CSE401: Semantic Analysis

Larry Snyder
Autumn 2003

Slide by Chambers, Eggens, Nozri, Ruzzo, Snyder and others
© L. Snyder and UW CSE, 1994-2003

Semantic analysis

- Perform final legality checking of input program
  - Properties not checked by lexical or syntactic checking
    - Example: type checking, ensuring break statement is in a loop, etc.
  - "Understand" program well enough to do the back-end synthesis activities
    - Example: relate particular names to particular declarations

Symbol tables

- Key data structure (at compile time, not run time)
  - Produced (and used) during semantic analysis
  - Used during code generation
  - Stores information about names used in the program
    - Declarations add entries to the symbol table
    - Uses of names look up appropriate symbol table entry

What information about names?

- Kind of declaration
  - var, const, proc, etc.
- Type
- For const: keep value
- For var: Where allocated in memory?
  - Static, stack, heap? Offset?
  - Not computed initially, but later on
- For formal parameter: passed by-value, by-ref...

Example: a PL/0 DeclList

var x : int;
var q : array[20] of bool;
procedure foo(a : int); begin end foo;
const z : int = 10;
PL/0 symbol table entries

```cpp
class SymTabEntry {
public:
    char* name();
    Type* type();
    virtual bool isConstant();
    virtual bool isVariable();
    virtual bool isFormal();
    virtual bool isProcedure();
    virtual int value(); // const only
    virtual int offset(SymTabScope* s); // var only
};
```

SymTab subclasses

```cpp
class VarSTE : public SymTabEntry { ... };
class FormalSTE : public VarSTE { ... };
class ConstSTE : public SymTabEntry { ... };
class ProcSTE : public SymTabEntry { ... };
```

Nested scopes: Example

```cpp
procedure foo(x:int, w:int);
    var z:bool;
    const y:boolean = true;
    procedure bar(x:array[5] of bool);
        var y:int;
        begin
            x[y] := z;
        end bar;
    begin
        while z do
            var z:int, y:int;
            y := z + x;
            output := x * y;
        end foo;
```

Nested scopes: How to handle?

1. What happens when the same name is declared in different scopes?
2. This is the first question of language design: what is the defined semantics?
3. Two standard choices
   - Lexical (static) scoping: use the block structure of the program
   - Do you remember choice #2 from 341?

Nested Scopes: Lexical/static

1. The syntactic (block) structure of the program determines how names are resolved
2. Given a name in a block
   - The nearest enclosing block with a declaration for that name is the relevant declaration
   - If none, it’s an error

Nested scopes: Dynamic
Lexical scope and symbol tables

§ Each scope has its own symbol table
§ Logically, for a block-structured program, there is a tree of symbol tables
  • Root = outermost block

Tree of symbol tables

procedure foo(x:int, w:int);
  var z:bool;
  const y:bool = true;
procedure bar(x:array[5] of bool);
  var y:int;
  begin
    y := z;
  end bar;
  begin
    while z do
      var z:int, y:int;
      y := z * x; end;
      output := x * y;
  end foo;

Nested scope operations

§ When encounter a new scope during semantic analysis
  • Create a new, empty scope
  • Its parent is the current scope (that of enclosing block)
  • New scope becomes “current”
§ When encounter a declaration
  • Add entry to the current scope
  • Check for duplicates in the current scope only (why?)
§ When encounter a use
  • Search scopes for declaration: current, its parent, grandparent,…
§ When exiting a scope
  • Parent becomes current again

PL/0 symbol table interface

class SymTabScope {
  public:
    SymTabScope(SymTabScope* enclosingScope);
    void enter(SymTabEntry* newSymbol);
    SymtabEntry* lookup(char* name);
    SymtabEntry* lookup(char* name, SymTabScope* & retScope);
  …
}

Implementing nested scopes

§ Each scope (instance of SymTabScope) keeps a pointer to its enclosing
SymTabScope (_parent)
§ Each scope maintains “down links”, too (_children, so we can walk the whole tree)
Symbol tables: Implementation

Abstractly, it’s simple: a mapping from names to information, aka key/value pairs

Concretely, there are lots of choices, each with different performance consequences, e.g.
- Linked list (or dynamic array)
- Binary search tree
- Hash table

So, we’ll take a brief trip down CSE326 memory lane…

Symbol tables: Complexity

<table>
<thead>
<tr>
<th></th>
<th>Enter</th>
<th>Lookup</th>
<th>Space cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Linked lists</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B. Binary search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C. Hash table</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Symbol tables: Other issues

- Linked lists must have keys that can be compared for equality
- Binary search trees must have keys that can be ordered
- Hash tables must have keys that can be hashed (well)
- Hash table size?

Symbol tables: Implementation Summary

In general
- Use a hash table for big mappings
- Use a binary tree or linked list for small mappings
- Ideally, use a self-reorganizing data structure

Types

Types are abstractions of values that share common properties
- What operations can be performed on them
- (Usually) how they are represented in memory
- Types usually guide how compilation proceeds

Taxonomy of types

- Basic/atomic types
  - int, bool, char, real, string, …
  - enum(v₁, v₂, …, vₙ)
- User-defined types: Stack, SymTabScope,…
  - Type constructors
  - Parameterized types
  - Type synonyms
Type constructors

- `ptr(type)`
- `array(index-range, element-type)`
- `record(name1:type1, ..., nameN:typeN)`
- `tuple(type1, ..., typeN)` or `type1 × ... × typeN`
- `union(type1, ..., typeN)` or `type1 + ... + typeN`
- `function(arg-types, result-type)` or `type1 × ... × typeN → result-type`

Parameterized types

- Functions returning types
  - `Array<T>`
  - `Stack<T>`
  - `HashTable<Key, Value>`

Type synonyms

Give alternative name to existing type

```c
typedef SymTabScope* SymTabReg
```

Type checking

- A key part of language implementation
  - Semantic analysis phase, linking, and/or runtime
- Verifies that operations on values will be legal
  - I.e., they compute values that will be legal in context

Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 4</td>
<td>3 + 4.0</td>
</tr>
<tr>
<td>3 + x</td>
<td>3 + 'x'</td>
</tr>
<tr>
<td>3[x]</td>
<td>x[3]</td>
</tr>
<tr>
<td>3 + TRUE</td>
<td>x.y-&gt;z*</td>
</tr>
</tbody>
</table>

Type checking terminology

- Static vs. dynamic typing
  - Static: checked prior to execution (e.g., compile-time)
  - Dynamic: checked during execution
- Strong vs. weak typing
  - Strong: guarantees no illegal operations performed
  - Weak: no such guarantee
- Caveats
  - Hybrids are common
  - Mistaken usages of these terms is common
    - Ex: "untyped", "typeless" could mean "dynamic" or "weak"

Type weaknesses in C/C++

```c
extern myfunc(double*);
main() {
    int i=42, j=0, *ip=i;
    double x=3.14, y[10];
    scanf("%d %f", &i, &j);
    x = (double) i;
    x = (double*) ip;
    (*ip) = 1;
    (*ip) = 1;
    y[11] = 1;
    myfunc(&x);
}
```

```c
myfunc(int *kp) {
    char c="1";
    union {
        int i;
        double x;
    } huh;
    c = sqrt(c);
    huh.x = 42.0;
    huh.i += 1;
    *kp = huh.i;
}
```
More on C++ type system

```c
Stmt* sp;
IfStmt* isp;
isp = new IfStmt(...);
sp = isp;
sp = (Stmt*) isp;
...
isp = (IfStmt*) sp;
// Better:
if (isp = dynamic_cast<IfStmt*> sp) {
    sp = isp -> _then_stmts -> fetch(14);
}
```

Fill in with real languages

<table>
<thead>
<tr>
<th>Statically typed</th>
<th>Dynamically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong typing</td>
<td>Weak typing</td>
</tr>
</tbody>
</table>

Type checking

5 Assume we have an AST for the source program
   • It is syntactically correct
   • The symbol table has been computed
5 Does it meet the type constraints of the language?
   • Ex: `a := 3 * b + fork(c + 3.14159)`
     - What are the types of `a`, `b`, and `c`?
     - What type does `fork` return?
     - What type does `fork` accept?
     - What happens when `c` is added to a `float`?
     - What happens when `b` is multiplied by `3`?
     - What happens when `fork`'s result is added to `3 * b`?

Type checking strategy

5 Traverse AST recursively, starting at root node
   • Most work is on the bottom-up pass
5 At each node
   • Recursively type check any subtrees
   • Check legality of current node, given children's types
   • Compute and return result type (if any) of current node

Example: `3 * b + fork(c + 3.14159)`

```
Symtab
b: int
  +
  |
+--------
3  b  +
     |
  |
  c + 3.14
```

Top-down information also:

5 Need to know types of variables referenced
   • Must pass down symbol table during traversal
5 Legality of (e.g.) `break` and `return` statements depends on context: pass down
   • whether in loop,
   • what the result type of the function must be,
   • etc.
Representing types in PL/0

```java
class Type {
  virtual bool same(Type* t);
};
class IntegerType : public Type {...};
class BooleanType : public Type {...};
class ProcedureType : public Type {
  TypeArray* _formalTypes;
};
IntegerType* integerType; // predefined instances
BooleanType* booleanType;
```

PL/0 type checking: overview

```java
Type* Expr::typecheck(SymTabScope* s);
void Stmt::typecheck(SymTabScope* s);
void Decl::typecheck(SymTabScope* s);
Type* LValue::
typecheck_Lvalue(SymTabScope* s);
int Expr::resolve_constant(SymTabScope* s);
Type* TypeAST::typecheck(SymTabScope* s);
```

Type checking PL/0 expressions

A simple case: integer literals (like "0" or "-17")

```java
Type* IntegerLiteral::typecheck(SymTabScope* s) {
  return integerType;
}
```

Type checking var references

```java
Type* VarRef::typecheck(SymTabScope* s) { ...
  SymTabEntry* ste = s->lookup_ident;
  char* errormsg = new char[errormsgbuffersize];
  sprintf(errormsg, "undeclared var \%s\ referenced\%, _ident);
  Plzero->typeError(errormsg, line);
  if (! ste->isConstant() &&
      ! ste->isVariable()) { ...
    char* errormsg = new char[errormsgbuffersize];
    sprintf(errormsg, "\%s not const or var\%, _ident);
    Plzero->typeError(errormsg, line);
    return ste->type();
  }
  return ste->type();
}
```

Type checking operators

```java
Type* BinOp::typecheck(SymTabScope* s) { ...
  switch (_op) { ...
    case PLUS:case MINUS:case MUL: case EQ: ...
    return integerType;
    case EQL:case NEQ:case LSS: case LTE: ...
    return booleanType;
    default:
      Plzero->fatal("unexpected BINOP");
      return NULL; // not actually executed
  }
}
```
Type checking assignments

```c
void AssignStmt::typecheck(SymTabScope* s) {
    Type* lhs = _lvalue->typecheck(_lvalue(s));
    Type* rhs = _expr->typecheck(s);
    if (lhs->different(rhs))
        Plzero->typeError("lhs type differs from rhs");
}
```

Type checking if statements

```c
void IfStmt::typecheck(SymTabScope* s) {
    Type* testType = _test->typecheck(s);
    if (testType->different(BooleanType))
        Plzero->typeError("test not Boolean");
    for (int i = 0; i < _then_stmts->length(); i++) {
        _then_stmts->fetch(i)->typecheck(s);
    }
}
```

Type checking call statements

```c
void CallStmt::typecheck(SymTabScope* s) {
    int i;
    TypeArray* argTypes = new TypeArray;
    for (i = 0; i < _args->length(); i++) {
        Type* argType = _args->fetch(i)->typecheck(s);
        argTypes->add(argType);
    }
    SymTabEntry* ste = s->lookup(_ident);
    if (ste == NULL) {
        Plzero->typeError("undeclared procedure");
    }
}
```

Type checking declarations

```c
void VarDecl::typecheck(SymTabScope* s) {
    for (int i = 0; i < _items->length(); i++) {
        _items->fetch(i)->typecheck(s);
    }
}
```

```c
void ConstDecl::typecheck(SymTabScope* s) {
    Type* t = _type->typecheck(s);
    Type* type = _expr->typecheck(s);
    Value* constant_value = _expr->resolve_constant(s);
    if (t->different(type)) {
        Plzero->typeError(...);
    }
    ConstSTE* constSTE = new ConstSTE(_name, t, constant_value);
    s->enter(constSTE, line);
}
```
Type checking

We've covered the basic issues in how to check semantic, type-oriented, properties for the data types and constructs in PL/0 (and some more)

But there are other features in languages richer than PL/0, and we'll look at some of them today

Records

Records (aka structs) group heterogeneous types into a single, usually named, unit

```
record R = begin
  x : int;
  a : array[10] of bool;
  m : char;
end record;

var t : R;
...
R.x
```

Type checking records

- Need to represent record type, including fields of record
- Need to name user-defined record types
- Need to access fields of record values
- May need to handle unambiguous but not fully qualified names (depending on language definition)

An implementation

- Representing record type using a symbol table for fields
  - class RecordType: public Type { ...);
  - CreateRecordTypeSTE
- To typecheck expr.x
  - Typecheck expr
    - Error if not record type
  - Lookup x in record type's symbol table
    - Error if not found
  - Extract and return type of x
Type checking classes & modules

5 A class/module is just like a record, except that it contains procedures in addition to simple variables
5 So they are already supported by using a symbol table to store record/class/module fields
5 Procedures in the class/module can access other fields of the class/module
   • Already supported: nest procs in record symbol table
5 Inheritance?

Type equivalence

5 When is one type equal to another?
   • Implemented in PL/0 with Type::same function
5 It’s generally “obvious” for atomic types like int, string, user-defined types (e.g., point2d vs complex)
5 What about type constructors like arrays?
   var a1 : array[10] of int;
   var a2,a3 : array[10] of int;
   var a4 : array[20] of int;
   var a5 : array[10] of bool
   var a6 : array[0:9] of int;

Equivalence, def I: Structural Eq.

5 Two types are structurally equivalent if they have the same structure
   • If atomic types, then obvious
   • If type constructors
      – Same constructor
      – Recursively, equivalent arguments to constructor
5 Implement with recursive same

Equivalence, def II: Name Eq.

5 Two types are name equivalent if they came from the same textual occurrence of a type constructor
5 Implement with pointer equality of Type instances
5 Special case: type synonyms don’t define new types

Implementing structural equivalence (details)

5 Problem: want to dispatch on two arguments, not just receiver
   • That is, choose what method to execute based on more than the class of the receiver
5 Why? There’s a symmetry that the OO dispatch approach skews
   • if (lhs->different(rhs)) {...error...}
5 Why not: if (different(lhs,rhs)) {...error...}
Multi-methods

§ Languages that support dispatching on more than one argument provide multi-methods

§ For example, they might look like
  • virtual bool same(type* t1, type* t2)
    (return false);
  • virtual bool same(intType* t1, intType* t2)
    (return true);
  • virtual bool same(ProcType* t1, ProcType* t2)
    (return same(t1->args, t2->args));

§ Different from static overloading in C++

Overloading: quick reminder

§ Overloading arises when the same operator or function is used to represent distinct operations
  • 3 + 4
  • 3.14159 + 2.71828
  • “mork” + “mindy”

§ The compiler statically decides which “+” to compile to based on the (type) context

Polymorphism: quick reminder

§ Polymorphism is different from overloading

§ In overloading the same operator means different things in different contexts

§ In polymorphism, the same operator works on different types of data
  • (length "a b c") vs. (length "((a) (b c) 3 4")
  • (sort "4 2 3") vs. (sort "c a g")

§ In polymorphism, the compiler compiles the same code regardless

But C++ has no multi-methods:
So we use double dispatching

class Type {
  virtual bool same(Type* t) = 0;
  virtual bool isInteger() {return false;}
  virtual bool isProc() {return false;}
};

class IntegerType : public Type {
  bool isInteger() {return t->isInteger();}
  bool isInteger() {return true;}
};

Type conversions and coercions

§ In C, can explicitly convert data of type float to data of type int (and some other examples)
  • Represent it explicitly as a unary operator
  • Type checking and code generation work as normal

§ In C, can also implicitly coerce
  • System must insert unary conversion operators as part of type checking
  • Code generation works as normal

Type casts

§ In C, Java (and some others) can explicitly cast an object of one type to another
  • Sometimes a cast means a conversion
    – E.g., casts between numeric types
    – Type-safe, but sometimes entails loss of accuracy
  • Sometimes a cast means just a change of static type without any computation
    – E.g., casts between pointer types
    – Generally NOT type-safe
Safety of casting

5 In C, the safety of casts is not checked
   • That is, it’s possible to convert into a representation that is illegal for the new type of data
   • Allows writing of low-level code that’s type-unsafe
   • More often used to work around limitations in C’s static type system

5 In Java, downcasts from superclass to subclass include a run-time type check to preserve type safety
   • This is the primary place where Java uses dynamic type checking

Where are we?

5 We now know, in principle, how to
   1. take a string of characters
   2. convert it into an AST with associated symbol table
   3. and know that it represents a legal source program (including semantic checks)

5 That is the complete set of responsibilities (at a high-level) of the front-end of a compiler

Next…

5 …what to do now that we have this wonderful AST representation

5 We’ll look mostly at interpreting it or compiling it
   • But you could also analyze it for program properties
   • Or you could “unparse” it to display aspects of the program on the screen for users
   • …