CS412/413

Introduction to Compilers
Radu Rugina

Lecture 12: Types and Type-Checking
15 Feb 02

Semantic Analysis

- Last time:
  - Semantic errors related to scopes
  - Symbol tables
- 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 "$x \in [-2^{31}, 2^{31})$"
- 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 (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 Checking

- Static type checking = perform type checking at compile-time
- Dynamic type checking = ensure the correct usage of types at run-time
  - Check type requirements during program execution
- Examples of dynamic checking:
  - Array bounds checking
  - Null pointer dereferences

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 at compile-time and do not change during the execution of the program
  - E.g. C, Java, Pascal
- Dynamically typed language: types defined at run-time, during program execution
  - E.g. Lisp, 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 which contain type errors
- These concepts refer to run-time (dynamic execution of the program)
  - 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 checking: when to check types?
- Static vs dynamic typing: when to define types?
- Strong vs weak typing: how many type errors?
- Sound type systems: statically catch all type errors

Classification

	Strong Typing	Weak Typing
Static Typing	ML Pascal Java Modula-3	C C++
Dynamic Typing	Scheme PostScript Smalltalk	assembly code

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

- Language type systems have basic types (also: primitive types, ground types)
- Basic types examples: int, string, bool
- Build type expressions using basic types:
  - Type constructors:
    - array types
    - structure types
    - pointer types
  - Type aliases
  - Function types

Type Expressions: Arrays

- Various kinds of array types in different programming languages
- array(T) : arrays without bounds
  - C, Java: T [ ], Modula-3: array of T
- array(T, S) : array with size
  - C: T[S], Modula-3: array[S] of T
  - May be indexed 0..S-1
- array(T,L,U) : array with upper/lower bounds
  - Pascal: array[L .. U] of T
- array(T, S_1, …, S_n) : multi-dimensional arrays
  - FORTRAN: T(L_1,…, L_n)

Type Expressions: Structures

- More complex type constructor
- Has form {id_1: T_1, … , id_n: T_n} for some identifiers id_i and types T_i
- Is essentially cartesian product:
  - (id_1 x T_1) x … x (id_n x T_n)
- Supports access operations on each field, with corresponding type
- Structures in C: struct { int a; float b; }
- Records in Pascal: record a: integer; b: real; end
- Objects: extension of structure types

Type Expressions: 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

Type Expressions: Pointers

- 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

Type Expressions: Functions

- Type: $T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$
- Function value can be invoked with some argument expressions with types T_i, returns return type T_r
- C functions : int f(float x, float y)
- Java: methods have function types
- Some languages have first-class function types (C, ML, Modula-3, Pascal, not Java)

Implementation

- Use a separate class hierarchy for types:

```
class BaseType extends Type { String name; }
class IntType extends BaseType { … }
class BoolType extends Base Type { … }
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
          {: RESULT = new BoolType(id); :}
     |  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 ::= expr:t1 PLUS expr:t2
   {: if (t1 == IntType && t2 == IntType)
           RESULT = IntType;
      else throw new TypeCheckError("+");
   :}
```

Type-Checking

- Option 2: first build the AST, then implement type-checking by recursive traversal of the AST nodes:

```
class Add extends Expr {
    Type typeCheck() {
        Type  t1 = e1.typeCheck(),
              t2 = e2.typeCheck();
        if (t1 == Int && t2 == Int) return Int;
        else throw new TypeCheckError("+");
    }
}
```


Type-Checking Identifiers

- Identifier expressions: lookup the type in the symbol table

```
class IdExpr extends Expr {
    Identifier id;
    Type typeCheck() { return id.lookupType(); }
}
```

- Using syntax-directed definitions for forward references: type-checking will fail

Next Time: Static Semantics

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