Type checking

Slides adapted from CS 412 (Cornell) and CS 164 (Berkeley)

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 {
      fs : FileSystem
      ...  
    }

    f(fs : FileSystem) {
      File fdesc <- fs.open("foo")
      ...
    } -- f cannot see inside fdesc!

Dynamic And Static Types

• 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,
    \[ \text{dynamic
type}(E) = \text{static
type}(E) \]
  (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 $B$ at runtime, if $B \leq A$

---

**Soundness theorem:**

$$\forall E. \ dynamic\_type(E) \leq static\_type(E)$$

**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**

- **Statically typed:** All or almost all checking of types is done as part of compilation (C#, Java). 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.

Conversion

- 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`).
- 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,
  
  ```java
  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
  - (eg) if \( E_1 \) and \( E_2 \) have type int, then \( E_1 + E_2 \) 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.
Type Judgments

• The type judgment:
  \[ \vdash E : T \]
  is read:
  "\(E\) is a well-typed construct of type \(T\)"

• Type checking program \(P\) is demonstrating the validity of
  the type judgment \(\vdash P : T\) for some type \(T\)

• Sample valid type judgments for program fragments:
  \[ \vdash 2 : \text{int} \]
  \[ \vdash 2 \times (3 + 4) : \text{int} \]
  \[ \vdash \text{true} : \text{bool} \]
  \[ \vdash (\text{true} ? 2 : 3) : \text{int} \]

Hypothetical Type Judgments

• The hypothetical type judgment
  \[ A \vdash E : T \]
  is read:
  "In type context \(A\) expression \(E\) is well-typed with type \(T\)"

• A type context is a mapping of identifiers to types (i.e., a symbol table). It's a set of assumptions about the types of identifiers.

• Sample valid hypothetical type judgments:
  \[ b : \text{bool} \vdash b : \text{bool} \]
  \[ 2 + 2 : \text{int} \]
  \[ b : \text{bool}, x : \text{int} \vdash (b ? 2 : x) : \text{int} \]
  \[ b : \text{bool}, x : \text{int} \vdash b : \text{bool} \]
  \[ b : \text{bool}, x : \text{int} \vdash 2 + 2 : \text{int} \]

• Type checking program \(P\) is demonstrating the validity of \(A \vdash P : T\) for some type \(T\) and the language's standard environment \(A\)

Deriving a Type Judgment

• Consider the judgment:
  \[ \vdash (b ? 2 : 3) : \text{int} \]
  What do we need in order to decide that this is a valid type judgment?
  • \(b\) must be a bool \(\vdash b : \text{bool} \)
  • \(2\) must be an int \(\vdash 2 : \text{int} \)
  • \(3\) must be an int \(\vdash 3 : \text{int} \)

• To show:
  \(b : \text{bool}, x : \text{int} \vdash (b ? 2 : x) : \text{int} \)

• Need to show:
  \(b : \text{bool}, x : \text{int} \vdash b : \text{bool} \)
  \(b : \text{bool}, x : \text{int} \vdash 2 : \text{int} \)
  \(b : \text{bool}, x : \text{int} \vdash x : \text{int} \)
**General Rule**

- For any type environment $A$, expressions $E$, $E_1$, and $E_2$, the judgment

  $A |- (E \ ? \ E_1 \ : \ E_2) : T$

  is valid if:

  $A |- E : \text{bool}$
  $A |- E_1 : T$
  $A |- E_2 : T$

**Inference Rule Schema**

Premises (a.k.a., antecedant)

$$
A |- E : \text{bool} \quad A |- E_1 : T \quad A |- E_2 : T
$$

Conclusion (a.k.a., consequent)

$$
A |- (E \ ? \ E_1 \ : \ E_2) : T
$$

- Premises: $A |- E : \text{bool}$, $A |- E_1 : T$, $A |- E_2 : T$
- Conclusion: $A |- (E \ ? \ E_1 \ : \ E_2) : T$

**Axioms**

- An axiom is an inference rule (schema) with no premises

  $A |- \text{true} : \text{bool}$

**Why Inference Rules?**

- Inference rules: compact, precise language for specifying static semantics (can specify languages in ~20 pages vs. 100's of pages of Java Language Specification)
- Inference rules are to type inference systems as productions are to context-free grammars
- Type judgments are to type inference systems as nonterminals are to context-free grammars
- Type checking is an attempt to prove that a type judgment is $A |- E : T$ is valid
Meaning of Inference Rule

- Inference rule says:
  - given that the antecedent judgments are derivable
  - with a uniform substitution for meta-variables (i.e., $A, E_1, E_2$)
  - then the consequent judgment is derivable
  - with the same uniform substitution for the meta-variables

Proof Tree

- A construct is well-typed if there exists a type derivation for a type judgment for the construct
- Type derivation is a proof tree where all the leaves are axioms
- Example: if $A_1 = b : \text{bool}, x : \text{int}$, then:

Proof Tree, cont.

- Axioms are analogous to production with epsilon on the right hand side
- A complete proof of $A |- E : T$ is like a derivation of epsilon from $A |- E : T$

Type Judgments for Statements

- Statements that have no value are said to have type void, i.e., judgment $|- S : \text{void}$
  - void means “$S$ is a well-typed statement with no result type”
- ML uses unit instead of void
While Statements

- Rule for while statements:

\[
A \vdash E : \text{bool} \\
A \vdash S : T \\
A \vdash \text{while}(E) \ S : \text{void}
\]

Assignment (Expression) Statements

\[
A, \text{id} : T \vdash E : T \\
A, \text{id} : T \vdash \text{id} = E : T
\]

\[
A \vdash E_3 : T \\
A \vdash E_2 : \text{int} \\
A \vdash E_1 : \text{array}[T] \\
A \vdash E_1[E_2] = E_3 : T
\]

Statement Sequences

- Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed too:

\[
A \vdash S_1 : T_1 \\
A \vdash (S_2, \ldots, S_n) : T_n \\
A \vdash (S_1 ; S_2 ; \ldots ; S_n) : T_n
\]

Identifier Declaration List

- What about variable declarations (with initialization)?
- Declarations add entries to the type environment in which the scope of the declared variable must type check

\[
A \vdash E : T \\
A, \text{id} : T \vdash (S_2, \ldots, S_n) : T' \\
A \vdash \text{id} = E ; S_2 ; \ldots ; S_n : T' \\
A \vdash (\text{id} = E) ; S_2 ; \ldots ; S_n : T'
\]
Function Calls

- If expression E is a function value, it has a type \( T_1 \times T_2 \times \ldots \times T_n \to T_r \).
- \( T_i \) are argument types; \( T_r \) is return type.
- How to type-check function call \( E(E_1, \ldots, E_n) \)?

\[
\begin{align*}
A \vdash E & : T_1 \times T_2 \times \ldots \times T_n \to T_r \\
A \vdash E_i & : T_i \quad (i \in 1..n) \\
A \vdash E(E_1, \ldots, E_n) & : T_r
\end{align*}
\]

Function Declarations

- Consider a function declaration of the form \( T_r \ f \ (T_1 \ a_1, \ldots, T_n \ a_n) \ { \{ \ E; \} } \).
- The body of the function must type check in an environment containing the type bindings for the formal parameters:

\[
\begin{align*}
A, a_1 : T_1, \ldots, a_n : T_n & \vdash T_r \ \ f (T_1 \ a_1, \ldots, T_n \ a_n) \ { \{ \ E; \} } : \text{void}
\end{align*}
\]

But what about recursion?

- Example:

```c
int fact(int x) {
    if (x==0) return 1;
    else return x * fact(x - 1);
}
```

- Need to prove: \( A \vdash x \cdot \text{fact}(x-1) : \text{int} \)

where: \( A = \{ \text{fact: int} \to \text{int}, x : \text{int} \} \)

And mutual recursion?

- Example:

```c
int f(int x) {
    return g(x) + 1;
}
int g(int x) {
    return f(x) - 1;
}
```

- Need environment containing at least:

\[ f : \text{int} \to \text{int}, g : \text{int} \to \text{int} \]

when checking both \( f \) and \( g \)

- Two-pass approach needed:
  - First pass: collect all function signatures into a type environment \( A \)
  - Second pass: type-check each function declaration using this global environment \( A \)
  - How do we express this in our type inference notation?
Solution

• Intuition:
  - Make one pass over program to add top level function signatures to symbol table
  - Use these signatures in a second pass to type-check program
  - Slight complication for object-oriented programs with methods inside classes:
    • functions are named using pair (Class, method name)

• Formalization:
  - Split the type environment into two parts, one for functions and one for variables
  - Type environment for functions does not change during the second pass
  - We will not show this to keep the notation simple.

How to Check Return?

\[
\frac{A \vdash E : T}{A \vdash \text{return } E : \text{void}} \quad \text{(return1)}
\]

• A return statement produces no value for its containing context to use
• Does not return control to containing context
• Suppose we use type void...
• ...then how to make sure T is the return type of the current function?

Put return type in environment

• Add a special entry \{ \text{return\_fun} : T \} when we start checking the function “f”, look up this entry when we hit a return statement.
• To check \( T, f (T_1 a_1, \ldots, T_n a_n) \{ \text{return } S; \} \) in environment \( A \), need to check:

\[
\begin{align*}
A, a_1 : T_1, \ldots, a_n : T_n, \text{return\_f} : T_r & \vdash E : \text{void} \\
A \vdash T_r f (T_1 a_1, \ldots, T_n a_n) \{ E; \} : \text{void} & \quad \text{(function-body)} \\
A, \text{return\_f} : T & \vdash E : T \\
A, \text{return\_f} : T & \vdash \text{return } E : \text{void} & \quad \text{(return)}
\end{align*}
\]

Example

\[
\begin{align*}
\{f : \text{int} \to \text{x : int, return\_f : \text{int}}\} & \vdash \text{x : int} & \quad \text{(return)} \\
\{f : \text{int} \to \text{x : int, return\_f : \text{int}}\} & \vdash \text{return } x : \text{void} & \quad \text{(function definition)} \\
\{f : \text{int} \to \text{int}\} & \vdash \text{int } (x:\text{int}) \{ \text{return } x; \} : \text{void}
\end{align*}
\]
**Arrays**

- Arrays:
  - array types are of form int[], float[] etc.

\[
\frac{A \vdash E_0 : T[1]}{A \vdash E_0[0] : T} \quad A \vdash E_1 : T[n]
\]

\[
\frac{A \vdash E : T[1]}{A \vdash E.length : int} \quad A \vdash E : T \quad A \vdash new T[E] : T[n]
\]

**Classes**

- Class would be represented in the type environment by a list of (name:type) pairs which has one entry for each field and method

```
class C1 {
    int x, y;
    int get_x() {return x;}
}
```

C1: {x:int, y:int, get_x:int -> int}

---

**Inference rules**

- Constructors:

  \[
  T \in C \quad \frac{}{A \vdash E : T() : T}
  \]

- Field accesses:

  \[
  \frac{A \vdash E : T \quad T \in C \quad (\forall i. T_i) \in T}{A \vdash E.x[i] : T}
  \]

- Method invocations

  \[
  \frac{A \vdash E_0 : T_1 \times \ldots \times T_n \rightarrow T \quad A \vdash E_i : T_i \quad 1 \leq i \leq n}{A \vdash E_0(E_1, \ldots, E_n) : T}
  \]

**Static Semantics Summary**

- Type inference system = formal specification of typing rules

- Concise form of static semantics: typing rules expressed as inference rules

- Expression and statements are well-formed (or well-typed) if a typing derivation (proof tree) can be constructed using the inference rules