JoeQ Framework
CS243, Winter 2016

Overview

● Java Intermediate representation
  - Bytecode
● JoeQ Framework
  - Quads: Instruction set used in JoeQ
  - JoeQ constructs
  - Writing analysis in JoeQ
● HW 2

Typical Compiler Infrastructure

Java compiler (javac)

Java

Front-end
- Parsing

Middle-end
  • Machine-Independent optimizations

Back-end
  • Machine-Dependent Optimizations

Front-end
- Parsing

Middle-end
  • Machine-Independent optimizations

Back-end
  • Machine-Dependent Optimizations

JVM (java)
Java Source Code

Input to the Java Front-end
- A very “rich” representation
  - Good for reading and writing (by human)
  - Hard to analyze (by computer)
- Many high-level concepts with no hardware counterparts
  - classes, generics, virtual function calls, exceptions, structured control flow, locks, etc.

Java Bytecode

- Machine-independent intermediate representation (.class files)
- Coarse program structure is still maintained
  - One file per class
  - A section per method or field
- Each method has a bytecode sequence for its implementation
- Still high level
  - Virtual methods, locks, etc.

Bytecode representation

- 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 parameters followed by locals
- Stack machine model
  - All intermediate values stored on a stack

What is a stack machine model?

- Each instruction pushes or pops values onto a stack.
- Eg :
  \[ x = y + 10 \]
  - push y
  - push 10
  - add
  - pop x
What is a stack machine model?

- Each instruction pushes or pops values onto a stack.
- Eg:
  
  \[ x = y + 10 \]
  
  → push \( y \)
  → push 10
  → add
  → pop \( x \)

*add - Implicitly pops the top 2 operands and pushes result
Bytecode Instructions

- Each instruction is prefixed by the types of operands.
- iload_1
  - Load the first parameter or local variable and push it on the stack
- bipush <n>
  - Push byte constant "n" onto the stack as an integer value
- 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

An example

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 ExprTest.java
> javap -c ExprTest

class ExprTest extends java.lang.Object {
    ExprTest() {
    }
}

ExprTest.ExprTest()

ExprTest():

Code:
0: aload_0
   // load address 'this' and push it onto the stack
1: invokespecial #1
   // invokes base class methods. #1 is constructor
2: return

ExprTest.test(int a)

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;
    }
}

int test(int)

Code:
0: iload_1
1: bipush 10
2: iadd
3: istore_3
4: iload_1
5: iload_3
6: iadd
7: istore 6
8: iload 6
9: iload 6
10: iload 6
11: iload_3
12: iadd
13: iload 6
14: iload 6
15: iload 6
16: iload 6
17: iload 6
18: iload 6
19: iload 6
20: iload 6
21: iload 6
22: iload 6
23: iload 6
24: ireturn
### JoeQ

- Compiler framework for analyzing and optimizing Java bytecode
  - Developed by John Whaley and others
  - Implemented in Java
  - Research project infrastructure: 10+ papers rely on Joeq
- Also see: [http://joeq.sourceforge.net](http://joeq.sourceforge.net)
- Etymology
  - “jyo-kyu-” like the name “Joe” and the letter “Q”

### JoeQ Intermediate Representation

- High-level representation: Joeq has classes for each component of the Java .class file
  - jq_Type, jq_Class, jq_Field, jq_Method, …
- We use a lower-level representation called “Quad”.
- Joeq translates bytecodes into quads.

### Quads

- Stanford version of the TAC (Three address code)
- One operator and up to four operands: four-address instructions
- `joeq.Compiler.Quad.Quad`
- Register machine model, not stack machine model
  - All temporary data stored in registers
  - Closer to (RISC) machine instructions than a stack model: Joeq is lower-level IR than Java Bytecode

### Register machine model

- This mode assumes an unbounded number of pseudo registers.
- Pseudo registers hold local variables of a method, as well as temporary variables
- All data must first be loaded into pseudo registers before they can be operated on.
- More conducive to program optimization than the stack architecture.
### JoeQ Operands
(joeq.Compiler.Quad.Operands)

- **Register Operand**
  - Abstract registers representing parameters, local variables, and temporal variables
- **Constant Operand**
  - int/float/string/etc. Constants
- **Target Operand**
  - Basic block target of a branch instruction
- **Method Operand, ParamListOperand**
  - Target and arguments to a method call
- **Field Operand, TypeOperand, ...**

### JoeQ Operators
(joeq.Compiler.Quad.Operator)

- **Operator.Move**
- **Operator.Unary, Operator.Binary**
- **Operator.Invoke**
- Have suffixes indicating return type
  - ADD_I adds two integers
  - L, F, D, A, and V refer to long, float, double, reference, and void

### JoeQ Runtime Checking Operators

- Runtime checks are explicit quads
  - Not implicit as in bytecodes: Joeq is lower-level IR than Java Bytecode
- **Operator.NullCheck**
- **Operator.BoundsCheck**
- **Operator.CheckCast, Operator.StoreCheck, ...**

### JoeQ CFGs
(joeq.Compiler.Quad.ControlFlowGraph)

- Graphs of basic blocks with entry and exit
  - Entry and exit basic blocks always exist.
  - They are empty.
JoeQ Basic Blocks
(joeq.Compiler.Quad.BasicBlock)

- Lists of quads
- Provide access to successors and predecessors
- 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 in this class

ExprTest

```java
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 ExprTest.java`
`java PrintQuads ExprTest`

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

Control flow graph for ExprTest.test (I)I:
...

ExprTest.ExprTest()

Code:
0: aload_0
// load address 'this' and push it onto the stack
1: invokespecial #1
// invokes base class methods. #1 is constructor
2: return

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

BB2 (in: BB0 (ENTRY), out: BB1 (EXIT))
1 NULL_CHECK T-1 <g>, R0 ExprTest
2 INVOKEVIRTUAL java.lang.Object.<init>()V, (R0 ExprTest)
3 RETURN_V

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

ExprTest.test(int a)

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;
    }
    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: 2, LE, BB4
        6 SUB_I T2 int, R4 int, R3 int
        7 MOVE_I R4 int, T2 int
        8 RETURN_I R4 int
    BB3 (in: BB2, out: BB4)
    BB4 (in: BB2, BB3, out: BB1 (EXIT))
    BB1 (EXIT) (in: BB4, out: <none>)
    ```

Quads

```
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: 2, LE, BB4
6 SUB_I T2 int, R4 int, R3 int
7 MOVE_I R4 int, T2 int
8 RETURN_I R4 int
BB1 (EXIT) (in: BB4, out: <none>)
```

Writing analysis with JoeQ

- Often 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
- QuadVisitor: interface for an analysis which makes a single pass over the quads in a CFG

Visitor Design Pattern

- Add an operation to existing objects without modifying the structure of objects
- Operations: variable, objects: fixed
- Control flow graph analysis 1, 2, ...
  - Add methods analysis1, analysis2, ... to ControlFlowGraph class vs.
  - Analysis1CfgVisitor, Analysis2CfgVisitor, ...
- In compiler, representation is (almost) fixed.
- Adding analysis/ transformation should be flexible.
Joeq.Main.Helper class

- A clean interface to the complexities of Joeq (Façade design pattern)
- runPass(CFG or quad, visitor) runs a ControlFlowGraphVisitor/QuadVisitor over ControlFlowGraphs/Quads.

QuadIterator

- An alternative to visitors
- Simple interface to iterate through all the quads in a reverse post-order
- Extends java.util.Iterator<Quad>

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

Helper Class Usage Example: QuadCounter

```
class CountQuads {
    public static void main(String[] args) {
        jq_Class[] classes = new jq_Class[1];
        for (String className : args) {
            jq_Class c = (jq_Class)Helper.load(className);
            System.out.println("Class: " + className);
            QuadCounter qc = new QuadCounter();
            Helper.runPass(c, qc);
            System.out.println(className + " has " + qc.count + " quads");
        }
    }
}
```

Java Beginners

- Java collections library
  - List, Set, Map, ...
- Java Generics
- Inner Class
- ...
- Make yourself familiar with these concepts
HW2 : Dataflow Framework

- We provide
  - Solver interface: Flow.Solver
  - Analysis interface: Flow.Analysis
  - Two analysis that extend Flow.Analysis: ConstantProp and Liveness

- Goal is to complete
  - Skeleton MySolver that extends Flow.Solver
    - Should work with ConstantProp and Liveness
  - Skeleton ReachingDefs that extends Flow.Analysis
  - Similarly Faintness analysis

More Details

- joes.jar is provided
  - Unjar this if you want to look at Joeq source code.
- Using Eclipse may make your life easier.
- Working with groups of two is encouraged.
- Output must match ours on Stanford Linux clusters such as myth.

- Get started early. It make take a long time to understand the Joeq framework, although you need to know only small part of it.
- Post questions on Piazza, so that we can answer them the session on Friday.