City University London

IN2009

Language Processors

Week 7

Semantic Analysis

Igor Siveroni

Session Plan

Session 5: Semantic analysis

- Announcement
- Extending MiniJava
- Symbol tables
  - Environments
  - Hash tables
  - Symbol table for MiniJava
- Typechecking

Announcement

- Coursework 2
  Deadline extension:
  Wednesday 25 March 2009
  11:59pm

Extending MiniJava

- A sample MiniJava program:

```
class Test {
  public static void main(String[] args)
   { System.out.println((new Pow()).f());
}}

class Pow {
    public int f() {
        int x; int y; int z;
        x =3; y = x * 10; return 5;
}}
```

- Suppose we want to extend MiniJava with the power operator “^” e.g. `return (y ^ x)`

Extending MiniJava

We need to extend the grammar:

```
Exp Expression():
 …
 | LOOKAHEAD( PrimaryExpression() ”^" )
   e=PowerExpression()
```

And

```
Exp PowerExpression() :
{ Exp e1,e2; }
{
  e1=PrimaryExpression() ”^”
 e2=PrimaryExpression()
  { return new Power(e1,e2); }
}
```


Extending MiniJava

Create the Java class for the Power AST:

```
package syntaxtree;
import visitor.Visitor;

public class Power extends Exp {
  public Exp e1,e2;
  public Power(Exp ae1, Exp ae2) {
    e1=ae1; e2=ae2;
  }
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

Extending MiniJava

Add to Visitor.java (interface):

```
public void visit(Power n);
```

Add to Visitor implementation:

```
// In DBPrettyVisitor.java
public void visit(Power n) {
        System.out.print("(");
        n.e1.accept(this);
        System.out.print(" ^ ");
        n.e2.accept(this);
        System.out.print(")");
     }
```


Semantic analysis

- Connects variable, type/class and function definitions to their uses (identifiers).
- Checks that each use matches an appropriate declaration
  - According to scope rules of language.
- Checks that each expression/part of the program is of correct type.
- Translates abstract syntax into simpler representation suitable for generating machine code (IR).

Semantic analysis

- Collects all identifiers in symbol table(s)
  - with attributes such as:
    - type and scope, for variables.
    - Parameter types & return type, for method declarations
  - By traversal of declaration ASTs
- Check that applied occurrences of identifiers have related declarations (are in table) at right scope
  - by traversal of rest of program ASTs

Semantic analysis

- Check that types of (sub-)expressions and parts of statements are of expected type
  - by traversal of rest of program ASTs and collecting types
  - For example:
    - actual parameter matches type of formal.
    - In A+B, A and B must be of type int.

Symbol tables (Environments)

- Map identifiers to their types & locations
- Identifiers (variable names, function/method names, type/class names) are bound to "meanings" (bindings) in symbol tables. For example:
  - Type of variable name
  - Number and type of parameters of function
  - Scope
- Perform lookup in symbol table when there is a *use* (non-defining occurrence) of identifier

Scope of an identifier: Parts of the program where the identifier is visible. For example:

```
class X {
  public int m(int y) {
  // scope of y }
}
```

Discard identifier bindings local to scope at end of scope analysis

Environment is a set of bindings → e.g. a type environment look like:

$$\sigma_0 = \{g \rightarrow \texttt{string}, a \rightarrow \texttt{int}\}$$

Example

```
class C {
  int a; int b; int c;
  public void m() {
    System.out.println(a+c);
    int j = a+b;
    String a = "hello"
    System.out.println(a);
    System.out.println(j);
    System.out.println(b);
  }
}
```

1 σ_0 is starting environment
2 $\sigma_1 = \sigma_0 + \{a \rightarrow \texttt{int}, b \rightarrow \texttt{int}, c \rightarrow \texttt{int}\}$
3
4 lookup ids `a`, `c` in σ_1
5 $\sigma_2 = \sigma_1 + \{j \rightarrow \texttt{int}\}$, lookup a, b in σ_1
6 $\sigma_3 = \sigma_2 + \{a \rightarrow \texttt{string}\}$
7 lookup `a` in σ_3
8 lookup `j` in σ_3
9 lookup `b` in σ_1
10 discard σ_3 revert to σ_1
11 discard σ_1 revert to σ_0

Need to deal with clashes (different bindings for same symbol)

$$\sigma_2 : a \rightarrow \texttt{int} \ , \ \sigma_3 : a \rightarrow \texttt{string}$$

Prefer *most recent binding* so as to implement scope rules of language

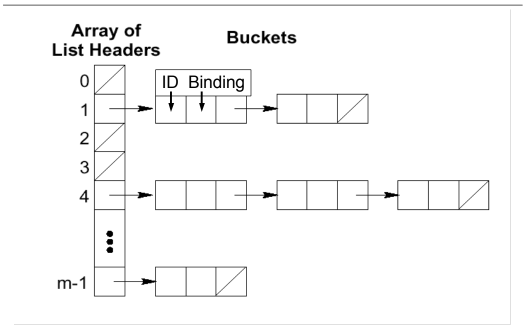
Imperative implementation style of Environments

- Modify σ_1 to σ_2 (destructive update)
- Undo modification to get back to σ_1
- Use a single global environment
- Implement with *hash tables*
- $\sigma' = \sigma + \{a \mapsto \tau\}$ Implement by inserting τ into hash table with key a
- Simple hash table with *external chaining*:
  i th bucket = linked list of all elements whose keys hash to *i* mod **`SIZE`**

Hashing

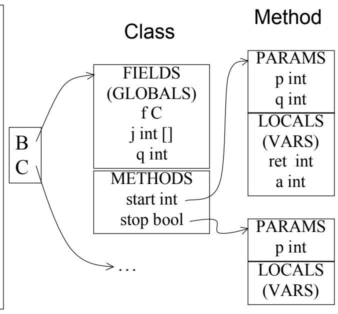
- **`java.util.Hashtable`** implements this for us.
- A hashtable will be used for storing classes.
- Each class will have two hashtables to store the fields (globals) and methods.
- Each method will have a hashtable to store the local variables.

Hashing with external chaining



Symbol table for MiniJava

```
class B {
  C f; int [] j; int q;
  public int start(int p, int q) {
    int ret; int a;
    ...
    return ret;
  }
  public boolean stop(int p) {
    ...
    return false;
  }
class C { ... }
```



MiniJava symbols & typechecking

- Each variable name and formal parameter bound to its type
- Each method name bound to its formal parameters, result type, and local variables
- Each class name bound to its variable (field) and method declarations
- Typechecking:
  - First, build the symbol table
  - Then typecheck the statements and expressions in the AST, consulting the symbol table for each identifier found.

Symbol table implementation

```
class SymbolTable {
  public SymbolTable();
  public boolean addClass(String id, String parent);
  public Class getClass(String id);
  public boolean containsClass(String id);
  public Type getVarType(Method m, Class c, String id);
  public Method getMethod(String id, String classScope);
  public Type getMethodType(String id, String
   classScope);
  public boolean compareTypes(Type t1, Type t2);
}
```

- SymbolTable contains a hashtable of Class objects…

Symbol table implementation

- **`getVarType(Method m, Class c, String id)`**
  - in **`c.m`**, find variable **`id`**
  - may be…
    - local variable in method
    - parameter in formal parameters in method
    - variable in the class
    - variable in a parent class

Symbol table implementation

- **`getMethod()`**, **`getMethodType()`**
  - may be defined in the parent classes
- **`compareTypes(t1,t2)`**
  - Primitive types **`IntegerType`**, **`BooleanType`**, **`IntegerArrayType`**
  - **`IdentifierType`**s (class types) stored as strings, returns true if identical or if equals a parent

Symbol table class `Class`

```
class Class {
  public Class(String id, String parent);
  public String getId();
  public Type type();
  public boolean addMethod(String id, Type type);
  public Method getMethod(String id);
  public boolean containsMethod(String id);
  public boolean addVar(String id, Type type);
  public Variable getVar(String id);
  public boolean containsVar(String id);
  public String parent();
}
```

- Class contains a hashtable of global variables (fields) and a hashtable of methods

Symbol table variable representation

```
class Variable {
  public Variable(String id, Type type);
  public String id();
  public Type type();
}
```


Symbol table method representation

```
class Method {
  public Method(String id, Type type);
  public String getId();
  public Type type();
  public boolean addParam(String id, Type type);
  public Variable getParamAt(int i);
  public boolean getParam(String id);
  public boolean containsParam(String id);
  public boolean addVar(String id, Type type);
  public Variable getVar(String id);
  public boolean containsVar(String id);
}
```

- Method contains a vector of parameters (formals) and a hashtable of (local) variables, and a return type

Implementation

- By visitors, just like the pretty printer
  - Interface **`Visitor`** as before.
  - A new visitor interface **`TypeVisitor`** is just the same but its methods return a Type
  - General classes **`DepthFirstVisitor`** and **`TypeDepthFirstVisitor`** implement these interfaces and traverse the AST depth-first visiting each node, but taking no action

Implementation

- Building the symbol table
  - **`BuildSymbolTableVisitor`** extends **`TypeDepthFirstVisitor`**, overriding methods so as to add classes, methods, vars, etc
- Typechecking
  - **`TypeCheckVisitor`** extends **`DepthFirstVisitor`** overriding methods so as to check statements, **`TypeCheckExpVisitor`** extends **`TypeDepthFirstVisitor`** overriding methods so as to check expressions

BuildSymbolTableVisitor

• Note that some checks are done as the table is built, eg for variable declarations (see below) a check is made on whether the variable is already declared

```
public class BuildSymbolTableVisitor extends
  TypeDepthFirstVisitor {
…
 private Class currClass;
 private Method currMethod;
…
 // Type t;
 // Identifier i;
  public Type visit(VarDecl n) {

     Type t =  n.t.accept(this);
     String id =  n.i.toString();
```


BuildSymbolTableVisitor (2)

```
  if (currMethod == null){
    if (!currClass.addVar(id,t)){
      System.out.println(id + "is already defined in "
                   + currClass.getId());
    }
  } else {
    if (!currMethod.addVar(id,t)){
      System.out.println(id + "is already defined in "
                   + currClass.getId() + "." +
                   currMethod.getId());
    }
  }
  return null;
}
```


TypeCheckVisitor

```
public class TypeCheckVisitor extends
  DepthFirstVisitor {

    static Class currClass;
    static Method currMethod;
    static SymbolTable symbolTable;

    public TypeCheckVisitor(SymbolTable s){
       symbolTable = s;
    }
…
```


TypeCheckVisitor (2)

```
…
// Identifier i;
// Exp e;
public void visit(Assign n) {
  Type t1 =
   symbolTable.getVarType(currMethod,currClass,n.i.toString());
  Type t2 = n.e.accept(new TypeCheckExpVisitor() );

  if (symbolTable.compareTypes(t1,t2)==false){
     System.out.println("Type error in assignment to
   "+n.i.toString());
  }
}
…
```


TypeCheckVisitor (3)

```
…
// Type t;
// Identifier i;
// FormalList fl;
// VarDeclList vl;
// StatementList sl;
// Exp e;
public void visit(MethodDecl n) {
  n.t.accept(this);
  String id = n.i.toString();
  currMethod = currClass.getMethod(id);
  Type retType = currMethod.type();
  for ( int i = 0; i < n.fl.size(); i++ )
       {n.fl.elementAt(i).accept(this); }
  for ( int i = 0; i < n.vl.size(); i++ )
       {n.vl.elementAt(i).accept(this); }
  for ( int i = 0; i < n.sl.size(); i++ )
       {n.sl.elementAt(i).accept(this); }
  if (symbolTable.compareTypes(retType, n.e.accept(new
 TypeCheckExpVisitor()))==false) {
      System.out.println("Wrong return type for method "+ id);
  }
}
```

TypeCheckExpVisitor

```
public class TypeCheckExpVisitor extends TypeDepthFirstVisitor {
…
 // Exp e1,e2;
 public Type visit(Plus n) {
  if (! (n.e1.accept(this) instanceof IntegerType) ) {
    System.out.println("Left side of Plus must be of type integer");
  }

  if (! (n.e2.accept(this) instanceof IntegerType) ) {
    System.out.println("Right side of Plus must be of type integer");
  }

    return new IntegerType();
 }
…
```


Method Calls `e.i(el₁,el₂,…)`

- Lookup method in the **`SymbolTable`** to get parameter list and result type
- Find **`i`** in class **`e`**
- The parameter types in the parameter list for the method must be matched against the actual arguments $\mathbf{el}_1$, $\mathbf{el}_2$, …
- Result type becomes the type of the method call as a whole.

TypeCheckExpVisitor (2)

```
  …
  // Exp e;
  // Identifier i;
  // ExpList el;
  public Type visit(Call n) {
    if (! (n.e.accept(this) instanceof IdentifierType)){
        System.out.println("method "+ n.i.toString()+ "called  on something
that is not a class or Object.");
    }

    String mname = n.i.toString();
    String cname = ((IdentifierType) n.e.accept(this)).s;
    Method calledMethod = TypeCheckVisitor.symbolTable.getMethod(mname,cname);

    for ( int i = 0; i < n.el.size(); i++ ) {
        Type t1 =null;
        Type t2 =null;

        if (calledMethod.getParamAt(i)!=null)
            t1 = calledMethod.getParamAt(i).type();
        t2 = n.el.elementAt(i).accept(this);
        if (!TypeCheckVisitor.symbolTable.compareTypes(t1,t2)){
            System.out.println("Type Error in arguments passed to " +cname+"."
+mname);
        }
    }
    return TypeCheckVisitor.symbolTable.getMethodType(mname,cname);
  }
```


What you should do now…

- Read and digest chapter 5
  - you don't need functional implementation styles or mutiple tables
- Get ready further to develop the MiniJava typechecker - Coursework 3.
- Next Lecture: Stack Frames