Type Checking

Lecture 17

(P. N. Hilfinger and G. Necula)

Administrivia

- Test is next Monday during class
- Please let me know soon if you need an alternative time for the test.
- Please use bug-submit to submit problems/questions
- Review session Saturday 306 Soda 3-5PM

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

```
addi  $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 !
}
```

Dynamic And Static Types (Review)

- A *dynamic type* attaches to an object reference or other value
  - A run-time notion
  - Applicable to any language
- The *static type* of an expression or variable is a notion that captures all possible dynamic types the value of the expression could take or the variable could contain
  - A compile-time notion

Dynamic and Static Types. (Cont.)

- In early type systems the set of static types correspond directly with the dynamic types:
  - 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

Subtyping

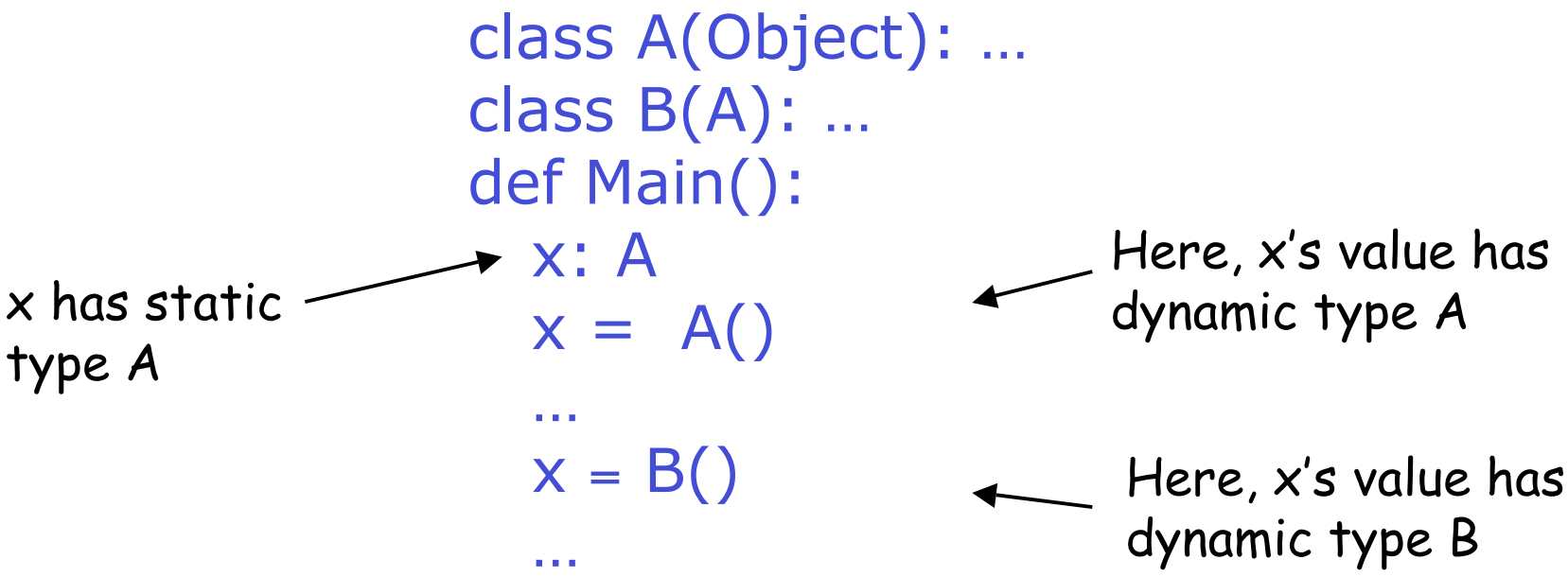
- Define a relation $X \leq Y$ on classes to say that:
  - An object (value) of type X could be used when one of type Y is acceptable, or equivalently
  - X conforms to Y
  - In Java this means that X extends Y
- Define a relation $\leq$ on classes

  $X \leq X$

  $X \leq Y$ if X inherits from Y

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

Dynamic and Static Types



- A variable of static type *A* can hold values of static type B, if B ≤ *A*

Dynamic and Static Types

Soundness theorem:

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

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 $C' \leq C$
  - Such as fetching the value of an attribute
  - Or invoking a method on the object
- Subclasses can *only add* attributes or methods
- Methods can be redefined but with same type !

Type Checking Overview

- Three kinds of languages:
  - *Statically typed*: All or almost all checking of types is done as part of compilation (C, Java, Cool). Static type system is rich.
  - *Dynamically typed*: Almost all checking of types is done as part of program execution (Scheme, Python). Static type system is trivial.
  - *Untyped*: No type checking (machine code). Static and dynamic type systems trivial.

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
  - Rapid prototyping easier in a dynamic type system

The Type Wars (Cont.)

- In practice, most code is written in statically typed languages with an "escape" mechanism
  - Unsafe casts in C, native methods in Java, unsafe modules in Modula-3
- Within the strongly typed world, are various devices, including subtyping, coercions, and type parameterization.
- Of course, each such wrinkle introduces its own complications.

The Use of Subtyping

- The idea then is that Y describes operations applicable to all its subtypes.
- Hence, relaxes static typing a bit: we may know that something "is a" Y without knowing precisely *which* subtype it has.

Conversion

- 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*).
- In general, might be a change of value or representation. Indeed **int→float** can lose information—a *narrowing conversion.*

Conversions: Implicit vs. Explicit

- Conversions, when automatic (implicit), another way to ease the pain of static typing.
- Typical rule (from Java):
  - Widening conversions are implicit; narrowing conversions require explicit cast.
- *Widening conversions* convert "smaller" types to "larger" ones (those whose values are a superset).
- *Narrowing conversions* go in opposite direction (and thus may lose information).

Examples

- Thus,

```
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

- *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* when inference is necessary

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 having the form
  - *If Hypothesis is true, then Conclusion is true*
- For type checking, this becomes:
  - *If E_1 and E_2 have certain types, then E_3 has a certain type*

Why Rules of Inference?

- Rules of inference are a compact notation for "If-Then" statements
- Given proper notation, easy to read (with practice), so easy to check that the rules are accurate.
- Can even be mechanically translated into programs.
- We won't do that in project 2 (unless you want to), but will illustrate today.

A Declarative Programming Language

- One possible notation uses Horn clauses, which are restricted logic rules for which there is an effective implementation: *logic programming*
- The Prolog language is one example (you may have learned another in CS61A).
- A form of *declarative programming:* we describe *effect* we want and system decides how to achieve it.

Horn Clauses in Prolog

- General form:

  Conclusion :- Hypothesis_1, …, Hypothesis_k.

  for $k \geq 0$.
- Means "may infer Conclusion by first establishing each Hypothesis."
- Each conclusion and hypothesis is a term.

Prolog Terms

- A term may either be
  - 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($term_1$, …, $term_k$)
- Structures are very general, can use them for ASTs, expressions, lists, etc.

Prolog Terms: Sugaring

- For convenience, allow 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

- "If e_1 has type int and e_2 has type int, then e_1+e_2 has type int"

```
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).

Soundness

- We'll say that our definition of typeof is *sound* if
  - Whenever rules show that typeof(e,t)
  - Then e 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).

Two More Rules (Not Pyth)

- typeof(not(X), 'Bool') :- typeof(X, 'Bool').
- typeof(while(E,S), 'Any') :-
  typeof(E,'Bool'), typeof(S, 'Any').

A Problem

- What is the type of a variable reference?
  - typeof(X,?) :- atom(X).

    (atom = "is identifier" builtin)
- This rules does not have enough information to give a type.
  - We need a hypothesis of the form "*we are in the scope of a declaration of x with type T*")

A Solution: Put more information in the rules

- *A type environment* gives types for *free* variables
  - A *type environment* is a mapping from Identifiers to Types
  - A variable is *free* in an expression if:
    - The expression contains an occurrence of the variable that refers to a declaration outside the expression
  - E.g. in the expression "x", the variable "x" is free
  - E.g. in "lambda x: x + y" only "y" is free (Python).
  - E.g. in "map(lambda x: g(x,y), x)" both "x" and "y" are free (along with map and g)

Type Environments

- 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)
- Now we make typeof 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).

Modified Rules

Redoing our examples so far:

```
typeof(E1 + E2, int, Env)
            :- typeof(E1, int, Env), typeof(E2,int, Env).
typeof(X, int, _) :- integer(X).
typeof(not(X), 'Bool', Env) :- typeof(X, 'Bool', Env).
typeof(while(E,S), 'Any', Env) :-
    typeof(E,'Bool',Env), typeof(S, 'Any', Env).
```

Example: Lambda (from Python)

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

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

Same Idea: Let in the Cool Language

- The statement let x : T_0 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:

  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 : $T_0 \leftarrow e_0$ in e_1
- 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]).

Loosening Things Up a Bit

- Too strict about E0:
  - typeof(let(X, T0, E0, E1), T1, Env) :-
    typeof(E0, T2, Env), T2 ≤ T0,
    typeof(E1, T1, [def(X, T0) | Env]).
- Have to define subtyping (≤), but that depends on other details of the language.

And, of course, Can Always Screw It Up

- typeof(let(X, T0, E0, E1), T1, Env) :-
  typeof(E0, T2, Env), T2 ≤ T0,
  typeof(E1, T1, Env).

- What's wrong with this?
- (It allows incorrect programs and disallows legal ones. Examples?)

Possible Rules for Assignments

- Java:
  - typeof (assign(X, E), T, Env) :-
    typeof(E, T), typeof(X, TX), T $\leq$ TX.
- Pyth:
  - typeof (assign(X, E), T, Env) :-
    typeof(E, T), typeof(X, TX), feasible(T, TX).

  feasible(T0, T1) :- T0 $\leq$ T1.

  feasible(T0, T1) :- T1 $\leq$ T0.
- Why does this imply run-time checks, while Java's rule does not?

Function Application in Pyth

- Let's just look at one-argument case (Pyth):
  - typeof(call(E1,[E2]), T, Env) :-
    typeof(E1, T1->T, Env), typeof(E2, T1a),
    feasible(T1a,T1).

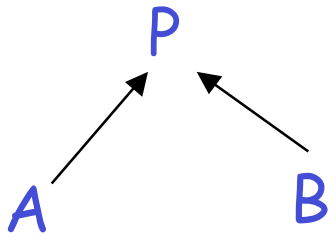
Conditional Expression

- Consider:

  if e_0 then e_1 else e_2 fi or e_0 ? e_1 : e_2 in C
- The result can be either e_1 or e_2
- The dynamic type is either e_1's or e_2's type
- The best we can do statically is the smallest supertype larger than the type of e_1 and e_2

If-Then-Else example

- Consider the class hierarchy



- … and the expression

  if … then new A else new B fi

- Its type should allow for the dynamic type to be both A or B
  - Smallest supertype is P

Least Upper Bounds

- lub(X,Y), the *least upper bound* of X and Y, is Z if
  - $X \leq Z \wedge Y \leq Z$

    Z is an upper bound

  - $X \leq Z' \wedge Y \leq Z' \Rightarrow Z \leq Z'$

    Z is least among upper bounds

- Typically, the least upper bound of two types is their least common ancestor in the inheritance tree

Possible Rules for Conditional Expressions

- Cool Language:
  - typeof(ifelse(C, E1, E2), T, Env) :-
    typeof(C, Bool),
    typeof(E1, T1), typeof(E2,T2),
    lubof(T, T1, T2).
- ML Language
  - typeof(ifelse(C, E1, E2), T, Env) :-
    typeof(C, Bool), typeof(E1, T), typeof(E2,T).