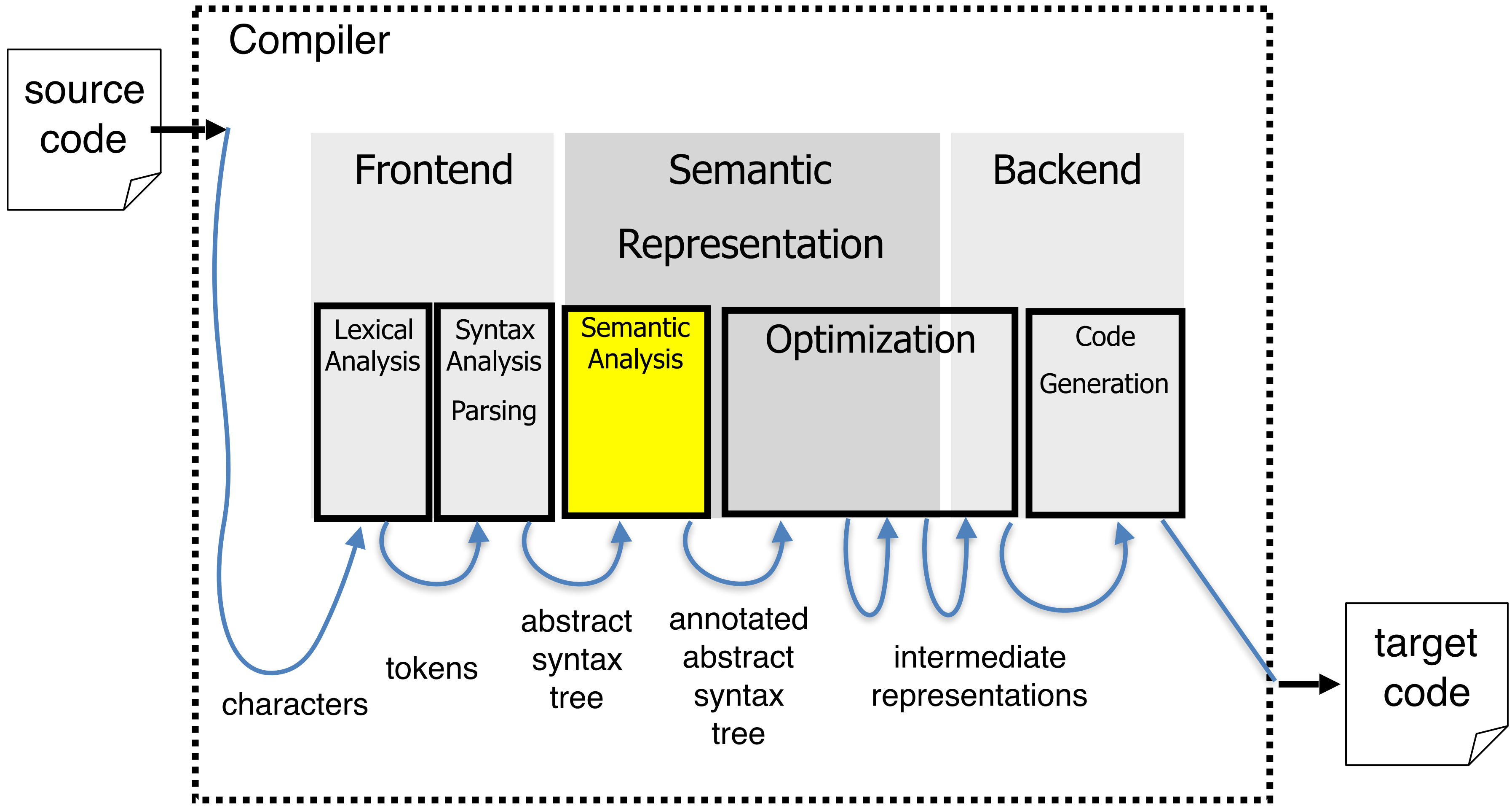
Type Checking

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

Types

- What is a type?
  - simplest answer: a set of values
  - examples: integers, real numbers, booleans, …

- Why do we care?
  - safety: guarantee that certain errors cannot occur at runtime
  - abstraction: hide implementation details

Type declarations

- Explicit type declarations

```
TYPE Int_Array = ARRAY [Integer 1..42] OF Integer;
```

- Anonymous types

```
Var a : ARRAY [Integer 1..42] OF Real;
```

Type declarations

```
Var a : ARRAY [Integer 1..42] OF Real;
```

```
TYPE #type01_in_line_73 = ARRAY [Integer 1..42] OF Real;
Var a : #type01_in_line_73;
```

Forward references

```
TYPE Ptr_List_Entry = POINTER TO List_Entry;
TYPE List_Entry =
    RECORD
        Element : Integer;
        Next : Ptr_List_Entry;
    END RECORD;
```

- Forward references must be resolved
- A forward ref is added to the symbol table (as forward ref), and **later updated when the type declaration is met**
- At the end of scope, check that all forward refs have been resolved
- Must add check for **circularity**

Type equivalence: **name** equivalence

```
Type t1 = ARRAY[Integer] OF Integer;
Type t2 = ARRAY[Integer] OF Integer;
```

t1 not (name) equivalent to t2

```
Type t3 = ARRAY[Integer] OF Integer;
Type t4 = t3
```

t3 equivalent to t4

Type equivalence: **structural** equivalence

```
Type t5 = RECORD c: Integer; p: POINTER TO t5; END
RECORD;
Type t6 = RECORD c: Integer; p: POINTER TO t6; END
RECORD;
Type t7 =
   RECORD
      c: Integer;
      p: POINTER TO
         RECORD
            c: Integer;
            p: POINTER to t5;
          END RECORD;
END RECORD;
```

t5, t6, t7 are all (structurally) equivalent

In practice…

- Almost all modern languages use **name equivalence**
- Why?

Types: strong vs. weak

- Coercion
- Strongly typed: C, C++, Java,…
- Weakly typed: Perl, PHP, …
- Not everybody agrees on this classification

Output: 73

perl

```
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

warning: initialization makes integer from pointer without a cast

C

```
main() {
 int a=31;
 char b[3]="42x";
 int c=a+b;
}
```

error: Incompatible type for declaration. Can't convert java.lang.String to int

Java

```
class A {
 public static void main() {
   int a=31;
   String b ="42x";
   int c=a+b;
 }
}
```

Types: strong vs. weak

- Coercion
- Strongly typed: C, C++, Java,…
- Weakly typed: Perl, PHP, …
- Not everybody agrees on this classification

Output: 73

perl

```
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

warning: initialization makes integer from pointer without a cast

C

```
main() {
 int a=31;
 char b[3]="42x";
 int c=a+b;
}
```

Java

```
public class… {
 public static void main() {
   int a=31;
   String b ="42x";
   String c=b+a;
 }
}
```

OK

Coercions

- Suppose that at some point in the program, we expect a value of type T1 and find a value of type T2
- Is that acceptable?

```
float x = 3.141;
int y = x;
```

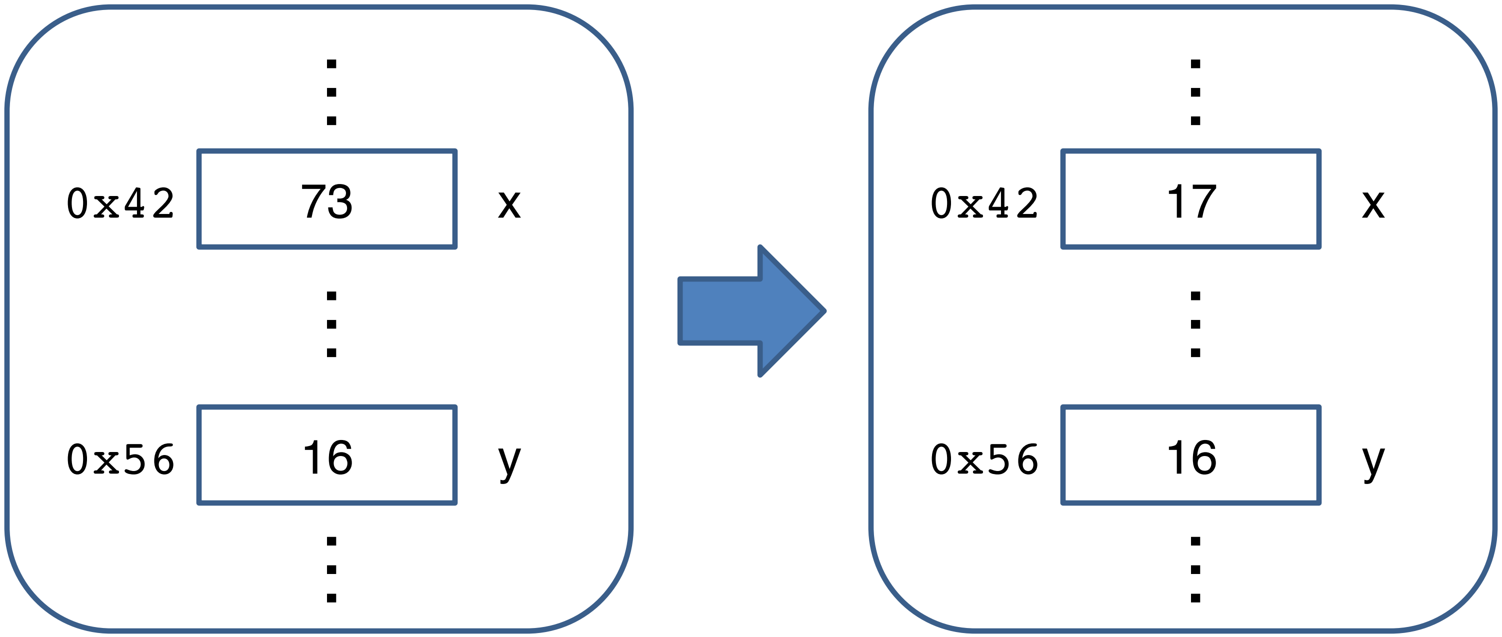
l-values and r-values

```
dst := src
```

- What is dst? dst is a **memory location** where the value should be stored
- What is src? src is a **value**
- “location” on the left of the assignment called an l-value
- “value” on the right of the assignment is called an r-value

Example: l-values and r-values

```
x:= y + 1
```



Example: l-values and r-values

```
x := A + 1
```

```
x := A[1]
```

```
x := A[A[1]]
```

l-values and r-values

expected

found

	lvalue	**rvalue**
lvalue	ok	deref
rvalue	error	ok

So far…

- Static correctness checking
- **Identification**: match applied occurrences of identifier to its defining occurrence
  - symbol table maintains this information
- **Checking**: which type combinations are legal
  - type equivalence: nominal vs structural
  - type coercion
  - each node in the AST of an expression represents either an l-value (location) or an r-value (value)

Type table

- All types in a compilation unit are collected in a type table

- For each type, its table entry contains
  - type constructor: basic, record, array, pointer,…
  - size and alignment requirements
    - to be used later in code generation
  - types of components (if applicable)
    - example: types of record fields

How does this magic happen?

- We probably need to go over the AST?
- How does this relate to the clean formalism of the parser?
- Different approaches
  - attribute grammars
  - type systems

TYPE RULES

Type system (textbook definition)

- “A type system is a tractable **syntactic** method for **proving the absence of certain program behaviors** by classifying phrases according to the kinds of values they compute”

-- Types and Programming Languages
/ Benjamin C. Pierce

Type system

- A type system of a programming language is a way to define how “good” programs behave
  - Good programs = well-typed programs
  - Bad programs = not well typed

- Type **checking**
  - Static typing: most checking at compile time
  - Dynamic typing: most checking at runtime
- Type **inference**
  - Automatically infer types for a program or show that there is no valid typing

Type checking: static vs. dynamic

- Static type checking is **conservative**
  - Any program that is determined to be well-typed is free from certain kinds of errors
  - May reject programs that cannot be statically determined as well typed
  - Why?

- Dynamic type checking
  - May accept more programs as valid (runtime info)
  - Errors not caught at compile time
  - Runtime cost

Type rules

- which types can be combined with certain operator
- assignment of expression to variable
- formal and actual parameters of a method call

string string
“drive” + “drink”

string

int string
42 + “the answer”

ERROR

Type rules

- Specify for each operator
  - types of operands
  - type of result
- Basic types
  - building blocks for the type rules
  - example: int, boolean, (sometimes) string
- Type expressions
  - array types
  - function types
  - record types and classes

Type rules

If E1 has type int and E2 has type int,
then E1 + E2 has type int

$$\frac{E1 : \text{int} \qquad\qquad E2 : \text{int}}{E1 + E2 : \text{int}}$$

Notations for rules

- An inference rule consists of **premises** and **conclusion**

$$\frac{A \qquad B}{C}$$

- An inference rule without any premises is an **axiom**

$$\frac{}{A} \qquad\qquad\qquad \frac{}{B}$$

- A proof is a sequence of lines, each of which is either an axiom or follows from earlier lines by an inference rule

$$\frac{\dfrac{}{A} \qquad \dfrac{}{B}}{C}$$

More type rules

$$\frac{}{\text{true : boolean}} \qquad \frac{}{\text{false : boolean}}$$

$$\frac{}{\text{int-literal : int}} \qquad \frac{}{\text{string-literal : string}}$$

$$\frac{\text{E1 : int} \qquad \text{E2 : int}}{\text{E1 op E2 : int}} \qquad \text{op} \in \{ +, -, /, *, \% \}$$

$$\frac{\text{E1 : int} \qquad \text{E2 : int}}{\text{E1 op E2 : boolean}} \qquad \text{op} \in \{ <=, <, >, >= \}$$

$$\frac{\text{E1 : T} \qquad \text{E2 : T}}{\text{E1 op E2 : boolean}} \qquad \text{op} \in \{ ==, != \}$$

And even more type rules

$$\frac{E1 : boolean \qquad E2 : boolean}{E1\ op\ E2 : boolean} \qquad op \in \{\ \&\&, ||\ \}$$

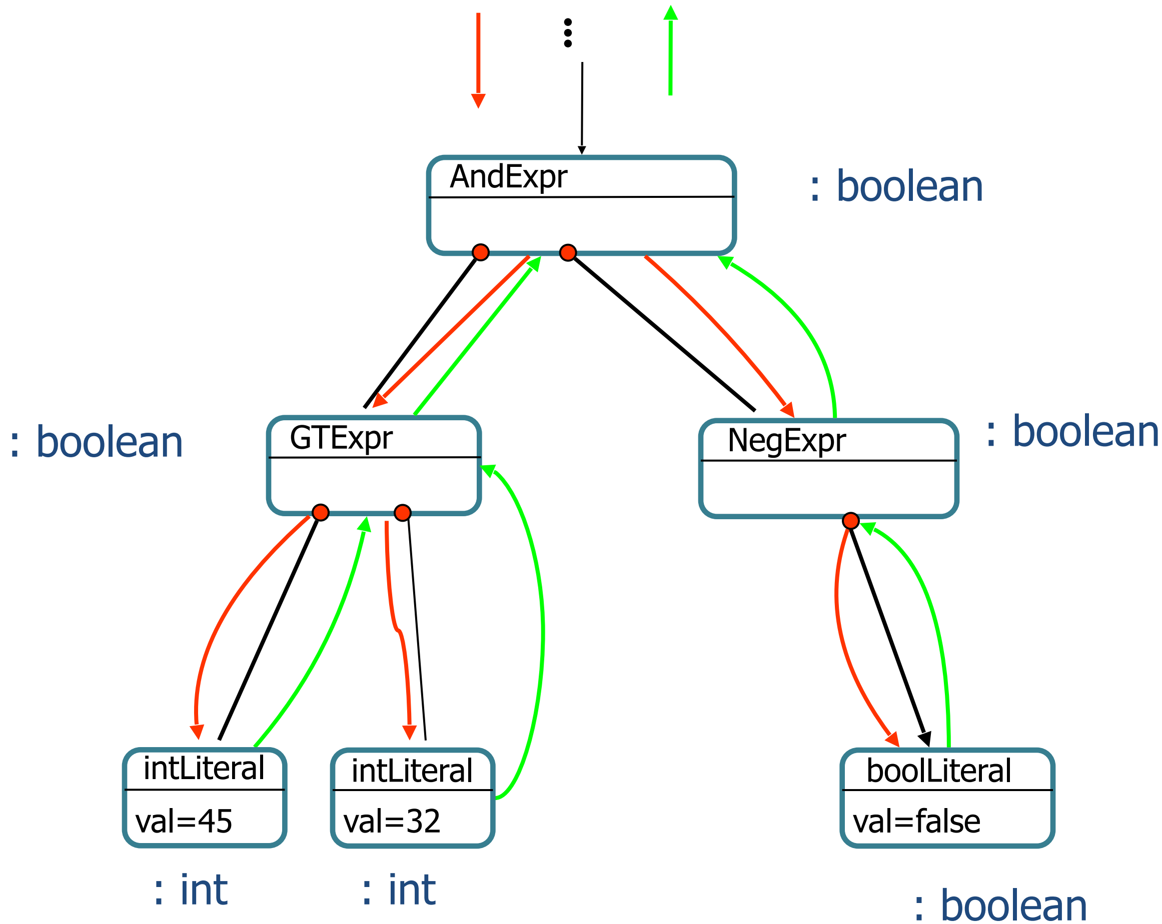
$$\frac{E1 : int}{- E1 : int} \qquad \frac{E1 : boolean}{!\ E1 : boolean}$$

$$\frac{E1 : T[]}{E1.length : int} \qquad \frac{E1 : T[] \qquad E2 : int}{E1[E2] : T} \qquad \frac{E1 : int}{new\ T[E1] : T[]}$$

Type Checking

- Traverse AST
- Assign types for AST nodes
- Use typing rules to compute node types

Example



45 > 32 && !false

$$\frac{E1 : \text{boolean} \qquad E2 : \text{boolean}}{E1 \text{ op } E2 : \text{boolean}}$$

$$\text{op} \in \{ \&\&, || \}$$

$$\frac{E1 : \text{boolean}}{!E1 : \text{boolean}}$$

$$\frac{E1 : \text{int} \qquad E2 : \text{int}}{E1 \text{ op } E2 : \text{boolean}}$$

$$\text{op} \in \{ <=, <, >, >= \}$$

$$\frac{}{\text{false} : \text{boolean}}$$

$$\frac{}{\text{int-literal} : \text{int}}$$

Cool Types

- The types are
  - Class names
  - SELF_TYPE
- The user **declares** types for identifiers
- The semantic analysis **infers** types for expressions
  - every expression has a unique type
- Cool **type rules** specify which operations are valid for which types
- The goal of **type checking** is to ensure that operations are used with the correct types
  - enforces intended interpretation of values
- Cool is **statically** typed: type checking during compilation

Plan

- Cool type rules
- Implementing type checking for Cool

Notations for rules

$$\frac{\begin{array}{l}\textbf{Environment}_1 \vdash \textbf{Statement}_1 \\ \textbf{Environment}_2 \vdash \textbf{Statement}_2 \\ \textbf{...} \\ \textbf{Environment}_n \vdash \textbf{Statement}_n\end{array}}{\textbf{Environment} \vdash \textbf{Statement}} \; \textbf{[NAME] conditions}$$

- $A \vdash B$ means "given A, it is provable that B"
- $A \vdash B$ is called a judgement
- A is called context or environment
- B is called statement

Cool type judgements

- Cool type rules have judgements of the form

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

- **O** gives types to free identifiers in the current scope
- **M** gives information about the formal parameters and return type of methods
- **C** is the class in which expression e appears

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(C, f) = (A, B, D)
    means there is a method f(a:A, b:B): D defined in class C (or its ancestor)
- **C** the class in which expression e appears
  - used when SELF_TYPE is involved

Cool type environment

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

- Why separate object/methods?
- In Cool, the method and object identifiers live in different name spaces

Type rules

- Rules are schemas for inferring types of expressions

$$\frac{O, M, K \vdash e1 : Int \qquad O, M, K \vdash e2 : Int}{O, M, K \vdash e1 + e2 : Int} \qquad \frac{}{O,M,C \vdash int\_const : Int} \qquad \frac{O(id) = T}{O,M,C \vdash id : T}$$

- Infer types by instantiating the schemas

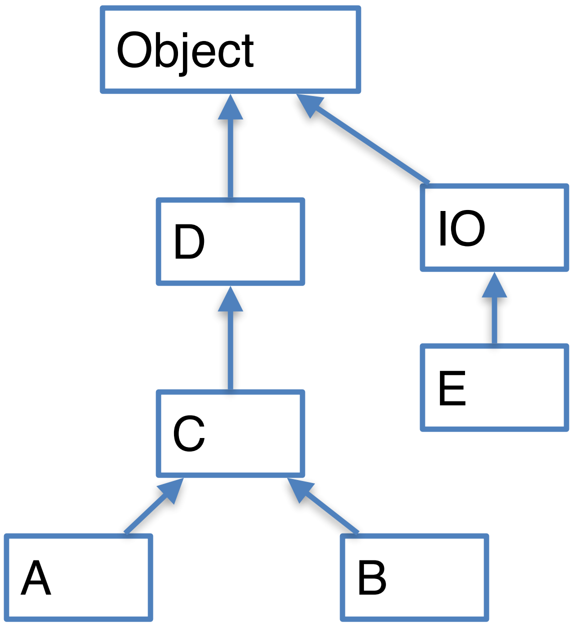
$$\frac{O, M, K \vdash 1 : Int \qquad O, M, K \vdash 2 : Int}{O, M, K \vdash 1 + 2 : Int} \qquad \frac{}{O,M,C \vdash 1 : Int} \qquad \frac{O(y) = Int}{O,M,C \vdash y : Int}$$

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

Subtyping

- Define a relation ≤ on classes
  - $X \leq X$
  - $X \leq Y$ if X inherits from Y
  - $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

- Example
  - $A \leq C$
  - $B \leq$ Object
  - $E \nleq D$ and $D \nleq E$



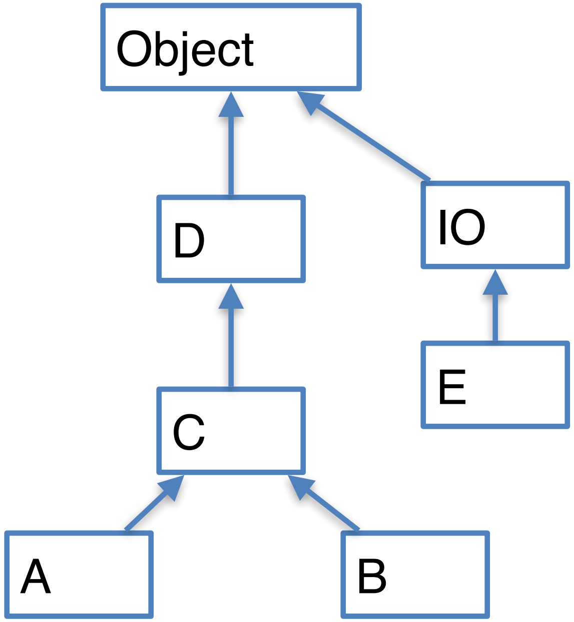
Least upper bounds

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

    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

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

```
class A {
        foo() : A  { … }
};
class B inherits A {  };
...
```

ERROR let x:B in x ← $\overbrace{\text{(new B).foo()}}^{A}$;

OK let x:A in x ← (new B).foo();

OK let x:Object in x ← (new B).foo();

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 $B \leq A$

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

- 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

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

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

```
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

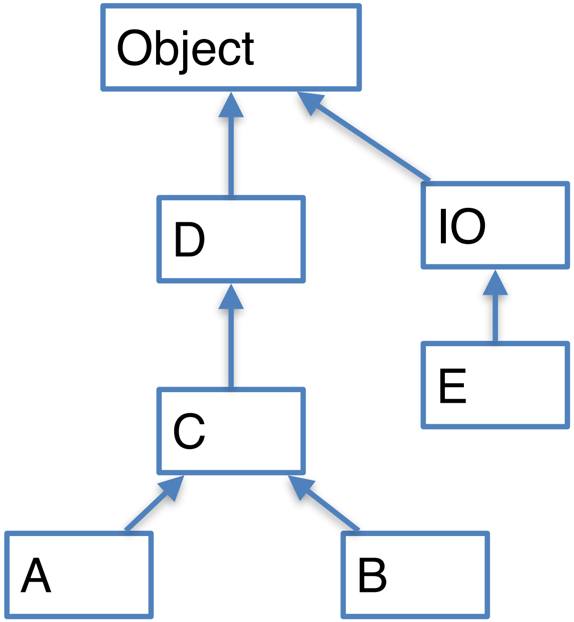
$$\frac{O,M,K \vdash e_0 : Bool \quad O,M,K \vdash e_1 : T_1 \quad O,M,K \vdash e_2 : T_2}{O,M,K \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : lub(T_1, T_2)}$$

```
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 ?



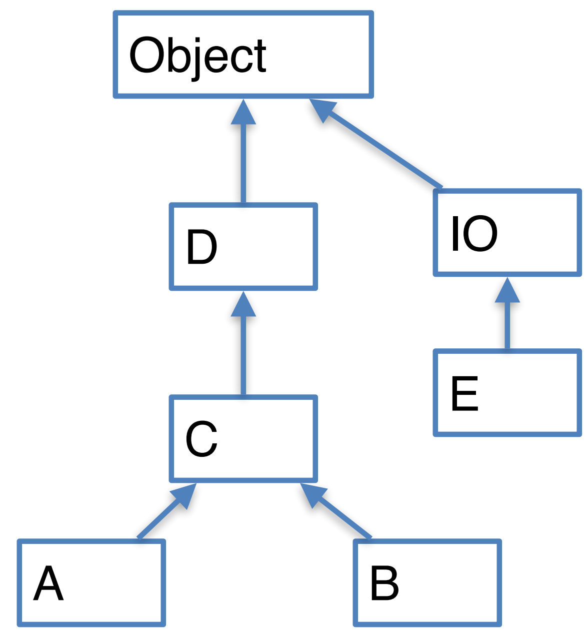
Case

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

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

IO

ERROR



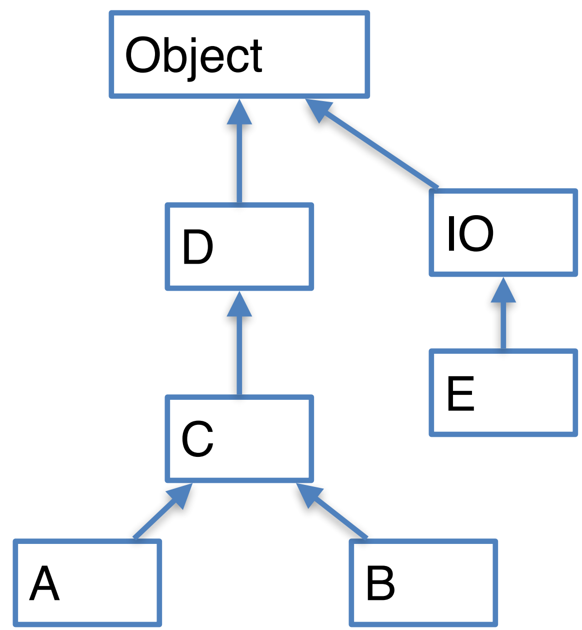
lub(IO,B,C) = Object   and Object ≤ D

Case

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

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

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
  - $C,M,K \vdash$ (new A).foo() : A
  - $C,M,K \vdash$ (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 C in which it appears, or any subtype of C

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

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

```
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() ?

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

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

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

```
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

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

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**

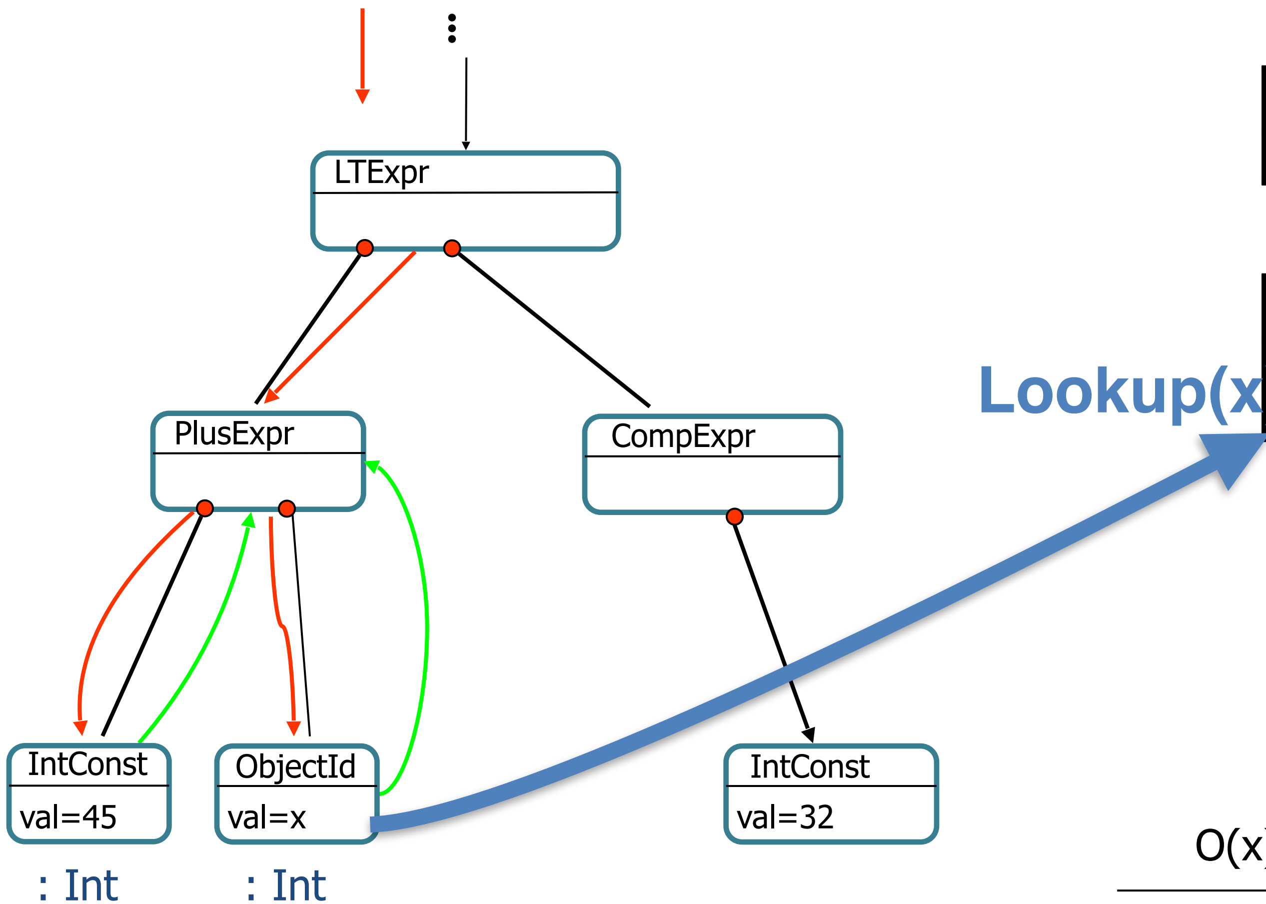
- When are types are created?
  - during lexer/parsing
  - predefined types

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

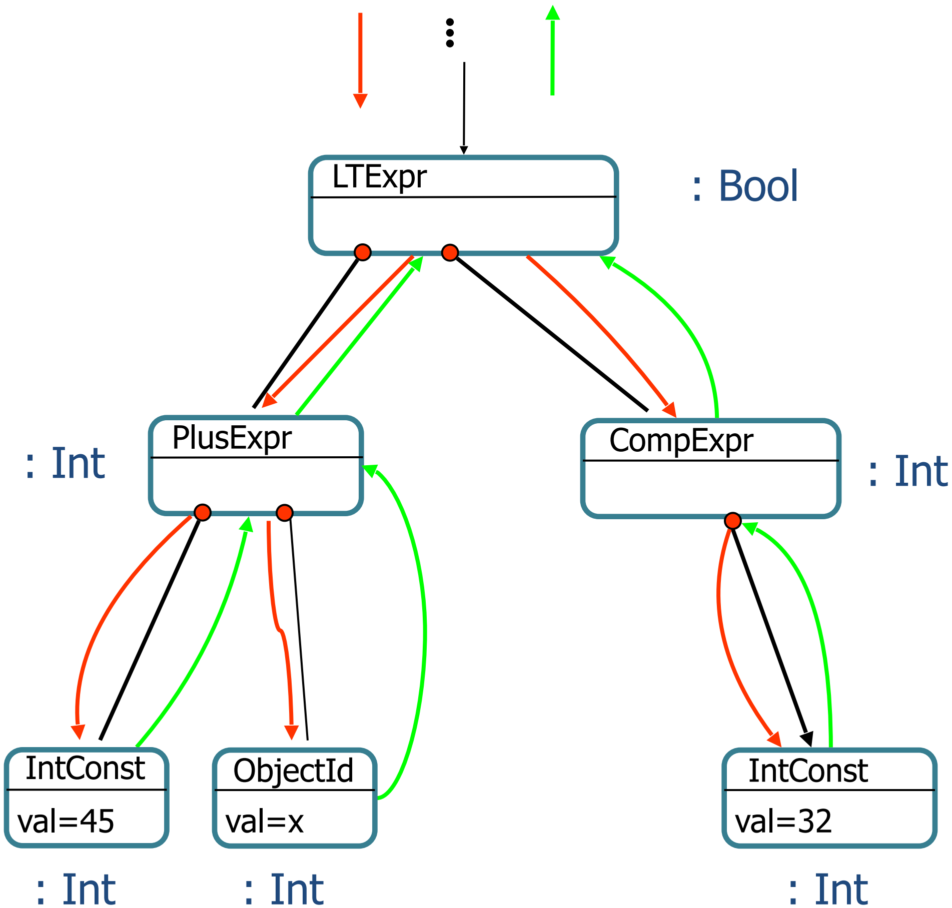
Symbol	kind	type
c	var	Int

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

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

(45 + x) < (~32)

Example



$(45 + x) < (\sim 32)$

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

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

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

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

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

Soundness of type rules

- Type is a set of values
- Soundness of static type rules
  - for every expression e,
    for every value v of e at runtime
    v $\in$values_of(static_type(e))
  - static_type(e) may actually describe more values
  - 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

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

Summary: type checking

- Type equivalence: nominal vs structural
- Expressions: locations (l-values) and values (r-values)
- Type coercions
- Types: strong vs weak types
- Types: dynamic vs static
- Type checking: dynamic vs static
- Type checking vs inference
- Subtyping relation and least upper bounds
- Soundness of type rules (conservative, provable)

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?