CS 4120
Introduction to Compilers

Andrew Myers

Cornell University

Lecture 12: Modules, Type representations, Visitors

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("+");
  }}
```


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); …
```

- (How we'd do it in a functional language)
- Now, code for a given *node* spread all over!

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);
  }
```

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

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	×		×

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.

Visitor pattern

```
class Node {
    void accept(Visitor v);
}
class FooNode extends Node {
    void accept(Visitor v) {
        invoke accept(c) for every child c
        v.visitFoo(this);
    }
}
```

```
class Visitor {
    void acceptFoo(Foo n) { }
    void acceptBar(Bar n) { }
    ...
}
class XVisitor extends Visitor {
    void acceptFoo(Foo n) {
        do whatever work pass X
        should do on Foo.
    }
}
```


Polyglot Visitors

- Allow rewriting AST lazily in functional style
- 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* (The “Gang of Four book”)