CS143: Semantic Analysis

David L. Dill

Stanford University

Semantic Analysis

- Introduction
- Scoped Symbol Tables
- Type Checking

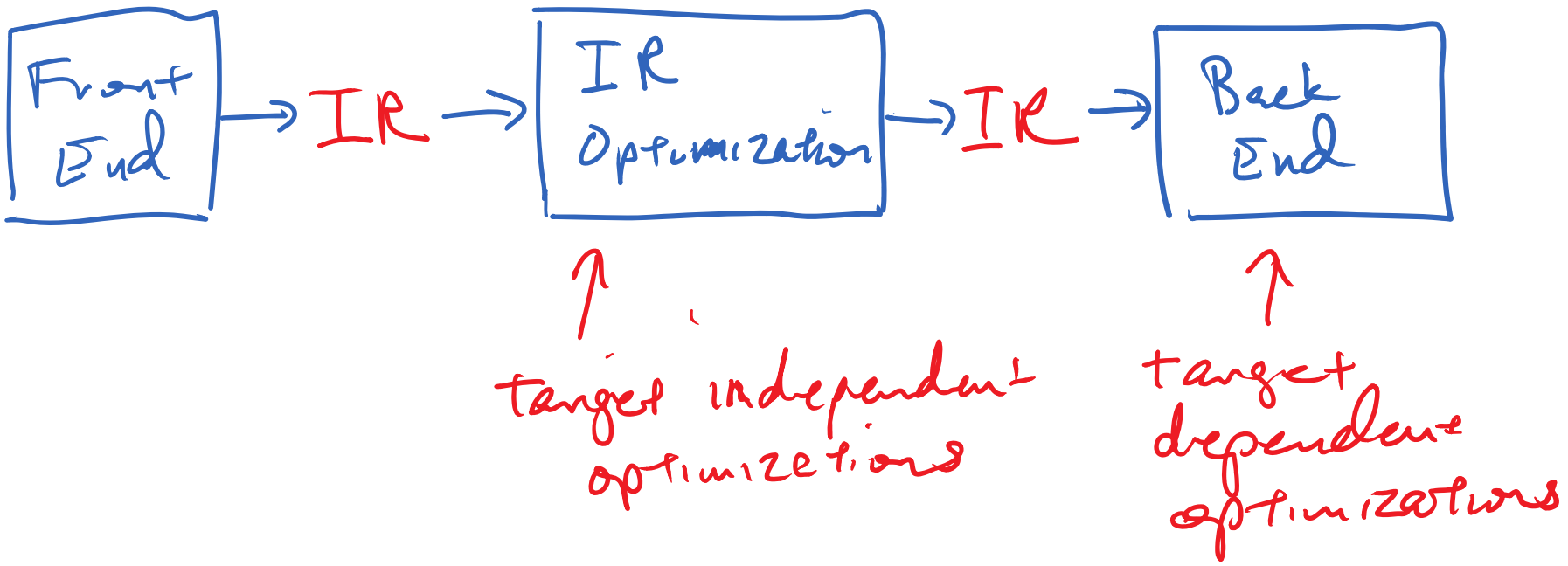
Introduction

"Run-time" vs "Compile-time"

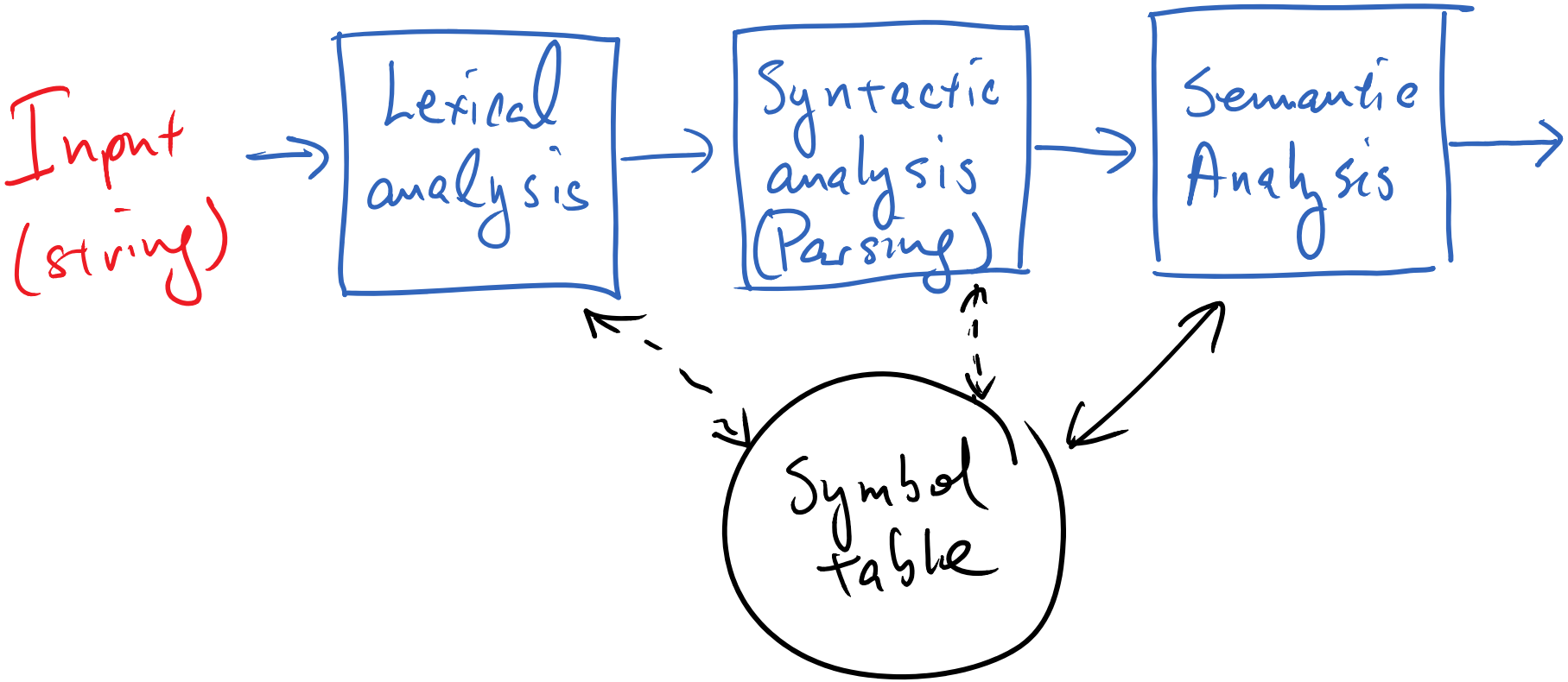
"Run-time" — happens when program is executed. "Dynamic"

"Compile-time" — happens when program is being compiled. "Static"

Better diagram



Front End


Input
(string)
Lexical analysis
Syntactic analysis (Parsing)
Semantic Analysis
Symbol table

Semantic Analysis

"Meaning" of program ← whatever that means

Machine-independent processing

"Static" - compiler can do it.

Not context-free — parser can't do it.

Semantic Analysis

- Process and store declarations

  Bind name to description

    Classes, attributes, methods,

    formal parameters, etc.

  Look up correct description
  for name

Errors: "undefined variable/class/etc."

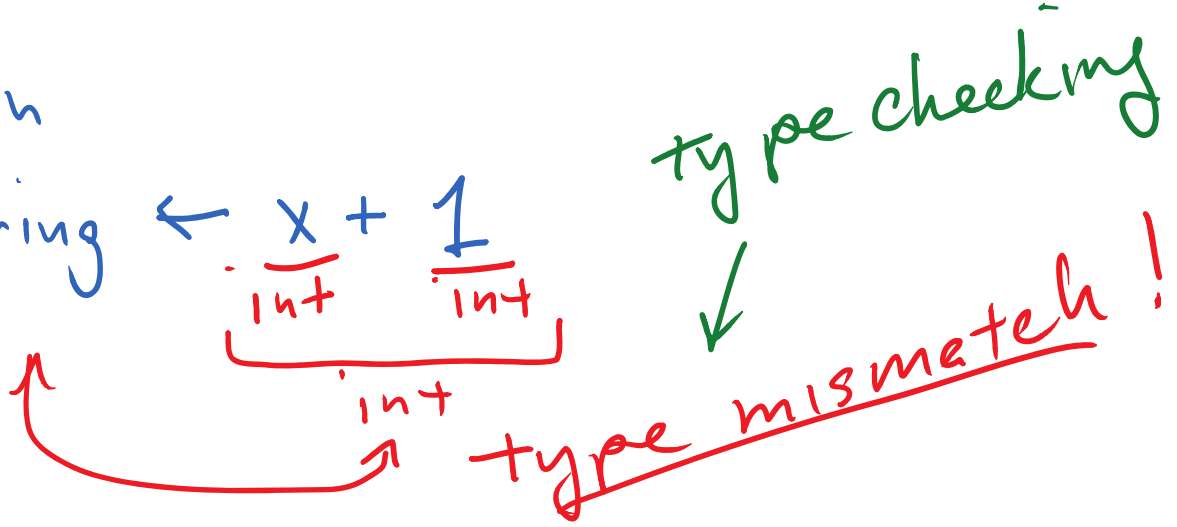
Semantic Analysis

Type checking

Inference - find type of expression

Checking - do types match?

```
Let x : Int in
  Let y : String ← x + 1
```



Semantic Analysis

Miscellaneous checks

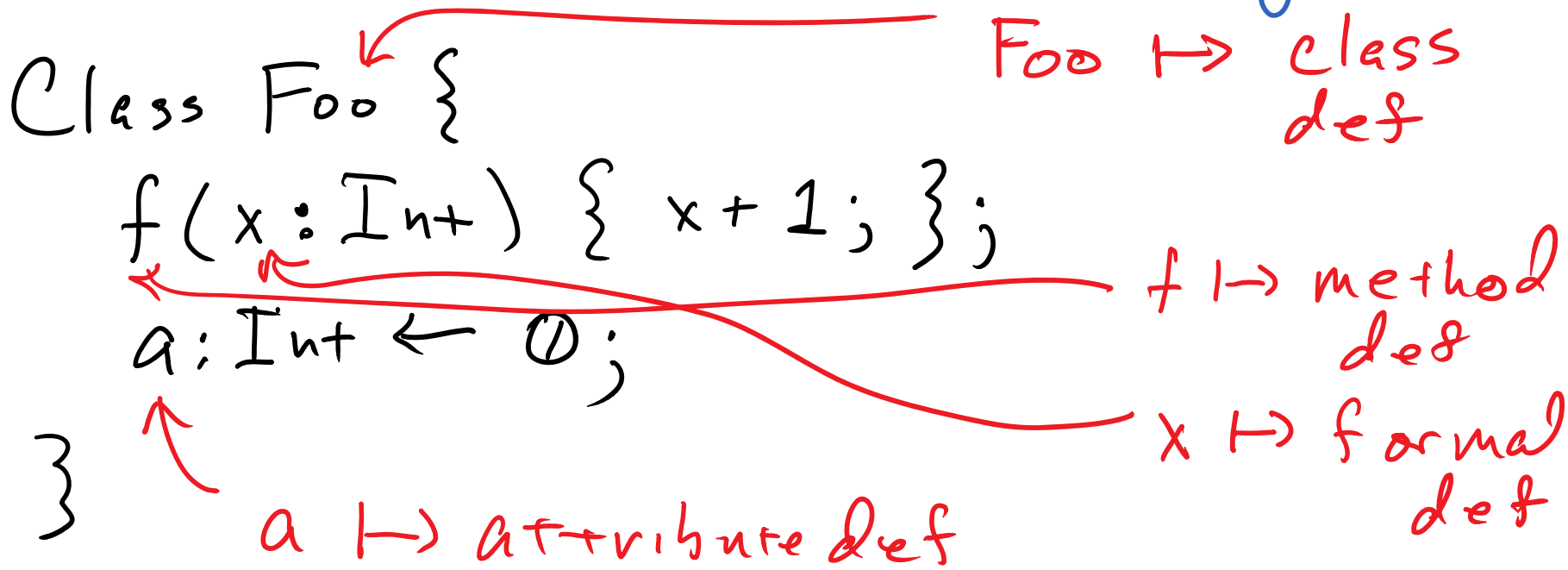
- Multiple definitions (if not allowed)
- Assignable values (true ← false)
- Proper inheritance (e.g, no cycles)
- Proper use of reserved words
- etc.

Highly language-dependent.

Scoped Symbol Tables

Declaration

Introduce a name for something.



Definitions (not complete)

Class — Parent class

Features: attribute & method declarations (private symbol table)

etc.

Attribute — type, initialization

Method — parameters, return type, body, etc.

Def: Name space - name → definition mapping
(i.e. "symbol table")

"Separate name spaces" - different symbol tables.

Choose based on identifiers context

Namespaces in Cool

Class names are in their own name space

Every class has a separate name space
methods, attributes are separate within
the class.

Def: Scope of a declaration - region of a program where declaration is accessible.

What declaration does ID refer to?

Def: Scope of a declaration = region of a program where declaration is accessible.

```
Let x : Int ← 0 in
  { x;
    Let x : Int ← 1 in
      x;
    x;
  }
```

nested scope

Example

Let x : Int ← 0 in e

declare x to be of type Int here

that definition applies in e (unless there is an inner Let x)

After e, decl of x is undone. Symbol table is restored to state before Let.

Symbol Table (symtab.h)

Logically, it is a stack of tables

enterscope() — start a new scope

exitscope() — restore symbol table to state just before matching enterscope

Symbol Table (symtab.h)

addid(s, d) – add a declaration d for symbol s to symtab.

lookup(s) – search for s in all enclosing scopes, starting with the innermost

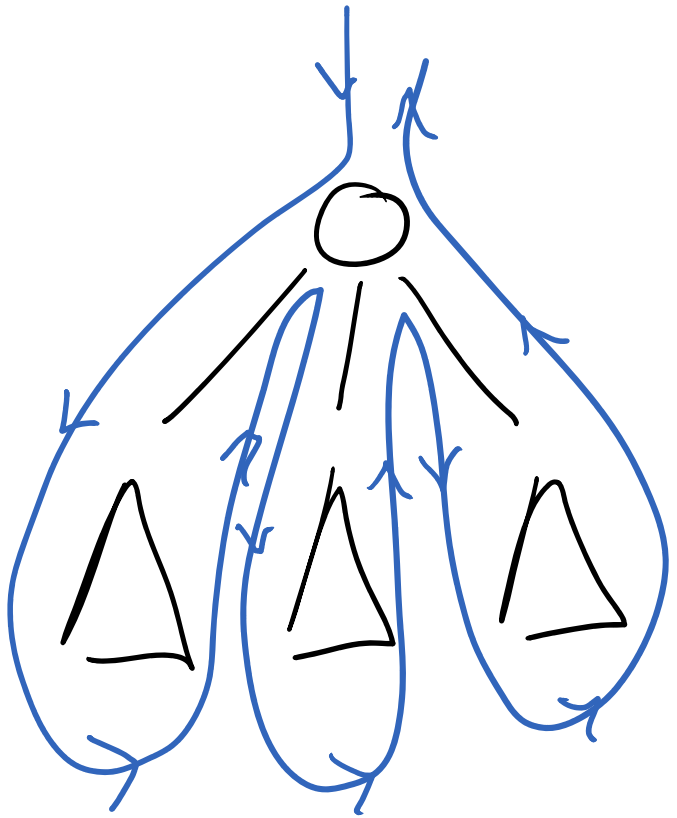
probe(s) – search for s in current scope only

Tree Traversal

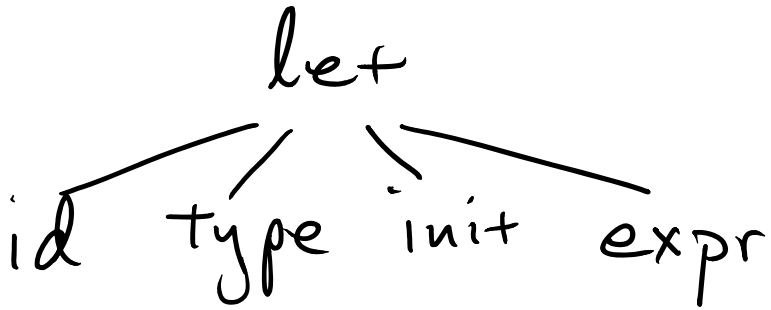
```
visit (n)
    do stuff
    visit (first child)
    do more stuff
    visit (next child)
    do more stuff
    do final stuff.
```

Tree Traversal

```
visit (n)
    do stuff
    visit (first child)
    do more stuff
    visit (next child)
    do more stuff
    do final stuff.
```



Tree Traversal



enter a scope

process type

add id/type to symtab

process expr

exit the scope

Def. Forward reference — use of a declaration appears earlier than the declaration itself.

Cool: Class names, features can be used before declaration

Enables recursive data structures, methods

Formals, let variables must be declared before uses.

Handling Class names, Features

Multiple Passes

Pass 1: Collect declarations

Use partial information if necessary

Pass 2: Processing that needs declarations

Use as many passes as is convenient.

Announcements

- Schedule revisions
  - Due dates pushed back a little
  - WA3/4 merged
- PA3 is hard. We're going to write up some advice this afternoon.
  - Start: Figure out how to find classes
  - Understand symbol table (and test it).

Type Checking

"Type" is very programming language dependent

Consensus: A type is

- A set of values
- A set of operations on those values

E.g. Classes

Goal of Type System

Machines think everything is a byte

No enforcement of reasonable operations for data.

No knowledge of size of data

Behavior of operations cannot change based on type of data.

Programming Language Types

Reduce application of operations to inappropriate data

Indicate size of data (e.g., for copying).

Allow behavior of operation to depend on data.

Dynamic type system – types are checked when operation is applied.

Run-time error reporting

Examples: Javascript, Python, LISP

Static typing – types are checked by compiler.

Examples: C++, Java (mostly), COOL

Dynamic types -

+ Fewer compile-time errors

- More run-time errors

(Less reliable code, more debugging)

- Greater runtime overhead in time and space.

Static typing - swap +, -

Def: Type Inference — fill in missing type information

If x : Int , y : Int , then (x + y) : Int

Def: Type checking — verify that all values are consistent with types.

Types in COOL

Class names

SELF_TYPE

User declares types of identifiers.

Compiler infers types of all expressions

Inference Rules

Formal notation borrowed from logic.

Often not used in language definitions.

No mainstream automated tools.

Soundness

No run-time type errors.

Whenever $\vdash e : T$

then run-time value of e has type T.

Conventional Notation

$$\frac{\vdash e_1 : \mathrm{Int} \qquad \vdash e_2 : \mathrm{Int}}{\vdash e_1 + e_2 : \mathrm{Int}}$$

premises

conclusion

If e_1 is of type Int and e_2 is of type Int, then $e_1 + e_2$ is of type Int.

Typing Complex Expressions

$$\frac{i \text{ is an int literal}}{\vdash i : \mathrm{Int}} \qquad \frac{\vdash e_1 : \mathrm{Int} \quad \vdash e_2 : \mathrm{Int}}{\vdash e_1 + e_2 : \mathrm{Int}}$$

$$\frac{\dfrac{1 \text{ is an int literal}}{\vdash 1 : \mathrm{Int}} \qquad \dfrac{2 \text{ is an int literal}}{\vdash 2 : \mathrm{Int}}}{\vdash 1 + 2 : \mathrm{Int}}$$

Code to Check Type Rules

Hand-written

One type rule per AST node

Premises are types of subexpressions

Types are computed bottom-up.

Rules for Constants

$$\frac{}{\vdash \text{false} : \text{Bool}}$$

$$\frac{s \text{ is a string literal}}{\vdash s : \text{String}}$$

More Rules

$$\frac{\vdash e : \text{Bool}}{\vdash \,!e : \text{Bool}}$$

$$\frac{\vdash e_1 : \text{Bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$

More Rules

$$\frac{\vdash e : \text{Bool}}{\vdash \text{!}e : \text{Bool}}$$

$$\frac{\vdash e_1 : \text{Bool} \quad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}}$$

↑ loop returns void

New

$$\frac{}{\text{new } T : T}$$

(ignore SELF_TYPE for now)

Variables

$$\frac{x \text{ is a variable}}{\vdash x : ?}$$

↑ we need to look up the declaration, somehow.

Solution: Add type environment (symbol table) to the rules.

O: maps Object IDs to types

If x is a variable, $O(x)$ is its type.

$$O \vdash e : T$$

"e has type T assuming the types of variables given in O."

We can add O to previous rules
(rules just ignore it)

E.g. $$\frac{O \vdash e_1 : \mathrm{Int} \qquad O \vdash e_2 : \mathrm{Int}}{O \vdash e_1 + e_2 : \mathrm{Int}}$$

Variable Rule

$$\frac{\Theta(x) = T}{\Theta \vdash x : T}$$

Let (without initialization)

Notation: $O[T/y]$ is the function O, modified to return T on argument y

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

type of e_1 is computed with $O[T/x]$.

Slope is limited to e_1

Let with initialization

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

Unnecessarily restrictive

It does not allow e_0 to be a class that inherits from T_0.

Subtyping

Partial order for inheritance

$T \leq T$ (reflexive)

$T \leq T'$ if T inherits from T'

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

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

Allows e_0 to have any subtype of declared type of X.

Assignment

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

Allows e_1 to be any subtype of declared type of x

Assignment

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

← assignment returns value of e_1, which has type T_1.

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

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

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

least $\forall T_4 \, (T_1 \leq T_4 \wedge T_2 \leq T_4 \rightarrow T_3 \leq T_4)$

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

$$\frac{\Gamma \vdash e_0 : \mathrm{Bool} \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1 \sqcup T_2}$$

Next

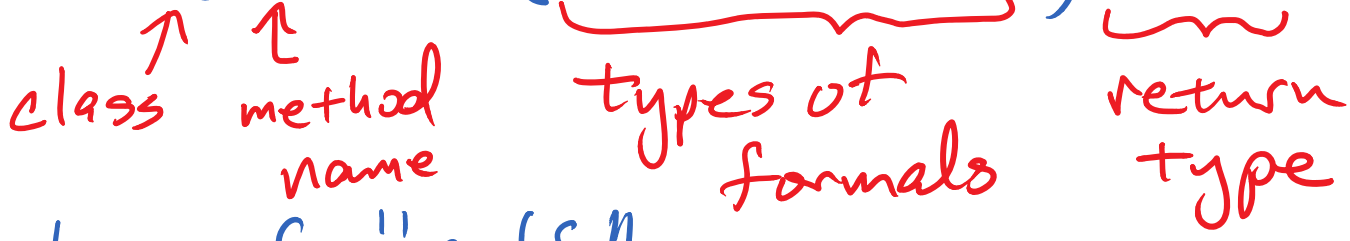
Need more stuff in environment

O – already have

M – methods

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

C – class of "self"



Implementation

Environment is passed down AST

Argument to recursive function

Types of expressions are computed bottom-up.

$tc(env, e_1 + e_2)$:

$T_1 = tc(env, e_1)$;

$T_2 = tc(env, e_2)$;

check $T_1 == T_2 == Int$

return Int;