Joeq Background

• Compiler backend for analyzing and optimizing Java bytecode
  – Developed by John Whaley and others
  – Implemented in Java
  – Research project infrastructure: 10+ papers rely on Joeq implementations

• Also see: http://joeq.sourceforge.net
Java Toolchain

- javac.exe is the frontend
- java.exe is the backend
Joeq Toolchain

- Analyze bytecode, not Java source
- Joeq can also act as a VM
  - Not using this functionality in this class
Rest of this Lecture

• Java: bytecode overview
• Joeq: quads overview
  – quads: instruction set used for Joeq analyses
• Joeq: translating bytecode -> quads
• Joeq: analyzing quads
• Homework 2
Java code

• Everything begins as Java source code
• A very “rich” representation
  – Good for reading and writing
  – Hard to analyze
• Lots of high-level concepts with no hardware counterparts
  – classes, virtual function calls, exceptions, threads, locks, structured control flow, etc.
Java bytecode (1)

• javac compiles source into machine-independent bytecode format (.class files)
• Coarse structure of the program is still maintained
  – Each class is a file
  – Sections for each method and field
• Each method has a bytecode sequence for its implementation
• Bytecode instructions are still high level
  – Know about virtual methods, locks, etc.
Java bytecode (2)

- Each bytecode instruction uses one byte
  - Instructions may have additional operands, stored immediately after the instruction
- Variables stored in abstract registers
  - r0 is ‘this’, r1 ... are params followed by locals
- Stack machine model
  - All intermediate values stored on a stack
Stack Machine Model

- Instructions push/pop values onto a stack
- \( x = y + 10: \)
  - push \( y; \)
  - push 10;
  - add;
  - pop \( x; \)
Bytecode instructions

• `iload_1`
  – Load first param/local and push it on the stack

• `bipush <n>`
  – Push byte constant ‘n’ onto the stack

• `iadd`
  – Add the top two values on the stack, and push the result back onto the stack

• `istore_2`
  – Pop the stack and store its value into the second param/local
Example (1)

class ExprTest {
    int test(int a) {
        int b, c;
        b = a + 10;
        c = a + b;
        if (c > 10) {
            c = 10;
        }
        return c;
    }
}

> javac ExprTest.java
> javap -c ExprTest

class ExprTest
    extends java.lang.Object {

ExprTest();
    Code:
    (omitted)

int test(int);
    Code:
    (omitted)
Example (2)

ExprTest2();

Code:

0:   aload_0
1:   invokespecial  #1;
4:   return

• **aload_0**
  – Load address ‘this’ and push it onto the stack

• **invokespecial**
  – Invokes base class methods (among other uses)
  – #1 is constructor
Example (3)

```java
int test(int); // Example function declaration

Code:
0: iload_1
1: bipush 10
3: iadd
4: istore_2
5: iload_1
6: iload_2
7: iadd
8: istore_3
9: iload_3
10: bipush 10
12: if_icmple 18
15: bipush 10
17: istore_3
18: iload_3
19: ireturn

class ExprTest {
    int test(int a) {
        int b, c;
        b = a + 10;
        c = a + b;
        if (c > 10) {
            c = 10;
        }
        return c;
    }
}
```
Joeq Class Representation

• Joeq has classes for each component of the Java .class file
  – jq_Type: a Java type
  – jq_Class: a Java class
  – jq_Field, jq_Method, ...
  – including the bytecodes

• Instead of analyzing bytecodes directly, we will use quads
Joeq Quads

• Joeq translates bytecodes to four-address instructions, called “quads“

• Quads: one operator, up to four operands
  – OPERATOR op1 op2 op3 op4
  – joeq.Compiler.Quad.Quad
  – joeq.Compiler.Quad.Operand

• There is no stack
  – All temporary data stored in registers
Joeq Operands

• **RegisterOperand**
  – Abstract registers represent parameters, local variables, and temporary values

• **ConstantOperand**
  – int/float/string/etc. constants

• **TargetOperand**
  – Basic block target of a branch instruction

• **MethodOperand, ParamListOperands**
  – Target and arguments to a method call

• **FieldOperand, TypeOperand, ...**
Joeq Operators (1)

• Operator.Move
  – Move a constant or register into another register

• Operator.Unary, Operator.Binary
  – Operations on constants/registers, storing result in another register

• Operator.Invoke
  – Method call with a result register, method operand and parameter list operand

Joeq Operators (2)

• Runtime checks are explicit quads
  – Not implicit as in bytecodes
  – Can throw exceptions (as can method calls)
• Operator_NullCheck
  – Check if an operand is NULL
• Operator_BoundsCheck
  – Check if an array access is out of bounds
• Operator_CheckCast, Operator_StoreCheck, …
Joeq CFGs (1)

• For each method, Joeq generates a control flow graph over basic blocks
  – joeq.Compiler.Quad.ControlFlowGraph
  – joeq.Compiler.Quad.BasicBlock

• CFGs are graphs of basic blocks with an entry and exit block

• Blocks are lists of quads, and know their successors and predecessors in the CFG
Joeq CFGs (2)

• Exception control flow is not explicit in Joeq basic blocks
  – An exception can jump out of the middle of a basic block

• You do not need to consider exceptions when writing Joeq analyses for this class
Example (1)

class ExprTest {
    int test(int a) {
        int b, c;
        b = a + 10;
        c = a + b;
        if (c > 10) {
            c = 10;
        }
        return c;
    }
}

> javac ExprTest.java
> java PrintQuads ExprTest

Class: ExprTest

Control flow graph for
ExprTest.<init> ()V:
    (omitted)

Control flow graph for
ExprTest.test (I)I:
    (omitted)
Example (2)

Control flow graph for ExprTest.<init> ()V:

BB0 (ENTRY) (in: <none>, out: BB2)

BB2 (in: BB0 (ENTRY), out: BB1 (EXIT))

2   NULL_CHECK  T-1 <g>,  R0 ExprTest
1   INVOKEESPECIAL_V% java.lang.Object.<init>()V,
    (R0 ExprTest)
3   RETURN_V

BB1 (EXIT)  (in: BB2, out: <none>)

Original bytecode:

```java
ExprTest2();
Code:
  0:   aload_0
  1:   invokespecial #1;
  4:   return
```
Example (3)

Control flow graph for ExprTest.test:
BB0 (ENTRY) (in: <none>, out: BB2)

BB2 (in: BB0 (ENTRY), out: BB3, BB4)
1   ADD_I T2 int, R1 int, IConst: 10
2   MOVE_I R3 int, T2 int
3   ADD_I T2 int, R1 int, R3 int
4   MOVE_I R4 int, T2 int
5   IFCMP_I R4 int, IConst: 10, LE, BB4

BB3 (in: BB2, out: BB4)
6   MOVE_I R4 int, IConst: 10

BB4 (in: BB2, BB3, out: BB1 (EXIT))
7   RETURN_I R4 int

BB1 (EXIT) (in: BB4, out: <none>)

class ExprTest {
    int test(int a) {
        int b, c;
        b = a + 10;
        c = a + b;
        if (c > 10) {
            c = 10;
        }
        return c;
    }
}

Example (4)

Control flow graph for ExprTest.test:
BB0 (ENTRY)  (in: <none>, out: BB2)

BB2     (in: BB0 (ENTRY), out: BB3, BB4)
1   ADD_I    T2 int, R1 int, IConst: 10
2   MOVE_I   R3 int, T2 int
3   ADD_I    T2 int, R1 int, R3 int
4   MOVE_I   R4 int, T2 int
5   IFCMP_I  R4 int, IConst: 10, LE, BB4

BB3     (in: BB2, out: BB4)
6   MOVE_I   R4 int, IConst: 10

BB4     (in: BB2, BB3, out: BB1 (EXIT))
7   RETURN_I   R4 int

BB1 (EXIT)   (in: BB4, out: <none>)

int test(int);
Code:
  0:   iload_1
  1:   bipush  10
  3:   iadd
  4:   istore_2
  5:   iload_1
  6:   iload_2
  7:   iadd
  8:   istore_3
  9:   iload_3
 10:  bipush  10
 12:  if_icmple 18
 15:  bipush  10
 17:  istore_3
 18:  iload_3
 19:  ireturn
Writing Analyses with Joeq

• Most analyses can be written with visitors
  – Traverse all the loaded CFGs, or all the quads in those CFGs
• ControlFlowGraphVisitor: interface for an analysis which makes a pass over the CFGs
• QuadIterator: iterate over every quad in a CFG
• QuadVisitor: interface for an analysis which makes a single pass over the quads in a CFG
joeq.Main.Helper class

• Helper class provides a clean interface to the complexities of Joeq
  – packages up reading of .class files, generating type and bytecode data, and translating to quads
• load(String) takes the name of a class, returns the corresponding jq_Class
• runPass(val, visitor) runs a CFG or quad visitor over val (which is a class, CFG, quad, etc.)
public static class QuadCounter
    extends QuadVisitor.EmptyVisitor
{
    public int count = 0;
    public void visitQuad(Quad q) {
        count++;
    }
}
class CountQuads
{
    public static class QuadCounter ...

    public static void main(String[] args)
    {
        jq_Class[] classes = new jq_Class[args.length];
        for (int i=0; i < classes.length; i++)
            classes[i] = (jq_Class)Helper.load(args[i]);

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

        for (int i=0; i < classes.length; i++) {
            System.out.println("Class: " + classes[i].getName());
            QuadCounter qc = new QuadCounter();
            Helper.runPass(classes[i], qc);
            System.out.println(classes[i].getName() + " has " +
                                qc.count + " quads");
        }
    }
}
Homework 2

• Implement a data flow framework with Joeq
  – We provide the interfaces for your framework
  – We provide two dataflow analyses that must work with your framework
• Write a Reaching Definitions analysis which works using your framework
• Due next Friday (1/30/2009)