Lecture 10: Static Semantics: Scope and Type[1]

[1]From material by G. Necula and P. Hilfinger

Scope Rules: Use Before Definition

- Languages have taken various decisions on where scopes start.
- In Java, C++, scope of a member (field or method) includes the entire class (textual uses may precede declaration).
- But scope of a local variable starts at its declaration.
- As for non-member and class declarations in C++: must write

```
extern int f(int);  // Forward declarations
class C;
int x = f(3)        // Would be illegal w/o forward decls.
void g(C* x) {
   ...
}

int f (int x) { ... }  // Full definitions
class C { ... }
```

Scope Rules: Overloading

- In Java or C++ (not Python or C), can use the same name for more than one method, as long as the number or types of parameters are unique.

```
int g(int a);          float g(float a);
```

- The declaration applies to the *signature*—name + argument types—not just name.
- But return type not part of signature, so this won't work:

```
int g(int a) { ... }      float g(int a) { ... }
```

- In Ada, it will, because the return type *is* part of signature. E.g., after

```
function g(Integer x) return Integer is begin return 2 * x; end;
function g(Integer x) return Float   is begin return Float(3 * x); end;
```

the declarations

```
a: Integer := g(2);        -- Sets a to 4
b: Float := g(2);          -- Sets a to 6.0
```

Dynamic Scoping

- Original Lisp, APL, Snobol use *dynamic scoping,* rather than static:

  Use of a variable refers to most recently executed, and still active, declaration of that variable.

- Makes static determination of declaration generally impossible.
- Example:

```
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); }
```

- With static scoping, illegal.
- With dynamic scoping, prints "10" and "hello"

Explicit vs. Implicit Declaration

- Java, C++ require explicit declarations of things.
- C is lenient: if you write `foo(3)` with no declaration of `foo` in scope, C will supply one.
- Python implicitly declares variables you assign to in a function to be local variables.
- Fortran implicitly declares any variables you use, and gives them a type depending on their first letter.
- But in all these cases, there *is* a declaration as far as the compiler is concerned.

So How Do We Annotate with Declarations?

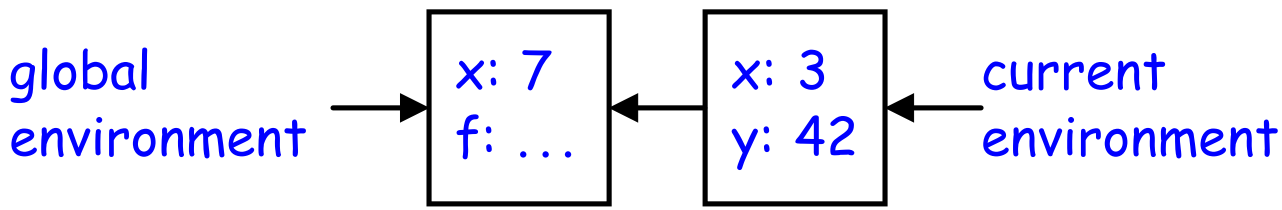
- Idea is to recursively navigate the AST,
  - in effect executing the program in simplified fashion,
  - extracting information that isn't data dependent.
- You saw it in CS61A (sort of).

Environment Diagrams and Symbol Entries

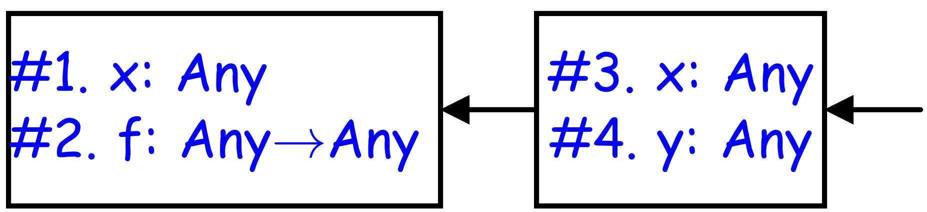
- In Python, executing

```
x = 7
def f(x):
    y = x + 39; return x + y
f(3)
```

would eventually give this environment at `(+ x y)`:



- Now abstract away values in favor of static type info:



- and voila! A data structure for mapping names to current declarations: a *block-structured symbol table*.

Type Checking Phase

- Determines the type of each expression in the program, (each node in the AST that corresponds to an expression)
- Finds type errors.
  - Examples?
- The *type rules* of a language define each expression's type and the types required of all expressions and subexpressions.

Types and Type Systems

- A type is a set of *values* together with a set of *operations* on those values.
- E.g., fields and methods of a Java class are meant to correspond to values and operations.
- A language's *type system* specifies which operations are valid for which types.
- Goal of type checking is to ensure that operations are used with the correct types, enforcing intended interpretation of values.
- Notion of "correctness" often depends on what programmer has in mind, rather than what the representation would allow.
- Most operations are legal only for values of some types
  - 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:

```
movl y, %eax;  addl x, %eax
```

Uses of Types

- Detect errors:
  - Memory errors, such as attempting to use an integer as a pointer.
  - Violations of abstraction boundaries, such as using a private field from outside a class.
- Help compilation:
  - When the Python compiler sees `x+y`, the *static* part of its type systems tells it almost nothing about types of `x` and `y`, so code must be general.
  - But during execution, the *dynamic part* of its type system, implemented by type information in the data structures, tells it what code to execute.
  - In C, C++, Java, code sequences for `x+y` are smaller and faster, because representations are known without runtime checks of type information.

Review: Dynamic vs. Static Types

- A *dynamic type* attaches to an object reference or other value. It's a run-time notion, applicable to any language.
- The *static type* of an expression or variable is a constraint on the possible dynamic types of its value, enforced at compile time.
- Language is *statically typed* if it enforces a "significant" set of static type constraints.
  - A matter of degree: assembly language might enforce constraint that "all registers contain 32-bit words," but since this allows just about any operation, not considered static typing.
  - C sort of has static typing, but rather easy to evade in practice.
  - Java's enforcement is pretty strict.
- In early type systems, $\text{dynamic\_type}(\text{value}(\mathcal{E})) = \text{static\_type}(\mathcal{E})$ for all expressions $\mathcal{E}$, so that in all executions, $\mathcal{E}$ evaluates to exactly type of value deduced by the compiler.
- Gets more complex in advanced type systems with subtyping.

Subtyping

- Define a relation $X \preceq Y$ on classes to say that:

  An object (value) of type X could be used when one of type Y is acceptable

  or equivalently

  X conforms to Y
- In Java this means that X extends Y.
- Properties:
  - $X \preceq X$
  - $X \preceq Y$ if X inherits from Y.
  - $X \preceq Z$ if $X \preceq Y$ and $Y \preceq Z$.

Example

```
class A { ... }
class B extends A { ... }
class Main {
   void f () {
      A x;           // x has static type A.
      x = new A();   // x's value has dynamic type A.
      ...
      x = new B();   // x's value has dynamic type B.
      ...
   }
}
```

Variables with static type A can hold values with dynamic type $\preceq A$, or in general...

Type Soundness

Soundness Theorem on Expressions.

$$\forall E.\ \text{dynamic\_type}(\underbrace{\text{value}(E)}_{\text{A value}}) \preceq \text{static\_type}(\underbrace{\ E\ }_{\text{An expression}})$$

- Compiler uses static_type(E) (call this type C).
- All operations that are valid on C (e.g., attribute (field) accesses, method calls) are also valid on values with types that are $\preceq C$.
- Subclasses only add attributes.
- Methods may be overridden, but only with same (or compatible) signature.

Typing Options

- *Statically typed:* almost all type checking occurs at compilation time (C, Java). Static type system is typically rich.
- *Dynamically typed:* almost all type checking occurs at program execution (Scheme, Python, Javascript, Ruby). Static type system can be trivial.
- *Untyped:* no type checking. What we might think of as type errors show up either as weird results or as various runtime exceptions.

“Type Wars”

- Dynamic typing proponents say:
  - Static type systems are restrictive; can require more work to do reasonable things.
  - Rapid prototyping easier in a dynamic type system.
  - Use *duck typing:* define types of things by what operations they respond to (“if it walks like a duck and quacks like a duck, it’s a duck”).
- Static typing proponents say:
  - Static checking catches many programming errors at compile time.
  - Avoids overhead of runtime type checks.
  - Use various devices to recover the flexibility lost by “going static:” *subtyping, coercions,* and *type parameterization.*
  - Of course, each such wrinkle introduces its own complications.

Example: Sort

Sorting in Python vs. Java:

```
def sort(v, lt = operator.lt):
  for i in range(1, len(v)):
    x = v[i]
    for j in range(i - 1, 0, -1):
      if lt(x, v[j]):
              ...
```

```
public static <T>
  void sort(T[] v,
            Comparator<? super T> comp) {
    for (int i = 1, i < a.length; i += 1) {
      x = v[i];
      for (int j = i - 1; j > 0; j -= 1) {
       if (comp.compare(x, v[j]) < 0) ...
```

- In Python, if v is not something that defines `__len__`, `__getitem__`, etc., or x does not define `__lt__`, we find out only at execution.
- In Java, one finds out earlier, but must write quite a bit more.
- Which makes all assumptions explicit, but isn't immediately clear. Furthermore, requires that v be a primitive array, not `ArrayList`.
- Interestingly, the Java library also contains:

```
public static void sort(Object[] v) {
    ...
    if (((Comparable) x).compareTo(v[j]) < 0) { ...
```

- To give a more Python-like dynamically checked version.

Using Subtypes

- In languages such as Java, can define types (classes) either to
  - Implement a type, or
  - Define the operations on a family of types without (completely) implementing them.
- Hence, relaxes static typing a bit: we may know that something *is a Y* without knowing precisely which subtype it has.

Implicit Coercions

- In Java, can write

```
int x = 'c';
float y = x;
```

- But relationship between **char** and **int**, or **int** and **float** not usually called subtyping, but rather *conversion* (or *coercion*).
- Such implicit coercions avoid cumbersome casting operations.
- Might cause a change of value or representation,
- But usually, such coercions allowed implicitly only if type coerced to contains all the values of the type coerced from (a *widening coercion*).
- Inverses of widening coercions, which typically lose information (e.g., **int**$\longrightarrow$**char**), are known as *narrowing coercions.* and typically required to be explicit.
- **int**$\longrightarrow$**float** a traditional exception (implicit, but can lose information and is neither a strict widening nor a strict narrowing.)

Coercion Examples

```
Object x = ...;   String y = ...;
int a = ...;  short b = 42;
x = y; a = b;     // OK
y = x; b = a;     // ERRORS
x = (Object) y;   // OK
a = (int) b;      // OK
y = (String) x;   // OK but may cause exception
b = (short) a;    // OK but may lose information
```

- Possibility of implicit coercion complicates type-matching rules.
- For example, in C++, if `x` has type `const T*` (pointer to constant T), can write `x = y` whether `y` has type `const T*` or `T*`.
- However, given the two declarations

```
void f(const T* z);
void f(T* z);
```

the call `f(y)` calls the second one if `y` is a `T*`, but would call the first one if the second `f` were not declared.

Type Inference

- Types of expressions and parameters need not be explicit to have static typing. With the right rules, might *infer* their types.
- The appropriate formalism for type checking is logical rules of inference having the form

  If Hypothesis is true, then Conclusion is true

- For type checking, this might become rules like

  If we can infer that E_1 and E_2 have types T_1 and T_2, then we can infer that E_3 has type T_3.

- The standard notation used in scholarly work looks like this:

$$\frac{\vdash\ E_1 : T_1, \qquad \vdash\ E_2 : T_2}{\vdash\ E_3 : T_3}$$

  where $A \vdash B$ means "B may be inferred from A." and $\vdash B$ means simply "B may be inferred."

- Given proper notation, easy to read (with practice), so easy to check that the rules are accurate.
- Can even be mechanically translated into programs.

Soundness

- We'll say that our rules are *sound* if
  - Whenever rules show that e:t, e always evaluates to a value of type t
- We only want sound rules,
- But some sound rules are better than others; here's one that's unnecessarily timid: Let E stand for any expression, then

$$\frac{\vdash E : \mathtt{int}}{\vdash [E] : [\mathtt{int}]}$$

  meaning that if we can show E is of type int, we can conclude that $[E]$ is of type list of int.

- Better simply to say that if T stands for some type, then

$$\frac{\vdash E : T}{\vdash [E] : [T]}$$

Example: A Few Rules for Java

$$\frac{\vdash X : \textsf{boolean}}{\vdash\ !X : \textsf{boolean}} \qquad \frac{\vdash E : \textsf{boolean} \qquad \vdash S : \textsf{void}}{\vdash \textsf{while}(E, S) : \textsf{void}} \qquad \frac{\vdash X : T}{\vdash X : \textsf{void}}$$

- The last rule describes what is known as *voiding:* any expression may appear in a context that requires no value (if syntactically allowed).
- Thus, one can write `someList.add(x)` as a standalone statement, even though `.add` returns a boolean value.
- Some languages (e.g., Fortran and Ada) do not have this rule.

The Type Environment

- What is the type of a variable instance? E.g., how do you show that $\vdash x : int$? for variable x.
- Ans: You can't, in general, without more information.
- We need a hypothesis of the form "we are in the scope of a declaration of x with type T."
- A *type environment* gives types for free names: a mapping from identifiers to types.
- [A variable is *free* in an expression if the expression contains an occurrence of the identifier that refers to a declaration outside the expression.
  - In the expression x, the variable x is free
  - In lambda x: x + y only y is free (Python).
  - In map(lambda x: g(x,y), x), x, y, map, and g are free.]

Notation for Type Environment

- We'll take the notation $O \vdash E : T$ to mean "E may be inferred to have type T in the type environment O."
- Such a type environment maps names to types, e.g., O(x) = int.
- We'll define the notation "$O[T/y]$" to refer to a modified type environment:

$$O[T/y](x) = \begin{cases} T, & \text{if x is the identifier y.} \\ O(x), & \text{otherwise.} \end{cases}$$

Examples:

$$\frac{O \vdash X : \text{boolean}}{O \vdash\ !X : \text{boolean}} \qquad \frac{O \vdash E : \text{boolean} \qquad O \vdash S : \text{void}}{O \vdash \text{while}(E, S) : \text{void}} \qquad \frac{O(x) = T}{O \vdash x : T}$$

$$\frac{O \vdash X : T}{O \vdash X : \text{void}} \qquad \frac{O \vdash E_1 : \text{int} \qquad O \vdash E_2 : \text{int}}{O \vdash E_1 + E2 : \text{int}} \qquad \frac{}{O \vdash I : \text{int}}$$

(where I is an integer literal and O is a type environment)

Example: lambda (Python)

- We may describe the type of a lambda expression with a rule like this:

$$\frac{O[D/X] \vdash E1 : T}{O \vdash \texttt{lambda X: E1} : \texttt{D} \rightarrow \texttt{T}}$$

- The notation $D \rightarrow T$ is standard mathematical notation for the set of functions from D to T.
- The rule above therefore,
  - "If we can infer that $E1$ has type T in a type environment modifying O so that X has type D,
  - Then we can infer that `lambda X: E1` has the function type $D \rightarrow T$ assuming just the assertions in O."

Example: Same Idea for 'let' in the Cool Language

- Cool is an object-oriented language sometimes used for the project in this course.
- The statement let x : T0 in e1 creates a variable x with given type T0 that is then defined throughout e1. Value is that of e1.
- Type rule:

$$\frac{O[T0/X] \ \vdash \ E1 : T1}{\text{let X : T0 in E1} : T1.}$$

"type of let X: T0 in E1 is T1, assuming that the type of E1 would be T1 if free instances of X were defined to have type T0".

Example of a Rule That's Too Conservative

- Let with initialization (also from Cool):

  let x : T0 $\leftarrow$ e0 in e1

- This gives the value of e1 after first evaluating e0 and using it to initialize a new local variable x of type `T0`.
- What's wrong with the following rule?

$$\frac{O \vdash e0 : T0, \quad O[T0/X] \vdash e1 : T1}{O \vdash \textbf{let X : T0} \leftarrow \textbf{e0 in e1} : T1.}$$

(Hint: I said Cool was an object-oriented language).

Loosening the Rule

- Problem is that we haven't allowed the type of the initializer expression to be subtype of T0.
- Here's how to do that:

$$\frac{O \vdash e0 : T2, \;\; T2 \leq T0, \;\; O[T0/X] \vdash e1 : T1}{O \vdash \textbf{let X : T0} \leftarrow \textbf{e0 in e1} : T1.}$$

- Still have to define subtyping (written here as $\leq$), but that depends on other details of the language.

As Usual, Can Always Screw It Up

$$\frac{O \vdash e0 : T2, \quad T2 \leq T0, \quad O \vdash e1 : T1}{O \vdash \textbf{let X : T0} \leftarrow \textbf{e0 in e1} : T1.}$$

This allows incorrect programs and disallows legal ones. Examples?

Function Application

- Consider only the one-argument case (Java):

$$\frac{??}{O \;\vdash\; e1(e2) : T.}$$

Function Application

- Consider only the one-argument case (Java):

$$\frac{O \vdash e1 : T1 \to T, \;\; O \vdash e2 : T2, \;\; T2 \le T1}{O \;\vdash\; e1(e2) : T.}$$

Conditional Expressions

- Consider:

  e1 if e0 else e2

  or (from C) e0 ? e1 : e2.

- The result can be value of either e1 or e2.
- The dynamic type is either e1's or e2's, so static type of result must be an *upper bound* of those types.
- We can constrain the types of e1 and e2 to be equal (as in ML):

$$\frac{??}{O \vdash \text{e1 if e0 else e2} : T}$$

Conditional Expressions

- Consider:

  e1 if e0 else e2

  or (from C) e0 ? e1 : e2.

- The result can be value of either e1 or e2.
- The dynamic type is either e1's or e2's, so static type of result must be an *upper bound* of those types.
- We can constrain the types of e1 and e2 to be equal (as in ML):

$$\frac{O \vdash e0 : \textbf{bool}, \quad O \vdash e1 : T, \quad O \vdash e2 : T}{O \vdash \textbf{e1 if e0 else e2} : T}$$

Conditional Expressions

- Consider:

  e1 if e0 else e2

  or (from C) e0 ? e1 : e2.

- The result can be value of either e1 or e2.
- The dynamic type is either e1's or e2's, so static type of result must be an *upper bound* of those types.
- We can constrain the types of e1 and e2 to be equal (as in ML):

$$\frac{O \vdash e0 : \textbf{bool}, \quad O \vdash e1 : T, \quad O \vdash e2 \ : \ T}{O \vdash \ \textbf{e1 if e0 else e2} : T}$$

- Or use any supertype of T1 and T2:

$$\frac{O \vdash e0 : \textbf{bool}, \quad O \vdash e1 : T1. \quad O \vdash e2 \ : \ T2, \quad T1 \leq T, \quad T2 \leq T}{O \vdash \ \textbf{e1 if e0 else e2} : T}$$

Conditional Expressions: A Question

- However, the last rule,

$$\frac{O \ \vdash \ e0 : \textsf{bool}, \quad O \ \vdash \ e1 : T1. \quad O \ \vdash e2 \ : \ T2, \quad T1 \leq T, \quad T2 \leq T}{O \ \vdash \ \textsf{e1 if e0 else e2} : T}$$

is unspecific as to exactly what type to assign to the construct.

- The compiler would like to pick a specific one, so ChocoPy uses the *least upper bound,* lub(T1, T2):

$$\frac{O \ \vdash \ e0 : \textsf{bool}, \quad O \ \vdash \ e1 : T1, \quad O \ \vdash e2 \ : \ T2,}{O \ \vdash \ \textsf{e1 if e0 else e2} : \textsf{lub}(T1, T2)}$$