joeq Compiler System

Oren Kerem
CS 243

Plan for Today

- Joeq System Overview
- Lifecycle of Analyzed Code
- Source Code Representation
- Writing and Running a Pass
- Assignment: Dataflow Framework

Background on joeq

- A compiler system for analyzing Java code
  - Developed by John Whaley and others
  - Used on a daily basis by the SUIF compiler group
  - An infrastructure for many research projects: 10+ papers rely on joeq implementations
- Visit http://joeq.sourceforge.net for more
- Or read http://www.stanford.edu/~jwhaley/papers/ivme03.pdf

joeq Design Choices

- Most of the system is implemented in pure Java
- Thus, analysis framework and bytecode processors work everywhere
- We treat joeq as a front-end and middle-end
- But it can be used as a VM as well
  - System-specific code is patched in when the joeq system compiles itself or its own runtime
  - These are ordinary C routines
  - Systems supported by full version: Linux and Windows under x86

joeq Components

- Full system is very large: ~100,000 lines of code
- Allocator
- Bootstrapper
- Classfile structure
- Compiler (Quad)
- Garbage Collector
- Quad Interpreters
- Memory Access
- Safe/Unsafe barriers
- Synchronization
- Assembler
- Class Library
- Compiler (Bytecode)
- Debugger
- Bytecode Interpreters
- Linkers
- Reflection support
- Scheduling
- UTF-8 Support

We restrict ourselves to only the compiler and classfile routines, which is closer to 40,000 lines of code

Starting at the Source

Lifecycle of Analyzed Code

- Everything begins as source code
- A very “rich” representation
  - Good for reading
  - Hard to analyze
- Lots of high-level concepts here with (probably) no counterparts in the hardware
  - Virtual function calls
  - Direct use of monitors and condition variables
  - Exceptions
  - Reflection
  - Anonymous classes
  - Threads

Source to Bytecode

- javac or jikes compiles source into a machine-independent bytecode format
- Coarse structure of the program is still maintained
  - Each class is a file
  - Split up into methods and fields
  - The bytecodes themselves are stored as a member attribute in methods that have them
  - Bytecode instructions are themselves high level:
    - invokevirtual
    - monitorenter
    - arraylength

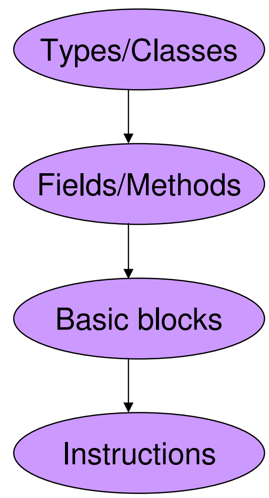
Analysis and Source Code

- No need to bother with source code, since structure is maintained in classfile format
- Moreover, bytecode is indifferent to language changes
- Reading in code:
  1. joeq finds and loads requested files through the classpath
  2. Each source component in the classfile has a corresponding object:
     - jq_Class
     - jq_Method
     - etc.
- Method bodies are transformed from bytecode arrays to more convenient representation

How Source Code is Represented Within joeq

Source Code Representation

- joeq is designed primarily to work with Java
  - Operates at all levels of abstraction
  - Has classes corresponding to each language component
- Relevant packages in joeq
  - joeq.Class package: classes that represent Java source components (classes, fields, methods, etc.)
  - joeq.Compiler.BytecodeAnalysis package: analysis of Java bytecode
  - joeq.Compiler.Quad package: Classes relevant to joeq's internal “quad” format
- Be careful with your imports
  - avoid name conflicts with java.lang.Class and java.lang.Compiler



joeq.Class: Types and Classes

- jq_Type: corresponds to any Java type
- jq_Primitive: static elements representing primitive types
- jq_Array: multidimensional arrays that have a component type, which itself is a jq_Array
- jq_Class: a defined class

joeq.Class: Fields and Methods

- Subclasses of jq_Field and jq_Method
  - Class hierarchy distinguishes between instance and class (static) members, but this detail is generally hidden from higher analyses
- Access to the types hierarchy: declaring types, parameter/return types, etc.
- Names are stored as UTF.Utf8 objects, so convert with toString() to make use of them

Analyzing Bytecode

- The Java Virtual Machine stores program code as *bytecodes* that serve as instructions to a stack machine of sorts
- Raw material for all analysis of Java code
- Preserves vast amounts of source information:
  - De-compilers can reconstruct source almost perfectly, down to variable names and line numbers

Example of Java Bytecode

```
class ExprTest {
    int test(int a){
      int b, c, d, e, f;
      c = a + 10;
      f = a + c;

      if(f > 2){
        f = f - c;
      }

      return f;
    }
}
```

- javac test.java
- javap -c ExprTest

```
int test(int);
  Code:
   0:   iload_1
   1:   bipush  10
   3:   iadd
   4:   istore_3
   5:   iload_1
   6:   iload_3
   7:   iadd
   8:   istore  6
   10:  iload   6
   12:  iconst_2
   13:  if_icmple       22
   16:  iload   6
   18:  iload_3
   19:  isub
   20:  istore  6
   22:  iload   6
   24:  ireturn
```

Bytecode Details

- The implied running model of the Java Virtual Machine is a stack machine
  - Local variables correspond to registers
  - Computation occurs on a stack
- This is hard to analyze!
- Fortunately, the JVM requires that bytecode pass strict type-checking and stack consistency checking
- **Gosling Property:** At each instruction, the types of every element on the stack, and every local variable, are all well defined
- By extension, the stack must have a specific height at each program point

Converting Bytecodes to Quads

- joeq converts bytecodes to four-address code, called "Quads"
- The highly abstract bytecode instructions have Quad counterparts
- One operator, up to four operands

  OPERATOR OP1 OP2 OP3 OP4
- Approximately 100 operators and 15 varieties of operands
- Details on the quads and relevant methods are on the course website's joeq documentation:
  - http://suif.stanford.edu/~courses/cs243/joeq/index.html

Operators

- Types of operators
  - Primitive operations: Moves, Adds, Bitwise AND, etc.
  - Memory access: Getfields and Getstatic
  - Control flow: Compares and conditional jumps, JSRs
  - Method invocation: OO and traditional
- Operators have suffixes indicating return type:
  - ADD_I adds two integers
  - L, F, D, A, and V refer to longs, floats, doubles, references, and voids respectively
  - Operators may have _DYNLINK (or %) appended, which means that a new class may need to be loaded

Operands

- Operands are split into 15 types
  - The ConstOperand classes (I, F, A, etc.) indicate constant values of the relevant type
  - RegisterOperands name pseudo-registers
  - MethodOperands and ParamListOperands are used to identify method targets
  - TypeOperands are passed to type-checking operators, or to "new" operators
  - TargetOperands indicate the target of a branch

Converting a Method to Quads

```
BB0 (ENTRY) (in: <none>, out: BB2)
BB2     (in: BB0 (ENTRY), out: BB3, BB4)
1   ADD_I      T0 int, R1 int, IConst: 10
2   MOVE_I     R3 int, T0 int
3   ADD_I      T0 int, R1 int, R3 int
4   MOVE_I     R6 int, T0 int
5   IFCMP_I    R6 int, IConst: 2, LE, BB4
BB3     (in: BB2, out: BB4)
6   SUB_I      T0 int, R6 int, R3 int
7   MOVE_I     R6 int, T0 int
BB4     (in: BB2, BB3, out: BB1 (EXIT))
8   RETURN_I   R6 int
BB1     (EXIT) (in: BB4, out: <none>)

Exception handlers: []
Register factory: Local: (I=7, F=7, L=7, D=7, A=7)
                  Stack: (I=2, F=2, L=2, D=2, A=2)
```

Control Flow and CFGs

- joeq.Compiler.Quad.ControlFlowGraph encapsulates most of the information we need
  - Don't confuse with the ControlFlowGraph in joeq.Compiler.BytecodeAnalysis
- Generated from jq_Methods by the underlying system's machinery

Basic Blocks

- Raw components of Control Flow Graphs
- Linked to predecessors and successors
- Contain a list of Quads
- And information about exception handlers
  - Which ones protect this basic block
  - Which blocks this one protects
- Exceptions violate traditional BB semantics
  - An exception can jump out of the middle of a basic block
  - We will ignore this subtlety

Safety Checks

- Java's safety checks are *implicit*: instructions may throw exceptions
- Joeq's safety checks are *explicit*: values of arguments are tested by operators such as NullCheck and BoundsCheck
  - Exceptions are thrown if checks fail
- When converting from bytecodes to quads, all necessary checks are automatically inserted

Iterating Over the Quads: QuadIterator

- Dealing with control flow graphs or basic blocks directly is tedious
- Dealing with individual quads tends to miss the forest for the trees
- Simple interface to iterate through all the quads in reverse post-order
- Predecessors and successors are still accessible

```
jq_Method m = ...
ControlFlowGraph cfg = CodeCache.getCode(m);
QuadIterator iter = new QuadIterator(cfg)
while(iter.hasNext()) {
        Quad quad = (Quad)iter.next();
        if(quad.getOperator() instanceof Operator.Invoke) {
            processCall(cfg.getMethod(), quad);
        }
}
```

Developing a joeq Compiler Pass

Writing and Running a Pass

- Passes themselves are written in Java, implementing various joeq interfaces
- Passes are invoked through library routines in the joeq.Main.Helper class

The joeq.Main.Helper Class

- joeq.Main.Helper provides a clean interface to the complexities of the joeq system
- load(String) takes the name of a class provides the corresponding jq_Class
- runPass(target, pass) lets you apply any pass to a target that's at least that big

Visitors in joeq

- joeq uses of the visitor design pattern
- The visitor for a level of the code hierarchy has methods visitFoo(code object) for each type of object in that level
- For some cases, types may overlap (e.g., visitStore and visitQuad) – the methods will be called from least-general to most-general (i.e., visitStore before visitQuad)
- Visitor interfaces with more than one method have internal abstract classes called "EmptyVisitor" to simplify implementation

Visitors: Some Examples

```
public class QuadCounter extends QuadVisitor.EmptyVisitor
    {
    public int count = 0;
    public void visitQuad(Quad q){
        count++;
    }
}

public class LoadStoreCounter extends
    QuadVisitor.EmptyVisitor {
    public int loadCount = 0, storeCount = 0;
    public void visitLoad(Quad q){ loadCount++;}
    public void visitStore(Quad q){ storeCount++;}
}
```

Running a Pass

```
public class RunQuadCounter {
    public static void main(String[] args){
        jq_Class[] c = new jq_Class[args.length];
        for(int i = 0; i < args.length; i++){
                c[i] = Helper.load(args[i]);
        }
        QuadCounter qc = new QuadCounter();
        for(int i = 0; i < args.length; i++){
                qc.count = 0;
                Helper.runPass(c[i], qc);
                System.out.println(
                        c[i].getName() + " has " +
                        qc.count + " Quads.");
        }
    }
}
```

Summary

- We're using the joeq compiler system
- Review of Java VM's code hierarchy
- Review of joeq's code hierarchy
- QuadIterators
- joeq.Main.Helper
- Visitor pattern
- Defining and running passes

Programming Assignment 1

- Implement a **basic data flow framework**
- We provide the interfaces for your framework
- Write the iterative Solver algorithm for any analysis matching these interfaces
- Phrase Reaching Definitions in terms that any Solver can understand

Programming Assignment 1

- Sample analyses are available in /usr/class/cs243/dataflow
- Flow.java contains the interfaces and the main() method
- ConstantProp.java contains classes that define a limited constant propagation algorithm
- Liveness.java contains classes that define a liveness detection algorithm

Programming Environment

- joeq.jar is provided
- We recommend you develop on Eclipse
- Your output must match ours on the publicly accessible Stanford Linux clusters
- Sample test case and matching output is provided
- The makefile will simplify your life

The LivePC Engine

- Import a complete Eclipse development platform, including joeq (~800MB)
- Go to www.moka5.com and download the LivePC Engine
- After installation, fetch the Java Program Analysis Toolset library
- (Your TAs are not proficient LivePC users)