```java
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import static jminusminus.CLConstants.*;
import java.util.ArrayList;
import java.util.BitSet;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Stack;
import java.util.TreeMap;
import java.util.Queue;

/**
 * A tuple representation of a JVM instruction.
 */

class NTuple {

    /** Program counter of the instruction. */
    public int pc;

    /** Opcode of the instruction. */
    public int opcode;

    /** Operands of the instructions. */
    public ArrayList<Short> operands;

    /** String representation (mnemonic) of the instruction. */
    public String mnemonic;

    /** Is this tuple the leader of the block containing it. */
    public boolean isLeader;

    /**
     * Construct a tuple representing the JVM instruction with the given program
     * counter, opcode, and operand list.
     *
     * @param pc
     *            program counter.
     * @param opcode
     *            opcode of the instruction.
     * @param operands
     *            list of operands of the instruction.
     */

    public NTuple(int pc, int opcode, ArrayList<Short> operands) {
        this.pc = pc;
        this.opcode = opcode;
        this.operands = operands;
        this.mnemonic = CLInstruction.instructionInfo[opcode].mnemonic;
        this.isLeader = false;
    }

    /**
     * Write the information pertaining to this tuple to STDOUT.
     *
     * @param p
     *            for pretty printing with indentation.
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("%s: %s", pc, mnemonic);
        for (short s : operands) {
            p.printf(" %s", s);
        }
```

```
67          p.printf("\n");
68      }
69
70  }
71
72  /**
73   * Representation of a block within a control flow graph.
74   */
75
76  class NBasicBlock {
77
78      /** The control flow graph (cfg) that this block belongs to. */
79      public NControlFlowGraph cfg;
80
81      /** Unique identifier of ths block. */
82      public int id;
83
84      /** List of tuples in this block. */
85      public ArrayList<NTuple> tuples;
86
87      /** List of predecessor blocks. */
88      public ArrayList<NBasicBlock> predecessors;
89
90      /** List of successor blocks. */
91      public ArrayList<NBasicBlock> successors;
92
93      /** List of high-level (HIR) instructions in this block. */
94      public ArrayList<Integer> hir;
95
96      /** List of low-level (LIR) instructions in this block. */
97      public ArrayList<NLIRInstruction> lir;
98
99      /**
100      * The state array for this block that maps local variable index to the HIR
101      * instruction that last affected it.
102      */
103     public int[] locals;
104
105     /** Has this block been visited? */
106     public boolean visited;
107
108     /** Is this block active? */
109     public boolean active;
110
111     /** Is this block a loop head? */
112     public boolean isLoopHead;
113
114     /** Is this block a loop tail? */
115     public boolean isLoopTail;
116
117     /** Index of a loop. */
118     public int loopIndex;
119
120     /** Depth of a loop. */
121     public int loopDepth;
122
123     /** Number of forward branches to this block. */
124     public int fwdBranches;
125
126     /** Number of backward branches to this block. */
127     public int bwdBranches;
128
129     /** Ref count of this block. */
130     public int ref;
131
132     /** The dominator of this block. */
133     public NBasicBlock dom;
134
135     /** All virtual registers locally defined within this block. */
```

```
136     public BitSet liveDef;
137
138     /**
139      * All virtual registers used before definition within this block.
140      */
141     public BitSet liveUse;
142
143     /** All virtual registers live in the block. */
144     public BitSet liveIn;
145
146     /** All virtual registers live outside the block. */
147     public BitSet liveOut;
148
149     /**
150      * Construct a block given its unique identifier.
151      *
152      * @param cfg
153      *            the cfg containing this block.
154      * @param id
155      *            id of the block.
156      */
157
158     public NBasicBlock(NControlFlowGraph cfg, int id) {
159         this.cfg = cfg;
160         this.id = id;
161         this.tuples = new ArrayList<NTuple>();
162         this.predecessors = new ArrayList<NBasicBlock>();
163         this.successors = new ArrayList<NBasicBlock>();
164         this.hir = new ArrayList<Integer>();
165         this.lir = new ArrayList<NLirInstruction>();
166         this.isLoopHead = false;
167     }
168
169     /**
170      * Return a string identifier of this block.
171      *
172      * @return string identifier of this block.
173      */
174
175     public String id() {
176         return "B" + id;
177     }
178
179     /**
180      * Is this block the same as the other block? Two blocks are the same if
181      * their ids are the same.
182      *
183      * @param other
184      *            the other block.
185      * @return true or false.
186      */
187
188     public boolean equals(NBasicBlock other) {
189         return this.id == other.id;
190     }
191
192     /**
193      * Return a string representation of this block.
194      *
195      * @return string representation of this block.
196      */
197
198     public String toString() {
199         return "[B" + id + "]";
200     }
201
202     /**
203      * Write the tuples in this block to STDOUT.
204      *
```

```
     * @param p
     *            for pretty printing with indentation.
     */

    public void writeTuplesToStdOut(PrettyPrinter p) {
        String s = id();
        p.printf("%s\n", s);
        for (NTuple tuple : tuples) {
            tuple.writeToStdOut(p);
        }
        p.printf("\n");
    }

    /**
     * Write the HIR instructions in this block to STDOUT.
     *
     * @param p
     *            for pretty printing with indentation.
     */

    public void writeHirToStdOut(PrettyPrinter p) {
        String s = id() + (isLoopHead ? " [LH]" : "")
                + (isLoopTail ? " [LT]" : "");
        if (tuples.size() > 0) {
            s += " [" + tuples.get(0).pc + ", "
                    + tuples.get(tuples.size() - 1).pc + "]";
        }
        if (dom != null) {
            s += " dom: " + dom.id();
        }
        if (predecessors.size() > 0) {
            s += " pred: ";
            for (NBasicBlock block : predecessors) {
                s += block.id() + " ";
            }
        }
        if (successors.size() > 0) {
            s += " succ: ";
            for (NBasicBlock block : successors) {
                s += block.id() + " ";
            }
        }
        p.printf(s + "\n");
        s = "Locals: ";
        if (locals != null) {
            for (int i = 0; i < locals.length; i++) {
                if (!(cfg.hirMap.get(locals[i]) instanceof NHIRLocal)) {
                    s += cfg.hirMap.get(locals[i]).id() + " ";
                }
            }
        }
        p.printf("%s\n", s);
        for (int ins : hir) {
            if (cfg.hirMap.get(ins) instanceof NHIRPhiFunction) {
                p.printf("%s: %s\n", ((NHIRPhiFunction) cfg.hirMap.get(ins))
                        .id(), ((NHIRPhiFunction) cfg.hirMap.get(ins)));
            }
        }
        for (int ins : hir) {
            if (!(cfg.hirMap.get(ins) instanceof NHIRPhiFunction)) {
                p.printf("%s\n", cfg.hirMap.get(ins));
            }
        }
        p.printf("\n");
    }

    /**
     * Write the LIR instructions in this block to STDOUT.
     *
```

```
     * @param p
     *            for pretty printing with indentation.
     */
    public void writeLirToStdOut(PrettyPrinter p) {
        p.printf("%s\n", id());
        for (NLIRInstruction ins : lir) {
            p.printf("%s\n", ins);
        }
        p.printf("\n");
    }

    /**
     * The instruction identifier for the first LIR instruction.
     *
     * @return the instruction identifier.
     */
    public int getFirstLIRInstId() {
        if (lir.isEmpty()) {
            return -1;
        }
        return lir.get(0).id;
    }

    /**
     * The instruction identifier for the last LIR instruction.
     *
     * @return the instruction identifier.
     */
    public int getLastLIRInstId() {
        if (lir.isEmpty()) {
            return -1;
        }
        return lir.get(lir.size() - 1).id;
    }

    /**
     * Iterates through the lir array of this block, returning an
     * NLIRInstruction with the specified id.
     *
     * @param id
     *            the id to look for.
     *
     * @return NLIRInstruction with the specified id, null if none matched.
     */
    public NLIRInstruction getInstruction(int id) {
        for (NLIRInstruction i : this.lir) {
            if (i.id == id) {
                return i;
            }
        }
        return null;
    }

    /**
     * Checks to see if there is an LIRInstruction with this id in the block's
     * lir.
     *
     * @return true or false.
     */
    public boolean idIsFree(int id) {
        if (this.getInstruction(id) != null) {
            return false;
        } else {
            return true;
        }
    }

    /**
     * Inserts an NLIRInstruction to the appropriate place in this block's lir
```

```
     * array based on its id -- preserving order by id.
     *
     * @param inst
     *            the NLIRInstruction to be inserted.
     */
    public void insertLIRInst(NLIRInstruction inst) {
        int idx = -1;
        for (int i = 0; i < this.lir.size(); i++) {
            if (this.lir.get(i).id < inst.id) {
                idx = i;
            }
        }
        if (++idx == this.lir.size()) {
            this.lir.add(inst);
        } else {
            this.lir.add(idx, inst);
        }
    }

}

/**
 * Representation of a control flow graph (cfg) for a method.
 */

class NControlFlowGraph {

    /** Constant pool for the class containing the method. */
    private CLConstantPool cp;

    /** Contains information about the method. */
    private CLMethodInfo m;

    /** Maps the pc of a JVM instruction to the block it's in. */
    private HashMap<Integer, NBasicBlock> pcToBasicBlock;

    /** block identifier. */
    public static int blockId;

    /** HIR instruction identifier. */
    public static int hirId;

    /** HIR instruction identifier. */
    public static int lirId;

    /** Virtual register identifier. */
    public static int regId;

    /** Stack offset counter.. */
    public int offset;

    /** Loop identifier. */
    public static int loopIndex;

    /** Name of the method this cfg corresponds to. */
    public String name;

    /** Descriptor of the method this cfg corresponds to. */
    public String desc;

    /**
     * List of blocks forming the cfg for the method.
     */
    public ArrayList<NBasicBlock> basicBlocks;

    /** Maps HIR instruction ids in this cfg to HIR instructions. */
    public TreeMap<Integer, NHIRInstruction> hirMap;

    /**
```

```
     * Registers allocated for this cfg by the HIR to LIR conversion algorithm.
     */
    public ArrayList<NRegister> registers;

    /**
     * The total number of intervals. This is used to name split children and
     * grows as more intervals are created by spills.
     */
    public int maxIntervals;

    /**
     * Physical registers allocated for this cfg by the HIR to LIR conversion
     * algorithm.
     */
    public ArrayList<NPhysicalRegister> pRegisters;

    /**
     * Intervals allocated by the register allocation algorithm.
     */
    public ArrayList<NInterval> intervals;

    /** Used to construct jump labels in spim output. */
    public String labelPrefix;

    /**
     * SPIM code for string literals added to the data segment.
     */
    public ArrayList<String> data;

    /**
     * Construct an NControlFlowGraph object for a method given the constant
     * pool for the class containing the method and the object containing
     * information about the method.
     *
     * @param cp
     *            constant pool for the class containing the method.
     * @param m
     *            contains information about the method.
     */

    public NControlFlowGraph(CLConstantPool cp, CLMethodInfo m) {
        this.cp = cp;
        this.m = m;
        name = new String(((CLConstantUtf8Info) cp.cpItem(m.nameIndex)).b);
        desc = new String(((CLConstantUtf8Info) cp.cpItem(m.descriptorIndex)).b);
        basicBlocks = new ArrayList<NBasicBlock>();
        pcToBasicBlock = new HashMap<Integer, NBasicBlock>();
        ArrayList<Integer> code = getByteCode();
        ArrayList<NTuple> tuples = bytecodeToTuples(code);
        if (tuples.size() == 0) {
            return;
        }
        NTuple[] tupleAt = new NTuple[code.size()];
        for (NTuple tuple : tuples) {
            tupleAt[tuple.pc] = tuple;
        }

        // Identify the leaders.
        tuples.get(0).isLeader = true;
        for (int j = 1; j < tuples.size(); j++) {
            NTuple tuple = tuples.get(j);
            boolean jumpInstruction = true;
            short operandByte1, operandByte2, operandByte3, operandByte4;
            int offset;
            switch (tuple.opcode) {
            case IFEQ:
            case IFNE:
            case IFLT:
            case IFGE:
```

```
            case IFGT:
            case IFLE:
            case IF_ICMPEQ:
            case IF_ICMPNE:
            case IF_ICMPLT:
            case IF_ICMPGE:
            case IF_ICMPGT:
            case IF_ICMPLE:
            case IF_ACMPEQ:
            case IF_ACMPNE:
            case GOTO:
            case JSR:
            case IFNULL:
            case IFNONNULL:
                operandByte1 = tuple.operands.get(0);
                operandByte2 = tuple.operands.get(1);
                offset = shortValue(operandByte1, operandByte2);
                tupleAt[tuple.pc + offset].isLeader = true;
                break;
            case GOTO_W:
            case JSR_W:
                operandByte1 = tuple.operands.get(0);
                operandByte2 = tuple.operands.get(1);
                operandByte3 = tuple.operands.get(2);
                operandByte4 = tuple.operands.get(3);
                offset = intValue(operandByte1, operandByte2, operandByte3,
                        operandByte4);
                tupleAt[tuple.pc + offset].isLeader = true;
                break;
            case IRETURN:
            case LRETURN:
            case FRETURN:
            case DRETURN:
            case ARETURN:
            case RETURN:
            case RET:
            case ATHROW:
                break;
            case TABLESWITCH: // TBD
                break;
            case LOOKUPSWITCH: // TBD
                break;
            default:
                jumpInstruction = false;
            }
            if (jumpInstruction) {
                if (j < tuples.size() - 1) {
                    tuples.get(j + 1).isLeader = true;
                }
            }
        }

        // Form blocks.
        {
            blockId = 0;
            NBasicBlock block = new NBasicBlock(this, blockId++);
            for (NTuple tuple : tuples) {
                if (tuple.isLeader) {
                    basicBlocks.add(block);
                    block = new NBasicBlock(this, blockId++);
                    if (!pcToBasicBlock.containsKey(tuple.pc)) {
                        pcToBasicBlock.put(tuple.pc, block);
                    }
                }
                block.tuples.add(tuple);
            }
            basicBlocks.add(block);
        }

```

```
        // Connect up the blocks for this method, that is, build
        // its control flow graph.
        basicBlocks.get(0).successors.add(basicBlocks.get(1));
        basicBlocks.get(1).predecessors.add(basicBlocks.get(0));
        NBasicBlock[] blockAt = new NBasicBlock[code.size()];
        for (NBasicBlock block : basicBlocks) {
            if (block.tuples.size() == 0) {
                continue;
            }
            blockAt[block.tuples.get(0).pc] = block;
        }
        for (int j = 0; j < basicBlocks.size(); j++) {
            NBasicBlock block = basicBlocks.get(j);
            if (block.tuples.size() == 0) {
                continue;
            }
            NTuple tuple = block.tuples.get(block.tuples.size() - 1);
            short operandByte1, operandByte2, operandByte3, operandByte4;
            int offset;
            NBasicBlock target;
            switch (tuple.opcode) {
            case IFEQ:
            case IFNE:
            case IFLT:
            case IFGE:
            case IFGT:
            case IFLE:
            case IF_ICMPEQ:
            case IF_ICMPNE:
            case IF_ICMPLT:
            case IF_ICMPGE:
            case IF_ICMPGT:
            case IF_ICMPLE:
            case IF_ACMPEQ:
            case IF_ACMPNE:
            case IFNULL:
            case IFNONNULL:
                operandByte1 = tuple.operands.get(0);
                operandByte2 = tuple.operands.get(1);
                offset = shortValue(operandByte1, operandByte2);
                target = blockAt[tuple.pc + offset];
                if (j < basicBlocks.size() - 1) {
                    block.successors.add(basicBlocks.get(j + 1));
                    basicBlocks.get(j + 1).predecessors.add(block);
                }
                block.successors.add(target);
                target.predecessors.add(block);
                break;
            case GOTO:
            case JSR:
                operandByte1 = tuple.operands.get(0);
                operandByte2 = tuple.operands.get(1);
                offset = shortValue(operandByte1, operandByte2);
                target = blockAt[tuple.pc + offset];
                block.successors.add(target);
                target.predecessors.add(block);
                break;
            case GOTO_W:
            case JSR_W:
                operandByte1 = tuple.operands.get(0);
                operandByte2 = tuple.operands.get(1);
                operandByte3 = tuple.operands.get(2);
                operandByte4 = tuple.operands.get(3);
                offset = intValue(operandByte1, operandByte2, operandByte3,
                        operandByte4);
                target = blockAt[tuple.pc + offset];
                block.successors.add(target);
                target.predecessors.add(block);
                break;
```

```
            case IRETURN:
            case LRETURN:
            case FRETURN:
            case DRETURN:
            case ARETURN:
            case RETURN:
            case RET:
            case ATHROW:
                break;
            case TABLESWITCH: // TBD
                break;
            case LOOKUPSWITCH: // TBD
                break;
            default:
                if (j < basicBlocks.size() - 1) {
                    block.successors.add(basicBlocks.get(j + 1));
                    basicBlocks.get(j + 1).predecessors.add(block);
                }
            }
        }

        // Calculate the ref count and number of forward branches
        // to each block in the this cfg.
        for (NBasicBlock block : basicBlocks) {
            block.ref = block.predecessors.size();
            block.fwdBranches = block.predecessors.size() - block.bwdBranches;
        }
    }

    /**
     * Implements loop detection algorithm to figure out if the specified block
     * is a loop head or a loop tail. Also calculates the number of backward
     * branches to the block.
     *
     * @param block
     *            a block.
     * @param pred
     *            block's predecessor or null.
     */

    public void detectLoops(NBasicBlock block, NBasicBlock pred) {
        if (!block.visited) {
            block.visited = true;
            block.active = true;
            for (NBasicBlock succ : block.successors) {
                detectLoops(succ, block);
            }
            block.active = false;
        } else if (block.active) {
            block.isLoopHead = true;
            pred.isLoopTail = true;
            block.bwdBranches++;
            block.loopIndex = NControlFlowGraph.loopIndex++;
        }
    }

    /**
     * Remove blocks that cannot be reached from the begin block (B0). Also
     * removes these blocks from the predecessor lists.
     */

    public void removeUnreachableBlocks() {
        // Create a list of blocks that cannot be reached.
        ArrayList<NBasicBlock> toRemove = new ArrayList<NBasicBlock>();
        for (NBasicBlock block : basicBlocks) {
            if (!block.visited) {
                toRemove.add(block);
            }
        }
```

```
688
689         // From the predecessor list for each blocks, remove
690         // the ones that are in toRemove list.
691         for (NBasicBlock block : basicBlocks) {
692             for (NBasicBlock pred : toRemove) {
693                 block.predecessors.remove(pred);
694             }
695         }
696
697         // From the list of all blocks, remove the ones that
698         // are in toRemove list.
699         for (NBasicBlock block : toRemove) {
700             basicBlocks.remove(block);
701         }
702     }
703
704     /**
705      * Compute the dominator of each block in this cfg recursively given the
706      * starting block and its predecessor.
707      *
708      * @param block
709      *            starting block.
710      * @param pred
711      *            block's predecessor.
712      */
713
714     public void computeDominators(NBasicBlock block, NBasicBlock pred) {
715         if (block.ref > 0) {
716             block.ref--;
717         }
718         if (block.dom == null) {
719             block.dom = pred;
720         } else {
721             block.dom = commonDom(block.dom, pred);
722         }
723         if (block.ref == block.bwdBranches) {
724             for (NBasicBlock s : block.successors) {
725                 computeDominators(s, block);
726             }
727         }
728     }
729
730     /**
731      * Convert tuples in each block to their high-level (HIR) representations.
732      */
733
734     public void tuplesToHir() {
735         clearBlockVisitations();
736         hirId = 0;
737         loopIndex = 0;
738         hirMap = new TreeMap<Integer, NHIRInstruction>();
739         int numLocals = numLocals();
740         int[] locals = new int[numLocals];
741         ArrayList<String> argTypes = argumentTypes(desc);
742         NBasicBlock beginBlock = basicBlocks.get(0);
743         for (int i = 0; i < locals.length; i++) {
744             NHIRInstruction ins = null;
745             if (i < argTypes.size()) {
746                 String lType = argTypes.get(i);
747                 ins = new NHIRLoadLocal(beginBlock, hirId++, i,
748                         shortType(lType), lType);
749                 beginBlock.hir.add(ins.id);
750             } else {
751                 ins = new NHIRLocal(beginBlock, hirId++, i, "", "");
752             }
753             beginBlock.cfg.hirMap.put(ins.id, ins);
754             locals[i] = ins.id;
755         }
756         beginBlock.locals = locals;
```

```
        Stack<Integer> operandStack = new Stack<Integer>();
        Queue<NBasicBlock> q = new LinkedList<NBasicBlock>();
        beginBlock.visited = true;
        q.add(beginBlock);
        while (!q.isEmpty()) {
            NBasicBlock block = q.remove();
            for (NBasicBlock succ : block.successors) {
                if (!succ.visited) {
                    succ.visited = true;
                    q.add(succ);
                }
            }

            // Convert tuples in block to HIR instructions.
            if (block.predecessors.size() == 1) {
                block.locals = block.predecessors.get(0).locals.clone();
            } else if (block.predecessors.size() > 1) {
                if (block.isLoopHead) {
                    for (NBasicBlock pred : block.predecessors) {
                        if (pred.locals != null) {
                            block.locals = pred.locals.clone();
                            break;
                        }
                    }
                    for (int i = 0; i < block.locals.length; i++) {
                        ArrayList<Integer> args = new ArrayList<Integer>();
                        NHIRPhiFunction phi = new NHIRPhiFunction(block,
                                hirId++, args, i);
                        block.hir.add(phi.id);
                        block.cfg.hirMap.put(phi.id, phi);
                        phi.arguments.add(block.locals[i]);
                        for (int j = 1; j < block.predecessors.size(); j++) {
                            phi.arguments.add(phi.id);
                        }
                        block.locals[i] = phi.id;
                        phi.inferType();
                    }
                } else {
                    block.locals = block.predecessors.get(0).locals.clone();
                    for (int i = 1; i < block.predecessors.size(); i++) {
                        NBasicBlock pred = block.predecessors.get(i);
                        mergeLocals(block, pred);
                    }
                }
            }
            for (NTuple tuple : block.tuples) {
                CLInsInfo insInfo = CLInstruction.instructionInfo[tuple.opcode];
                int localVariableIndex = insInfo.localVariableIndex;
                NHIRInstruction ins = null;
                short operandByte1 = 0, operandByte2 = 0, operandByte3 = 0,
offset = 0;
                int operand1 = 0, operand2 = 0, operand3 = 0;
                switch (insInfo.opcode) {
                case MULTIANEWARRAY: {
                    operandByte1 = tuple.operands.get(0);
                    operandByte2 = tuple.operands.get(1);
                    operandByte3 = tuple.operands.get(2);
                    int index = shortValue(operandByte1, operandByte2);
                    int classIndex = ((CLConstantClassInfo)
cp.cpItem(index)).nameIndex;
                    String type = new String(((CLConstantUtf8Info) cp
                            .cpItem(classIndex)).b);
                    ins = new NHIRNewArray(block, hirId++, insInfo.opcode,
                            (int) operandByte3, shortType(type), type);
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    operandStack.push(ins.id);
                    break;
                }
```

```
824                 case AALOAD: {
825                     operand2 = operandStack.pop();
826                     operand1 = operandStack.pop();
827 
828                     // Compute base address.
829                     NHIRInstruction ins1 = new NHIRIntConstant(block, hirId++,
830                             12);
831                     NHIRInstruction ins2 = new NHIRArithmetic(block, hirId++,
832                             IADD, operand1, ins1.id);
833                     block.cfg.hirMap.put(ins1.id, ins1);
834                     block.hir.add(ins1.id);
835                     block.cfg.hirMap.put(ins2.id, ins2);
836                     block.hir.add(ins2.id);
837 
838                     // Compute index.
839                     NHIRInstruction ins3 = new NHIRIntConstant(block, hirId++,
840                             4);
841                     NHIRInstruction ins4 = new NHIRArithmetic(block, hirId++,
842                             IMUL, operand2, ins3.id);
843                     block.cfg.hirMap.put(ins3.id, ins3);
844                     block.hir.add(ins3.id);
845                     block.cfg.hirMap.put(ins4.id, ins4);
846                     block.hir.add(ins4.id);
847 
848                     ins = new NHIRALoad(block, hirId++, insInfo.opcode,
849                             ins2.id, ins4.id, "L", "L");
850                     block.cfg.hirMap.put(ins.id, ins);
851                     block.hir.add(ins.id);
852                     operandStack.push(ins.id);
853                     break;
854                 }
855                 case IALOAD: {
856                     operand2 = operandStack.pop();
857                     operand1 = operandStack.pop();
858 
859                     // Compute base address.
860                     NHIRInstruction ins1 = new NHIRIntConstant(block, hirId++,
861                             12);
862                     NHIRInstruction ins2 = new NHIRArithmetic(block, hirId++,
863                             IADD, operand1, ins1.id);
864                     block.cfg.hirMap.put(ins1.id, ins1);
865                     block.hir.add(ins1.id);
866                     block.cfg.hirMap.put(ins2.id, ins2);
867                     block.hir.add(ins2.id);
868 
869                     // Compute index.
870                     NHIRInstruction ins3 = new NHIRIntConstant(block, hirId++,
871                             4);
872                     NHIRInstruction ins4 = new NHIRArithmetic(block, hirId++,
873                             IMUL, operand2, ins3.id);
874                     block.cfg.hirMap.put(ins3.id, ins3);
875                     block.hir.add(ins3.id);
876                     block.cfg.hirMap.put(ins4.id, ins4);
877                     block.hir.add(ins4.id);
878 
879                     ins = new NHIRALoad(block, hirId++, insInfo.opcode,
880                             ins2.id, ins4.id, "I", "I");
881                     block.cfg.hirMap.put(ins.id, ins);
882                     block.hir.add(ins.id);
883                     operandStack.push(ins.id);
884                     break;
885                 }
886                 case IASTORE: {
887                     operand3 = operandStack.pop();
888                     operand2 = operandStack.pop();
889                     operand1 = operandStack.pop();
890 
891                     // Compute base address.
892                     NHIRInstruction ins1 = new NHIRIntConstant(block, hirId++,
```

```
                    12);
            NHIRInstruction ins2 = new NHIRArithmetic(block, hirId++,
                    IADD, operand1, ins1.id);
            block.cfg.hirMap.put(ins1.id, ins1);
            block.hir.add(ins1.id);
            block.cfg.hirMap.put(ins2.id, ins2);
            block.hir.add(ins2.id);

            // Compute index.
            NHIRInstruction ins3 = new NHIRIntConstant(block, hirId++,
                    4);
            NHIRInstruction ins4 = new NHIRArithmetic(block, hirId++,
                    IMUL, operand2, ins3.id);
            block.cfg.hirMap.put(ins3.id, ins3);
            block.hir.add(ins3.id);
            block.cfg.hirMap.put(ins4.id, ins4);
            block.hir.add(ins4.id);

            ins = new NHIRAStore(block, hirId++, insInfo.opcode,
                    ins2.id, ins4.id, operand3, "I", "I");
            block.cfg.hirMap.put(ins.id, ins);
            block.hir.add(ins.id);
            break;
        }
        case ICONST_0:
        case ICONST_1:
        case ICONST_2:
        case ICONST_3:
        case ICONST_4:
        case ICONST_5: {
            ins = new NHIRIntConstant(block, hirId++, tuple.opcode - 3);
            block.cfg.hirMap.put(ins.id, ins);
            block.hir.add(ins.id);
            operandStack.push(ins.id);
            break;
        }
        case ILOAD: {
            operandByte1 = tuple.operands.get(0);
            localVariableIndex = operandByte1;
            operandStack.push(block.locals[localVariableIndex]);
            break;
        }
        case ILOAD_0:
        case ILOAD_1:
        case ILOAD_2:
        case ILOAD_3:
        case ALOAD_0:
        case ALOAD_1:
        case ALOAD_2:
        case ALOAD_3: {
            operandStack.push(block.locals[localVariableIndex]);
            break;
        }
        case ISTORE: {
            operandByte1 = tuple.operands.get(0);
            localVariableIndex = operandByte1;
            block.locals[localVariableIndex] = operandStack.pop();
            break;
        }
        case ISTORE_0:
        case ISTORE_1:
        case ISTORE_2:
        case ISTORE_3:
        case ASTORE_0:
        case ASTORE_1:
        case ASTORE_2:
        case ASTORE_3: {
            block.locals[localVariableIndex] = operandStack.pop();
            break;
```

```
                }
                case BIPUSH: {
                    operandByte1 = tuple.operands.get(0);
                    ins = new NHIRIntConstant(block, hirId++, operandByte1);
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    operandStack.push(ins.id);
                    break;
                }
                case SIPUSH: {
                    operandByte1 = tuple.operands.get(0);
                    operandByte2 = tuple.operands.get(1);
                    ins = new NHIRIntConstant(block, hirId++, shortValue(
                            operandByte1, operandByte2));
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    operandStack.push(ins.id);
                    break;
                }
                case LDC: {
                    operandByte1 = tuple.operands.get(0);

                    // Only allowing ldc of string constants for
                    // now.
                    int stringIndex = ((CLConstantStringInfo) cp
                            .cpItem(operandByte1)).stringIndex;
                    String s = new String(((CLConstantUtf8Info) cp
                            .cpItem(stringIndex)).b);
                    ins = new NHIRStringConstant(block, hirId++, s);
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    operandStack.push(ins.id);
                    break;
                }
                case IADD:
                case ISUB:
                case IMUL: {
                    operand2 = operandStack.pop();
                    operand1 = operandStack.pop();
                    ins = new NHIRArithmetic(block, hirId++, insInfo.opcode,
                            operand1, operand2);
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    operandStack.push(ins.id);
                    break;
                }
                case IINC: {
                    operandByte1 = tuple.operands.get(0);
                    operandByte2 = tuple.operands.get(1);
                    operand1 = block.locals[operandByte1];
                    NHIRInstruction ins1 = new NHIRIntConstant(block, hirId++,
                            (byte) operandByte2);
                    ins = new NHIRArithmetic(block, hirId++, IADD, operand1,
                            ins1.id);
                    block.locals[operandByte1] = ins.id;
                    block.hir.add(ins1.id);
                    block.cfg.hirMap.put(ins1.id, ins1);
                    block.hir.add(ins.id);
                    block.cfg.hirMap.put(ins.id, ins);
                    break;
                }
                case IF_ICMPNE:
                case IF_ICMPGT:
                case IF_ICMPLE: {
                    operandByte1 = tuple.operands.get(0);
                    operandByte2 = tuple.operands.get(1);
                    offset = shortValue(operandByte1, operandByte2);
                    int rhs = operandStack.pop();
                    int lhs = operandStack.pop();
```

```
                    NBasicBlock trueDestination = pcToBasicBlock.get(tuple.pc
                            + offset);
                    NBasicBlock falseDestination = pcToBasicBlock
                            .get(tuple.pc + 3);
                    ins = new NHIRConditionalJump(block, hirId++, lhs, rhs,
                            insInfo.opcode, trueDestination, falseDestination);
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    break;
                }
                case GOTO: {
                    operandByte1 = tuple.operands.get(0);
                    operandByte2 = tuple.operands.get(1);
                    offset = shortValue(operandByte1, operandByte2);
                    NBasicBlock destination = pcToBasicBlock.get(tuple.pc
                            + offset);
                    ins = new NHIRGoto(block, hirId++, destination);
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    break;
                }
                case GETSTATIC:
                case PUTSTATIC: {
                    operandByte1 = tuple.operands.get(0);
                    operandByte2 = tuple.operands.get(1);
                    int index = shortValue(operandByte1, operandByte2);
                    int classIndex = ((CLConstantFieldRefInfo)
cp.cpItem(index)).classIndex;
                    int nameAndTypeIndex = ((CLConstantFieldRefInfo) cp
                            .cpItem(index)).nameAndTypeIndex;
                    int nameIndex = ((CLConstantClassInfo) cp
                            .cpItem(classIndex)).nameIndex;
                    String target = new String(((CLConstantUtf8Info) cp
                            .cpItem(nameIndex)).b);
                    int fieldNameIndex = ((CLConstantNameAndTypeInfo) cp
                            .cpItem(nameAndTypeIndex)).nameIndex;
                    int fieldDescIndex = ((CLConstantNameAndTypeInfo) cp
                            .cpItem(nameAndTypeIndex)).descriptorIndex;
                    String name = new String(((CLConstantUtf8Info) cp
                            .cpItem(fieldNameIndex)).b);
                    String desc = new String(((CLConstantUtf8Info) cp
                            .cpItem(fieldDescIndex)).b);
                    if (insInfo.opcode == PUTSTATIC) {
                        ins = new NHIRPutField(block, hirId++, insInfo.opcode,
                                target, name, shortType(desc), desc,
                                operandStack.pop());
                    } else {
                        ins = new NHIRGetField(block, hirId++, insInfo.opcode,
                                target, name, shortType(desc), desc);
                        operandStack.push(ins.id);
                    }
                    block.cfg.hirMap.put(ins.id, ins);
                    block.hir.add(ins.id);
                    break;
                }
                case INVOKESPECIAL:
                case INVOKESTATIC: {
                    operandByte1 = tuple.operands.get(0);
                    operandByte2 = tuple.operands.get(1);
                    int index = shortValue(operandByte1, operandByte2);
                    int classIndex = ((CLConstantMethodRefInfo) cp
                            .cpItem(index)).classIndex;
                    int nameAndTypeIndex = ((CLConstantMethodRefInfo) cp
                            .cpItem(index)).nameAndTypeIndex;
                    int nameIndex = ((CLConstantClassInfo) cp
                            .cpItem(classIndex)).nameIndex;
                    String target = new String(((CLConstantUtf8Info) cp
                            .cpItem(nameIndex)).b);
                    int methodNameIndex = ((CLConstantNameAndTypeInfo) cp
```

```
                        .cpItem(nameAndTypeIndex)).nameIndex;
                int methodDescIndex = ((CLConstantNameAndTypeInfo) cp
                        .cpItem(nameAndTypeIndex)).descriptorIndex;
                String name = new String(((CLConstantUtf8Info) cp
                        .cpItem(methodNameIndex)).b);
                String desc = new String(((CLConstantUtf8Info) cp
                        .cpItem(methodDescIndex)).b);
                ArrayList<Integer> args = new ArrayList<Integer>();
                int numArgs = argumentCount(desc);
                for (int i = 0; i < numArgs; i++) {
                    int arg = operandStack.pop();
                    args.add(0, arg);
                }
                String returnType = returnType(desc);
                ins = new NHIRInvoke(block, hirId++, insInfo.opcode,
                        target, name, args, shortType(returnType),
                        returnType);
                if (!returnType.equals("V")) {
                    operandStack.push(ins.id);
                }
                block.cfg.hirMap.put(ins.id, ins);
                block.hir.add(ins.id);
                break;
            }
            case IRETURN:
            case ARETURN: {
                ins = new NHIRReturn(block, hirId++, insInfo.opcode,
                        operandStack.pop());
                block.cfg.hirMap.put(ins.id, ins);
                block.hir.add(ins.id);
                break;
            }
            case RETURN: {
                ins = new NHIRReturn(block, hirId++, insInfo.opcode, -1);
                block.cfg.hirMap.put(ins.id, ins);
                block.hir.add(ins.id);
                break;
            }
        }
    }
}

/**
 * Carry out optimizations on the high-level instructions.
 */

public void optimize() {
    // TBD
}

/**
 * Eliminate redundant phi functions of the form x = (y, x, x, ..., x) with
 * y.
 */

public void eliminateRedundantPhiFunctions() {
    for (int ins : hirMap.keySet()) {
        NHIRInstruction hir = hirMap.get(ins);
        if (hir instanceof NHIRPhiFunction) {
            NHIRPhiFunction phi = (NHIRPhiFunction) hir;
            int firstArg = phi.arguments.get(0);
            boolean match = true;
            NBasicBlock block = phi.block;
            if (!block.isLoopHead) {
                continue;
            }
            for (int i = 1; i < phi.arguments.size(); i++) {
                if (phi.arguments.get(i) !=
```

```
block.predecessors.get(i).locals[phi.local]) {
                        match = false;
                        phi.arguments.set(i,
                                block.predecessors.get(i).locals[phi.local]);
                    }
                }
                if (match && firstArg != phi.id) {
                    hirMap.put(phi.id, hirMap.get(firstArg));
                    phi.block.hir.remove((Integer) phi.id);
                }
            }
        }
    }

    /**
     * Convert the hir instructions in this cfg to lir instructions.
     */

    public void hirToLir() {
        lirId = 0;
        regId = 32;
        offset = 0;
        registers = new ArrayList<NRegister>();
        data = new ArrayList<String>();
        for (int i = 0; i < 32; i++) {
            registers.add(null);
        }
        pRegisters = new ArrayList<NPhysicalRegister>();
        for (int ins : hirMap.keySet()) {
            hirMap.get(ins).toLir();
        }

        // We now know how many virtual registers are needed, so
        // we can initialize bitset fields in each block that are
        // needed for interval calculation.
        int size = registers.size();
        for (NBasicBlock block : basicBlocks) {
            block.liveDef = new BitSet(size);
            block.liveUse = new BitSet(size);
            block.liveIn = new BitSet(size);
            block.liveOut = new BitSet(size);
        }
    }

    /**
     * Resolve the phi functions in this cfg, i.e., for each x = phi(x1, x2,
     * ..., xn) generate an (LIR) move xi, x instruction at the end of the
     * predecessor i of thte block defining the phi function; if the instruction
     * there is a branch, add the instruction prior to the branch.
     */

    public void resolvePhiFunctions() {
        for (int ins1 : hirMap.keySet()) {
            NHIRInstruction hir = hirMap.get(ins1);
            if (hir instanceof NHIRPhiFunction) {
                NHIRPhiFunction phi = (NHIRPhiFunction) hir;
                NBasicBlock block = phi.block;
                for (int i = 0; i < phi.arguments.size(); i++) {
                    NHIRInstruction arg = hirMap.get(phi.arguments.get(i));
                    if (arg.sType.equals("")) {
                        continue;
                    }
                    NBasicBlock targetBlock = block.predecessors.get(i);
                    NLIRMove move = new NLIRMove(arg.block, lirId++, arg.lir,
                            phi.lir);
                    int len = targetBlock.hir.size();
                    if (hirMap.get(targetBlock.hir.get(len - 1)) instanceof
NHIRGoto
                            || hirMap.get(targetBlock.hir.get(len - 1))
```

```
instanceof NHIRConditionalJump) {
                        targetBlock.lir.add(len - 1, move);
                    } else {
                        targetBlock.lir.add(move);
                    }
                }
            }
        }
    }

    /**
     * Compute optimal ordering of the basic blocks in this cfg.
     */

    public void orderBlocks() {
        // TBD
    }

    /**
     * The basic block at a particular instruction id.
     *
     * @param id
     *            the (LIR) instruction id.
     * @return the basic block.
     */

    public NBasicBlock blockAt(int id) {
        for (NBasicBlock b : this.basicBlocks) {
            if (b.getFirstLIRInstId() <= id && b.getLastLIRInstId() >= id)
                return b;
        }
        return null;
    }

    /**
     * Assign new ids to the LIR instructions in this cfg.
     */

    public void renumberLIRInstructions() {
        int nextId = 0;
        for (NBasicBlock block : basicBlocks) {
            ArrayList<NLIRInstruction> newLir = new ArrayList<NLIRInstruction>();
            for (NLIRInstruction lir : block.lir) {
                if (lir instanceof NLIRLoadLocal
                        && ((NLIRLoadLocal) lir).local < 4) {
                    // Ignore first four formals.
                    continue;
                }
                lir.id = nextId;
                nextId += 5; // an extra slot for spills though we
                // don't use it
                newLir.add(lir);
            }
            block.lir = newLir;
        }
    }

    /**
     * Replace references to virtual registers in LIR instructions with
     * references to physical registers.
     */

    public void allocatePhysicalRegisters() {
        for (NBasicBlock block : basicBlocks) {
            for (NLIRInstruction lir : block.lir) {
                lir.allocatePhysicalRegisters();
            }
        }
    }
```

```
1303
1304    /**
1305     * Write the tuples in this cfg to STDOUT.
1306     *
1307     * @param p
1308     *             for pretty printing with indentation.
1309     */
1310
1311    public void writeTuplesToStdOut(PrettyPrinter p) {
1312        p.indentRight();
1313        p.printf("========== TUPLES ==========\n\n");
1314        for (NBasicBlock block : basicBlocks) {
1315            block.writeTuplesToStdOut(p);
1316        }
1317        p.indentLeft();
1318    }
1319
1320    /**
1321     * Write the hir instructions in this cfg to STDOUT.
1322     *
1323     * @param p
1324     *             for pretty printing with indentation.
1325     */
1326
1327    public void writeHirToStdOut(PrettyPrinter p) {
1328        p.indentRight();
1329        p.printf("========== HIR ==========\n\n");
1330        for (NBasicBlock block : basicBlocks) {
1331            block.writeHirToStdOut(p);
1332        }
1333        p.indentLeft();
1334    }
1335
1336    /**
1337     * Write the lir instructions in this cfg to STDOUT.
1338     *
1339     * @param p
1340     *             for pretty printing with indentation.
1341     */
1342
1343    public void writeLirToStdOut(PrettyPrinter p) {
1344        p.indentRight();
1345        p.printf("========== LIR ==========\n\n");
1346        for (NBasicBlock block : basicBlocks) {
1347            block.writeLirToStdOut(p);
1348        }
1349        p.indentLeft();
1350    }
1351
1352    /**
1353     * Write the intervals in this cfg to STDOUT.
1354     *
1355     * @param p
1356     *             for pretty printing with indentation.
1357     */
1358
1359    public void writeIntervalsToStdOut(PrettyPrinter p) {
1360        p.indentRight();
1361        p.printf("========== INTERVALS ==========\n\n");
1362        for (NInterval interval : intervals) {
1363            interval.writeToStdOut(p);
1364        }
1365        p.indentLeft();
1366        p.printf("\n");
1367    }
1368
1369    /**
1370     * Clear the visitation information in each block in this cfg.
1371     */
```

```
1372
1373    private void clearBlockVisitations() {
1374        for (NBasicBlock block : basicBlocks) {
1375            block.visited = false;
1376        }
1377    }
1378
1379    /**
1380     * Given a basic block and its predecessor, return their common dominator.
1381     *
1382     * @param b
1383     *            a basic block.
1384     * @param pred
1385     *            predecessor of b.
1386     *
1387     * @return common dominator of the given block and its predecessor.
1388     */
1389
1390    private NBasicBlock commonDom(NBasicBlock b, NBasicBlock pred) {
1391        NBasicBlock dom = b;
1392        clearBlockVisitations();
1393        while (dom != null) {
1394            dom.visited = true;
1395            dom = dom.dom;
1396        }
1397        dom = pred;
1398        while (!dom.visited) {
1399            dom = dom.dom;
1400        }
1401        return dom;
1402    }
1403
1404    /**
1405     * Merge the locals from each of the predecessors of the specified block
1406     * with the locals in the block.
1407     *
1408     * @param block
1409     *            block to merge into.
1410     */
1411
1412    private void mergeLocals(NBasicBlock block) {
1413        for (NBasicBlock other : block.predecessors) {
1414            mergeLocals(block, other);
1415        }
1416    }
1417
1418    /**
1419     * Merge the locals in block b with the locals in block a.
1420     *
1421     * @param a
1422     *            block to merge into.
1423     * @param b
1424     *            block to merge from.
1425     */
1426
1427    private void mergeLocals(NBasicBlock a, NBasicBlock b) {
1428        for (int i = 0; i < a.locals.length; i++) {
1429            if (a.cfg.hirMap.get(a.locals[i]).equals(
1430                    b.cfg.hirMap.get(b.locals[i]))) {
1431                continue;
1432            } else {
1433                ArrayList<Integer> args = new ArrayList<Integer>();
1434                args.add(a.locals[i]);
1435                args.add(b.locals[i]);
1436                NHIRInstruction ins = new NHIRPhiFunction(a,
1437                        NControlFlowGraph.hirId++, args, i);
1438                a.locals[i] = ins.id;
1439                a.hir.add(ins.id);
1440                a.cfg.hirMap.put(ins.id, ins);
```

```
                ((NHIRPhiFunction) ins).inferType();
            }
        }
    }

    /**
     * Convert the bytecode in the specified list to their tuple
     * representations.
     *
     * @param code
     *            bytecode to convert.
     *
     * @return list of tuples.
     */

    private ArrayList<NTuple> bytecodeToTuples(ArrayList<Integer> code) {
        ArrayList<NTuple> tuples = new ArrayList<NTuple>();
        for (int i = 0; i < code.size(); i++) {
            int pc = i;
            int opcode = code.get(i);
            int operandBytes =
CLInstruction.instructionInfo[opcode].operandCount;
            short operandByte1, operandByte2, operandByte3, operandByte4;
            int pad, deflt;
            ArrayList<Short> operands = new ArrayList<Short>();
            switch (operandBytes) {
            case 0:
                break;
            case 1:
                operandByte1 = code.get(++i).shortValue();
                operands.add(operandByte1);
                break;
            case 2:
                operandByte1 = code.get(++i).shortValue();
                operandByte2 = code.get(++i).shortValue();
                operands.add(operandByte1);
                operands.add(operandByte2);
                break;
            case 3:
                operandByte1 = code.get(++i).shortValue();
                operandByte2 = code.get(++i).shortValue();
                operandByte3 = code.get(++i).shortValue();
                operands.add(operandByte1);
                operands.add(operandByte2);
                operands.add(operandByte3);
                break;
            case 4:
                operandByte1 = code.get(++i).shortValue();
                operandByte2 = code.get(++i).shortValue();
                operandByte3 = code.get(++i).shortValue();
                operandByte4 = code.get(++i).shortValue();
                operands.add(operandByte1);
                operands.add(operandByte2);
                operands.add(operandByte3);
                operands.add(operandByte4);
                break;
            case DYNAMIC: // TBD
                break;
            }
            tuples.add(new NTuple(pc, opcode, operands));
        }
        return tuples;
    }

    /**
     * Construct and return a short integer from two unsigned bytes specified.
     *
     * @param a
     *            unsigned byte.
```

```
1509     * @param b
1510     *            unsigned byte.
1511     *
1512     * @return a short integer constructed from the two unsigned bytes
1513     *         specified.
1514     */
1515
1516    private short shortValue(short a, short b) {
1517        return (short) ((a << 8) | b);
1518    }
1519
1520    /**
1521     * Construct and return an integer from the four unsigned bytes specified.
1522     *
1523     * @param a
1524     *            unsigned byte.
1525     * @param b
1526     *            unsigned byte.
1527     * @param c
1528     *            unsigned byte.
1529     * @param d
1530     *            unsigned byte.
1531     * @return an integer constructed from the four unsigned bytes specified.
1532     */
1533
1534    private int intValue(short a, short b, short c, short d) {
1535        return (a << 24) | (b << 16) | (c << 8) | d;
1536    }
1537
1538    /**
1539     * Extract and return the JVM bytecode for the method denoted by this cfg.
1540     *
1541     * @return JVM bytecode for the method denoted by this cfg.
1542     */
1543
1544    private ArrayList<Integer> getByteCode() {
1545        ArrayList<Integer> code = null;
1546        for (CLAttributeInfo info : m.attributes) {
1547            if (info instanceof CLCodeAttribute) {
1548                code = ((CLCodeAttribute) info).code;
1549                break;
1550            }
1551        }
1552        return code;
1553    }
1554
1555    /**
1556     * Return short form of the specified type descriptor.
1557     *
1558     * @param descriptor
1559     *            type descriptor.
1560     *
1561     * @return short form of type descriptor.
1562     */
1563
1564    private String shortType(String descriptor) {
1565        String sType = "V";
1566        char c = descriptor.charAt(0);
1567        switch (c) {
1568        case 'B':
1569        case 'C':
1570        case 'I':
1571        case 'F':
1572        case 'S':
1573        case 'Z':
1574        case 'J':
1575        case 'D':
1576            sType = c + "";
1577            break;
```

```
1578        case '[':
1579        case 'L':
1580            sType = "L";
1581            break;
1582        }
1583        return sType;
1584    }
1585
1586    /**
1587     * Return the number of local variables in the method denoted by this cfg.
1588     *
1589     * @return number of local variables.
1590     */
1591
1592    private int numLocals() {
1593        ArrayList<Integer> code = null;
1594        int numLocals = 0;
1595        for (CLAttributeInfo info : m.attributes) {
1596            if (info instanceof CLCodeAttribute) {
1597                code = ((CLCodeAttribute) info).code;
1598                numLocals = ((CLCodeAttribute) info).maxLocals;
1599                break;
1600            }
1601        }
1602        return numLocals;
1603    }
1604
1605    /**
1606     * Return the argument count (number of formal parameters) for the specified
1607     * method. 0 is returned if the descriptor is invalid.
1608     *
1609     * @param descriptor
1610     *            method descriptor.
1611     * @return argument count for the specified method.
1612     */
1613
1614    private int argumentCount(String descriptor) {
1615        int i = 0;
1616
1617        // Extract types of arguments and the return type from
1618        // the method descriptor
1619        String argTypes = descriptor.substring(1, descriptor.lastIndexOf(")"));
1620
1621        // Find number of arguments
1622        for (int j = 0; j < argTypes.length(); j++) {
1623            char c = argTypes.charAt(j);
1624            switch (c) {
1625            case 'B':
1626            case 'C':
1627            case 'I':
1628            case 'F':
1629            case 'S':
1630            case 'Z':
1631                i += 1;
1632                break;
1633            case '[':
1634                break;
1635            case 'J':
1636            case 'D':
1637                i += 2;
1638                break;
1639            case 'L':
1640                int k = argTypes.indexOf(";", j);
1641                j = k;
1642                i += 1;
1643                break;
1644            }
1645        }
1646        return i;
```

```
1647    }
1648
1649    /**
1650     * Return the argument count (number of formal parameters) for the specified
1651     * method. 0 is returned if the descriptor is invalid.
1652     *
1653     * @param descriptor
1654     *            method descriptor.
1655     * @return argument count for the specified method.
1656     */
1657
1658    private ArrayList<String> argumentTypes(String descriptor) {
1659        ArrayList<String> args = new ArrayList<String>();
1660        int i = 0;
1661
1662        // Extract types of arguments and the return type from
1663        // the method descriptor
1664        String argTypes = descriptor.substring(1, descriptor.lastIndexOf(")"));
1665
1666        String type = "";
1667
1668        // Find number of arguments
1669        for (int j = 0; j < argTypes.length(); j++) {
1670            char c = argTypes.charAt(j);
1671            switch (c) {
1672            case 'B':
1673            case 'C':
1674            case 'I':
1675            case 'F':
1676            case 'S':
1677            case 'Z':
1678            case 'J':
1679            case 'D':
1680                args.add(type + String.valueOf(c));
1681                type = "";
1682                break;
1683            case '[':
1684                type += c;
1685                break;
1686            case 'L':
1687                int k = argTypes.indexOf(";", j);
1688                args.add(type + argTypes.substring(j, k));
1689                type = "";
1690                j = k;
1691                break;
1692            }
1693        }
1694        return args;
1695    }
1696
1697    /**
1698     * Return the return type of a method given its descriptor.
1699     *
1700     * @param descriptor
1701     *            descriptor of the method.
1702     *
1703     * @return return type, "V" if void.
1704     */
1705
1706    private String returnType(String descriptor) {
1707        String returnType = descriptor
1708                .substring(descriptor.lastIndexOf(")") + 1);
1709        return returnType;
1710    }
1711
1712}
1713
```