CS412/413
Introduction to Compilers
Radu Rugina

Lecture 12: Types
17 Feb 06

Types

• Today’s topics
  – Type errors
  – Type system concepts
  – Types constructors
  – 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 ∈ [-2^31, 2^31)”
  – Think: “type = set of possible values”

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

• Type-safety: absence of type errors

How to Ensure Type-Safety

• Bind (assign) types, then check types

• Type binding: defines type of 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
  – Consists of a set of type-checking rules

Type Checking

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

• Examples:
  – Unary and binary operators (+, =, ==, [ ] ) 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
  – 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, ML, Java, Pascal, Modula-3

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

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

Soundness

- Sound type systems: all programs that satisfy the typing rules are free of type errors
  - i.e., if program type-checks, then there are no type errors
- 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><strong>Static Typing</strong></td>
<td></td>
</tr>
<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><strong>Dynamic Typing</strong></td>
<td></td>
</tr>
<tr>
<td>Scheme</td>
<td>assembly code</td>
</tr>
<tr>
<td>PostScript</td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td></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 (at least 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

- Type $array(T) =$ type of arrays with elements of type $T$
  - C, Java: int[], Modula-3: array of integer
- $array(T, S)$ : array with size
  - C: int[10], Modula-3: array[10] of integer
  - Indexed from 0 to size-1
- $array(T, L, L')$ : array with upper/lower bounds
  - Ada: array (2..5) of integer
- $array(T, S_1, \ldots, S_n)$ : multi-dimensional arrays
  - FORTRAN: real(3,5)

Record Types

- A record is \{id$_1$, $T_1$, \ldots, id$_n$, $T_n$\} for some identifiers id$_i$ and types $T_i$
- 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

Type Aliases

- Some languages allow type aliases (a.k.a. type definitions, equates)
  - C: typedef int int_array;
  - Modula-3: type int_array = array of int;
  - Java does not have type aliases
- Aliases are not type constructors!
  - int_array is the same type as int[
- Different type expressions denote the same type

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 and array references (everything is a pointer)

Function Types

- Type: $T_1 \times T_2 \times \ldots \times T_n 
\rightarrow T'$
- Function value can be invoked with some argument expressions with types $T_i$, returns return type $T'$
- C functions : int pow(int x, int y)
  - Type int x int 
- Java: methods have function types
- Some languages have first-class functions
  - usually in functional languages, e.g. ML, Lisp
  - C/C++ have function pointers
  - Java doesn’t
Implementation

- Use a class hierarchy for types:
  
  ```java
  abstract class Type { ... 
  class IntType extends Type { ... } 
  class BoolType extends Type { ... } 
  class ArrayType extends Type { 
      Type elemType; ... } 
  class FunctionType extends Type { 
      Type[] paramTypes; 
      Type returnType; ... } 
  class ClassType extends Type { 
      ClassSymbol sym; ... } 
  }
  ```

Type Comparison

- Option 1: use a unique object for each distinct type
  - each type expression (e.g. array[int]) resolved to same type object everywhere
  - Use reference equality (==) for comparison

- Option 1: implement a method t1.equals(t2)
  - Must compare type trees of t1 and t2

- For object-oriented languages, also need sub-typing: t1.subtypeOf(t2)

Creating Type Objects

- Build types while parsing – use a syntax-directed definition:
  
  ```java
  non terminal Type 
  type ::= BOOLEAN 
          | ARRAY LBRAKT type:T RBRAKT 
          
  Type objects = AST nodes for type expressions
  ```

Type-Checking (1)

- 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)
  
  ```java
  expr ::= expr1 PLUS expr2 
  
  if ( t1 == Type.integer && t2 == Type.integer ) 
  
  RESULT = Type.integer; 
  
  else throw new TypeCheckError("+");
  ```

Type-Checking (2)

- Option 2: first build the AST, then implement type-checking by recursive traversal of the AST nodes:
  
  ```java
  class AddExpr extends Expr { 
      Type typeCheck() { 
          if ( a1.typeCheck() == Type.integer && 
               a2.typeCheck() == Type.integer) 
              return Type.integer; 
          else 
              throw new TypeCheckingError(this); 
          }
  }
  ```

Type-Checking (2)

- Identifier expressions: lookup the type in the symbol table
  
  ```java
  class IdExpr extends Expr { 
      Symbol id; 
      
      Type typeCheck() { 
          return id.getType();
      }
  }
  ```
Possible Strategy

- Separate AST construction from type checking phase
- Traverse the AST and perform semantic checks (or other actions) only after the tree has been built and its structure is stable
- This approach is less error-prone
  - It is better when efficiency is not a critical issue

Next Time: Static Semantics

- Visitors = a methodology for designing passes over the AST
- Static semantics = mathematical description of typing rules for the language
- Static semantics formally defines types for all legal language ASTs