Data Flow Analysis using JoeQ and HW2 Introduction

Lovish Chopra
Outline

- Background
  - JoeQ
  - Java
- Code Structure of JoeQ
  - The Quad Intermediate Representation of JoeQ
  - Operator & Operand
  - QuadIterator
  - Visitor methods
- HW2 and tips
Background on JoeQ

- A compiler system for analyzing Java code
- Developed by John Whaley and others
- An infrastructure for many research projects: 10+ papers rely on JoeQ implementations
- The joeq system is extremely large (~100,000 lines of code). We will restrict ourselves to a small, standalone subset of the complete system.
Background on Java

- Similar syntax to C/C++
- Typical routine of running a Java program
  - Compile source code (*.java) -> machine-independent Java bytecode (*.class) via javac
  - Run Java bytecode via java in a JVM (Java Virtual Machine)
- The JDK (Java Development Kit) provides you with the programs javac, java, etc.
Java Bytecode

- The coarse structure of the program is preserved!
  - Each class is a file
  - Split up into methods and fields
  - Bytecoded instructions are themselves high level:
    - invokevirtual
    - monitorenter
    - arraylength
- Stack-oriented: operands are pushed on the operand stack and arithmetic operations are applied to the top variables on the stack
- javac does not perform any optimization for the bytecode
Java Bytecode

- Use javap to disassemble java bytecode

```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);
    }
}
class ExprTest extends java.lang.Object {
    ExprTest();
    int test(int);
}
```
Java Bytecode: A deeper look into the test method

- **signature**
  - test accepts an integer and returns an integer

- A frame is created for each invocation. Location 0 holds the `this` pointer; the parameter and local variables a,b,c,d,e,f are numbered 1 to 6, respectively.

- Instructions are labeled by their position in the array of bytecode representing the procedure.
Java Bytecode: A deeper look into the test method

- Instructions such as load are prefixed by the result type: a, b, c, d, f, i, j, s, and z
  - They represent reference, byte, character, double, float, integer, long, short, boolean, respectively.

- An instruction's parameter is either represented as a suffix or an extra operand.
  - iload_1 and iload 6 load the 1st and 6th variables from the frame onto the stack, respectively.
  - The difference is just an optimization in encoding; the former, which is more common, is encoded in one byte and the latter is encoded in two.
The Quad Intermediate Representation (IR)

- JoeQ translates bytecode into its own intermediate representation – Quads
- Named quad the instructions each accept up to three input operands and one result variable
- Assumes you have an unbounded number of pseudo registers
The Quad Intermediate Representation (IR)

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);
    }
}

<table>
<thead>
<tr>
<th>Quad Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB0 (ENTRY) (in: &lt;none&gt;, out: BB2)</td>
<td></td>
</tr>
<tr>
<td>BB2 (in: BB0 (ENTRY), out: BB3, BB4)</td>
<td></td>
</tr>
<tr>
<td>1 ADD_I</td>
<td>T2 int, R1 int, IConst: 10</td>
</tr>
<tr>
<td>2 MOVE_I</td>
<td>R3 int, T2 int</td>
</tr>
<tr>
<td>3 ADD_I</td>
<td>T2 int, R1 int, R3 int</td>
</tr>
<tr>
<td>4 MOVE_I</td>
<td>R4 int, T2 int</td>
</tr>
<tr>
<td>5 IFCMP_I</td>
<td>R4 int, IConst: 2, LE, BB4</td>
</tr>
<tr>
<td>BB3 (in: BB2, out: BB4)</td>
<td></td>
</tr>
<tr>
<td>6 SUB_I</td>
<td>T2 int, R4 int, R3 int</td>
</tr>
<tr>
<td>7 MOVE_I</td>
<td>R4 int, T2 int</td>
</tr>
<tr>
<td>BB4 (in: BB2, BB3, out: BB1 (EXIT))</td>
<td></td>
</tr>
<tr>
<td>8 RETURN_I</td>
<td>R4 int</td>
</tr>
<tr>
<td>BB1 (EXIT) (in: BB4, out: &lt;none&gt;)</td>
<td></td>
</tr>
</tbody>
</table>
Breaking down the Quads!

What do we need to know about a quad?

Information of the quad itself:

- Operator: the “type” of the quad
- Operand: could be register, constant, memory, etc.

Placing the quad in the context of CFG:

- The ID of the quad in CFG
- Relationship with other quads
Operator (joeq.Compiler.Quad.Operator)

- Operators are hierarchically ordered
  - For example, there are different operations to write to a field in a class, depending on the type of the field
  - `PUTFIELD_{i,f,l,d,a,b,c,s,z}` operators represent array load instructions for the types integer, floating-point, long, double, reference, byte, character, short, and boolean
  - All the various `PUTFIELD` variants inherit from the `Putfield` class, which inherits from `Operator`
- Each operator class has a set of methods, shared or specific to its type
Operator (joeq.Compiler.Quad.Operator)

Some shared methods:

- `getThrownExceptions()`: gets the exception handlers for the operator.
- `getDefinedRegisters(q)`: gets all the registers defined by quad q.
- `getUsedRegisters(q)`: gets all the registers used by quad q.
- `getReg1(q), getReg2(q), getReg3(q)`: gets the named register from quad q.
Operator (joeq.Compiler.Quad.Operator)

- **Moves**
- **Arithmetic and conversion ops** (UnaryOperator, BinaryOperator)
- **Memory ops** (GETFIELD, PUTFIELD, ALOAD, ASTORE)
- **Function calls** (INVOKEVIRTUAL, INVOKEESPECIAL, INVOKEINTERFACE)
- **Control flow** (cond jumps, table switches)
- **Special Java operations** (MONITORENTRY/EXIT, INSTANCEOF, etc.)
- **A few weird ones** (PEEK, POKE)

---

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Subclasses</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>Move between pseudo registers</td>
<td>MOVE_{I,F,L,D,A}</td>
<td>getMoveOp, getDest, getSrc</td>
</tr>
<tr>
<td>Binary</td>
<td>Binary operation, with two sources and one destination</td>
<td>ADD_{I,L,F,D}, SUB_{I,L,F,D}, MUL_{I,L,F,D}, DIV_{I,L,F,D}, REM_{I,L,F,D}, AND_{I,L}, OR_{I,L}, XOR_{I,L}, SHL_{I,L}, SHR_{I,L}, USHR_{I,L}, CMP_{I,L,F,D}</td>
<td>getDest, getSrc1, getSrc2</td>
</tr>
<tr>
<td>Unary</td>
<td>Unary operation, with one source and one destination</td>
<td>INT_2LONG, INT_2FLOAT, INT_2DOUBLE, LONG_2INT, LONG_2FLOAT, LONG_2DOUBLE, FLOAT_2INT, DOUBLE_2LONG, DOUBLE_2FLOAT, INT_2BYTE, INT_2CHAR, INT_2SHORT, OBJECT_2INT, _2OBJECT, FLOAT_2INTBITS, INTBITS_2FLOAT, DOUBLE_2FLOAT, DOUBLE_2LONGBITS, LONGBITS_2DOUBLE</td>
<td>getDest, getSrc</td>
</tr>
</tbody>
</table>
Operand (joeq.Compiler.Quad.Operand)

Operand type depends on the operator

- RegisterOperand (most common in the assignment)
- ConstOperand (may come with prefix A, D, F, I, L, P)
- ...

Interface Operand

All Known Subinterfaces:
Operand.Const4Operand, Operand.Const8Operand, Operand.ConstOperand

All Known Implementing Classes:
Operand.AConstOperand, Operand.BasicBlockTableOperand, Operand.ConditionOperand, Operand.DConstOperand,
Operand.FConstOperand, Operand.FieldOperand, Operand.IConstOperand, Operand.IntValueTableOperand, Operand.LConstOperand,
Operand.MethodOperand, Operand.ParamListOperand, Operand.PConstOperand, Operand.RegisterOperand, Operand.TargetOperand,
Operand.TypeOperand, Operand.UnnecessaryGuardOperand
Examples – Get All Registers Defined in a Quad

- q is a quad
- Need to use method `getRegister()` to get the register object (joeq.Compiler.Quad.RegisterFactory.Register)
- What does it mean if we change `getDefinedRegisters()` to `getUsedRegisters()`?

```java
for (RegisterOperand reg: q.getDefinedRegisters()) {
    System.out.println(reg.getRegister().toString());
}
```
Examples – Get All Registers Defined in a Quad

- Register T2 is the only defined register in the quad
- Register R1 is the only used register in the quad

ADD_I T2 int, R1 int, IConst: 10
Examples – Get All Registers Defined in a Quad

- No registers defined or used

RETURN_V
Examples: Implement a Function that Conditions on Operator Type

- Code snippet from starter code
- Ignore all the `val.setXXX` stuff, the code compares the operator to an instance of `NEG_I` (singleton design pattern, INSTANCE holds the object of the operator, easy for equality check)

```java
public void visitUnary (Quad q) {
    Operand op = Operator.Unary.getSrc(q);
    String key = Operator.Unary.getDest(q).getRegister().toString();
    Operator opr = q.getOperator();

    if (opr == Operator.Unary.NEG_I._INSTANCE) {
        if (isUndef(op)) {
            val.setUndef(key);
        } else if (isConst(op)) {
            val.setConst(key, -getConst(op));
        } else {
            val.setNAC(key);
        }
    } else {
        val.setNAC(key);
    }
}
```
Examples: Implement a Function that Conditions on Operator Type

- *visitUnary* is a method of the interface *QuadVisitor*
- Our implementation overrides the empty implementation of *EmptyVisitor*, which implements the interface
- Implicit to our program, *visitUnary* will be called in

```java
public void processQuad(Quad q) {
    transferfn.val.copy(in[q.getID()]);
    Helper.runPass(q, transferfn);
    out[q.getID()].copy(transferfn.val);
}
```
Relationship Between Quads – BasicBlock and Predecessors/Successors

- Quads are organized into BasicBlocks
- BasicBlocks are single-entry, meaning that control flow can only enter at the start of a basic block
- So we can iterate over all quads/basic blocks
- For the programming assignment, you can ignore the difference of basic blocks and quad, and only iterate over all quads using QuadIterator
QuadIterator

- A container for successors/predecessors in order to update the value of IN/OUT
- A container for all quads in the program we analyze
- Implemented in JoeQ using `java.util.Iterator`
Useful Methods of QuadIterator

- `next()/previous()`
- `hasNext()/hasPrevious()`
- `successors()`
- `predecessors()`

Note: we do not care about the order of the quads; `next()/hasNext()` are used for traversing the QuadIterator, and `successors()/predecessors()` actually store the edges of the CFG
Walking Through a Flow Graph

- Possible order of the iterator of whole flow graph: [f1, f2, f3]
- Successors of f1: [f2, f3]
- Successors of f2: [null] (means boundary)
Examples of QuadIterator

- A program that loops over all quads in a CFG and stores all register names, used or defined

```java
qit = new QuadIterator(cfg);
while (qit.hasNext()) {
    Quad q = qit.next();
    for (RegisterOperand def : q.getDefinedRegisters()) {
        s.add(def.getRegister().toString());
    }
    for (RegisterOperand use : q.getUsedRegisters()) {
        s.add(use.getRegister().toString());
    }
}
```
Examples of QuadIterator

- A program that prints out all the successors of the current quad iterator

```java
Iterator<Quad> successorIterator = quadIterator.successors();
while (successorIterator.hasNext()) {
    Quad curSuccessor = successorIterator.next();
    System.out.println(curSuccessor.toString());
}
```
JoeQ Representation for Java Class Components

- Quad is not enough – JoeQ eats a java file and should maintain a hierarchical representation of the file, quads are basic building blocks
- The joeq.Class package
  - Types: \texttt{jq\_Type} and \texttt{jq\_Primitive}
  - Arrays: \texttt{jq\_Array}
  - Classes: \texttt{jq\_Class}
  - Methods: \texttt{jq\_Method}
  - Fields: \texttt{jq\_Field}
- \texttt{Helper.load(<class\_name>)} constructs a \texttt{jq\_Class} object from the java file, then we could run \texttt{Helper.runPass()} on it
- Other \texttt{jq\_Xxx} not in our interest
Putting it Together – Flow Analysis

Take a glimpse at starter code:

- Defines a Analysis interface, which basically consists of all information specific to a type of flow analysis (e.g. meet operator, bottom, top, initialization)
- Several implementations of the Analysis interface
  - Liveness & Constant Propagation is given to you
  - You need to fill in Reaching Definitions & Faintness
- `visitCFG` ideally should call the methods of the Analysis object registered via `registerAnalysis` during the traversal of the CFG (the iterative algorithm!)

```java
public static interface Solver extends ControlFlowGraphVisitor {
    void visitCFG(ControlFlowGraph cfg);
    void registerAnalysis(Analysis a);
}
```
Putting it Together – Flow Analysis

<table>
<thead>
<tr>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
</tr>
<tr>
<td><strong>Direction</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Transfer function</strong></td>
</tr>
<tr>
<td><strong>Meet Operation (\land)</strong></td>
</tr>
<tr>
<td><strong>Boundary Condition</strong></td>
</tr>
<tr>
<td><strong>Initial interior points</strong></td>
</tr>
</tbody>
</table>

```c
void preprocess (ControlFlowGraph cfg);
void postprocess (ControlFlowGraph cfg);

/**
 * Is this a forward dataflow analysis?
 */
boolean isForward ();

/**
 * Sets the IN value of a quad
 */
void setIn(Quad q, DataflowObject value);

/**
 * Sets the OUT value of a quad
 */
void setOut(Quad q, DataflowObject value);

/**
 * Sets the entry value
 */
void setEntry(DataflowObject value);

/**
 * Sets the exit value
 */
void setExit(DataflowObject value);

/**
 * Actually performs the transfer operation on the given
 * quad.
 */
void processQuad(Quad q);
```
Putting it Together – Flow Analysis

- The place where `visitCFG` is called lies in

```
Helper.runPass(classes[i], solver);
```

- JoeQ makes heavy use of the visitor design pattern. The second argument fed to `Helper.runPass` should support a `visitXXX` method (depends on the type of the object), and it will be called.
HW2: General Structure

- All the interesting pieces are in `src/flow/`.

- **Flow.java**: main class
  - Also defines the `Flow.Analysis` and `Flow.DataflowObject` interface

- **MySolver.java**: the solver you will be writing for Task 1

- **ConstantProp.java, Liveness.java, ReachingDefs.java, Faintness.java**: individual analyses
  - You will need to fill in ReachingDefs.java (Task 2) and Faintness.java (Task 3)
public static interface DataflowObject {
    void setToTop();
    void setToBottom();
    void meetWith(DataflowObject o);
    void copy(DataflowObject o);
}

public static interface Analysis {
    void preprocess(ControlFlowGraph cfg);
    ... boolean isForward();
    ...
    DataflowObject newTempVar();
    void processQuad(Quad q);
    void setToTop();
    void setToBottom();
}
Task 1: MySolver: algorithms

Inputs:

• The specific analysis to use (as a Flow.Analysis object)
• Control-flow graph of a function to analyze

Main task: Translate the iterative algorithms in the textbook to the language of JoeQ, Flow.Analysis, and DataflowObject. No need to use reverse post-order.

Examples:

• for (each basic block) { ... }
  • Use joeq.Compiler.Quad.QuadIterator
  • OUT[B] = T
  • Use setToTop()
  • OUT[B] = fB(IN[B])
  • analysis.processQuad(q)

1) OUT[ENTRY] = vENTRY;
2) for (each basic block B other than ENTRY) OUT[B] = T;
3) while (changes to any OUT occur)
   4) for (each basic block B other than ENTRY) {
          IN[B] = \bigwedge_P \text{a predecessor of } B \text{ OUT}[P];
          OUT[B] = f_B(IN[B]);
      }

(a) Iterative algorithm for a forward data-flow problem.

1) IN(EXIT] = vEXIT;
2) for (each basic block B other than EXIT) IN[B] = T;
3) while (changes to any IN occur)
   4) for (each basic block B other than EXIT) {
          OUT[B] = \bigwedge_S \text{a successor of } B \text{ IN}[S];
          IN[B] = f_B(OUT[B]);
      }

(b) Iterative algorithm for a backward data-flow problem.

Figure 9.23: Forward and backward versions of the iterative algorithm
Task 2: Implement Reaching Definitions


Much like last week:

• Step 1: Choose a lattice
  • (i.e., implement DataflowObject, newTempVar)
• Step 2: Find the transfer function
  • (i.e., implement processQuad, isForward)
• Step 3: Determine the initial values
  • (i.e., implement preprocess, postprocess)

<table>
<thead>
<tr>
<th>Domain</th>
<th>Reaching Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Sets of definitions</td>
</tr>
<tr>
<td>Transfer function</td>
<td>Forwards</td>
</tr>
<tr>
<td>Boundary</td>
<td>gen_B U (x - kill_B)</td>
</tr>
<tr>
<td>Meet (Λ)</td>
<td>OUT[ENTRY] = Ø</td>
</tr>
<tr>
<td>Equations</td>
<td>OUT[B] = f_B(IN[B])</td>
</tr>
<tr>
<td></td>
<td>IN[B] = T_P,pre(B)OUT[P]</td>
</tr>
<tr>
<td>Initialize</td>
<td>OUT[B] = Ø</td>
</tr>
</tbody>
</table>
Task 3: Implement Faintness

Similar to Task 2, but you don’t have the dataflow analysis table.

• Step 1: Work out what the analysis is on paper
• Step 2: Implement it (like in Task 2)
Useful Hints

• Exit and Entry are not really defined as specific quads. So you would have to find the predecessors of EXIT/the successors of ENTRY at the end of your iterative data flow algorithm to take their meet.
  • **Suggestion:** For each quad, check if any of its predecessors is NULL. In that case, it is a successor of ENTRY. Similarly, if you check for any of its successors being NULL, in that case, it is a predecessor of EXIT.

For forward data flow algorithm, you need to call setExit at the end of all iterations explicitly. Similarly, you need to call setEntry at the end of the backward data flow algorithm explicitly.

• Reaching definitions: could you use preprocess to store a data structure to map defined variables and definitions: it will make it easier to kill conflicting definitions in the transfer function.
IDE Recommendation

- IntelliJ IDEA (free to students)
- VSCode
- Remote SSH over myth/rice cluster works conveniently
Useful Links

- A written tutorial
- The full API doc
- A very ancient slides
- The JoeQ paper
- Java tutorial (be careful with the Java version)