CS412/413

Introduction to Compilers and Translators
Spring ’00

Lecture 9: Types and static semantics

Administration

- Programming Assignment 1 due now
- Programming Assignment 2 handout

Review

- Semantic analysis performed on representation of program as AST
- Implemented as a recursive traversal of abstract syntax tree

Semantic Analysis

- Catching errors in a syntactically valid program
  - Identifier errors: unknown identifier, duplicate identifier, used before declaration
  - Flow control errors: unreachable statements, invalid goto/break/continue statements
  - Expressions have proper type for using context
- This lecture:
  - What kinds of checks are done (particularly type checks)
  - How to implement types
  - Not covered in Appel or Dragon Book

Type checking

- Bulk of semantic checking
- Operators (e.g. +, !, [ ] ) must receive operands of the proper type
- Functions must be called w/ right number & type of arguments
- Return statements must agree w/ return type
- In assignments, assigned value must be compatible with type of variable on LHS.
- Class members accessed appropriately

Static vs. Strong Typing

- Many languages statically typed (e.g. C, Java, but not Scheme, Dylan): expressions, variables have a static type
- Static type is a predicate on values might occur at run time. int x; in Java means x ∈ [-2^{31}, 2^{31}). Types = efficiently decidable predicates
- Strongly typed language: operations unsupported by a value never performed at run time.
- In strongly typed language with sound static type system: run-time values of expressions, variables characterized conservatively by static type
**Type safety**

<table>
<thead>
<tr>
<th>Strongly typed</th>
<th>Not strongly typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML, Pascal, Iota, Java, Modula-3, Iota+</td>
<td>C, C++</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static type</th>
<th>Not static type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Scheme, PostScript, Smalltalk, SELF, Dylan, CLOS</td>
</tr>
</tbody>
</table>

**Why Static Typing?**

- Compiler can reason more effectively
- Allows more efficient code: don’t have to check for unsupported operations
- Allows error detection by compiler
- But:
  - requires at least some type declarations
  - type decls often can be inferred (ML)

**Dynamic checks**

- Even statically-typed languages have some dynamic checking
  - Array index out of bounds
  - null in Java, null pointers in C
  - Inter-module type checking in Java
- Sometimes can be eliminated through static analysis
  - harder than type checking: undecidable
  - theorem proving
  - can’t always eliminate these checks

**Type Systems**

- Type is predicate on values
- Arbitrary predicates: type checking intractable (theorem proving)
- Languages have type systems that define what types can be expressed and what static types expressions have
- Types described in program by type expressions: int, string, array[int], Object, InputStream[], Vector<int>

**Example: Iota type system**

- Language type systems have primitive types (also: basic types, base types, ground types)
- Iota: int, string, bool
- Also have type constructors that operate on types to produce other types
- Iota: for any type $T$, array[$T$] is a type. Java: $T[ ]$ is a type for any $T$

**Type expressions: aliases**

- Some languages (not Java) allow type aliases (type definitions, equates)
  - C: typedef int int_array[
  - Modula-3: type int_array = array of int;
- int_array is type expression denoting same type as int [ ] -- not a type constructor
- Different type expressions may denote the same type
Type Expressions: Arrays

- Different languages have various kinds of array types
- w/o bounds: array(T)
  - C, Java: T[ ], Modula-3: array T
- size: array(T, L) (may be indexed 0..L-1)
  - C: T[L], Modula-3: array[L] of T
- upper & lower bounds: array(T, L, U)
  - Pascal, Modula-3: indexed L..U
- Multi-dimensional arrays (FORTRAN)

Records/Structures

- More complex type constructor
- Has form \{id_i: T_i, id_j: T_j, ...\} for some ids and types T
- Supports access operations on each field, with corresponding type
- C: struct \{ int a; float b; \} corresponds to type \{ a: int, b: float \}
- Class types (e.g., Java) extension of record types

Functions

- Some languages have first-class function types (C, ML, Modula-3, Pascal, not Java)
- Function value can be invoked with some argument expressions with types T_i, returns return type T_r
- Type: T_1 \times T_2 \times ... \times T_n \rightarrow T_r
- C: int f(float x, float y)
  - f: float \times float \rightarrow int
- Function types useful for describing methods, as in Java, even though not values
  - extensions needed for exceptions.

Representing types

- Type-checking routine returned a Type object – what is it?
- Type typeCheck(SymTab s)

Option 1: make Type an AST node

abstract class Type extends Node
  [ abstract boolean equals(Type t); ]
class IdType extends Type [ String name; ]
class ArrayType extends Type [ Type elemType; ]
class FunctionType extends Type [ ... ]
- Type equality requires tree comparisons
- Must look in symbol table to interpret IdType; must make sure the right symbol table is available!

Representing type

- Type aliases, class definitions must be added to symbol table (usu. top-level) during semantic analysis
- Class defn ::= CLASS ID:id [ decls:d ]
  - AST for class_defn should be checked once for validity – mutual references can require multiple passes over AST to collect legal names
  - Sem. analysis binds (in ST) class names to objects representing checked type definitions:
    class IotaClass [ String name; SymTab decls; ... ]

Creating Type AST nodes

non terminal Type type_expr
  or Type parseType();

  type ::= ID:id
      [: RESULT = new IdType(id); :]
    | ARRAY LBRACKET type:t RBRACKET
      [: RESULT = new ArrayType(t); :]

Processing type declarations

- Type aliases, class definitions must be added to symbol table (usu. top-level) during semantic analysis
  - AST for class_defn should be checked once for validity – mutual references can require multiple passes over AST to collect legal names
  - Sem. analysis binds (in ST) class names to objects representing checked type definitions: class IotaClass [ String name; SymTab decls; ... ]
Another approach: type objects

- Option 2: resolve AST trees representing types to unique objects for each distinct type
  ```java
class BaseType extends Type {
    String name;
  }
  static BaseType Int, Char, Float, ...
class IotaClass extends Type { ... }
class ArrayType extends Type { Type elemType; }
```
- `array[int]` resolved to same type object everywhere
- Semantic analysis resolves all type expressions to type objects; symbol table binds name to type object
- Faster type equality: can use `==`, mostly
- Type meaning is independent of symbol table

Static Semantics

- Can describe the types used in a program. How to describe type checking?
- Formal description: *static semantics* for the programming language
- Static semantics defines types for all legal language ASTs
- We will write ordinary language syntax to mean the corresponding AST