Lab 1 - Dataflow in Joeq

CS243
Joeq

- A collection of compilers, interpreters, and run time routines

- Analyzing and optimizing Java bytecode
  - Machine independent optimizations
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;
    }
}

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: icast _2
13: if _icmple 22
16: iload 6
18: iload 3
19: isub
20: istore 6
22: iload 6
24: ireturn
Joeq Quad

- Converts bytecodes to three-address code
- One operator and up to four operands
  - Operator
    - Operator.Binary
    - Operator.Move
  - Operand
    - Operand.ILConstOperand
    - Operand.RegisterOperand
Joeq Intermediate Representation

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

test (int a) {
    int b, c, d;
    int e, f;
    c = a + 10;
    f = a + c;
    if (f > 2) {
        f = f - c;
    }
    return f;
}

$ bin/parun examples.PrintQuads examples.ExprTest
Representing Code in Joeq

- **Quad**
  - `getDefinedRegisters()`
  - `getUsedRegisters()`

- **BasicBlock**
  - Lists of quads
  - `getPredecessors()`
  - `getSuccessors()`
  - `iterator()`

- **ControlFlowGraph**
  - A method
  - Graphs of basic blocks
  - Use `QuadIterator` to iterate over quads
Example - Liveness

VarSet val;
public void visitQuad(Quad q) {
    for (RegisterOperand def : q.getDefinedRegisters()) {
        val.killVar(def.getRegister().toString());
    }
    for (RegisterOperand use : q.getUsedRegisters()) {
        val.genVar(use.getRegister().toString());
    }
}
Visitor Pattern

- ControlFlowGraphVisitor
  - visitCFG(ControlFlowGraph cfg)

- BasicBlockVisitor
  - visitBasicBlock(BasicBlock bb)

- QuadVisitor
  - visitQuad(Quad obj)
  - visitBinary(Quad obj)
  - ...

You should extend *Visitor.
For Visit...
Examples

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

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

class CountQuads {
    public static void main(String[] args) {
        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(qc.count);
        }
    }
}

runPass(ControlFlowGraph c, BasicBlockVisitor bbv)
runPass(ControlFlowGraph c, QuadVisitor qv)
runPass(BasicBlock b, QuadVisitor qv)
Quads in Reverse Post Order

QuadIterator(ControlFlowGraph cfg, boolean direction)

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

- ConstantProp
- ReachingDefs
- Faintness
- Liveness
- Flow.Solver
- Flow.Analysis
- MySolver
Lab 1 - Requirement 1

Implement MySolver

- Which implements interface Solver
- Represent iterative dataflow algorithm

Iterate through all basic blocks (quads)

- Compute IN of the block using the meet operator
- Compute OUT of the block using IN and transfer function
- Repeat until converge
Lab 1 - Requirement 2

Implement ReachingDefs and Faintness
  - Which implements Analysis

Specify properties
  - Direction
  - Lattice value
  - Boundary condition
  - Meet operator
  - Transfer function
  - ...

See Flow.Liveness for more details
Lab 1 - Requirement 3

Implement `TestFaintness`
  - Implement each test case in a method

Run `make` to produce bytecode for `TestFaintness`
Method: test
Initialization completed.
entry: [R1]
1 in: [R1]
1 out: [R1, T2]
2 in: [R1, T2]
2 out: [R1, R3]
3 in: [R1, R3]
3 out: [R3, T2]
4 in: [R3, T2]
4 out: [R3, R4]
5 in: [R3, R4]
5 out: [R3, R4]
6 in: [R3, R4]
6 out: [T2]
7 in: [T2]
7 out: [R4]
8 in: [R4]
8 out: []
exit: []

Liveness Output

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
BB3 (in: BB2, out: BB4)
6 SUB_I T2 int, R4 int, R3 int
7 MOVE_I R4 int, T2 int
BB4 (in: BB2, BB3, out: BB1 (EXIT))
8 RETURN_I R4 int
BB1 (EXIT) (in: BB4, out: <none>)
Lab 1

- Due next Friday – Feb 7
- Working in groups of two is encouraged
- Make sure the solution works on myth

As always, **Get started early!**