NHIRInstruction.java

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

package jminusminus;

import static jminusminus.CLConstants.*;
import static jminusminus.NPhysicalRegister.*;
import java.util.ArrayList;

/**
 * High-level intermediate representation (HIR) of a JVM instruction.
 */

abstract class NHIRInstruction {

    /**
     * Maps JVM opcode to a string mnemonic for HIR instructions. For example,
     * the opcode imul is mapped to the string "*".
     */
    protected static String[] hirMnemonic;
    static {
        hirMnemonic = new String[256];
        hirMnemonic[IADD] = "+";
        hirMnemonic[ISUB] = "-";
        hirMnemonic[IMUL] = "*";
        hirMnemonic[MULTIANEWARRAY] = "multianewarray";
        hirMnemonic[AALOAD] = "aaload";
        hirMnemonic[IALOAD] = "iaload";
        hirMnemonic[IASTORE] = "iastore";
        hirMnemonic[IF_ICMPNE] = "!=";
        hirMnemonic[IF_ICMPGT] = ">";
        hirMnemonic[IF_ICMPLE] = "<=";
        hirMnemonic[GETSTATIC] = "getstatic";
        hirMnemonic[PUTSTATIC] = "putstatic";
        hirMnemonic[INVOKESPECIAL] = "invokespecial";
        hirMnemonic[INVOKESTATIC] = "invokestatic";
        hirMnemonic[ARETURN] = "areturn";
        hirMnemonic[RETURN] = "return";
        hirMnemonic[IRETURN] = "ireturn";
    }

    /** The block containing this instruction. */
    public NBasicBlock block;

    /** Unique identifier of this instruction. */
    public int id;

    /** Short type name for this instruction. */
    public String sType;

    /** Long type name for this instruction. */
    public String lType;

    /** The LIR instruction corresponding to this HIR instruction. */
    public NLIRInstruction lir;

    /**
     * Construct an NHIRInstruction object.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     */

    protected NHIRInstruction(NBasicBlock block, int id) {
        this(block, id, "", "");
    }

    /**
     * Construct an NHIRInstruction object.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param sType
     *            short type name of the instruction.
     * @param lType
     *            long type name of the instruction.
     */

    protected NHIRInstruction(NBasicBlock block, int id, String sType,
            String lType) {
        this.block = block;
        this.id = id;
        this.sType = sType;
        this.lType = lType;
    }

    /**
     * Return true if this instruction is the same as the other, false
     * otherwise. Two instructions are the same if their ids are the same.
     *
     * @param other
     *            the instruction to compare to.
     * @return true if the instructions are the same, false otherwise.
     */

    public boolean equals(NHIRInstruction other) {
        return this.id == other.id;
    }

    /**
     * Convert and return a low-level representation (LIR) of this HIR
     * instruction. Also adds the returned LIR instruction to the list of LIR
     * instructions for the block containing this instruction, along with any
     * other intermediate LIR instructions needed.
     *
     * @return LIR instruction corresponding to this HIR instruction.
     */

    public NLIRInstruction toLir() {
        return null;
    }

    /**
     * Return the identifier of this instruction with the short type name
     * prefixed.
     *
     * @return identifier of this IR instruction with the short type name
     *         prefixed.
     */

    public String id() {
        return sType + id;
    }

    /**
     * Return a string representation of this instruction.
     *
     * @return string representation of this instruction.
     */

    public String toString() {
        return sType + id;
    }

}

/**
 * HIR instruction corresponding to the JVM arithmetic instructions.
 */

class NHIRArithmetic extends NHIRInstruction {

    /** Opcode for the arithmetic operator. */
    public int opcode;

    /** Lhs HIR id. */
    public int lhs;

    /** Rhs HIR id. */
    public int rhs;

    /**
     * Construct an NHIRArithmetic instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            opcode for the arithmetic operator.
     * @param lhs
     *            lhs HIR id.
     * @param rhs
     *            rhs HIR id.
     */

    public NHIRArithmetic(NBasicBlock block, int id, int opcode, int lhs,
            int rhs) {
        super(block, id, "I", "I");
        this.opcode = opcode;
        this.lhs = lhs;
        this.rhs = rhs;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        NLIRInstruction ins1 = block.cfg.hirMap.get(lhs).toLir();
        NLIRInstruction ins2 = block.cfg.hirMap.get(rhs).toLir();
        lir = new NLIRArithmetic(block, NControlFlowGraph.lirId++, opcode,
                ins1, ins2);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + block.cfg.hirMap.get(lhs).id() + " "
                + hirMnemonic[opcode] + " " + block.cfg.hirMap.get(rhs).id();
    }

}
```

```java
/**
 * HIR instruction corresponding to the JVM instructions representing integer
 * constants.
 */

class NHIRIntConstant extends NHIRInstruction {

    /** The constant int value. */
    public int value;

    /**
     * Construct an NHIRIntConstant instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param value
     *            the constant int value.
     */

    public NHIRIntConstant(NBasicBlock block, int id, int value) {
        super(block, id, "I", "I");
        this.value = value;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        lir = new NLIRIntConstant(block, NControlFlowGraph.lirId++, value);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + value;
    }

}

/**
 * HIR instruction corresponding to the JVM instructions representing string
 * constants.
 */

class NHIRStringConstant extends NHIRInstruction {

    /** The constant string value. */
    public String value;

    /**
     * Construct an NHIRStringConstant instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier for the instruction.
     * @param value
     *            the constant string value.
     */

    public NHIRStringConstant(NBasicBlock block, int id, String value) {
        super(block, id, "L", "Ljava/lang/String;");
        this.value = value;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        lir = new NLIRStringConstant(block, NControlFlowGraph.lirId++, value);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + value;
    }

}

/**
 * HIR instruction representing an conditional jump instructions in JVM.
 */

class NHIRConditionalJump extends NHIRInstruction {

    /** Lhs HIR id. */
    public int lhs;

    /** Rhs HIR id. */
    public int rhs;

    /** Test expression opcode. */
    public int opcode;

    /** Block to jump to on true. */
    public NBasicBlock onTrueDestination;

    /** Block to jump to on false. */
    public NBasicBlock onFalseDestination;

    /**
     * Construct an NHIRConditionalJump instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param lhs
     *            Lhs HIR id.
     * @param rhs
     *            Rhs HIR id.
     * @param opcode
     *            opcode in the test.
     * @param onTrueDestination
     *            block to jump to on true.
     * @param onFalseDestination
     *            block to jump to on false.
     */

    public NHIRConditionalJump(NBasicBlock block, int id, int lhs, int rhs,
            int opcode, NBasicBlock onTrueDestination,
            NBasicBlock onFalseDestination) {
        super(block, id, "", "");
        this.lhs = lhs;
        this.rhs = rhs;
        this.opcode = opcode;
        this.onTrueDestination = onTrueDestination;
        this.onFalseDestination = onFalseDestination;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        NLIRInstruction ins1 = block.cfg.hirMap.get(lhs).toLir();
        NLIRInstruction ins2 = block.cfg.hirMap.get(rhs).toLir();
        lir = new NLIRConditionalJump(block, NControlFlowGraph.lirId++, ins1,
                ins2, opcode, onTrueDestination, onFalseDestination);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": if " + block.cfg.hirMap.get(lhs).id() + " "
                + hirMnemonic[opcode] + " " + block.cfg.hirMap.get(rhs).id()
                + " then " + onTrueDestination.id() + " else "
                + onFalseDestination.id();
    }

}

/**
 * HIR instruction representing an unconditional jump instruction in JVM.
 */

class NHIRGoto extends NHIRInstruction {

    /** The destination block to unconditionally jump to. */
    public NBasicBlock destination;

    /**
     * Construct an NHIRGoto instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param destination
     *            the block to jump to.
     */

    public NHIRGoto(NBasicBlock block, int id, NBasicBlock destination) {
        super(block, id, "", "");
        this.destination = destination;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        lir = new NLIRGoto(block, NControlFlowGraph.lirId++, destination);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": goto " + destination.id();
    }

}

/**
 * HIR instruction representing method invocation instructions in JVM.
 */

class NHIRInvoke extends NHIRInstruction {

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

    /** Target for the method. */
    public String target;

    /** Name of the method being invoked. */
    public String name;

    /** List of HIR ids of arguments for the method. */
    public ArrayList<Integer> arguments;

    /**
     * Construct an NHIRInvoke instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            opcode of the JVM instruction.
     * @param target
     *            target of the method.
     * @param name
     *            name of the method.
     * @param arguments
     *            list of HIR ids of arguments for the method.
     * @param sType
     *            return type (short name) of the method.
     * @param lType
     *            return type (long name) of the method.
     */

    public NHIRInvoke(NBasicBlock block, int id, int opcode, String target,
            String name, ArrayList<Integer> arguments, String sType,
            String lType) {
        super(block, id, sType, lType);
        this.opcode = opcode;
        this.target = target;
        this.name = name;
        this.arguments = arguments;
    }

    /**
```

```java
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }

        // First four arguments are stored in physical registers
        // (a0, ..., a3) and the rest are on the stack.
        // Allocate space on stack for arguments fourth or
        // above; [0, block.cfg.offset - 1].
        if (this.arguments.size() - 4 > block.cfg.offset) {
            block.cfg.offset = this.arguments.size() - 4;
        }

        ArrayList<NRegister> arguments = new ArrayList<NRegister>();
        ArrayList<NPhysicalRegister> froms = new ArrayList<NPhysicalRegister>();
        ArrayList<NVirtualRegister> tos = new ArrayList<NVirtualRegister>();
        for (int i = 0; i < this.arguments.size(); i++) {
            int arg = this.arguments.get(i);
            NLIRInstruction ins = block.cfg.hirMap.get(arg).toLir();
            if (i < 4) {
                // Generate an LIR move instruction (move1) to save
                // away the physical register a0 + i into a virtual
                // register, and another LIR move instruction (move2)
                // to copy the argument from the virtual register
                // it's in to the physical register a0 + i.
                String sType = block.cfg.hirMap.get(arg).sType;
                String lType = block.cfg.hirMap.get(arg).lType;
                NPhysicalRegister from = NPhysicalRegister.regInfo[A0 + i];
                block.cfg.registers.set(A0 + i, from);
                NVirtualRegister to = new NVirtualRegister(
                        NControlFlowGraph.regId++, sType, lType);
                block.cfg.registers.add(to);
                NLIRMove move1 = new NLIRMove(block, NControlFlowGraph.lirId++,
                        from, to);
                block.lir.add(move1);
                NLIRMove move2 = new NLIRMove(block, NControlFlowGraph.lirId++,
                        ins.write, from);
                block.lir.add(move2);
                arguments.add(NPhysicalRegister.regInfo[A0 + i]);

                // Remember the froms and the tos so we can restore
                // the
                // values of a0 + i registers.
                froms.add(from);
                tos.add(to);
            } else {
                NLIRStore store = new NLIRStore(block,
                        NControlFlowGraph.lirId++, i - 4, OffsetFrom.SP,
                        ins.write);
                block.lir.add(store);
                arguments.add(ins.write);
            }
        }

        lir = new NLIRInvoke(block, NControlFlowGraph.lirId++, opcode, target,
                name, arguments, sType, lType);
        block.lir.add(lir);

        // If the function returns a value, generate an LIR move
        // instruction to save away the value in the physical
        // register v0 into a virtual register.
        if (lir.write != null) {
            NVirtualRegister to = new NVirtualRegister(
                    NControlFlowGraph.regId++, sType, lType);
            NLIRMove move = new NLIRMove(block, NControlFlowGraph.lirId++,
                    NPhysicalRegister.regInfo[V0], to);
            block.cfg.registers.add(to);
            block.lir.add(move);
            lir = move;
        }

        // Generate LIR move instructions to restore the a0, ..., a3
        // instructions.
        for (int i = 0; i < tos.size(); i++) {
            NLIRMove move = new NLIRMove(block, NControlFlowGraph.lirId++, tos
                    .get(i), froms.get(i));
            block.lir.add(move);
        }

        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        String s = id() + ": " + hirMnemonic[opcode] + " " + target + "."
                + name + "( ";
        for (int arg : arguments) {
            s += block.cfg.hirMap.get(arg).id() + " ";
        }
        s += ")";
        return s;
    }

}

/**
 * HIR instruction representing a JVM return instruction.
 */

class NHIRReturn extends NHIRInstruction {

    /** JVM opcode for the return instruction. */
    public int opcode;

    /** Return value HIR id. */
    public int value;

    /**
     * Construct an NHIRReturn instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the return instruction.
     * @param value
     *            return value HIR id.
     */

    public NHIRReturn(NBasicBlock block, int id, int opcode, int value) {
        super(block, id,
                (value == -1) ? "" : block.cfg.hirMap.get(value).sType,
                (value == -1) ? "" : block.cfg.hirMap.get(value).lType);
        this.opcode = opcode;
        this.value = value;
    }

    /**
     * @inheritDoc
     */

```

```java
    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        NLIRInstruction result = null;
        if (value != -1) {
            result = block.cfg.hirMap.get(value).toLir();
            NLIRMove move = new NLIRMove(block, NControlFlowGraph.lirId++,
                    result.write, NPhysicalRegister.regInfo[V0]);
            block.lir.add(move);
            block.cfg.registers.set(V0, NPhysicalRegister.regInfo[V0]);
        }
        lir = new NLIRReturn(block, NControlFlowGraph.lirId++, opcode,
                (result == null) ? null : NPhysicalRegister.regInfo[V0]);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        if (value == -1) {
            return id() + ": " + hirMnemonic[opcode];
        }
        return id() + ": " + hirMnemonic[opcode] + " "
                + block.cfg.hirMap.get(value).id();
    }

}

/**
 * HIR instruction representing JVM (put) field instructions.
 */

class NHIRPutField extends NHIRInstruction {

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

    /** Target for the field. */
    public String target;

    /** Name of the field being accessed. */
    public String name;

    /** HIR id of the value of the field. */
    public int value;

    /**
     * Construct an NHIRPutField instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param target
     *            target for the field.
     * @param name
     *            name of the field.
     * @param sType
     *            type (short name) of the field.
     * @param lType
     *            type (long name) of the field.
     * @param value
     *            HIR id of the value of the field.
     */

    public NHIRPutField(NBasicBlock block, int id, int opcode, String target,
            String name, String sType, String lType, int value) {
        super(block, id, sType, lType);
        this.opcode = opcode;
        this.target = target;
        this.name = name;
        this.value = value;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        NLIRInstruction result = block.cfg.hirMap.get(value).toLir();
        lir = new NLIRPutField(block, NControlFlowGraph.lirId++, opcode,
                target, name, sType, lType, result);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + hirMnemonic[opcode] + " " + target + "." + name
                + " = " + block.cfg.hirMap.get(value).id();
    }

}

/**
 * HIR instruction representing JVM (get) field instructions.
 */

class NHIRGetField extends NHIRInstruction {

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

    /** Target for the field. */
    public String target;

    /** Name of the field being accessed. */
    public String name;

    /**
     * Construct an NHIRGetField instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param target
     *            target for the field.
     * @param name
     *            name of the field.
     * @param sType
     *            type (short name) of the field.
     * @param lType
     *            type (long name) of the field.
     */

    public NHIRGetField(NBasicBlock block, int id, int opcode, String target,
            String name, String sType, String lType) {
        super(block, id, sType, lType);
        this.opcode = opcode;
        this.target = target;
        this.name = name;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        lir = new NLIRGetField(block, NControlFlowGraph.lirId++, opcode,
                target, name, sType, lType);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + hirMnemonic[opcode] + " " + target + "." + name;
    }

}

/**
 * HIR instruction representing JVM array creation instructions.
 */

class NHIRNewArray extends NHIRInstruction {

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

    /** Dimension of the array. */
    public int dim;

    /**
     * Construct an NHIRNewArray instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param dim
     *            dimension of the array.
     * @param sType
     *            type (short name) of the array.
     * @param lType
     *            type (long name) of the array.
     */

    public NHIRNewArray(NBasicBlock block, int id, int opcode, int dim,
            String sType, String lType) {
        super(block, id, lType, sType);
        this.opcode = opcode;
        this.dim = dim;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        lir = new NLIRNewArray(block, NControlFlowGraph.lirId++, opcode, dim,
                sType, lType);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + hirMnemonic[opcode] + " " + lType + " [" + dim
                + "]";
    }

}

/**
 * HIR instruction representing JVM array load instructions.
 */

class NHIRALoad extends NHIRInstruction {

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

    /** HIR id of the array reference. */
    public int arrayRef;

    /** HIR id of the array index. */
    public int index;

    /**
     * Construct an NHIRALoad instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param arrayRef
     *            HIR id of the array reference.
     * @param index
     *            HIR id of the the array index.
     * @param sType
     *            type (short name) of the array.
     * @param lType
     *            type (long name) of the array.
     */

    public NHIRALoad(NBasicBlock block, int id, int opcode, int arrayRef,
            int index, String sType, String lType) {
        super(block, id, sType, lType);
        this.opcode = opcode;
        this.arrayRef = arrayRef;
        this.index = index;
    }

    /**
```

```
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        NLIRInstruction arrayRef = block.cfg.hirMap.get(this.arrayRef).toLir();
        NLIRInstruction index = block.cfg.hirMap.get(this.index).toLir();
        lir = new NLIRALoad(block, NControlFlowGraph.lirId++, opcode, arrayRef,
                index, sType, lType);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + hirMnemonic[opcode] + " "
                + block.cfg.hirMap.get(arrayRef).id() + "["
                + block.cfg.hirMap.get(index).id() + "]";
    }

}

/**
 * HIR instruction representing JVM array store instructions.
 */

class NHIRAStore extends NHIRInstruction {

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

    /** HIR id of the array reference. */
    public int arrayRef;

    /** HIR id of the array index. */
    public int index;

    /** HIR id of the value to store. */
    public int value;

    /**
     * Construct an NHIRAStore instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param arrayRef
     *            HIR id of the array reference.
     * @param index
     *            HIR id of the array index.
     * @param value
     *            HIR id of the value to store.
     * @param sType
     *            type (short name) of the array.
     * @param lType
     *            type (long name) of the array.
     */

    public NHIRAStore(NBasicBlock block, int id, int opcode, int arrayRef,
            int index, int value, String sType, String lType) {
        super(block, id, sType, lType);
```

```java
        this.opcode = opcode;
        this.arrayRef = arrayRef;
        this.index = index;
        this.value = value;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        NLIRInstruction arrayRef = block.cfg.hirMap.get(this.arrayRef).toLir();
        NLIRInstruction index = block.cfg.hirMap.get(this.index).toLir();
        NLIRInstruction value = block.cfg.hirMap.get(this.value).toLir();
        lir = new NLIRAStore(block, NControlFlowGraph.lirId++, opcode,
                arrayRef, index, value, sType, lType);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": " + hirMnemonic[opcode] + " "
                + block.cfg.hirMap.get(arrayRef).id() + "["
                + block.cfg.hirMap.get(index).id() + "] = "
                + block.cfg.hirMap.get(value).id();
    }

}

/**
 * HIR instruction representing phi functions.
 */

class NHIRPhiFunction extends NHIRInstruction {

    /** List of HIR ids of arguments for the phi function. */
    public ArrayList<Integer> arguments;

    /** Local variable index. */
    public int local;

    /**
     * Construct an NHIRPhiFunction instruction.
     *
     * @param block
     *                  enclosing block.
     * @param id
     *                  identifier of the instruction.
     * @param arguments
     *                  list of HIR ids of arguments for the phi function.
     * @param local
     *                  local variable index.
     */

    public NHIRPhiFunction(NBasicBlock block, int id,
            ArrayList<Integer> arguments, int local) {
        super(block, id, "", "");
        this.arguments = arguments;
        this.local = local;
    }

    /**
```

```java
     * Infer type for this phi function. It is essentially the type of the
     * arguments.
     */

    public void inferType() {
        for (int arg : arguments) {
            if (!block.cfg.hirMap.get(arguments.get(0)).sType.equals("")) {
                sType = block.cfg.hirMap.get(arguments.get(0)).sType;
                lType = block.cfg.hirMap.get(arguments.get(0)).lType;
                break;
            }
        }
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        lir = new NLIRPhiFunction(block, NControlFlowGraph.lirId++, sType,
                lType);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        String s = "[ ";
        for (int ins : arguments) {
            if (block.cfg.hirMap.get(ins) != null)
                s += block.cfg.hirMap.get(ins).sType + ins + " ";
        }
        s += "]";
        return s;
    }

}

/**
 * HIR instruction representing a formal parameter.
 */

class NHIRLoadLocal extends NHIRInstruction {

    /** Local variable index. */
    public int local;

    /**
     * Construct an NHIRLoadLocal instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param local
     *            local variable index.
     * @param sType
     *            short type name of the instruction.
     * @param lType
     *            long type name of the instruction.
     */

    public NHIRLoadLocal(NBasicBlock block, int id, int local, String sType,
            String lType) {
```

```java
        super(block, id, sType, lType);
        this.local = local;
    }

    /**
     * @inheritDoc
     */

    public NLIRInstruction toLir() {
        if (lir != null) {
            return lir;
        }
        lir = new NLIRLoadLocal(block, NControlFlowGraph.lirId++, local, sType,
                lType);
        block.lir.add(lir);
        return lir;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": LDLOC " + local;
    }

}

/**
 * HIR instruction representing a local (not formal) variable.
 */

class NHIRLocal extends NHIRInstruction {

    /** Local variable index. */
    public int local;

    /**
     * Construct an NHIRLocal instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param local
     *            local variable index.
     * @param sType
     *            short type name of the instruction.
     * @param lType
     *            long type name of the instruction.
     */

    public NHIRLocal(NBasicBlock block, int id, int local, String sType,
            String lType) {
        super(block, id, sType, lType);
        this.local = local;
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id() + ": LOC " + lType;
    }

}
```