Semantic Analysis

Slides based on material
by Ras Bodik available at
http://inst.eecs.berkeley.edu/~cs164/fa04

Symbol Table

Outline

- How to build symbol tables

- How to use them to find
  - multiply-declared and
  - undeclared variables.

The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal tokens
    - e.g.: main£ ();
- Parsing
  - Detects inputs with ill-formed parse trees
    - e.g.: missing semicolons
- Semantic analysis
  - Last “front end” phase
  - Catches all remaining errors

Introduction

- typical semantic errors:
  - multiple declarations: a variable should be declared (in the same scope) at most once
  - undeclared variable: a variable should not be used before being declared.
  - type mismatch: e.g. type of the left-hand side of an assignment should match the type of the right-hand side.
  - wrong argument number/types: methods should be called with the right number and types of arguments.

A simple semantic analyzer

- works in **two phases**
  - by **visiting** the AST created by the parser

1. Generation of **Enriched AST** (performed top-down)

   For each **scope** in the program:

   - **process the declarations** =
     - add new entries to the **symbol table** and
     - report any variable/method that is **multiply declared**
   - **process the statements =**
     - find uses of **undeclared variables**, and
     - in "**ID" nodes** (variable/method **usages**) of the AST add a pointer to the appropriate **symbol table entry**.

2. **Type** Checking (performed bottom-up)

   Process all of the statements in the program again,

   - use **attached symbol-table entry information** to determine the **type of each expression**, and to find type errors.

Symbol Table = map from names to entries

- purpose:
  - keep track of **names** declared in the program
  - names of
    - variables, classes, fields, methods,
- symbol table **entry**:
  - set of **attributes** associated with a **name**, e.g.:
    - kind of name (variable, class, field, method, etc)
    - type (int, float, etc)
    - nesting level
    - memory location (i.e., where it will be found at runtime).

Scoping

- symbol table design influenced by what kind of **scoping** is used by the compiled language

- In most languages, the **same name can be declared multiple times**
  - if its declarations occur in **different scopes**, and/or
  - involve **different kinds of names**.

Scoping: example

- Java: can use same name for
  - a class,
  - field of the class,
  - a method of the class, and
  - a local variable of the method
- ***legal Java program:***

```
class Test {
    int Test;
    void Test( ) { double Test; }
}
```

Scoping: overloading

- Java and C++ (but not in Pascal or C):
  - can use the same name for more than one method
  - as long as the number and/or types of parameters are unique.

```
int add(int a, int b);
float add(float a, float b);
```

Scoping: general rules

- The **scope rules** of a language:
  - determine **which declaration** of a named element (e.g. a variable) corresponds to each **use of the element.**
  - i.e., scoping rules map **uses of element** to their **declarations**.
- C++ and Java use ***static scoping***:
  - mapping from uses to declarations is **made at compile time**.
  - C++ uses the "most closely nested" rule
    - a use of variable x matches the declaration in the **most closely enclosing scope** such that the declaration precedes the use.
    - inner scope variable x declaration **hides** x declared in an outer scope.
  - in Java:
    - inner scopes cannot define variables defined in outer scopes

Scope levels

- Each function has one or more scopes:
  - one for the parameters and the function body,
  - and possibly additional scopes in the function
    - each nested block (delimited by curly braces)

Example (assume C++ rules)

```
void f( int y ) {           // y is a parameter
    int k = 0;              // k is a local variable
    while (….) {
        int k = 1;          // another local var, in a loop (not ok in Java)
    }
}
```

- the outmost scope includes just the name "f", and
- function f itself has two inner (nested) scopes:
  1. The scope for the body of f, which includes parameter y and local variable k that is initialized to 0.
  2. The innermost scope is for the body of the while loop, and includes the variable k that is initialized to 1.

TEST YOURSELF

- This is a C++ program. Match each use to its declaration, or say why it is a use of an undeclared variable.

```
class Foo {
   int k=10, x=20;
   void foo(int k) {
      int a = x;
      int x = k;
      int b = x;
      while (...) {
         int x=11;
         if (x == k) {
            int k, y;
            k = (y = x);
         }
         if (x == k) { int x, y; }
      }
}}
```

Dynamic scoping

- Not all languages use static scoping.
- Lisp, APL, and Snobol use **dynamic** scoping.
- Other languages, like Common Lisp or Perl, allow the programmer to chose among static or dynamic scoping
- Dynamic scoping:
  - Use of a variable with no corresponding declaration in the same function (**non-local** reference)
    - corresponds to the declaration in **most-recently-called still active** function.

Example

- For example, consider the following code:

```
char x;
void main() { f1(); f2(); }
void f1() { int x = 10; g(); }
void f2() { String x = "hello"; f3(); g(); }
void f3() { double x = 30.5; }
void g() { print(x); }
```

TEST YOURSELF

- Assuming that static (resp. dynamic) scoping is used, what is output by the following program?

```
{ int x = 0;
  void f() {int x=1; g();}
  void g() {print x;}
  f(); }
```

Static vs dynamic scoping

- generally, dynamic scoping is a bad idea
  - can make a program difficult to understand
  - a single **use** of a variable can correspond to
    - many **different declarations**
    - with **different types**!

Used before declared?

- can a name be used before it is defined?
  - Java: a method or field name *can* be used before the definition appears,
    - *not* true for a variable!

Example

```
class Test {
    void f() {
        val = 0;
        // field val has not yet been declared -- OK
        g();
        // method g has not yet been declared -- OK
        x = 1;
        // var x has not yet been declared -- ERROR!
        int x;
    }
    void g() {}
    int val;
}
```

Simplification

- From now on, assume that our language:
  - uses static scoping
  - requires that *all* names be declared before they are used
  - does not allow multiple declarations of a name in the same scope
    - e.g. no method overloading
    - even for different kinds of names
      - e.g. field and method with the same name not allowed
  - *does* allow the same name to be declared in multiple nested scopes
    - but only once per scope

Symbol Table Implementations

- In addition to the above simplification, assume that the **symbol table will be used to answer two questions**:

  1. Given a **declaration** of a name, is there already a declaration of the same name in the **current scope**
     - i.e., is it multiply declared?

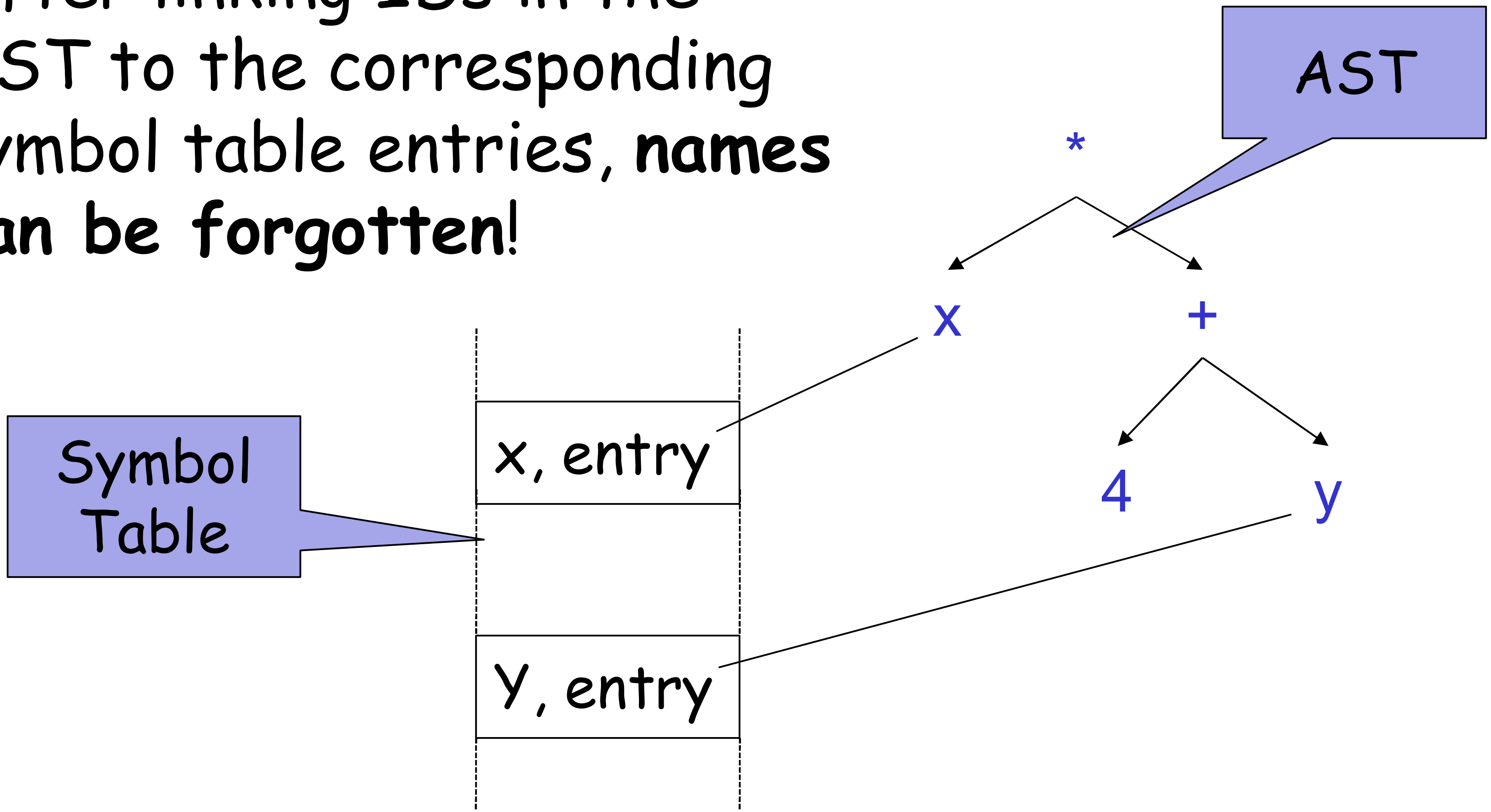
  2. Given a **use** of a name, to **which declaration does it correspond** (using the "most closely nested" rule), or is it undeclared?

Note

- The symbol table is **only needed** to answer those two questions, i.e.
  - once all **declarations** have been processed to **build the symbol table**,
  - and all uses have been processed to **link** each **ID** node in the abstract-syntax tree with the corresponding **symbol-table entry**,
  - then the symbol table itself **is no longer needed**
    - because no more **lookups based on name** will be performed

Graphically…

- After linking IDs in the AST to the corresponding symbol table entries, **names can be forgotten**!



What operation do we need?

- Given the above assumptions, we will need:
  1. (for name declarations)
     Look up a name in the current scope only
     - to check if it is multiply declared

     and **insert a new name** into the symbol table **mapped to an entry** containing its attributes.

  2. (for name usages)
     Look up a name in the **current** and **enclosing scopes**
     - to check for a use of an undeclared name, and
     - to link a use with the corresponding symbol-table entry.

  3. Do what must be done when a **new scope is entered**.
  4. Do what must be done when a **scope is exited**.

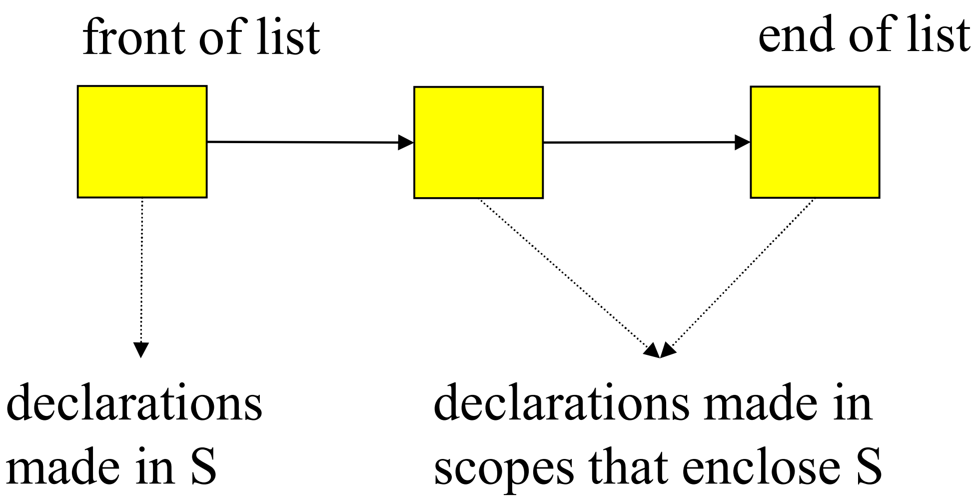
Two possible symbol table implementations

1. a list of tables
2. a table of lists

- For each approach, we will consider
  - what must be done when processing a declaration,
  - when processing a use, and
  - when entering and exiting a scope.
- Simplification:
  - assume each symbol-table **entry** includes only:
    - its **type** and the **nesting level** of its declaration

Method 1: List of Hashtables

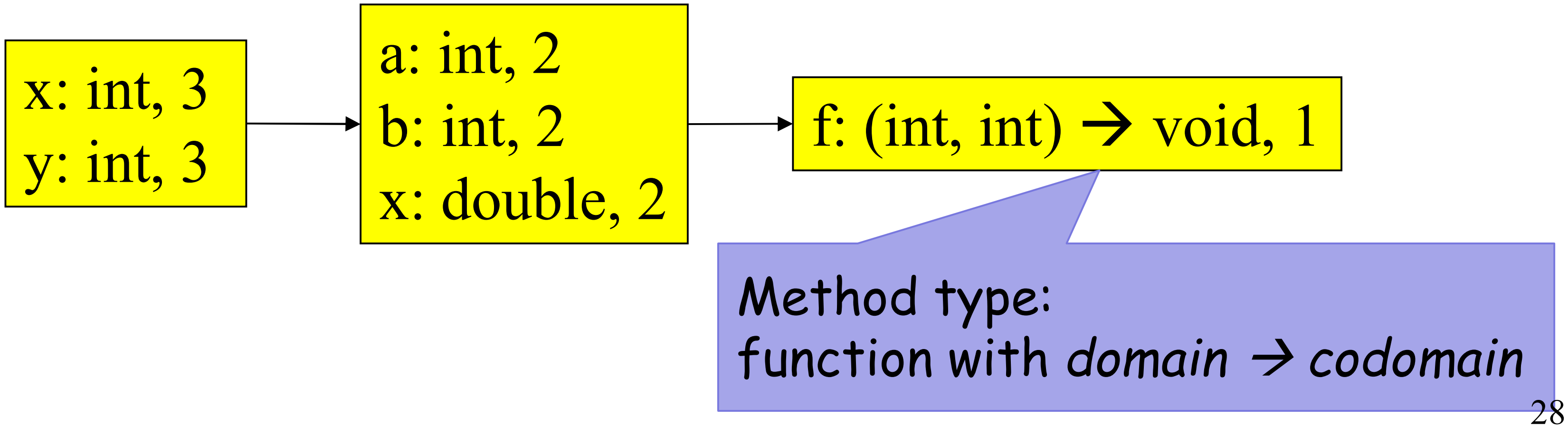
- The idea:
  - symbol table = a list of hashtables,
  - one hashtable for each currently visible scope.
- When processing a scope S:



Example:

```
void f(int a, int b) {
   double x;
   while (...) { int x, y; ... }
}
void g() { f(); }
```

- *After processing declarations inside the while loop:*



List of hashtables: the operations

1. On scope entry:
   - increment the **current level number** and add a new empty hashtable to the front of the list.
2. To process a declaration of x:
   - look up x in the first table in the list.
     - If **it is there**, then issue a "**multiply declared variable**" error;
     - otherwise, **add x** to the first table in the list **mapped** to a **new entry** containing x type and nesting level.

… continued

3. To process a use of x:
   - look up x starting in the first table in the list;
     - if it is not there, then look up x in **each successive table in the list**.
     - **link** the **use of x** with the found symbol-table **entry**
     - If, instead, **x is not in *any* table** then issue an "**undeclared variable**" error.
4. On scope exit,
   - remove the first table from the list and decrement the **current level number**.

Remember

- method names belong to the hashtable for the outer scope (w.r.t. inner method scopes)
  - i.e. not to the same table as the method's variables/parameters

- For instance, in the example above:
  - method name f is in the symbol table for the outermost scope
  - name f is *not* in the same scope as parameters a and b, and variable x.
  - This is so that when the use of name f in method g is processed, the name is found in an enclosing scope's table.

The running times for each operation:

1. **Scope entry**:
   - time to initialize a new, empty hashtable;
   - probably proportional to the size of the hashtable.
2. **Process a declaration:**
   - using hashing, constant expected time (O(1)).
3. **Process a use**:
   - using hashing to do the lookup in each table in the list, the worst-case time is O(depth of nesting), when every table in the list must be examined.
4. **Scope exit**:
   - time to remove a table from the list, which should be O(1)

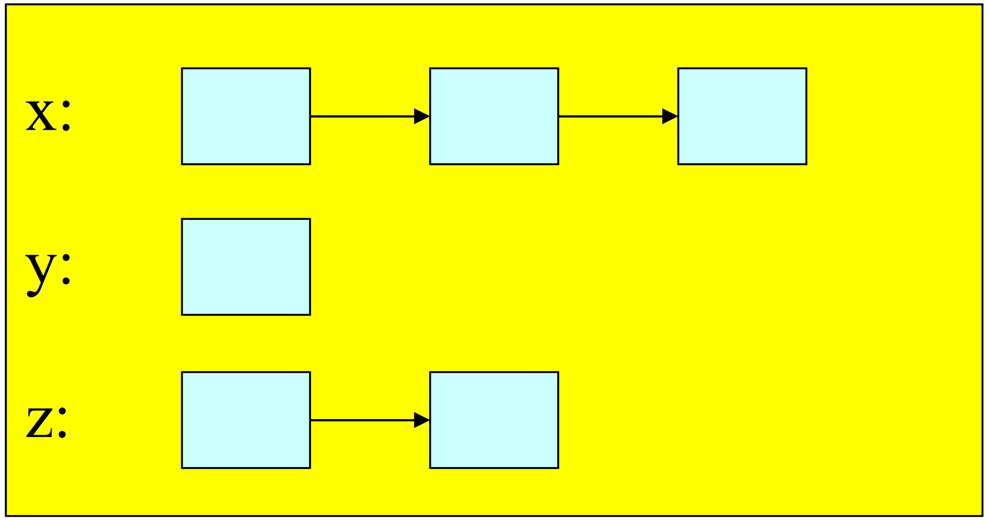
TEST YOURSELF

- Assume that the symbol table is implemented using a list of hashtables.
- Draw pictures to show how the symbol table changes as each declaration in the following code is processed.

```
void g(int x, int a) {
    double d;
    while (...) {
        int d, w;
        double x, b;
        if (...) { int a,b,c; }
    }
    while (...) { int x,y,z; }
}
```

Method 2: Hashtable of Lists

- the idea:
  - when processing a scope S, the structure of the symbol table is:



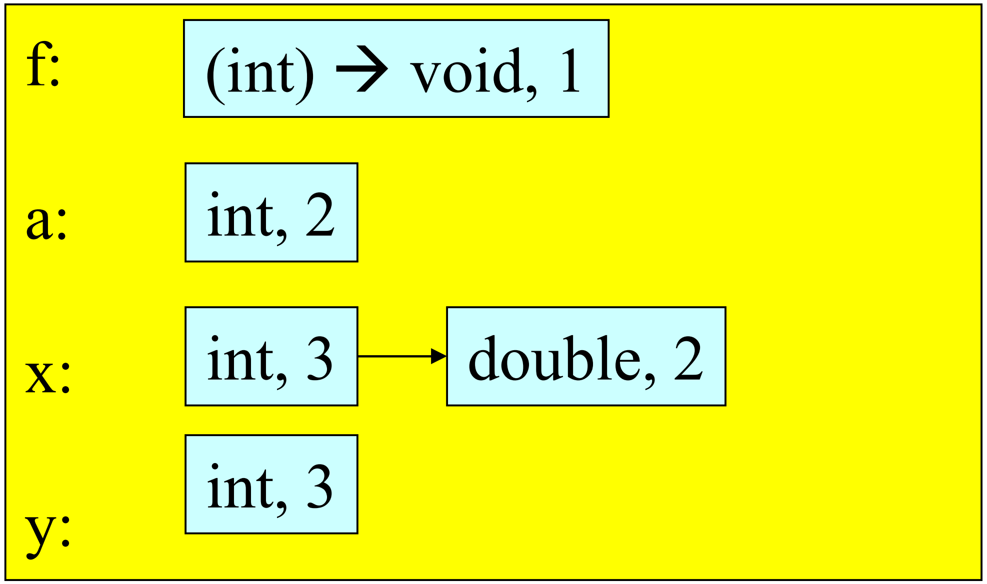
Definition

- there is just one big hashtable, containing an entry for each name for which there is
  - some declaration in scope S or
  - in a scope that encloses S.
- Associated with each name is a list of symbol-table entries.
  - The first list item corresponds to the most closely enclosing declaration;
  - the other list items correspond to declarations in enclosing scopes.

Example

```
void f(int a) {
   double x;
   while (...) { int x, y; ... }
}
void g() { f(); }
```

- After processing the declarations inside the while loop:



Nesting level information is crucial

- the **level-number attribute** stored in each list item enables us to determine **whether the most closely enclosing declaration was made**
  - in the current scope or
  - in an enclosing scope.

Hashtable of lists: the operations

1. On scope entry:
   - increment the **current level number**.
2. To process a declaration of x:
   - look up x in the symbol table.
     - If **x is there**, fetch **the level number** from the **first** list item.
       - If that **level number = current level** then issue a "**multiply declared variable**" error;
       - otherwise, **add a new item** to the front of the list with the appropriate **type** and the **current level number**.

… continue

3. To process a use of x:
   - Look up x in the symbol table.
     - If it is there, **link** the **use of x** with the symbol-table **entry** at the front of the list
     - If it is not there, then issue an "undeclared variable" error.
4. On scope exit:
   - **Scan all the names** in the symbol table, looking at the **first item** on each list.
   - If that item's **level number = current level** number, then **remove it** from its list
     - and if the **list becomes empty**, **remove the name**.
   - Finally, decrement the **current level number**.

Running times

1. **Scope entry:**
   - time to increment the level number, O(1).
2. **Process a declaration:**
   - using hashing, constant expected time (O(1)).
3. **Process a use:**
   - using hashing, constant expected time (O(1)).
4. **Scope exit:**
   - time proportional to the number of names in the symbol table.

TEST YOURSELF

- Assume that the symbol table is implemented using a hashtable of lists.
- Draw pictures to show how the symbol table changes as each declaration in the following code is processed.

```
void g(int x, int a) {
    double d;
    while (...) {
        int d, w;
        double x, b;
        if (...) { int a,b,c; }
    }
    while (...) { int x,y,z; }
}
```

Type Checking

Types

- What is a type?
  - The notion varies from language to language

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

- Classes are one instantiation of the modern notion of type

Why Do We Need Type Systems?

Consider the assembly language fragment

add $r1, $r2, $r3

What are the types of $r1, $r2, $r3?

Types and Operations

- Most operations are legal only for values of some types
  - It doesn't make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies 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, because nothing else will!

- Type systems provide a concise formalization of the semantic checking rules

What Can Types do For Us?

- Can detect certain kinds of errors
- Memory errors:
  - Reading from an invalid pointer, etc.
- Violation of abstraction boundaries:

```
class FileSystem {
   open(x : String) : File {
      …
   }
…
}
```

```
class Client {
   f(fs : FileSystem) {
      File fdesc ← fs.open("foo")
      …
   } -- f cannot see inside fdesc !
}
```

Type Checking Overview

- Three kinds of languages:
  - *Statically typed:* All or almost all checking of types is done as part of compilation (C)
  - *Dynamically typed:* Almost all checking of types is done as part of program execution (Scheme)
  - *Untyped:* No type checking (machine code)

The Type Wars

- Competing views on static vs. dynamic typing
- Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks
- Dynamic typing proponents say:
  - Static type systems are restrictive
    - e.g. due to the type **statically associated/fixed** for a variable
      - for variable "**double x**" we cannot do "**int y=x**" even if at run-time "**x**" **actually contains an integer**
      - programmers end up using **casts** escaping the type system
  - Rapid prototyping easier in a dynamic type system

Type Checking and Type Inference

- *Type Checking* is the process of checking that the program obeys the type system

- Often involves inferring types for parts of the program
  - Some people call the process *type inference*

Rules of Inference

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions (for the lexer)
  - Context-free grammars (for the parser)

- The appropriate formalism for type checking is logical rules of inference

Why Rules of Inference?

- Inference rules have the form
  If Hypothesis is true, then Conclusion is true

- Type checking computes via reasoning
  If E_1 and E_2 have certain types, then E_3 has a certain type

From English to an Inference Rule

- The notation is easy to read (with practice)

- Start with a simplified system and gradually add features

- Building blocks
  - Symbol $\wedge$ is "and"
  - Symbol $\Rightarrow$ is "if-then"
  - x:T is "x has type T"

From English to an Inference Rule (2)

If e_1 has type Int and e_2 has type Int,
then $e_1 + e_2$ has type Int

(e_1 has type Int $\wedge$ e_2 has type Int) $\Rightarrow$
$e_1 + e_2$ has type Int

$(e_1\text{: Int} \wedge e_2\text{: Int}) \Rightarrow e_1 + e_2\text{: Int}$

From English to an Inference Rule (3)

The statement

$$(e_1\!: \text{Int} \wedge e_2\!: \text{Int}) \Rightarrow e_1 + e_2\!: \text{Int}$$

is a special case of

$$(\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n) \Rightarrow \text{Conclusion}$$

The latter is an inference rule usually written:

$$\frac{\vdash \text{Hypothesis}_1 \quad \ldots \quad \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

...and the above rules is written:

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

Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\vdash \text{Hypothesis}_1 \quad \ldots \quad \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

- Type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means "we can prove that . . ."

An example

- An inference system for proving $x < y$

$$\frac{}{\vdash x < (x+1)}$$

[A]
(Axiom stating that every number is strictly smaller than its successor)

$$\frac{\vdash x < y \quad \vdash y < z}{\vdash x < z}$$

[T]
(Transitivity of <)

An example (continued)

- How to prove that $6 < 10$?

$$\vdash 6 < 10$$

An example (continued)

- How to prove that $6 < 10$?

$$\dfrac{\vdash 6 < 7 \qquad \vdash 7 < 10}{\vdash 6 < 10}\,[T]$$

An example (continued)

- How to prove that $6 < 10$?

$$\cfrac{\cfrac{}{\vdash 6 < 7}\,[A] \qquad\qquad \vdash 7 < 10}{\vdash 6 < 10}\,[T]$$

An example (continued)

- How to prove that $6 < 10$?

$$
\dfrac{\dfrac{}{\vdash 6 < 7}\,[A] \qquad \dfrac{\vdash 7 < 9 \qquad \vdash 9 < 10}{\vdash 7 < 10}\,[T]}{\vdash 6 < 10}\,[T]
$$

An example (continued)

- How to prove that $6 < 10$?

$$\dfrac{\dfrac{}{\vdash 6 < 7}\,[A] \qquad \dfrac{\vdash 7 < 9 \qquad \dfrac{}{\vdash 9 < 10}\,[A]}{\vdash 7 < 10}\,[T]}{\vdash 6 < 10}\,[T]$$

An example (continued)

- How to prove that $6 < 10$?

$$\dfrac{\dfrac{}{\vdash 6 < 7}\,[A] \qquad \dfrac{[T]\,\dfrac{\vdash 7 < 8 \qquad \vdash 8 < 9}{\vdash 7 < 9} \qquad \dfrac{}{\vdash 9 < 10}\,[A]}{\vdash 7 < 10}\,[T]}{\vdash 6 < 10}\,[T]$$

An example (continued)

- How to prove that $6 < 10$?

$$
\dfrac{\dfrac{}{\vdash 6 < 7}\,[A] \qquad \dfrac{\dfrac{\dfrac{}{\vdash 7 < 8}\,[A] \quad \dfrac{}{\vdash 8 < 9}\,[A]}{\vdash 7 < 9}\,[T] \qquad \dfrac{}{\vdash 9 < 10}\,[A]}{\vdash 7 < 10}\,[T]}{\vdash 6 < 10}\,[T]
$$

Type system: Two Rules

$$\frac{(i \text{ is an integer token})}{\vdash i : \text{Int}} \; [\text{Int}]$$

(this kind of premises are usually written as side conditions, as they do not require other inferences)

$$\frac{}{\vdash i : \text{Int}} \; [\text{Int}] \quad (i \text{ is an integer token})$$

$$\frac{\begin{array}{c}\vdash e_1 : \text{Int} \\ \vdash e_2 : \text{Int}\end{array}}{\vdash e_1 + e_2 : \text{Int}} \; [\text{Add}]$$

Type system: Two Rules (Cont.)

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions

$$\dfrac{\dfrac{}{\vdash 1 : \text{Int}} \qquad \dfrac{}{\vdash 2 : \text{Int}}}{\vdash 1 + 2 : \text{Int}}$$

Soundness

- A type system is <u>sound</u> if
  - Whenever we can infer $\vdash e : T$
  - Then at run-time e actually evaluates to a value of type T
- We only want sound rules

Type Checking Proofs

- Type checking proves facts $e : T$
  - **One type rule** is used for **each kind** of expression, e.g. previous rule used if e is in the form $e_1 + e_2$

- In the **type rule used** for typing e:
  - The hypotheses are the proofs of types of e's subexpressions
  - The conclusion is the proof of type of e

Rules for Constants

$$\frac{}{\vdash \text{false} : \text{Bool}} \quad [\text{Bool}]$$

$$\frac{}{\vdash s : \text{String}} \quad [\text{String}]$$

(s is a string constant)

Object Creation Example

$$\frac{}{\vdash \text{new } C() : C} \text{ [New]}$$

(*C* denotes a class with parameterless constructor)

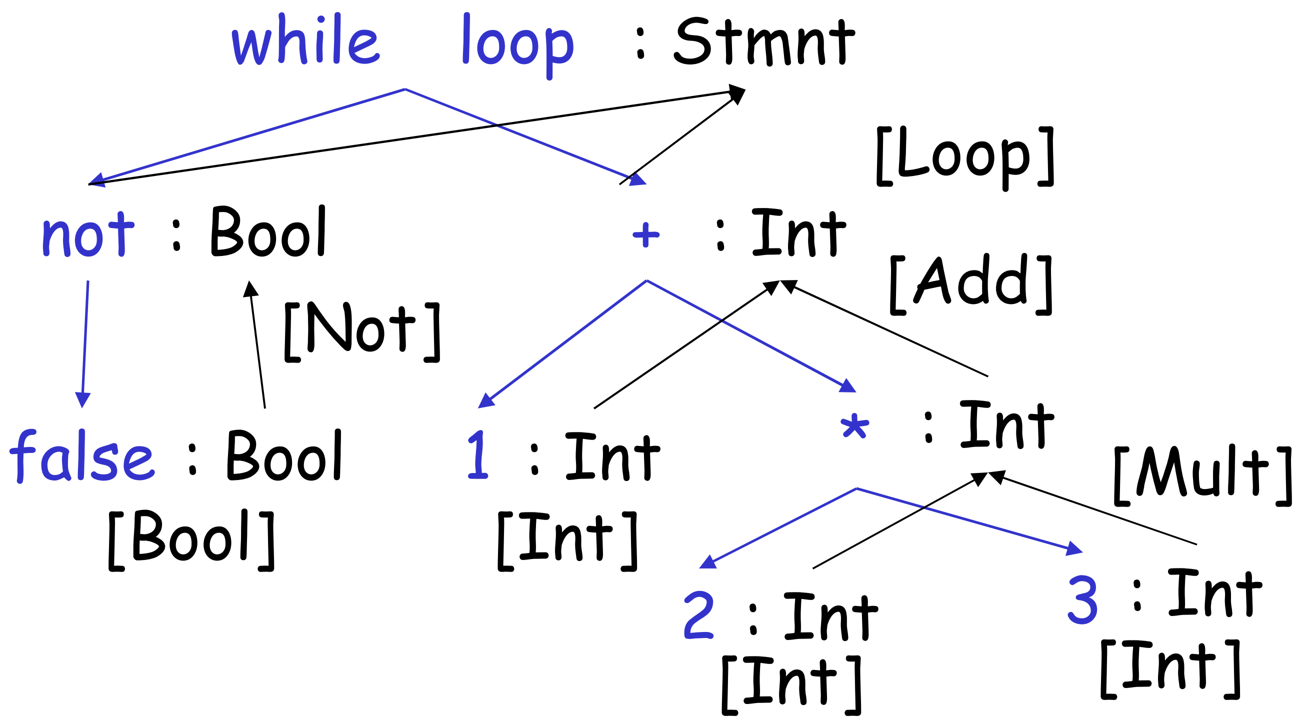
Two More Rules

$$\frac{\vdash e : Bool}{\vdash \text{not } e : Bool} \quad [\text{Not}]$$

$$\frac{\begin{array}{c}\vdash e_1 : Bool \\ \vdash e_2 : T\end{array}}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : Stmnt} \quad [\text{Loop}]$$

Typing: Example

- Typing for while not false loop 1 + 2 * 3 pool



Typing Derivations

- The typing reasoning can be expressed as a tree:

$$\dfrac{\dfrac{\vdash \text{false} : \text{Bool}}{\vdash \text{not false} : \text{Bool}} \qquad \dfrac{\vdash 1 : \text{Int} \qquad \dfrac{\vdash 2 : \text{Int} \qquad \vdash 3 : \text{Int}}{\vdash 2 * 3 : \text{Int}}}{\vdash 1 + 2 * 3 : \text{Int}}}{\vdash \text{while not false loop } 1 + 2 * 3 \text{ pool} : \text{Stmnt}}$$

- The root of the tree is the whole expression
- Each node is an instance of a typing rule
- Leaves are the rules with no hypotheses

A Problem

- What is the type of a variable usage?

$$\frac{}{\vdash x : ?} \text{[Id]}$$

(x is an identifier)

- This rule does not have enough information to give a type.
  - We need an **assumption** of the form "*we are in the scope of a declaration of x with type T*")

A Solution: Put more information in the rules!

Let O be a function from Identifiers to Types

The sentence $O \vdash e : T$

is read:

Under the **assumption** that the **variables in the current scope have the types given by** O, it is provable that expression e has type T

- NOTE: Corresponds to the information stored in the **symbol table**!

Type Environments

- *A type environment* O gives **types** for ***free* variables**
  - A variable is *free* in an (sub)expression if:
    - The expression contains an occurrence of the variable that refers to a declaration outside the expression
  - E.g. in "int f(int x) {return x+y}" only "y" is free
  - E.g. in the (sub)expression "x", the variable "x" is free

Modified Rules

The type environment is added to the earlier rules:

$$\frac{}{O \vdash i : \text{Int}} \quad \text{[Int]} \quad \text{(i is an integer)}$$

$$\frac{\begin{array}{c} O \vdash e_1 : \text{Int} \\ O \vdash e_2 : \text{Int} \end{array}}{O \vdash e_1 + e_2 : \text{Int}} \quad \text{[Add]}$$

New Rules

And we can write new rules:

$$\frac{}{O \vdash x : T} \quad [\text{Id}] \qquad (\text{if } O(x) = T)$$

Function invocation

The type of a function is usually written:

- $T_1,\ldots,T_n \to T$
- with $T_1,\ldots,T_n$ types of the input parameters
- and T the output return type

$$\frac{O \vdash f:(T_1,\ldots,T_n \to T) \quad O \vdash e_1:T_1 \quad \ldots \quad O \vdash e_n:T_n}{O \vdash f(e_1,\ldots,e_n) : T} \quad [\text{FunCall}]$$

Let (from languages like ML)

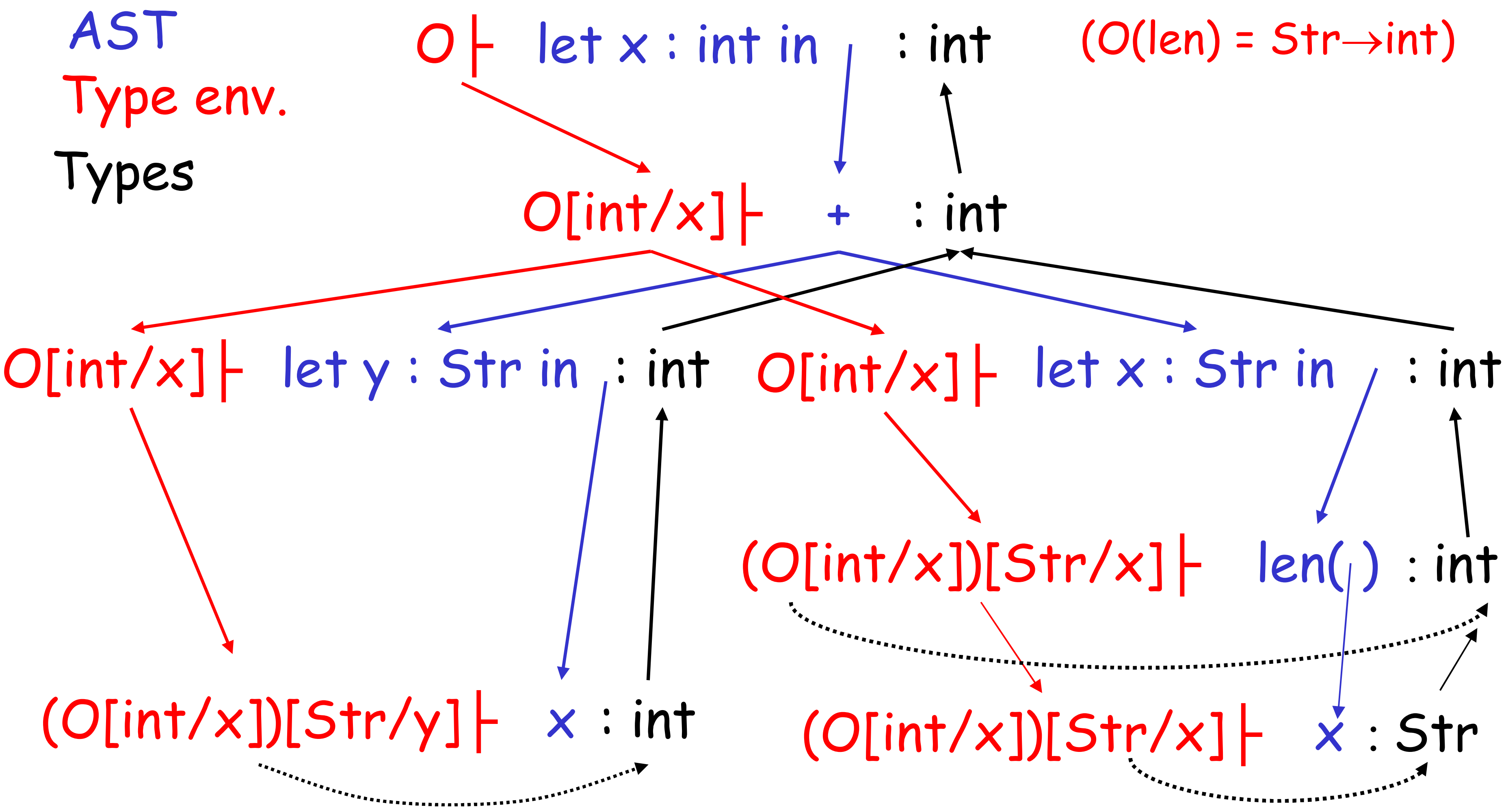
- Let statement declares a variable x with given type T_0 that is then defined throughout e_1 :
  - let $x : T_0$ in e_1 (without initialization)
  - let $x : T_0 \leftarrow e_0$ in e_1 (with initialization)

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \text{ in } e_1 : T_1} \quad \text{[Let-No-Init]}$$

Let. Example.

- Consider the expression
  let $x : T_0$ in ((let $y : T_1$ in $E_{x,y}$) + (let $x : T_2$ in $F_{x,y}$))
  (where $E_{x,y}$ and $F_{x,y}$ are some expressions that may contain occurrences of "x" and "y")
- Declaration
  - of "y" applies to $E_{x,y}$
  - of outer "x" applies to $E_{x,y}$
  - of inner "x" applies to $F_{x,y}$
- This is captured precisely in the typing rule.

Let Example.


AST
Type env.
Types
(O(len) = Str→int)
O ⊢ let x : int in : int
O[int/x] ⊢ + : int
O[int/x] ⊢ let y : Str in : int
O[int/x] ⊢ let x : Str in : int
(O[int/x])[Str/y] ⊢ x : int
(O[int/x])[Str/x] ⊢ len( ) : int
(O[int/x])[Str/x] ⊢ x : Str

Notes

- The type environment gives types to the free identifiers in the current scope
- The type environment is passed down the AST from the root towards the leaves
- Types are computed up the AST from the leaves towards the root

  - NOTE: in compiler implementations, the "downward" phase generates an **enriched version of the AST** where the **identifiers** are linked to their **symbol table entry** (indicating their type)

Let with Initialization

Consider a let with initialization:

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \text{ [Let-Init]}$$

This rule is too "restrictive". Why?

Let with Initialization

- Consider the example:

    class C inherits P { … }

    …

    let x : P ← new C in …

     …

- The previous let rule does not allow this code
  - We say that the rule is too "restrictive"

Subtyping

- Define a relation $X \leq Y$ on classes to say that:
  - An object of type **X can be used** when one of type **Y is acceptable**, or equivalently
  - X conforms with Y
- Also known as "X is a subclass of Y"
  - Consider a relation $\leq$ on classes

    $X \leq X$

    $X \leq Y$ if X **inherits** from Y

    $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

Let with Initialization (Again)

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ T \leq T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \text{[Let-Init]}$$

- Both rules for let are sound
- But more programs type check with the latter, which is more “flexible”

Let with Subtyping. Notes.

- There is a tension between
  - “Flexible” rules that do not constrain programming
  - “Restrictive” rules that ensure safety of execution

Expressiveness of Static Type Systems

- A **static type system** enables a compiler to detect many common programming errors
- The cost is that **some correct programs are disallowed**
  - Some argue for dynamic type checking instead
  - Others argue for more expressive static type checking

- But more expressive type systems are also **more complex**

Static and Dynamic Types

- The *static type* of an object **variable** is the class *C* used to **declare** the variable.
  - This notion is then extended to **expressions** over object variables.
  - *A* **compile-time notion**
- The *dynamic type* of an object **variable** is the class *C* that is used to create its object **value** ("new *C*").
  - This notion is then extended to **expressions** over object variables.
  - *A* **run-time notion**

Static and Dynamic Types. (Cont.)

- In early type systems static types correspond directly with dynamic types
- Soundness theorem: for all expressions E

  dynamic_type(E) = static_type(E)

  (in **all** executions, E evaluates to values of the type inferred by the compiler)

- This gets more complicated in **advanced** type systems

Ex. of Code that Type Checks with the Rules

```
class A: …
class B extends A: …
main() {
  A x = new A();
  …
  x = new B();
  …
}
```

x has static type A

Here, x has dynamic type A

Here, x has dynamic type B

- According to rules: variable x with static type A can be assigned values of type B if $B \leq A$
  - **Liskov** substitution principle:
    subtypes can be used in place of supertypes
- Hence: dynamic_type(x) $\leq$ static_type(x)

Soundness of the Type Checking System

Soundness theorem:

$$\forall E. \quad dynamic\_type(E) \leq static\_type(E)$$

Why is this Ok?

- For E, compiler uses static_type(E) (call it C)
- All operations that can be used on an object of type C can also be used on an object of type C' ≤ C
  - Objects of type **C' must expose all the public methods and public fields exposed by C**
  - Objects of type **C' could have additional public methods and fields**, but they will be not used if the static type is C

Let. Examples.

- Consider the following class definitions

```
Class A { a() : Int { return 0 }; }
Class B inherits A { b() : Int { return 1 }; }
```

- An instance of B has methods "a" and "b"
- An instance of A has method "a"
  - A run-time error occurs if we try to invoke method "b" on an instance of A
- Supertype cannot be used in place of a subtype!

Example of Wrong Let Rule (1)

- Now consider a hypothetical let rule:

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

- How is it different from the correct rule?

- The following good program does not typecheck

$$\text{let } x : \text{Int} \leftarrow 0 \text{ in } x + 1$$

- And some bad programs do typecheck

$$\text{foo}(x : B) : \text{Int} \; \{ \text{ let } x : A \leftarrow \text{new } A() \text{ in } x.b() \; \}$$

Example of Wrong Let Rule (2)

- Now consider another hypothetical let rule:

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

- How is it different from the correct rule?

- The following bad program is well typed

  let x : B ← new A() in x.b()

- Why is this program bad?

Recall the correct Let-Init rule

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ T \leq T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \quad \text{[Let-Init]}$$

- We also need a rule **for just assigning a value** to **an already declared** variable x

Assignment of an already declared x

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ T \leq T_0 \\ O \vdash e_1 : T_1 \end{array}}{O \vdash \; x \leftarrow e_0 \; ; \; e_1 \; : T_1} \quad \text{[Assign]} \quad (\text{if } O(x) = T_0)$$

- E.g. the following program is well typed:
  let x : A ← new B() in {… x ← new A(); x.a() }

Example of Wrong Let Rule (3)

- Now consider another hypothetical let rule:

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

- How is it different from the correct rule?

- The following good program is not well typed

  let x : A ← new B() in {… x ← new A(); x.a() }

- Why is this program not well typed?

Function invocation with subtyping

Function f with type:

- $T_{0,1},\ldots,T_{0,n} \to T$
- with $T_{0,1},\ldots,T_{0,n}$ types of the input parameters
- and T the output return type

$$\frac{O \vdash f{:}(T_{0,1},\ldots,T_{0,n} \to T) \quad O \vdash e_1{:}T_1 \ldots O \vdash e_n{:}T_n \quad \forall i\ T_i \leq T_{0,i}}{O \vdash f(e_1,\ldots,e_n) : T}\ [\text{Fun}]$$

A (generally) wrong subtyping rule

- Let *array of A* be the type of an array containing data of type *A*
- Intuitively, one could assume:
  - If **B** $\leq$ **A** then **array of B** $\leq$ **array of A**

  (known as "covariant" arrays, present e.g. in Java)
- But, consider the following program (well typed according to this assumption):

  let function f(x:**array of A**) {x[1]←new A}

  in let z:**array of B**

  in {f(z); z[1].b();};
- What is wrong with this program?
  - see next slide…

A (generally) wrong subtyping rule (cont.)

- Problem:
  - When the array of subtypes is used in place of an array of supertypes…
  - …it is possible to **insert** a supertype in the array…
  - …and then the supertype can be used in place of a subtype (type error!)
- But if arrays cannot be written/modified, "covariance" is sound!
  - It is only possible to **use** the content of array cells, that is of subtype in place of supertype

Class subtyping

- Subclasses can usually **override some declarations** of the superclass
  - Usually the body of methods
- Assume it is possible to **override both fields and methods**, by changing also their types
  - This will be possible in our FOOL language

Field overriding

- Let's start by overriding fields
  - Class A{…,T:f,…}
  - Class B inherits A{…,T':f,…}
- Fields can be seen as array cells
  - So if they can be dynamically modified, their type T cannot be changed in the subclass
    - IN GENERAL FIELD SUBTYPING IS NOT SOUND (for this reason in Java we cannot override fields)
  - But if they are **immutable** (the fields cannot be modified in FOOL) subtyping is admitted:
    - if $T' \leq T$ (for all fields) then $B \leq A$

    we call these "**covariant fields**"

Method overriding

- Consider now the possibility to override methods by changing the return and, also, the parameter types (not possible in Java):
  - Class $A\{\ldots, T{:}m(T_1{:}p_1, \ldots, T_n{:}p_n)\ \{e\}, \ldots\}$
  - Class B inherits $A\{\ldots, T'{:}m(T'_1{:}p'_1, \ldots, T'_n{:}p'_n)\ \{e'\}, \ldots\}$
- Consider "let $x{:}T \leftarrow y.m(e_1, \ldots, e_n)$ in e" assuming "y" with static type A and dynamic type B
  - The B return type T' must be usable in place of the A return type T
    - we need $T' \leq T$
  - A parameter types must be usable in place of B parameter types
    - we need $T_i \leq T'_i$

Method overriding (cont.)

- Summarising:
  - if $T_1 \leq T'_1 \ldots T_n \leq T'_n$ and $T' \leq T$ (for all methods) then $B \leq A$

  - The type of m in A is: $T_1,\ldots,T_n \rightarrow T$
  - The type of m in B is: $T'_1,\ldots,T'_n \rightarrow T'$
  - The **general subtyping rule for functions** is:

$$\frac{T_1 \leq T'_1 \ \ldots \ T_n \leq T'_n \text{ and } T' \leq T}{(T'_1,\ldots,T'_n \rightarrow T') \leq (T_1,\ldots,T_n \rightarrow T)}$$

"**covariant**" output (return) type
"**contravariant**" input (parameter) types

Comments

- The typing rules use very concise notation
- They are very carefully constructed
- Virtually any change in a rule either:
  - Makes the type system unsound
    (bad programs are accepted as well typed)
  - Or, makes the type system less usable
    (good programs are rejected)

- But some good programs will be rejected anyway
  - The notion of a good program is undecidable
    (we will study this when we will talk about "computability theory")