Lecture 17: Types

Administrivia

- Reminder: Test #1 in class on Wednesday, 11 March.

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 Python sees $x+y$, its type systems tells it almost nothing about types of $x$ and $y$, so code must be general.
  - In C, C++, Java, code sequences for $x+y$ are smaller and faster, because representations are known.
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, \( \text{dynamic	extunderscore type}(E) = \text{static	extunderscore type}(E) \) for all expressions \( E \), so that in all executions, \( E \) evaluates to exactly type of value deduced by the compiler.
- Gets more complex in advanced type systems.

Subtyping

- Define a relation \( X \leq Y \) 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:
  - \( X \leq X \)
  - \( X \leq Y \) if \( X \) inherits from \( Y \).
  - \( X \leq Z \) if \( X \leq Y \) and \( Y \leq Z \).

Type Soundness

Soundness Theorem on Expressions.

\[ \forall E. \text{dynamic	extunderscore type}(E) \leq \text{static	extunderscore type}(E) \]

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

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 \( \leq A \), or in general...
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
  ```java
  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*→*char*), are known as *narrowing coercions* and typically required to be explicit.
- *int*→*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 (see C++).

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 $E_1$ and $E_2$ have types $T_1$ and $T_2$, then $E_3$ has type $T_3$.
- The standard notation used in scholarly work looks like this:
  \[ \Gamma \vdash E_1 : T_1, \Gamma \vdash E_2 : T_2 \]
  \[ \Gamma \vdash E_3 : T_3 \]

Here, $\Gamma$ stands for some set of assumptions about the types of free names, generically known as a type environment and $A \vdash B$ means "from $A$ we may infer that $B$" or "$A$ entails $B$.”
- 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.
- Its statements “declare” facts about the desired solution to a problem. The system then figures out the solution from these facts.
- You saw this in CS61A.
- General form:
  \[ \text{Conclusion} :- \text{Hypothesis}_1, \ldots, \text{Hypothesis}_k. \]
  for $k \geq 0$ means “we 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 $a$, $\text{foo}$, $\text{bar12}$, $\varepsilon$, '+', '(', 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 $\text{likes(brian, potstickers)}$.
- *Universally quantified* facts, such as $\text{eats(brian, X)}$.
  (for all $X$, brian eats $X$).
- Rules of inference, such as $\text{eats(brian, X) :- isfood(X), likes(brian, X)}$.
  (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

- "If $e_1$ has type int and $e_2$ has type int, then $e_1 + e_2$ has type int."
  $\text{typeof(E1 + E2, int) :- typeof(E1, int), typeof(E2, int)}$.
- "All integer literals have type int."
  $\text{typeof(X, int) :- integer(X)}$.
  (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 $\text{typeof(e, t)}$, 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:
  $\text{typeof(X,any) :- integer(X)}$.
  Instead, would be better to be more general, as in
  $\text{typeof(X,any)}$.
  (that is, any expression $X$ is an any.)
Example: A Few Rules for Java (Classic Notation)

⊢ X : boolean
⊢ !X : boolean

⊢ E : boolean ⊢ S : void
⊢ while(E,S) : void

⊢ X : T
⊢ E₁ : int ⊢ E₂ : int
⊢ E₁ + E₂ : int

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 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.

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:
  ```prolog
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 (Classic)

\[
\begin{align*}
\Gamma \vdash X : \text{boolean} & \quad \Gamma \vdash \Gamma \vdash X : \text{boolean} & \quad \Gamma \vdash \Gamma \vdash S : \text{void} & \quad \Gamma \vdash \Gamma \vdash \text{while}(E, S) : \text{void} \\
\Gamma \vdash E : \text{boolean} & \quad \Gamma \vdash E_1 : \text{int} & \quad \Gamma \vdash E_2 : \text{int} & \quad \Gamma \vdash E_1 + E_2 : \text{int} \\
\Gamma \vdash E_1 : \text{int} & \quad \Gamma \vdash E_2 : \text{int} & \quad \Gamma \vdash \Gamma \vdash I : \text{int} & \quad \Gamma \vdash \Gamma \vdash S : \text{void} \\
\end{align*}
\]

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

Examples Revisited (Prolog)

\[
\begin{align*}
\text{typeof}(E_1 + E_2, \text{int}, \text{Env}) & \quad :- \text{typeof}(E_1, \text{int}, \text{Env}), \text{typeof}(E_2, \text{int}, \text{Env}). \\
\text{typeof}(X, \text{int}, \_\_) & \quad :- \text{integer}(X) . \\
\text{typeof}(!X, \text{boolean}, \text{Env}) & \quad :- \text{typeof}(X, \text{boolean}, \text{Env}). \\
\text{typeof}(\text{while}(E, S), \text{void}, \text{Env}) & \quad :- \text{typeof}(E, \text{boolean}, \text{Env}), \text{typeof}(S, \text{void}, \text{Env}).
\end{align*}
\]

Example: lambda (Python)

\[
\text{typeof}(\lambda(X,E_1), \text{any}\rightarrow\text{T}, \text{Env}) :- \\
\text{typeof}(E_1, \text{T}, [\text{def}(X, \text{any}) \mid \text{Env}]).
\]

In effect, \([\text{def}(X, \text{any}) \mid \text{Env}]\) means "\(\text{Env}\) modified to map \(X\) to \text{any} and behaving like \(\text{Env}\) 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 \(\text{let } x : T_0 \text{ in } e_1\) creates a variable \(x\) with given type \(T_0\) that is then defined throughout \(e_1\). Value is that of \(e_1\).
- Rule (assuming that \("\text{let}(X,T_0,E_1)\"\) is the AST for \(\text{let}\)):
  \[
  \text{typeof}(\text{let}(X,T_0,E_1), T_1, \text{Env}) :- \\
  \text{typeof}(E_1, T_1, [\text{def}(X, T_0) \mid \text{Env}]).
  \]

"type of \(\text{let } X : T_0 \text{ in } E_1\) is \(T_1\), assuming that the type of \(E_1\) would be \(T_1\) if free instances of \(X\) were defined to have type \(T_0\)."
Example of a Rule That's Too Conservative

- Let with initialization (also from Cool):
  
  \[
  \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1
  \]

- What's wrong with this rule?

  typeof(let(X, T_0, E_0, E_1), T_1, Env) :-
  typeof(E_0, T_0, Env),
  typeof(E_1, T_1, [def(X, T_0) | 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 T_0.

- Here's how to do that:

  typeof(let(X, T_0, E_0, E_1), T_1, Env) :-
  typeof(E_0, T_2, Env), T_2 <= T_0,
  typeof(E_1, T_1, [def(X, T_0) | 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, T_0, E_0, E_1), T_1, Env) :-
  typeof(E_0, T_2, Env), T_2 <= T_0,
  typeof(E_1, T_1, 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(E_1, [E_2]), T, Env) :-
  typeof(E_1, T_1\rightarrow T, Env), typeof(E_2, T_1a, Env),
  T_1a <= T_1.
Conditional Expressions

- Consider:
  
  \[ e_1 \text{ if } e_0 \text{ else } e_2 \]

  or (from C)
  
  \[ e_0 \ ? \ e_1 : e_2 \].

- The result can be value of either \( e_1 \) or \( e_2 \).
- The dynamic type is either \( e_1 \)'s or \( e_2 \)'s.
- Either constrain these to be equal (as in ML):
  
  \[
  \text{typeof(if}(E_0,E_1,E_2), T, \text{Env}) :- \\
  \text{typeof}(E_0,\text{bool},\text{Env}), \text{typeof}(E_1,T,\text{Env}), \text{typeof}(E_2,T,\text{Env}).
  \]

- Or use the smallest supertype at least as large as both of these types— the least upper bound (lub) (as in Cool):
  
  \[
  \text{typeof(if}(E_0,E_1,E_2), T, \text{Env}) :- \\
  \text{typeof}(E_0,\text{bool},\text{Env}), \text{typeof}(E_1,T_1,\text{Env}), \text{typeof}(E_2,T_2,\text{Env}), \\
  \text{lub}(T,T_1,T_2).
  \]