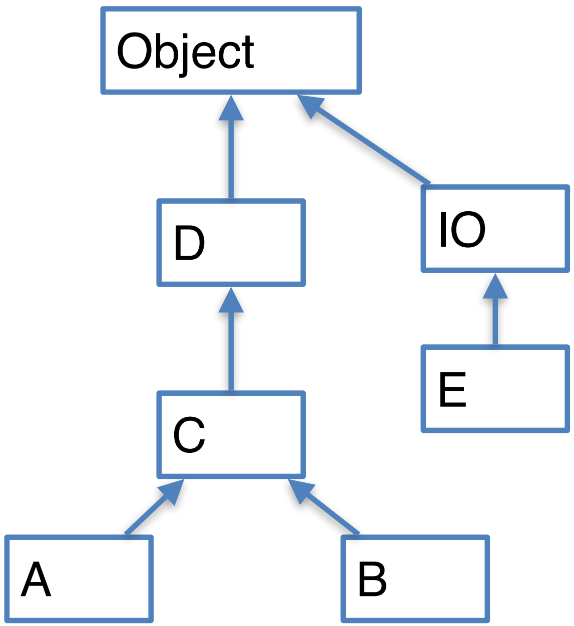
$$\frac{\begin{array}{l} O,M,K \vdash e_0 : Bool \\ O,M,K \vdash e_1 : T_1 \\ O,M,K \vdash e_2 : T_2 \end{array}}{O,M,K \vdash \text{ if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi } : lub(T_1, T_2)}$$

```
foo(a:A, b:B, c:C, e:E) : D {
  if (a < b) then e else c fi
}
```

ERROR

lub(E,C) = Object

Is Object ≤ D ?

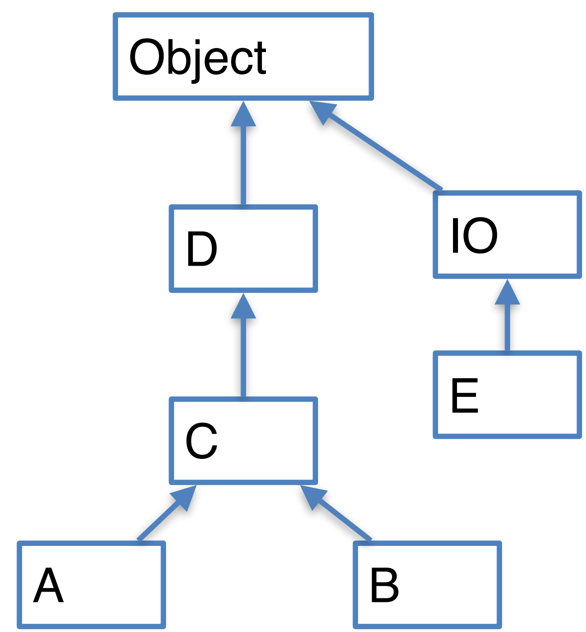


Case

$$\frac{\begin{array}{l} O,M,K \vdash e : T \\ O[X/x],M,K \vdash e_1 : E \\ O[Y/y],M,K \vdash e_2 : F \\ O[Z/z],M,K \vdash e_3 : G \end{array}}{O,M,K \vdash \ \text{case } e \text{ of } \ x : X \Rightarrow e_1;\ y : Y \Rightarrow e_2\ ;\ z : Z \Rightarrow e_3 \text{ esac} : \text{lub}(E,F,G)}$$

foo(d:D) : D {
  case d of
  x : IO => $\overbrace{\text{let a:A} \leftarrow \text{(new A) in x}}^{\text{IO}}$;
  y : E => (new B);
  z : C => z;
  esac
};

ERROR



lub(IO,B,C) = Object   and Object ≤ D ?

Case

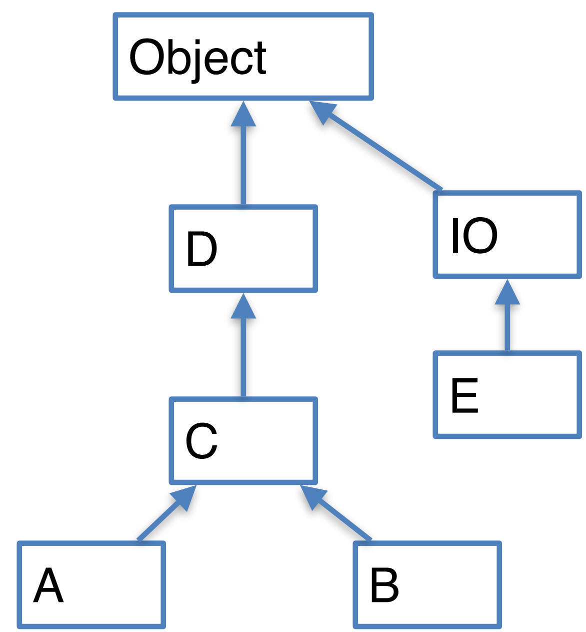
$$\frac{\begin{array}{l} O,M,K \vdash e : T \\ O[X/x],M,K \vdash e_1 : E \\ O[Y/y],M,K \vdash e_2 : F \\ O[Z/z],M,K \vdash e_3 : G \end{array}}{O,M,K \vdash \text{case } e \text{ of } x : X \Rightarrow e_1;\ y : Y \Rightarrow e_2\ ;\ z : Z \Rightarrow e_3 \text{ esac} : \mathrm{lub}(E,F,G)}$$

```
foo(d:D) : D {
  case d of
  x : IO => let a:A ←(new A) in a;
  y : E => (new B);
  z : C => z;
  esac
};
```

A

OK

lub(A,B,C) = C   and C ≤ D



Cool type rules

✓ Arithmetic and boolean expressions
✓ Object identifiers
✓ Conditionals
✓ Let
✓ Case
- SELF_TYPE and self
- Allocation: new
- Dispatch: dynamic and static
- Error handling

Motivation for SELF_TYPE

- What can be the dynamic type of object returned by foo() ?
    - any subtype of A

```
class A {
  foo() : A  { self } ;
};
class B inherits A { ... };
class Main {
          B x ← (new B).foo();
};          ERROR
```

```
class A {
  foo() : SELF_TYPE  { self } ;
};
class B inherits A { ... };
class Main {
          B x ← (new B).foo();
};          OK
```

SELF_TYPE

- Research idea
- Helps type checker to accept more correct programs
  - O,M,K $\vdash$ (new A).foo() : A
  - O,M,K $\vdash$ (new B).foo() : B

- SELF_TYPE is **NOT a dynamic type**
  - Meaning of SELF_TYPE depends on where it appears textually
  - SELF TYPE may refer to the class K in which it appears, or any subtype of K

Where can SELF_TYPE appear?

- Parser checks that SELF_TYPE appears only where a type is expected (How ?)
- But SELF_TYPE is not allowed everywhere a type can appear

Where can SELF_TYPE appear ?

- class T1 inherits T2 { … }
  - T1, T2 cannot be SELF_TYPE
- x : SELF_TYPE
  - attribute
  - let
  - not in case
- new SELF_TYPE
  - creates an object of the same type as self
- e@T.foo(e1)
  - T cannot be SELF_TYPE
- foo(x:T1):T2 {…}
  - only T2 can be SELF_TYPE

Example: new

```
class A {
   foo() : A {  new SELF_TYPE  };
};
class B inherits A { … }
…
(new A).foo();        creates A object
(new B).foo();        creates B object
```

Subtyping for SELF_TYPE

- SELF_TYPEc ≤ SELF_TYPEc

- SELF_TYPEc ≤ C
- It is always safe to replace SELF_TYPEc with C
- SELF_TYPEc ≤ T if C ≤ T

- T ≤ SELF_TYPEc is always false
  - because SELF_TYPEc can denote **any** subtype of C

lub(T,T') for SELF_TYPE

- lub(SELF_TYPEc, SELF_TYPEc) = SELF_TYPEc
- lub(T, SELF_TYPEc) = lub(T,C)
  - the best we can do

Type rules for **self** and **new**

$$\frac{}{O, M, K \vdash \text{ self} : \text{SELF\_TYPE}_k}$$

$$\frac{}{O, M, K \vdash \text{ new SELF\_TYPE} : \text{SELF\_TYPE}_k}$$

Other rules

- A use of SELF_TYPE refers to any subtype of the current class
- Except in dispatch
  - because the method return type of SELF_TYPE might have nothing to do with the current class

Dispatch

$$\frac{\begin{array}{l} O, M, K \vdash c : C \\ O, M, K \vdash a : A \\ O, M, K \vdash b : B \\ M(C, foo) = (A_1, B_1, D_1) \\ A \leq A_1 \quad , \quad B \leq B_1, \quad \mathbf{D_1 \neq SELF\_TYPE} \end{array}}{O, M, K \vdash c.foo(a, b) : D_1}$$

```
class C1 {
  foo(a:A1, b:B1) : D1 { new D1 ; };
};
class C inherits C1 {…};
…
(new  C).foo( (new A) , (new B) );
```

which class is used to find the declaration of foo() ?

$$\frac{\begin{array}{l} O, M, K \vdash c : C \\ O, M, K \vdash a : A \\ O, M, K \vdash b : B \\ M(C, foo) = (A1, B1, \mathbf{SELF\_TYPE}) \\ A \leq A1 \quad , \quad B \leq B1 \end{array}}{O, M, K \vdash c.foo(a, b): C}$$

Example: self

```
class A {
   foo() : A {  self  };
};
class B inherits A { … }
…
(new A).foo();        returns A object
(new B).foo();        returns B object
```

Static Dispatch

$$\frac{\begin{array}{l} O, M, K \vdash c : C \\ O, M, K \vdash a : A \\ O, M, K \vdash b : B \\ M(C_1, f) = (A_1, B_1, D_1) \\ A \leq A_1,\ B \leq B_1,\ C \leq C_1,\ \mathbf{D_1 \neq SELF\_T} \end{array}}{O, M, K \vdash c@C_1.f(a, b) : D_1}$$

```
class C1 {
  foo(a:A1, b:B1) : D1 { new D1 ; };
};
class C inherits C1 {…} ;
…
(new  C)@C1.foo( (new A) , (new
B) );
```

if we dispatch a method returning SELF_TYPE in class C_1, do we get back C_1 ?

No. SELF_TYPE is the type of self, which may be a subtype of the class in which the method appears

$$\frac{\begin{array}{l} O, M, K \vdash c : C \\ O, M, K \vdash a : A \\ O, M, K \vdash b : B \\ M(C_1, f) = (A_1, B_1, \mathbf{SELF\_TYPE}) \\ A \leq A_1,\ B \leq B_1,\ C \leq C_1 \end{array}}{O, M, K \vdash c@C_1.f@(a, b) : C}$$

SELF_TYPE Example

```
class A {
    delegate : B;
    callMe() SELF_TYPE
        { delegate.callMe(); } ;  ERROR
};
class B {
    callMe() : SELF_TYPE { self };
};
class Main {
        A a ←(new A).callMe();
};
```

Error Handling

- Error detection is easy
- Error recovery: what type is assigned to an expression with no legitimate type ?
  - influences type of enclosing expressions
  - cascading errors

```
let y : Int ← x + 2 in y + 3
```

- Better solution: special type No_Type
  - inheritance graph can be cyclic

Implementation of Cool Types

- How are types represented ?
  - **Symbol**
  - compare types by comparing **Symbol**

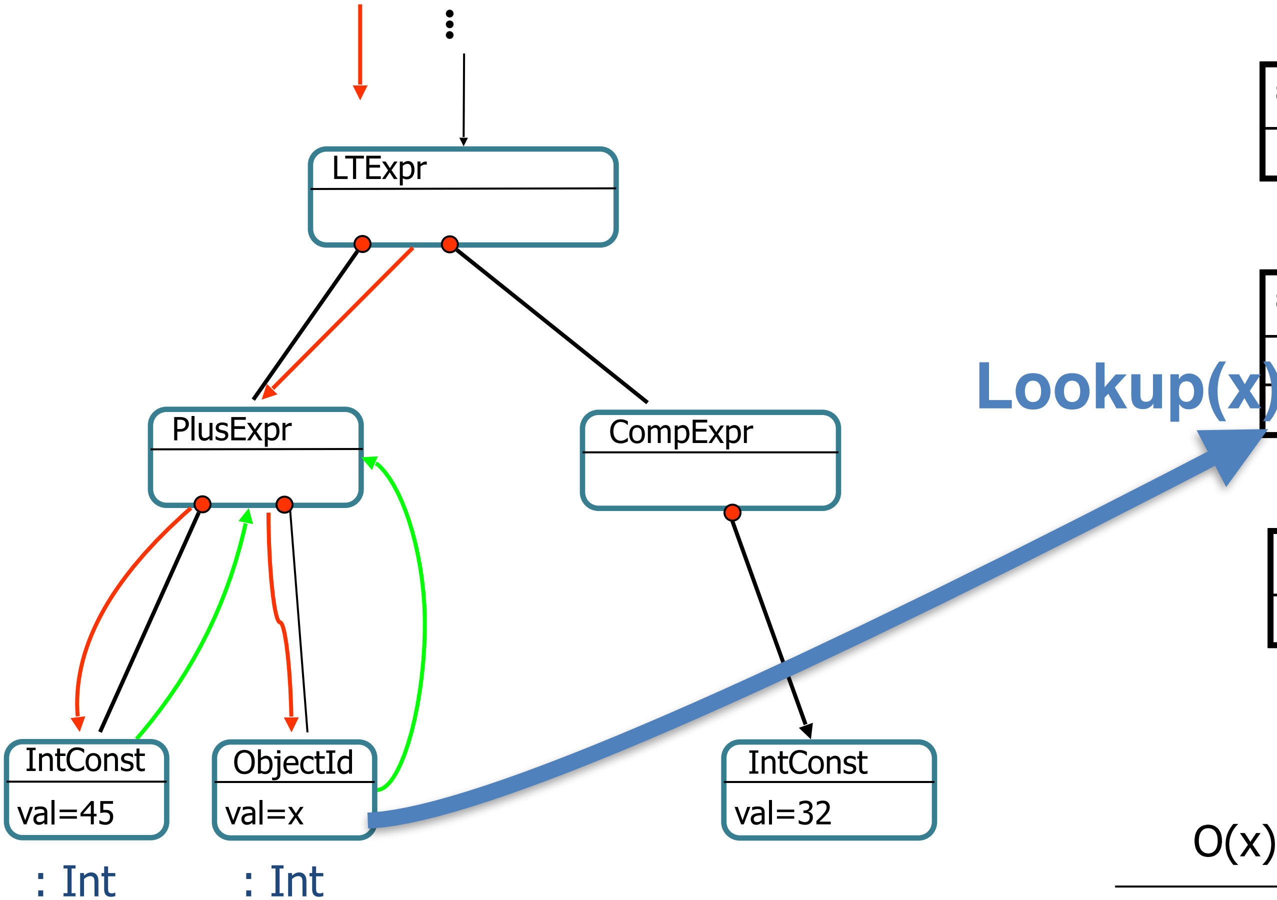
- When are types are created?
  - during lexer/parsing
  - predefined types

Type checking implementation

- Single traversal over AST

- Types passed **up** the tree
- Type environment passed **down** the tree

Example



globals

Symbol	kind		
Foo	class		

Foo

Symbol	kind	type
test	method	Int->Int
x	var	Int

test

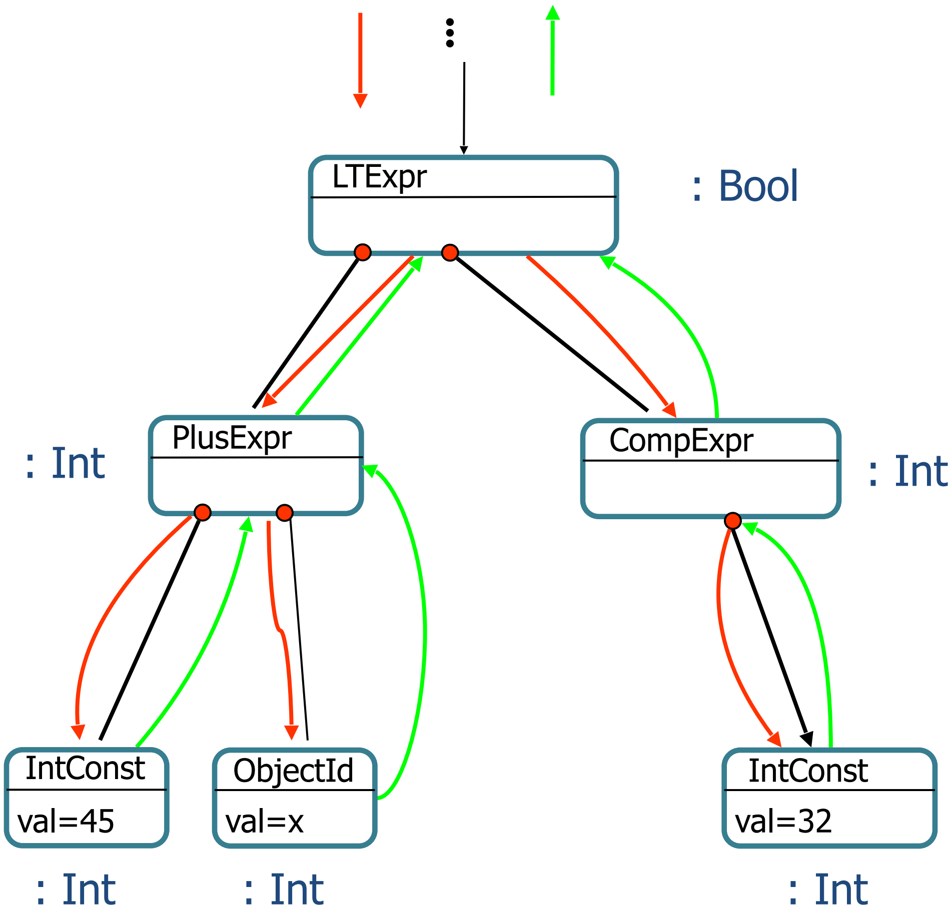
Symbol	kind	type
c	var	Int

$$\frac{O(x)=Int}{O, M, K \vdash x : Int}$$

$$\frac{}{O, M, K \vdash \text{int-literal} : Int}$$

(45 + x) < (~32)

Example



(45 + x) < (~32)

$$\frac{O, M, K \vdash e1 : Int \qquad O, M, K \vdash e2 : Int}{O, M, K \vdash e1 < e2 : Bool}$$

$$\frac{O, M, K \vdash e : Int}{O, M, K \vdash \sim e : Int}$$

$$\frac{O, M, K \vdash e1 : Int \qquad O, M, K \vdash e2 : Int}{O, M, K \vdash e1 + e2 : Int}$$

$$\frac{O(x) = Int}{O, M, K \vdash x : Int}$$

$$\frac{}{O, M, K \vdash \text{int-literal} : Int}$$

Quick Quiz

- Which type rules use subtyping relation ≤ ?
- Which type rules use lub?
- Which type rules have a special case for SELF_TYPE?
- Where can SELF_TYPE appear in Cool program?
- How to extend subtying for SELF_TYPE?