CS 412/413
Introduction to Compilers
Tim Teitelbaum

Lecture 13: Types and Type-Checking
19 Feb 07

Semantic Analysis

• Last time:
  - Semantic errors related to scopes
  - Symbol tables
  - Name resolution

• This lecture:
  - Semantic errors related to types
  - Type system concepts
  - Types and type-checking

What Are Types?

• Types describe the values computed during the execution of the program

• Essentially, types are predicate on values
  - E.g., “int x” in Java means “x ∈ [-231, 231)”
  - Think: “type = set of possible values”

• Type errors: improper, type-inconsistent operations during program execution

• Type-safety: absence of type errors at run time

How to Ensure Type-Safety

• Bind (assign) types, then check types

  - Type binding: defines type for constructs in the program (e.g., variables, functions)
    • Can be either explicit (int x) or implicit (x = 1)
    • Type consistency (safety) = correctness with respect to the type bindings

  - Type checking: determine if the program correctly uses the type bindings
    • Enforce a set of type-checking rules

Type Checking

• Type checking: static semantic checks to enforce the type safety of the program

• Examples:
  - Unary and binary operators (e.g., +, ==, [ ]) must receive operands of the proper type
  - Functions must be invoked with the right number and type of arguments
  - Return statements must agree with the return type
  - In assignments, assigned value must be compatible with type of variable on LHS.
  - Class members accessed appropriately

Static vs. Dynamic Typing

• Static and dynamic typing refer to type definitions (i.e., bindings of types to variables, expressions, etc.)

  - Statically typed language: types are defined and checked at compile-time, and do not change during the execution of the program
    • E.g., C, Java, Pascal

  - Dynamically typed language: types defined and checked at run-time, during program execution
    • E.g., Lisp, Scheme, Smalltalk
Strong vs. Weak Typing

- Strong and weak typing refer to how much type consistency is enforced
  - Strongly typed languages: guarantees that accepted programs are type-safe
  - Weakly typed languages: allow programs that contain type errors
- Can achieve strong typing using either static or dynamic typing

Soundness

- Sound type systems: can statically ensure that the program is type-safe
- Soundness implies strong typing
- Static type safety requires a conservative approximation of the values that may occur during all possible executions
  - May reject type-safe programs
  - Need to be expressive: reject as few type-safe programs as possible

Concept Summary

- Static vs dynamic typing: when to define/check types?
- Strong vs weak typing: how many type errors?
- Sound type systems: statically catch all type errors

Classification

<table>
<thead>
<tr>
<th>Strong Typing</th>
<th>Weak Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>C</td>
</tr>
<tr>
<td>Pascal</td>
<td>C++</td>
</tr>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>Modula-3</td>
<td></td>
</tr>
<tr>
<td>Static Typing</td>
<td>Dynamic Typing</td>
</tr>
<tr>
<td>Scheme</td>
<td>PostScript</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>assembly code</td>
</tr>
</tbody>
</table>

Why Static Checking?

- Efficient code
  - Dynamic checks slow down the program
- Guarantees that all executions will be safe
  - Dynamic checking gives safety guarantees only for some execution of the program
- But is conservative for sound systems
  - Needs to be expressive: reject few type-safe programs

Type Systems

- Type is predicate on value
- Type expressions: describe the possible types in the program: int, string, array[], Object, etc.
- Type system: defines types for language constructs (e.g., expressions, statements)
### Type Expressions

- Languages have **basic types** (a.k.a. primitive types or ground types)
  - E.g., int, char, boolean
- Build **type expressions** using basic types:
  - Type constructors
  - Type aliases

### Array Types

- Various kinds of array types in different programming languages
  - `array(T)` : array with elements of type T and no bounds
    - C, Java: `int []`, Modula-3: `array of integer`
  - `array(T, S)` : array with size
    - May be indexed `0..size-1`
  - `array(T,L,U)` : array with upper/lower bounds
    - Pascal or Ada: `array[2 .. 5] of integer`
  - `array(T, S1, ..., Sn)` : multi-dimensional arrays
    - FORTRAN: `real(3,5)`

### Record Types

- A record is `{id1: T1, ..., idn: Tn}` for some identifiers idi and types Ti
- Supports access operations on each field, with corresponding type
- C: `struct { int a; float b; }`
- Pascal: `record a: integer; b: real; end`
- Objects: generalize the notion of records

### Pointer Types

- Pointer types characterize values that are addresses of variables of other types
- **Pointer(T)** : pointer to an object of type T
- C pointers: `T*` (e.g., `int *x;`)
- Pascal pointers: `^T` (e.g., `x: ^integer;`)
- Java: `object references`

### Function Types

- Type: `T1 x T2 x ... x Tn → Tr`
- Function value can be invoked with some argument expressions with types Ti, returns return type Tr
- C functions: `int pow(int x, int y)`
  - type: `int x int → int`
- Java: methods have function types
- Some languages have first-class functions
  - usually in functional languages, e.g., ML, LISP
  - C and C++ have function pointers
  - Java doesn’t

### Type Aliases

- Some languages allow type aliases (type definitions, equates)
  - C: `typedef int int_array[] ;`
  - Modula-3: `type int_array = array of int ;`
  - Java doesn’t allow type aliases
- Aliases are not type constructors!
  - `int_array` is the same type as `int [ ]`
- Different type expressions may denote the same type
Implementation

• Use a separate class hierarchy for types:
  class BaseType extends Type { ... }
  class IntType extends BaseType { ... }
  class BoolType extends BaseType { ... }
  class ArrayType extends Type { Type elemType; }
  class FunctionType extends Type { ... }

• Semantic analysis translates all type expressions to type objects
• Symbol table binds name to type object

Type Comparison

• Option 1: implement a method T1.Equals(T2)
  - Must compare type trees of T1 and T2
  - For object-oriented language: also need sub-typing: T1.SubtypeOf(T2)

• Option 2: use unique objects for each distinct type
  - each type expression (e.g., array[int] ) resolved to same type object everywhere
  - Faster type comparison: can use ==
  - Object-oriented: check subtyping of type objects

Creating Type Objects

• Build types while parsing – use a syntax-directed definition:
  non terminal Type type
  type ::= BOOLEAN
      | ARRAY LBRACKET type:t RBRACKET
  { RESULT = new ArrayType(t); : }

• Type objects = AST nodes for type expressions

Processing Type Declarations

• Type declarations add new identifiers and their types in the symbol tables
• Class definitions must be added to symbol table:
  class_defn ::= CLASS ID:id { decls:d }

• Forward references require multiple passes over AST to collect legal names
  class A { B b; }
  class B { ... }

Type-Checking

• Type-checking = verify typing rules
  "operands of + must be integer expressions; the result is an integer expression"

• Option 1: Implement using syntax-directed definitions (type-check during the parsing)
  expr ::= expr1 PLUS expr2
  { if (t1 == IntType & t2 == IntType)
      RESULT = IntType;
    else throw new TypeCheckError("+");
  :}

• Option 2: Implement type-checking by an AST visitor
  class typeCheck implements Visitor {
    Object visit(Add e, Object symbolTable) {
      Type t1 = (int) e.e1.accept(this, symbolTable);
      Type t2 = (int) e.e2.accept(this, symbolTable);
      if (t1 == Int & t2 == Int) return Int;
      else throw new TypeCheckError("+");
    }
    Object visit(Num e, Object symbolTable) {
      return Int;
    }
    Object visit(Id e, Object symbolTable) {
      return (SymbolTable)symbolTable.lookupType(e);
    }
  }
Next Time: Static Semantics

- Static semantics = mathematical description of typing rules for the language
- Static semantics formally defines types for all legal language ASTs