Lecture #16: Types[1]

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

"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

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

  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

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

- 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:

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

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

(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:

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

"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 ← e0 in e1

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

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

(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:

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

- 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

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

This allows incorrect programs and disallows legal ones. Examples?

Function Application

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

$$\frac{O \vdash e1 : T1 \rightarrow T, \quad O \vdash e2 : T2, \quad T2 \leq T1}{O \ \vdash \ e1(e2) : T.}$$

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.
- We can constrain the types of e1 and e2 to be equal (as in ML):

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

- Or use the *smallest supertype* at least as large as both of these types—the *least upper bound (lub)* (as in Chocopy):

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