Lecture 17: Types[1]

Administrivia

- Reminder: Test #1 in class on Wednesday, 7 March.

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

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, $\text{dynamic\_type}(\mathcal{E}) = \text{static\_type}(\mathcal{E})$ 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.

$$\forall E.\ \text{dynamic\_type}(E) \preceq \text{static\_type}(E)$$

- Compiler uses $\text{static\_type}(E)$ (call this type C).
- All operations that are valid on C are also valid on values with types $\preceq C$ (e.g., attribute (field) accesses, method calls).
- 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.

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.

Prolog: A Declarative Programming Language

- Prolog is the most well-known *logic programming language*.
- It is a *declarative programming language:* its statements “declare” facts about the desired solution to a problem. The system then figures out the solution from these facts.
- General form:

    Conclusion :- Hypothesis$_1$, $\ldots$, Hypothesis$_k$.

  for $k \geq 0$ means Means “may infer Conclusion by first establishing each Hypothesis.” (when $k = 0$, we generally leave off the ‘:-’).

Prolog: Terms

- Each conclusion and hypothesis is a kind of *term*, represent both programs and data. A term is:
  - A constant, such as a, foo, bar12, =, +, '(', 12, 'Foo'.
  - A variable, denoted by an unquoted symbol that starts with a capital letter or underscore: E, Type, _foo.
  - The nameless variable (_) stands for a different variable each time it occurs.
  - A structure, denoted in prefix form: symbol(term$_1$, ..., term$_k$). Very general: can represent ASTs, expressions, lists, facts.
- Constants and structures can also represent conclusions and hypotheses, just as some list structures in Scheme can represent programs.

Prolog Sugaring

- For convenience, allows structures written in infix notation, such as a + X rather than +(a,X).
- List structures also have special notation:
  - Can write as .(a,.(b,.(c,[]))) or .(a,.(b,.(c,X)))
  - But more commonly use [a, b, c] or [a, b, c | X].

Inference Databases

- Can now express *ground* facts, such as

  likes(brian, potstickers).

- *Universally quantified* facts, such as

  eats(brian, X).

  (for all X, brian eats X).

- Rules of inference, such as

  eats(brian, X) :- isfood(X), likes(brian, X).

  (you may infer that brian eats X if you can establish that X is a food and brian likes it.)

- A collection (database) of these constitutes a Prolog program.

Examples: From English to an Inference Rule

- Consider the type rule:

$$\frac{\vdash\ E_1 : \mathsf{int}, \quad \vdash\ E_2 : \mathsf{int}}{\vdash\ E_1 + E_2 : \mathsf{int}}$$

- Or in English "If e1 has type int and e2 has type int, then e1+e2 has type int"
- The Prolog version is then

```
typeof(E1 + E2, int) :- typeof(E1, int), typeof(E2,int).
```

- "All integer literals have type int:"

```
typeof(X, int) :- integer(X).
```

  (integer is a built-in predicate on terms).
- In general, our typeof predicate will take an AST and a type as arguments.

Soundness

- We'll say that our definition of typeof is *sound* if
  - Whenever rules show that typeof(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 not very useful:

  ```
  typeof(X,any) :- integer(X).
  ```

  Instead, would be better to be more general, as in

  ```
  typeof(X,any).
  ```

  (that is, any expression X is an any.)

Example: A Few Rules for Java (Classic Notation)

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

- 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.

Example: A Few Rules for Java (Prolog)

- typeof(! X, boolean) :- typeof(X, boolean).
- typeof(while(E, S), void) :- typeof(E, boolean), typeof(S, void).
- typeof(X,void) :- typeof(X,Y)

The Type Environment

- What is the type of a variable instance? E.g., how do you show that typeof(x, int)?
- 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.]

Classical Notation for Type Environment

- The notation $\Gamma \ \vdash \ E : T$ means "E may be inferred to have type T in the type environment Γ."
- A type environment (such as Γ) consists of type assertions such as x : int, and "$\Gamma, y : T$" means the type environment Γ augmented by the assertion that y is a T.

Examples:

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

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

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

Defining the Environment in Prolog

- Can define a predicate, say, defn(I,T,E), to mean "I is defined to have type T in environment E."
- We can implement such a defn in Prolog like this:

```
defn(I, T, [def(I,T) | _]).
defn(I, T, [def(I1,_)|R]) :- dif(I,I1), defn(I,T,R).
```

  (dif is built-in, and means that its arguments differ).

- Now we revise typeof to have a 3-argument predicate: typeof(E, T, Env) means "E is of type T in environment Env," allowing us to say

```
typeof(I, T, Env) :- defn(I, T, Env).
```

Examples Revisited (Prolog)

```
typeof(E1 + E2, int, Env)
              :- typeof(E1, int, Env), typeof(E2,int, Env).
typeof(X, int, _) :- integer(X).
typeof(!X, boolean, Env) :- typeof(X, boolean, Env).
typeof(while(E,S), void, Env) :-
         typeof(E, boolean,Env), typeof(S, void, Env).
```

Example: lambda (Python)

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

$$\frac{\Gamma,\ X : D \ \vdash\ E1 : T}{\Gamma\ \vdash\ \texttt{lambda X: E1} : \texttt{D} \rightarrow \texttt{T}}$$

- Which means,
  - "If we can infer that $E1$ has type T in a type environment containing the assertions in Γ plus the assertion 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 Γ."
- Or in Prolog:

```
typeof(lambda(X,E1), D->T, Env) :-
            typeof(E1,T, [def(X,D) | Env]).
```

In effect, [def(X,any) | Env] means "Env modified to map x to any and behaving like Env on all other arguments."

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.
- Prolog rule (assuming that "let(X,T0,E1)" is the AST for **let**):

```
typeof(let(X,T0,E1),  T1, Env) :-
             typeof(E1, T1, [def(X, T0)|Env]).
```

"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

- What's wrong with this rule?

```
typeof(let(X, T0, E0, E1), T1, Env) :-
          typeof(E0, T0, Env),
          typeof(E1, T1, [def(X, T0) | Env]).
```

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

Loosening the Rule

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

```
typeof(let(X, T0, E0, E1), T1, Env) :-
          typeof(E0, T2, Env), T2 <= T0,
          typeof(E1, T1, [def(X, T0) | Env]).
```

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

As Usual, Can Always Screw It Up

```
typeof(let(X, T0, E0, E1), T1, Env) :-
        typeof(E0, T2, Env), T2 <= T0,
        typeof(E1, T1, Env).
```

This allows incorrect programs and disallows legal ones. Examples?

Function Application

- Consider only the one-argument case (Java).
- AST uses 'call', with function and list of argument types.

```
typeof(call(E1,[E2]),  T, Env) :-
     typeof(E1, T1->T, Env), typeof(E2, T1a, Env),
     T1a <= T1.
```

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.
- Either constrain these to be equal (as in ML):

```
typeof(if(E0,E1,E2), T, Env) :-
     typeof(E0,bool,Env), typeof(E1,T,Env), typeof(E2,T,Env).
```

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

```
typeof(if(E0,E1,E2), T, Env) :-
     typeof(E0,bool,Env), typeof(E1,T1,Env), typeof(E2,T2,Env),
     lub(T,T1,T2).
```