CS 4120
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 8: Semantic analysis

14 Sep 09

AST

- **A**bstract **S**yntax **T**ree is a tree representation of the program. Used for
  - semantic analysis (type checking)
  - some optimization (e.g. constant folding)
  - intermediate code generation (sometimes intermediate code = AST with somewhat different set of nodes)
- Compiler phases = recursive tree traversals
  - building new tree or modifying tree in place for next compiler phase
- Object-oriented and functional languages both convenient for defining AST nodes

Goals of Semantic Analysis

- Find all possible remaining errors that would make program invalid
  - undefined variables, types
  - type errors that can be caught *statically*
  - uninitialized variables, unreachable code
- Figure out useful information for later compiler phases
  - types of all expressions
  - data layout: memory sizes

Recursive semantic checking

- Program is tree, so...
  - recursively traverse tree, checking each component
  - traversal routine returns information about node checked

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

Type-checking identifiers

```
class Id extends Expr {
   String name;
   Type typeCheck() {
        return ?
   }
}
```

Need a **environment** that keeps track of types of all identifiers in scope: **symbol table**

Symbol table

- Can write formally as set of *identifier* : *type* pairs: { x: int, y: array[string] }

```
{
   int i, n = ...;
   for (i = 0; i < n; i++) {
       boolean b = ...
   }
```

{ i: int, n: int }

{ i: int, n: int, b: boolean }

?

Specification

- Symbol table maps identifiers to types

```
class SymTab {
  Type lookup(String id) ...
  void add(String id, Type binding) ...
}
```


Using the symbol table

- Symbol table is argument to all checking routines

```
class Id extends Expr {
   String name;
   Type typeCheck(SymTab s) {
     try {
        return s.lookup(name);
     } catch (NotFound exc) {
        throw new UndefinedIdentifier(this);
     }
   }
}
```

Propagation of symbol table

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

- Same variables in scope – same symbol table used
- When do we add new entries to symbol table?

Adding entries

- Java, Iota9: statement may declare new variables.
  `{ a = b; int x = 2; a = a + x }`
- Suppose $\{stmt_1;\ stmt_2;\ stmt_3 \ldots\}$ represented by AST nodes:
  ```
  abstract class Stmt { … }
  class Block { Vector/*Stmt*/ stmts; … }
  ```
- And declarations are a kind of statement:
  ```
  class Decl extends Stmt {
      String id; TypeExpr typeExpr; …
  ```


A stab at adding entries

```
class Block { Vector stmts;
    Type typeCheck(SymTab s) { Type t;
        for (int i = 0; i < stmts.length(); i++) {
            t = stmts[i].typeCheck(s);
            if (stmts[i] instanceof Decl)
                Decl d = (Decl) stmts[i];
                s.add(d.id, d.typeExpr.interpret());
        }
        return t;
    }
}
```

Does it work?

Restoring Symbol Table

scope of y

```
{ int x = 5;
   { int y = 1; }
   x = y;          // should be illegal!
}
```

Handling declarations

```
class Block { Vector stmts;
   Type typeCheck(SymTab s) { Type t;
        SymTab s1 = s.clone();
        for (int i = 0; i < stmts.length(); i++) {
            t = stmts[i].typeCheck(s1);
            Decl d = (Decl) stmts[i];
                s1.add(d.id, d.typeExpr.interpret());
        }
        return t;
   }
}
```

Declarations added in block (to s1) don't affect code after the block

Storing Symbol Tables

- Many symbol tables constructed during checking
  - May keep track of more than just variables: type definitions, break & continue labels, ...
  - Top-level symbol table contains global variables, type & module declarations,
  - Nested scopes result in extended symbol tables containing add'l definitions for those scopes.
- Can reconstruct symbol tables, but useful to save in corresponding AST nodes to avoid recomputation

How to implement Symbol Table?

- Imperative? Three operations:

```
Object lookup(String name);
void add (String name, Object type);
SymTab clone(); // expensive?
```

- Functional? Two operations:

```
Object lookup(String name);
SymTab add (String, Object); // expensive?
```


Imperative: Linked list of tables

```
class SymTab {
   SymTab parent;
   HashMap table;
   Object lookup(String id) {
     if (table.get(id) != null) return table.get(id);
     else return parent.lookup(id); // can cache..
   }
   void add(String id, Object t)
     { table.add(id,t); }
   SymTab(Symtab p)
     { parent = p; } // =clone
}
```

Functional: Binary trees

- Discussed in Appel Ch. 5
- Implements the two-operation interface

```
Object lookup(String name);
SymTab add (String, Object);
```

  - non-destructive add so no cloning is needed
  - O(lg n) performance: clones only the path from added node to the root.

Decorating the tree

- How to remember expression type?
- One approach: record in the node

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

- Maybe useful to record: symbol table used (if not destructively modified later...)

Structuring Analysis

- Analysis is a traversal of AST
- Technique used in lecture: recursion using methods of AST node objects—object-oriented style

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


Redundancy

- There will be several more compiler phases like typeCheck and foldConstants
  - constant folding
  - translation to intermediate code
  - optimization
  - final code generation
- Object-oriented style: each phase is a method in AST node objects
- Weakness 1: code for each phase spread
- Weakness 2: traversal logic replicated

Separating Syntax, Impl.

- Can write each traversal in a *single* method

```
Type typeCheck(Node n, SymTab s) {
   if (n instanceof Add) {
       Add a = (Add) n;
              Type t1 = typeCheck(a.e1, s),
                    t2 = typeCheck(a.e2, s);
              if (t1 == Int && t2 == Int) return Int;
              else throw new TypeCheckError("+");
   } else if (n instanceof Id) {
              Id id = (Id)n;
              return s.lookup(id.name); …
```

- Now, code for a given *node* spread all over!

Modularity Conflict

- No good answer!
- Two orthogonal organizing principles: node types and phases (rows or columns)

phases →

node types ↓

	typeCheck	foldConst	codeGen
Add	×	×	×
Num	×	×	×
Id	×	×	×
Stmt	×	×	×

Constant Folding

- AST optimization: replaces constant expressions with constants they would compute
- Traverses (and modifies) AST

```
abstract class Expr {
  Expr foldConstants();
}
class Add extends Expr { Expr e1, e2;
  Expr foldConstants() {
    e1 = e1.foldConstants(); e2 = e2.foldConstants();
    if (e1 instanceof IntConst && e2 instanceof IntConst)
        return new IntConst(e1.value + e2.value);
    else return new Add(e1, e2);
  }
```


Which is better?

- Neither completely satisfactory
- Both involve repetitive code
  - modularity by objects (rows): different traversals share basic traversal code—boilerplate code
  - modularity by operations (columns): lots of boilerplate:

```
if (n instanceof Add)  { Add a = (Add) n; …}
else if (n instanceof Id)  { Id x = (Id) n; … }
else …
```

Visitors

- Idea: avoid repetition by providing one set of standard traversal code.
- Knowledge of particular phase embedded in **visitor** object.
- Standard traversal code is done by object methods, reused by every phase.
- Visitor invoked at every step of traversal to allow it to do phase-specific work.

A Visitor Methodology

- Class Node is superclass for all AST nodes
- NodeVisitor is superclass for all visitor classes (one visitor class per phase)

```
abstract class Node {
  public final Node visit (NodeVisitor v) {
     Node n = v.override (this);        // default: null
     if (n != null) return n;
     else {
       NodeVisitor v_ = v.enter(this);  // default: v_=v
       n = visitChildren (v_);          // visit children
       return v.leave(this, n, v_);     // default: n
     }}
  abstract Node visitChildren(NodeVisitor v);
```


Folding constants with visitors

```
public class ConstantFolder extends NodeVisitor {
  public Node leave (Node old, Node n, NodeVisitor v) {
    return n.foldConstants();
         // note: all children of n already folded
  }
}

class Node { Node foldConstants( ) { return this; } }
class BinaryExpression {
   Node foldConstants( ) { switch(op) {...} } }
class UnaryExpression {
   Node foldConstants( ) { switch(op) {...} } }
```


Summary

- Semantic analysis: traversal of AST
- Symbol tables needed to provide context during traversal
- Traversals can be modularized differently
- Visitor pattern avoids repetitive code
- Read Appel, Ch. 4 & 5
- See also: *Design Patterns*