NRegisterAllocator.java

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

package jminusminus;

import java.util.ArrayList;
import java.util.BitSet;

/**
 * A register allocator maps virtual registers (from LIR code) to physical
 * registers on the target machine. That there are a limited number of physical
 * registers makes this interesting.
 */

public abstract class NRegisterAllocator {

    /** The control flow graph for a method. */
    protected NControlFlowGraph cfg;

    /**
     * Construct an NRegisterAllocator object given the control flow graph for
     * method.
     * 
     * @param cfg
     *            control flow graph for a method.
     */

    protected NRegisterAllocator(NControlFlowGraph cfg) {
        this.cfg = cfg;
        this.cfg.intervals = new ArrayList<NInterval>();
        for (int i = 0; i < cfg.registers.size(); i++) {
            this.cfg.intervals.add(new NInterval(i, cfg));
        }
        this.cfg.maxIntervals = this.cfg.intervals.size();
    }

    /**
     * The work horse that does the allocation, implemented in the concrete
     * sub-classes of NRegisterAllocator.
     */

    public abstract void allocation();

    /**
     * Build the intervals for a control flow graph.
     */

    protected void buildIntervals() {
        this.computeLocalLiveSets();
        this.computeGlobalLiveSets();
        for (int i = cfg.basicBlocks.size() - 1; i >= 0; i--) {
            NBasicBlock currBlock = cfg.basicBlocks.get(i);
            if (currBlock.lir.size() == 0) {
                continue;
            }
            int blockStart = currBlock.lir.get(0).id;
            int blockEnd = currBlock.lir.get(currBlock.lir.size() - 1).id;
            BitSet liveOut = currBlock.liveOut;
            for (int idx = liveOut.nextSetBit(0); idx >= 0; idx = liveOut
                    .nextSetBit(idx + 1)) {
                cfg.intervals.get(idx).addOrExtendNRange(
                        new NRange(blockStart, blockEnd));
            }
            for (int j = currBlock.lir.size() - 1; j >= 0; j--) {
                int currLIRid = currBlock.lir.get(j).id;
                NRegister output = currBlock.lir.get(j).write;
                if (output != null) {
                    cfg.intervals.get(output.number).newFirstRangeStart(
                            currLIRid);
                    cfg.intervals.get(output.number).addUsePosition(currLIRid,
                            InstructionType.write);
                }
                ArrayList<NRegister> inputs = currBlock.lir.get(j).reads;
                for (NRegister reg : inputs) {
                    cfg.intervals.get(reg.number).addOrExtendNRange(
                            new NRange(blockStart, currLIRid));
                    cfg.intervals.get(reg.number).addUsePosition(currLIRid,
                            InstructionType.read);
                }
            }
        }
    }

    /**
     * Iterate through a list of basic blocks in order, and sets their liveUse
     * and liveDef BitSet fields to represent the appropriate virtual registers
     * that are locally defined to each block. It works internally with the
     * cfg's basicBlock structure.
     */

    private void computeLocalLiveSets() {
        for (NBasicBlock block : cfg.basicBlocks) {
            block.liveUse = new BitSet(cfg.registers.size());
            block.liveDef = new BitSet(cfg.registers.size());
            for (NLIRInstruction inst : block.lir) {
                for (NRegister reg : inst.reads) {
                    if (!block.liveDef.get(reg.number())) {
                        block.liveUse.set(reg.number());
                    }
                }
                if (inst.write != null) {
                    block.liveDef.set(inst.write.number());
                }
            }
        }
    }

    /**
     * Iterate through a list of basic blocks in reverse order, and sets their
     * lliveIn and liveOut bit sets to reflect global use-def information. It
     * works internally with the cfg's basicBlock structure.
     */

    private void computeGlobalLiveSets() {
        boolean changed = false;
        for (NBasicBlock b : cfg.basicBlocks) {
            b.liveOut = new BitSet(cfg.registers.size());
        }

        // note: we only check for changes in liveOut.
        do {
            changed = false;
            for (int i = cfg.basicBlocks.size() - 1; i >= 0; i--) {
                NBasicBlock currBlock = cfg.basicBlocks.get(i);
                BitSet newLiveOut = new BitSet(cfg.registers.size());
                for (NBasicBlock successor : currBlock.successors) {
                    newLiveOut.or(successor.liveIn);
                }
                if (!currBlock.liveOut.equals(newLiveOut)) {
                    currBlock.liveOut = newLiveOut;
                    changed = true;
                }
                currBlock.liveIn = (BitSet) currBlock.liveOut.clone();
                currBlock.liveIn.andNot(currBlock.liveDef);
                currBlock.liveIn.or(currBlock.liveUse);
            }
        } while (changed);
    }

}

```