

NControlFlowGraph.java

```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import static jminusminus.CLConstants.*;
6  import java.util.ArrayList;
7  import java.util.BitSet;
8  import java.util.HashMap;
9  import java.util.LinkedList;
10 import java.util.Stack;
11 import java.util.TreeMap;
12 import java.util.Queue;
13
14 /**
15  * A tuple representation of a JVM instruction.
16  */
17
18 class NTuple {
19
20     /** Program counter of the instruction. */
21     public int pc;
22
23     /** Opcode of the instruction. */
24     public int opcode;
25
26     /** Operands of the instructions. */
27     public ArrayList<Short> operands;
28
29     /** String representation (mnemonic) of the instruction. */
30     public String mnemonic;
31
32     /** Is this tuple the leader of the block containing it. */
33     public boolean isLeader;
34
35     /**
36      * Construct a tuple representing the JVM instruction with the given program
37      * counter, opcode, and operand list.
38      *
39      * @param pc
40      *         program counter.
41      * @param opcode
42      *         opcode of the instruction.
43      * @param operands
44      *         list of operands of the instruction.
45      */
46
47     public NTuple(int pc, int opcode, ArrayList<Short> operands) {
48         this.pc = pc;
49         this.opcode = opcode;
50         this.operands = operands;
51         this.mnemonic = CLInstruction.instructionInfo[opcode].mnemonic;
52         this.isLeader = false;
53     }
54
55     /**
56      * Write the information pertaining to this tuple to STDOUT.
57      *
58      * @param p
59      *         for pretty printing with indentation.
60      */
61
62     public void writeToStdOut(PrettyPrinter p) {
63         p.printf("%s: %s", pc, mnemonic);
64         for (short s : operands) {
65             p.printf(" %s", s);
66         }
67     }
68 }
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

67         p.printf("\n");
68     }
69 }
70
71 /**
72  * Representation of a block within a control flow graph.
73  */
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 this 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. */

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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.instructions = new ArrayList<NLTInstruction>();
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  */

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

205     * @param p
206     *         for pretty printing with indentation.
207     */
208
209     public void writeTuplesToStdOut(PrettyPrinter p) {
210         String s = id();
211         p.printf("%s\n", s);
212         for (NTuple tuple : tuples) {
213             tuple.writeToStdOut(p);
214         }
215         p.printf("\n");
216     }
217
218     /**
219     * Write the HIR instructions in this block to STDOUT.
220     *
221     * @param p
222     *         for pretty printing with indentation.
223     */
224
225     public void writeHirToStdOut(PrettyPrinter p) {
226         String s = id() + (isLoopHead ? " [LH]" : "")
227             + (isLoopTail ? " [LT]" : "");
228         if (tuples.size() > 0) {
229             s += " [" + tuples.get(0).pc + ", "
230                 + tuples.get(tuples.size() - 1).pc + "]";
231         }
232         if (dom != null) {
233             s += " dom: " + dom.id();
234         }
235         if (predecessors.size() > 0) {
236             s += " pred: ";
237             for (NBasicBlock block : predecessors) {
238                 s += block.id() + " ";
239             }
240         }
241         if (successors.size() > 0) {
242             s += " succ: ";
243             for (NBasicBlock block : successors) {
244                 s += block.id() + " ";
245             }
246         }
247         p.printf(s + "\n");
248         s = "Locals: ";
249         if (locals != null) {
250             for (int i = 0; i < locals.length; i++) {
251                 if (!(cfg.hirMap.get(locals[i]) instanceof NHIRLocal)) {
252                     s += cfg.hirMap.get(locals[i]).id() + " ";
253                 }
254             }
255         }
256         p.printf("%s\n", s);
257         for (int ins : hir) {
258             if (cfg.hirMap.get(ins) instanceof NHIRPhiFunction) {
259                 p.printf("%s: %s\n", ((NHIRPhiFunction) cfg.hirMap.get(ins))
260                     .id(), ((NHIRPhiFunction) cfg.hirMap.get(ins)));
261             }
262         }
263         for (int ins : hir) {
264             if (!(cfg.hirMap.get(ins) instanceof NHIRPhiFunction)) {
265                 p.printf("%s\n", cfg.hirMap.get(ins));
266             }
267         }
268         p.printf("\n");
269     }
270
271     /**
272     * Write the LIR instructions in this block to STDOUT.
273     *

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

274     * @param p
275     *         for pretty printing with indentation.
276     */
277     public void writeLirToStdOut(PrettyPrinter p) {
278         p.printf("%s\n", id());
279         for (NLIRInstruction ins : lir) {
280             p.printf("%s\n", ins);
281         }
282         p.printf("\n");
283     }
284
285     /**
286     * The instruction identifier for the first LIR instruction.
287     *
288     * @return the instruction identifier.
289     */
290     public int getFirstLIRInstId() {
291         if (lir.isEmpty()) {
292             return -1;
293         }
294         return lir.get(0).id;
295     }
296
297     /**
298     * The instruction identifier for the last LIR instruction.
299     *
300     * @return the instruction identifier.
301     */
302     public int getLastLIRInstId() {
303         if (lir.isEmpty()) {
304             return -1;
305         }
306         return lir.get(lir.size() - 1).id;
307     }
308
309     /**
310     * Iterates through the lir array of this block, returning an
311     * NLIRInstruction with the specified id.
312     *
313     * @param id
314     *         the id to look for.
315     *
316     * @return NLIRInstruction with the specified id, null if none matched.
317     */
318     public NLIRInstruction getInstruction(int id) {
319         for (NLIRInstruction i : this.lir) {
320             if (i.id == id) {
321                 return i;
322             }
323         }
324         return null;
325     }
326
327     /**
328     * Checks to see if there is an LIRInstruction with this id in the block's
329     * lir.
330     *
331     * @return true or false.
332     */
333     public boolean idIsFree(int id) {
334         if (this.getInstruction(id) != null) {
335             return false;
336         } else {
337             return true;
338         }
339     }
340
341     /**
342     * Inserts an NLIRInstruction to the appropriate place in this block's lir

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

343     * array based on its id -- preserving order by id.
344     *
345     * @param inst
346     *         the NLIRInstruction to be inserted.
347     */
348     public void insertNLIRInst(NLIRInstruction inst) {
349         int idx = -1;
350         for (int i = 0; i < this.lir.size(); i++) {
351             if (this.lir.get(i).id < inst.id) {
352                 idx = i;
353             }
354         }
355         if (++idx == this.lir.size()) {
356             this.lir.add(inst);
357         } else {
358             this.lir.add(idx, inst);
359         }
360     }
361 }
362
363 /**
364  * Representation of a control flow graph (cfg) for a method.
365  */
366
367 class NControlFlowGraph {
368
369     /** Constant pool for the class containing the method. */
370     private ClConstantPool cp;
371
372     /** Contains information about the method. */
373     private CLMethodInfo m;
374
375     /** Maps the pc of a JVM instruction to the block it's in. */
376     private HashMap<Integer, NBasicBlock> pcToBasicBlock;
377
378     /** block identifier. */
379     public static int blockId;
380
381     /** HIR instruction identifier. */
382     public static int hirId;
383
384     /** HIR instruction identifier. */
385     public static int lirId;
386
387     /** Virtual register identifier. */
388     public static int regId;
389
390     /** Stack offset counter.. */
391     public int offset;
392
393     /** Loop identifier. */
394     public static int loopIndex;
395
396     /** Name of the method this cfg corresponds to. */
397     public String name;
398
399     /** Descriptor of the method this cfg corresponds to. */
400     public String desc;
401
402     /**
403      * List of blocks forming the cfg for the method.
404      */
405     public ArrayList<NBasicBlock> basicBlocks;
406
407     /** Maps HIR instruction ids in this cfg to HIR instructions. */
408     public TreeMap<Integer, NHIRInstruction> hirMap;
409
410     /**

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

412     * Registers allocated for this cfg by the HIR to LIR conversion algorithm.
413     */
414     public ArrayList<NRegister> registers;
415
416     /**
417     * The total number of intervals. This is used to name split children and
418     * grows as more intervals are created by spills.
419     */
420     public int maxIntervals;
421
422     /**
423     * Physical registers allocated for this cfg by the HIR to LIR conversion
424     * algorithm.
425     */
426     public ArrayList<NPhysicalRegister> pRegisters;
427
428     /**
429     * Intervals allocated by the register allocation algorithm.
430     */
431     public ArrayList<NInterval> intervals;
432
433     /** Used to construct jump labels in spim output. */
434     public String labelPrefix;
435
436     /**
437     * SPIM code for string literals added to the data segment.
438     */
439     public ArrayList<String> data;
440
441     /**
442     * Construct an NControlFlowGraph object for a method given the constant
443     * pool for the class containing the method and the object containing
444     * information about the method.
445     *
446     * @param cp      constant pool for the class containing the method.
447     * @param m        contains information about the method.
448     */
449
450     public NControlFlowGraph(CLConstantPool cp, CLMethodInfo m) {
451         this.cp = cp;
452         this.m = m;
453         name = new String(((CLConstantUtf8Info) cp.cpItem(m.nameIndex)).b);
454         desc = new String(((CLConstantUtf8Info) cp.cpItem(m.descriptorIndex)).b);
455         basicBlocks = new ArrayList<NBasicBlock>();
456         pcToBasicBlock = new HashMap<Integer, NBasicBlock>();
457         ArrayList<Integer> code = getByteCode();
458         ArrayList<NTuple> tuples = bytecodeToTuples(code);
459         if (tuples.size() == 0) {
460             return;
461         }
462         NTuple[] tupleAt = new NTuple[code.size()];
463         for (NTuple tuple : tuples) {
464             tupleAt[tuple.pc] = tuple;
465         }
466
467         // Identify the leaders.
468         tuples.get(0).isLeader = true;
469         for (int j = 1; j < tuples.size(); j++) {
470             NTuple tuple = tuples.get(j);
471             boolean jumpInstruction = true;
472             short operandByte1, operandByte2, operandByte3, operandByte4;
473             int offset;
474             switch (tuple.opcode) {
475                 case IFEQ:
476                 case IFNE:
477                 case IFLT:
478                 case IFGE:

```

```

481     case IFGT:
482     case IFLE:
483     case IF_ICMPEQ:
484     case IF_ICMPNE:
485     case IF_ICMPLT:
486     case IF_ICMPGE:
487     case IF_ICMPGT:
488     case IF_ICMPLE:
489     case IF_ACMPEQ:
490     case IF_ACMPLT:
491     case GOTO:
492     case JSR:
493     case IFNULL:
494     case IFNONNULL:
495         operandByte1 = tuple.operands.get(0);
496         operandByte2 = tuple.operands.get(1);
497         offset = shortValue(operandByte1, operandByte2);
498         tupleAt[tuple.pc + offset].isLeader = true;
499         break;
500     case GOTO_W:
501     case JSR_W:
502         operandByte1 = tuple.operands.get(0);
503         operandByte2 = tuple.operands.get(1);
504         operandByte3 = tuple.operands.get(2);
505         operandByte4 = tuple.operands.get(3);
506         offset = intValue(operandByte1, operandByte2, operandByte3,
507             operandByte4);
508         tupleAt[tuple.pc + offset].isLeader = true;
509         break;
510     case LRETURN:
511     case LRETURN:
512     case FRETURN:
513     case DRETURN:
514     case ARETURN:
515     case RETURN:
516     case RET:
517     case ATHROW:
518         break;
519     case TABLESWITCH: // TBD
520         break;
521     case LOOKUPSWITCH: // TBD
522         break;
523     default:
524         jumpInstruction = false;
525     }
526     if (jumpInstruction) {
527         if (j < tuples.size() - 1) {
528             tuples.get(j + 1).isLeader = true;
529         }
530     }
531 }
532
533 // Form blocks.
534 {
535     blockId = 0;
536     NBasicBlock block = new NBasicBlock(this, blockId++);
537     for (NTuple tuple : tuples) {
538         if (tuple.isLeader) {
539             basicBlocks.add(block);
540             block = new NBasicBlock(this, blockId++);
541             if (!pcToBasicBlock.containsKey(tuple.pc)) {
542                 pcToBasicBlock.put(tuple.pc, block);
543             }
544         }
545         block.tuples.add(tuple);
546     }
547     basicBlocks.add(block);
548 }
549

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```

550 // Connect up the blocks for this method, that is, build
551 // its control flow graph.
552 basicBlocks.get(0).successors.add(basicBlocks.get(1));
553 basicBlocks.get(1).predecessors.add(basicBlocks.get(0));
554 NBasicBlock[] blockAt = new NBasicBlock[code.size()];
555 for (NBasicBlock block : basicBlocks) {
556     if (block.tuples.size() == 0) {
557         continue;
558     }
559     blockAt[block.tuples.get(0).pc] = block;
560 }
561 for (int j = 0; j < basicBlocks.size(); j++) {
562     NBasicBlock block = basicBlocks.get(j);
563     if (block.tuples.size() == 0) {
564         continue;
565     }
566     NTuple tuple = block.tuples.get(block.tuples.size() - 1);
567     short operandByte1, operandByte2, operandByte3, operandByte4;
568     int offset;
569     NBasicBlock target;
570     switch (tuple.opcode) {
571     case IFEQ:
572     case IFNE:
573     case IFLT:
574     case IFGE:
575     case IFGT:
576     case IFLE:
577     case IF_ICMPEQ:
578     case IF_ICMPNE:
579     case IF_ICMPLT:
580     case IF_ICMPGE:
581     case IF_ICMPGT:
582     case IF_ICMPLE:
583     case IF_ACMPEQ:
584     case IF_ACMPLT:
585     case IFNULL:
586     case IFNONNULL:
587         operandByte1 = tuple.operands.get(0);
588         operandByte2 = tuple.operands.get(1);
589         offset = shortValue(operandByte1, operandByte2);
590         target = blockAt[tuple.pc + offset];
591         if (j < basicBlocks.size() - 1) {
592             block.successors.add(basicBlocks.get(j + 1));
593             basicBlocks.get(j + 1).predecessors.add(block);
594         }
595         block.successors.add(target);
596         target.predecessors.add(block);
597         break;
598     case GOTO:
599     case JSR:
600         operandByte1 = tuple.operands.get(0);
601         operandByte2 = tuple.operands.get(1);
602         offset = shortValue(operandByte1, operandByte2);
603         target = blockAt[tuple.pc + offset];
604         block.successors.add(target);
605         target.predecessors.add(block);
606         break;
607     case GOTO_W:
608     case JSR_W:
609         operandByte1 = tuple.operands.get(0);
610         operandByte2 = tuple.operands.get(1);
611         operandByte3 = tuple.operands.get(2);
612         operandByte4 = tuple.operands.get(3);
613         offset = intValue(operandByte1, operandByte2, operandByte3,
614             operandByte4);
615         target = blockAt[tuple.pc + offset];
616         block.successors.add(target);
617         target.predecessors.add(block);
618         break;

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

619         case IRETURN:
620         case LRETURN:
621         case FRETURN:
622         case DRETURN:
623         case ARETURN:
624         case RETURN:
625         case RET:
626         case ATHROW:
627             break;
628         case TABLESWITCH: // TBD
629             break;
630         case LOOKUPSWITCH: // TBD
631             break;
632         default:
633             if (j < basicBlocks.size() - 1) {
634                 block.successors.add(basicBlocks.get(j + 1));
635                 basicBlocks.get(j + 1).predecessors.add(block);
636             }
637     }
638 }
639
640 // Calculate the ref count and number of forward branches
641 // to each block in the this cfg.
642 for (NBasicBlock block : basicBlocks) {
643     block.ref = block.predecessors.size();
644     block.fwdBranches = block.predecessors.size() - block.bwdBranches;
645 }
646 }
647
648 /**
649  * Implements loop detection algorithm to figure out if the specified block
650  * is a loop head or a loop tail. Also calculates the number of backward
651  * branches to the block.
652  *
653  * @param block
654  *     a block.
655  * @param pred
656  *     block's predecessor or null.
657  */
658
659 public void detectLoops(NBasicBlock block, NBasicBlock pred) {
660     if (!block.visited) {
661         block.visited = true;
662         block.active = true;
663         for (NBasicBlock succ : block.successors) {
664             detectLoops(succ, block);
665         }
666         block.active = false;
667     } else if (block.active) {
668         block.isLoopHead = true;
669         pred.isLoopTail = true;
670         block.bwdBranches++;
671         block.loopIndex = NControlFlowGraph.loopIndex++;
672     }
673 }
674
675 /**
676  * Remove blocks that cannot be reached from the begin block (B0). Also
677  * removes these blocks from the predecessor lists.
678  */
679
680 public void removeUnreachableBlocks() {
681     // Create a list of blocks that cannot be reached.
682     ArrayList<NBasicBlock> toRemove = new ArrayList<NBasicBlock>();
683     for (NBasicBlock block : basicBlocks) {
684         if (!block.visited) {
685             toRemove.add(block);
686         }
687     }

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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;

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

757 Stack<Integer> operandStack = new Stack<Integer>();
758 Queue<NBasicBlock> q = new LinkedList<NBasicBlock>();
759 beginBlock.visited = true;
760 q.add(beginBlock);
761 while (!q.isEmpty()) {
762     NBasicBlock block = q.remove();
763     for (NBasicBlock succ : block.successors) {
764         if (!succ.visited) {
765             succ.visited = true;
766             q.add(succ);
767         }
768     }
769
770     // Convert tuples in block to HIR instructions.
771     if (block.predecessors.size() == 1) {
772         block.locals = block.predecessors.get(0).locals.clone();
773     } else if (block.predecessors.size() > 1) {
774         if (block.isLoopHead) {
775             for (NBasicBlock pred : block.predecessors) {
776                 if (pred.locals != null) {
777                     block.locals = pred.locals.clone();
778                     break;
779                 }
780             }
781             for (int i = 0; i < block.locals.length; i++) {
782                 ArrayList<Integer> args = new ArrayList<Integer>();
783                 NHIRPhiFunction phi = new NHIRPhiFunction(block,
784                     hirId++, args, i);
785                 block.hir.add(phi.id);
786                 block.cfg.hirMap.put(phi.id, phi);
787                 phi.arguments.add(block.locals[i]);
788                 for (int j = 1; j < block.predecessors.size(); j++) {
789                     phi.arguments.add(phi.id);
790                 }
791                 block.locals[i] = phi.id;
792                 phi.inferType();
793             }
794         } else {
795             block.locals = block.predecessors.get(0).locals.clone();
796             for (int i = 1; i < block.predecessors.size(); i++) {
797                 NBasicBlock pred = block.predecessors.get(i);
798                 mergeLocals(block, pred);
799             }
800         }
801     }
802     for (NTuple tuple : block.tuples) {
803         CLInsInfo insInfo = CLInstruction.instructionInfo[tuple.opcode];
804         int localVariableIndex = insInfo.localVariableIndex;
805         NHIRInstruction ins = null;
806         short operandByte1 = 0, operandByte2 = 0, operandByte3 = 0,
offset = 0;
807         int operand1 = 0, operand2 = 0, operand3 = 0;
808         switch (insInfo.opcode) {
809             case MULTIANEWARRAY: {
810                 operandByte1 = tuple.operands.get(0);
811                 operandByte2 = tuple.operands.get(1);
812                 operandByte3 = tuple.operands.get(2);
813                 int index = shortValue(operandByte1, operandByte2);
814                 int classIndex = ((CLConstantClassInfo)
cp.cpItem(index)).nameIndex;
815                 String type = new String(((CLConstantUtf8Info) cp
816                     .cpItem(classIndex)).b);
817                 ins = new NHIRNewArray(block, hirId++, insInfo.opcode,
818                     (int) operandByte3, shortType(type), type);
819                 block.cfg.hirMap.put(ins.id, ins);
820                 block.hir.add(ins.id);
821                 operandStack.push(ins.id);
822                 break;
823             }

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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++,

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

893         12);
894         NHIRInstruction ins2 = new NHIRArithmetic(block, hirId++,
895             IADD, operand1, ins1.id);
896         block.cfg.hirMap.put(ins1.id, ins1);
897         block.hir.add(ins1.id);
898         block.cfg.hirMap.put(ins2.id, ins2);
899         block.hir.add(ins2.id);
900
901         // Compute index.
902         NHIRInstruction ins3 = new NHIRIntConstant(block, hirId++,
903             4);
904         NHIRInstruction ins4 = new NHIRArithmetic(block, hirId++,
905             IMUL, operand2, ins3.id);
906         block.cfg.hirMap.put(ins3.id, ins3);
907         block.hir.add(ins3.id);
908         block.cfg.hirMap.put(ins4.id, ins4);
909         block.hir.add(ins4.id);
910
911         ins = new NHIRASTore(block, hirId++, insInfo.opcode,
912             ins2.id, ins4.id, operand3, "I", "I");
913         block.cfg.hirMap.put(ins.id, ins);
914         block.hir.add(ins.id);
915         break;
916     }
917     case ICONST_0:
918     case ICONST_1:
919     case ICONST_2:
920     case ICONST_3:
921     case ICONST_4:
922     case ICONST_5: {
923         ins = new NHIRIntConstant(block, hirId++, tuple.opcode - 3);
924         block.cfg.hirMap.put(ins.id, ins);
925         block.hir.add(ins.id);
926         operandStack.push(ins.id);
927         break;
928     }
929     case ILOAD: {
930         operandByte1 = tuple.operands.get(0);
931         localVariableIndex = operandByte1;
932         operandStack.push(block.locals[localVariableIndex]);
933         break;
934     }
935     case ILOAD_0:
936     case ILOAD_1:
937     case ILOAD_2:
938     case ILOAD_3:
939     case ALOAD_0:
940     case ALOAD_1:
941     case ALOAD_2:
942     case ALOAD_3: {
943         operandStack.push(block.locals[localVariableIndex]);
944         break;
945     }
946     case ISTORE: {
947         operandByte1 = tuple.operands.get(0);
948         localVariableIndex = operandByte1;
949         block.locals[localVariableIndex] = operandStack.pop();
950         break;
951     }
952     case ISTORE_0:
953     case ISTORE_1:
954     case ISTORE_2:
955     case ISTORE_3:
956     case ASTORE_0:
957     case ASTORE_1:
958     case ASTORE_2:
959     case ASTORE_3: {
960         block.locals[localVariableIndex] = operandStack.pop();
961         break;

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

962     }
963     case BIPUSH: {
964         operandByte1 = tuple.operands.get(0);
965         ins = new NHIRIntConstant(block, hirId++, operandByte1);
966         block.cfg.hirMap.put(ins.id, ins);
967         block.hir.add(ins.id);
968         operandStack.push(ins.id);
969         break;
970     }
971     case SIPUSH: {
972         operandByte1 = tuple.operands.get(0);
973         operandByte2 = tuple.operands.get(1);
974         ins = new NHIRIntConstant(block, hirId++, shortValue(
975             operandByte1, operandByte2));
976         block.cfg.hirMap.put(ins.id, ins);
977         block.hir.add(ins.id);
978         operandStack.push(ins.id);
979         break;
980     }
981     case LDC: {
982         operandByte1 = tuple.operands.get(0);
983
984         // Only allowing ldc of string constants for
985         // now.
986         int stringIndex = ((CLConstantStringInfo) cp
987             .cpItem(operandByte1)).stringIndex;
988         String s = new String(((CLConstantUtf8Info) cp
989             .cpItem(stringIndex)).b);
990         ins = new NHIRStringConstant(block, hirId++, s);
991         block.cfg.hirMap.put(ins.id, ins);
992         block.hir.add(ins.id);
993         operandStack.push(ins.id);
994         break;
995     }
996     case IADD:
997     case ISUB:
998     case IMUL: {
999         operand2 = operandStack.pop();
1000        operand1 = operandStack.pop();
1001        ins = new NHIRArithmetic(block, hirId++, insInfo.opcode,
1002            operand1, operand2);
1003        block.cfg.hirMap.put(ins.id, ins);
1004        block.hir.add(ins.id);
1005        operandStack.push(ins.id);
1006        break;
1007    }
1008    case IINC: {
1009        operandByte1 = tuple.operands.get(0);
1010        operandByte2 = tuple.operands.get(1);
1011        operand1 = block.locals[operandByte1];
1012        NHIRInstruction ins1 = new NHIRIntConstant(block, hirId++,
1013            (byte) operandByte2);
1014        ins = new NHIRArithmetic(block, hirId++, IADD, operand1,
1015            ins1.id);
1016        block.locals[operandByte1] = ins.id;
1017        block.hir.add(ins1.id);
1018        block.cfg.hirMap.put(ins1.id, ins1);
1019        block.hir.add(ins.id);
1020        block.cfg.hirMap.put(ins.id, ins);
1021        break;
1022    }
1023    case IF_ICMPNE:
1024    case IF_ICMPGT:
1025    case IF_ICMPLE: {
1026        operandByte1 = tuple.operands.get(0);
1027        operandByte2 = tuple.operands.get(1);
1028        offset = shortValue(operandByte1, operandByte2);
1029        int rhs = operandStack.pop();
1030        int lhs = operandStack.pop();

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```

1031         NBasicBlock trueDestination = pcToBasicBlock.get(tuple.pc
1032             + offset);
1033         NBasicBlock falseDestination = pcToBasicBlock
1034             .get(tuple.pc + 3);
1035         ins = new NHIRConditionalJump(block, hirId++, lhs, rhs,
1036             insInfo.opcode, trueDestination, falseDestination);
1037         block.cfg.hirMap.put(ins.id, ins);
1038         block.hir.add(ins.id);
1039         break;
1040     }
1041     case GOTO: {
1042         operandByte1 = tuple.operands.get(0);
1043         operandByte2 = tuple.operands.get(1);
1044         offset = shortValue(operandByte1, operandByte2);
1045         NBasicBlock destination = pcToBasicBlock.get(tuple.pc
1046             + offset);
1047         ins = new NHIRGoto(block, hirId++, destination);
1048         block.cfg.hirMap.put(ins.id, ins);
1049         block.hir.add(ins.id);
1050         break;
1051     }
1052     case GETSTATIC:
1053     case PUTSTATIC: {
1054         operandByte1 = tuple.operands.get(0);
1055         operandByte2 = tuple.operands.get(1);
1056         int index = shortValue(operandByte1, operandByte2);
1057         int classIndex = ((CLConstantFieldRefInfo)
cp.cpItem(index)).classIndex;
1058         int nameAndTypeIndex = ((CLConstantFieldRefInfo) cp
1059             .cpItem(index)).nameAndTypeIndex;
1060         int nameIndex = ((CLConstantClassInfo) cp
1061             .cpItem(classIndex)).nameIndex;
1062         String target = new String(((CLConstantUtf8Info) cp
1063             .cpItem(nameIndex)).b);
1064         int fieldNameIndex = ((CLConstantNameAndTypeInfo) cp
1065             .cpItem(nameAndTypeIndex)).nameIndex;
1066         int fieldDescIndex = ((CLConstantNameAndTypeInfo) cp
1067             .cpItem(nameAndTypeIndex)).descriptorIndex;
1068         String name = new String(((CLConstantUtf8Info) cp
1069             .cpItem(fieldNameIndex)).b);
1070         String desc = new String(((CLConstantUtf8Info) cp
1071             .cpItem(fieldDescIndex)).b);
1072         if (insInfo.opcode == PUTSTATIC) {
1073             ins = new NHIRPutField(block, hirId++, insInfo.opcode,
1074                 target, name, shortType(desc), desc,
1075                 operandStack.pop());
1076         } else {
1077             ins = new NHIRGetField(block, hirId++, insInfo.opcode,
1078                 target, name, shortType(desc), desc);
1079             operandStack.push(ins.id);
1080         }
1081         block.cfg.hirMap.put(ins.id, ins);
1082         block.hir.add(ins.id);
1083         break;
1084     }
1085     case INVOKESPECIAL:
1086     case INVOKESTATIC: {
1087         operandByte1 = tuple.operands.get(0);
1088         operandByte2 = tuple.operands.get(1);
1089         int index = shortValue(operandByte1, operandByte2);
1090         int classIndex = ((CLConstantMethodRefInfo) cp
1091             .cpItem(index)).classIndex;
1092         int nameAndTypeIndex = ((CLConstantMethodRefInfo) cp
1093             .cpItem(index)).nameAndTypeIndex;
1094         int nameIndex = ((CLConstantClassInfo) cp
1095             .cpItem(classIndex)).nameIndex;
1096         String target = new String(((CLConstantUtf8Info) cp
1097             .cpItem(nameIndex)).b);
1098         int methodNameIndex = ((CLConstantNameAndTypeInfo) cp

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```

1099         .cpItem(nameAndTypeIndex)).nameIndex;
1100     int methodDescIndex = ((CLConstantNameAndTypeInfo) cp
1101         .cpItem(nameAndTypeIndex)).descriptorIndex;
1102     String name = new String(((CLConstantUtf8Info) cp
1103         .cpItem(methodNameIndex)).b);
1104     String desc = new String(((CLConstantUtf8Info) cp
1105         .cpItem(methodDescIndex)).b);
1106     ArrayList<Integer> args = new ArrayList<Integer>();
1107     int numArgs = argumentCount(desc);
1108     for (int i = 0; i < numArgs; i++) {
1109         int arg = operandStack.pop();
1110         args.add(0, arg);
1111     }
1112     String returnType = returnType(desc);
1113     ins = new NHIRInvoke(block, hirId++, insInfo.opcode,
1114         target, name, args, shortType(returnType),
1115         returnType);
1116     if (!returnType.equals("V")) {
1117         operandStack.push(ins.id);
1118     }
1119     block.cfg.hirMap.put(ins.id, ins);
1120     block.hir.add(ins.id);
1121     break;
1122 }
1123 case IRETURN:
1124 case ARETURN: {
1125     ins = new NHIRReturn(block, hirId++, insInfo.opcode,
1126         operandStack.pop());
1127     block.cfg.hirMap.put(ins.id, ins);
1128     block.hir.add(ins.id);
1129     break;
1130 }
1131 case RETURN: {
1132     ins = new NHIRReturn(block, hirId++, insInfo.opcode, -1);
1133     block.cfg.hirMap.put(ins.id, ins);
1134     block.hir.add(ins.id);
1135     break;
1136 }
1137 }
1138 }
1139 }
1140 }
1141
1142 /**
1143  * Carry out optimizations on the high-level instructions.
1144  */
1145
1146 public void optimize() {
1147     // TBD
1148 }
1149
1150 /**
1151  * Eliminate redundant phi functions of the form x = (y, x, x, ..., x) with
1152  * y.
1153  */
1154
1155 public void eliminateRedundantPhiFunctions() {
1156     for (int ins : hirMap.keySet()) {
1157         NHIRInstruction hir = hirMap.get(ins);
1158         if (hir instanceof NHIRPhiFunction) {
1159             NHIRPhiFunction phi = (NHIRPhiFunction) hir;
1160             int firstArg = phi.arguments.get(0);
1161             boolean match = true;
1162             NBasicBlock block = phi.block;
1163             if (!block.isLoopHead) {
1164                 continue;
1165             }
1166             for (int i = 1; i < phi.arguments.size(); i++) {
1167                 if (phi.arguments.get(i) !=

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

block.predecessors.get(i).locals[phi.local]) {
1168         match = false;
1169         phi.arguments.set(i,
1170             block.predecessors.get(i).locals[phi.local]);
1171     }
1172 }
1173 if (match && firstArg != phi.id) {
1174     hirMap.put(phi.id, hirMap.get(firstArg));
1175     phi.block.hir.remove((Integer) phi.id);
1176 }
1177 }
1178 }
1179 }
1180
1181 /**
1182  * Convert the hir instructions in this cfg to lir instructions.
1183  */
1184
1185 public void hirToLir() {
1186     lirId = 0;
1187     regId = 32;
1188     offset = 0;
1189     registers = new ArrayList<NRegister>();
1190     data = new ArrayList<String>();
1191     for (int i = 0; i < 32; i++) {
1192         registers.add(null);
1193     }
1194     pRegisters = new ArrayList<NPhysicalRegister>();
1195     for (int ins : hirMap.keySet()) {
1196         hirMap.get(ins).toLir();
1197     }
1198
1199     // We now know how many virtual registers are needed, so
1200     // we can initialize bitset fields in each block that are
1201     // needed for interval calculation.
1202     int size = registers.size();
1203     for (NBasicBlock block : basicBlocks) {
1204         block.liveDef = new BitSet(size);
1205         block.liveUse = new BitSet(size);
1206         block.liveIn = new BitSet(size);
1207         block.liveOut = new BitSet(size);
1208     }
1209 }
1210
1211 /**
1212  * Resolve the phi functions in this cfg, i.e., for each x = phi(x1, x2,
1213  * ..., xn) generate an (LIR) move xi, x instruction at the end of the
1214  * predecessor i of the block defining the phi function; if the instruction
1215  * there is a branch, add the instruction prior to the branch.
1216  */
1217
1218 public void resolvePhiFunctions() {
1219     for (int ins1 : hirMap.keySet()) {
1220         NHIRInstruction hir = hirMap.get(ins1);
1221         if (hir instanceof NHIRPhiFunction) {
1222             NHIRPhiFunction phi = (NHIRPhiFunction) hir;
1223             NBasicBlock block = phi.block;
1224             for (int i = 0; i < phi.arguments.size(); i++) {
1225                 NHIRInstruction arg = hirMap.get(phi.arguments.get(i));
1226                 if (arg.sType.equals("")) {
1227                     continue;
1228                 }
1229                 NBasicBlock targetBlock = block.predecessors.get(i);
1230                 NLIRMove move = new NLIRMove(arg.block, lirId++, arg.lir,
1231                     phi.lir);
1232                 int len = targetBlock.hir.size();
1233                 if (hirMap.get(targetBlock.hir.get(len - 1)) instanceof
1234                     NHIRGoto

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

instanceof NHIRConditionalJump) {
1235         targetBlock.lir.add(len - 1, move);
1236     } else {
1237         targetBlock.lir.add(move);
1238     }
1239 }
1240 }
1241 }
1242 }
1243
1244 /**
1245  * Compute optimal ordering of the basic blocks in this cfg.
1246  */
1247
1248 public void orderBlocks() {
1249     // TBD
1250 }
1251
1252 /**
1253  * The basic block at a particular instruction id.
1254  *
1255  * @param id
1256  *         the (LIR) instruction id.
1257  * @return the basic block.
1258  */
1259
1260 public NBasicBlock blockAt(int id) {
1261     for (NBasicBlock b : this.basicBlocks) {
1262         if (b.getFirstLIRInstId() <= id && b.getLastLIRInstId() >= id)
1263             return b;
1264     }
1265     return null;
1266 }
1267
1268 /**
1269  * Assign new ids to the LIR instructions in this cfg.
1270  */
1271
1272 public void renumberLIRInstructions() {
1273     int nextId = 0;
1274     for (NBasicBlock block : basicBlocks) {
1275         ArrayList<NLIRInstruction> newLir = new ArrayList<NLIRInstruction>();
1276         for (NLIRInstruction lir : block.lir) {
1277             if (lir instanceof NLIRLoadLocal
1278                 && ((NLIRLoadLocal) lir).local < 4) {
1279                 // Ignore first four formals.
1280                 continue;
1281             }
1282             lir.id = nextId;
1283             nextId += 5; // an extra slot for spills though we
1284             // don't use it
1285             newLir.add(lir);
1286         }
1287         block.lir = newLir;
1288     }
1289 }
1290
1291 /**
1292  * Replace references to virtual registers in LIR instructions with
1293  * references to physical registers.
1294  */
1295
1296 public void allocatePhysicalRegisters() {
1297     for (NBasicBlock block : basicBlocks) {
1298         for (NLIRInstruction lir : block.lir) {
1299             lir.allocatePhysicalRegisters();
1300         }
1301     }
1302 }

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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  */

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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);

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

1441         ((NHIRPhiFunction) ins).inferType();
1442     }
1443 }
1444 }
1445
1446 /**
1447  * Convert the bytecode in the specified list to their tuple
1448  * representations.
1449  *
1450  * @param code
1451  *         bytecode to convert.
1452  *
1453  * @return list of tuples.
1454  */
1455
1456 private ArrayList<NTuple> bytecodeToTuples(ArrayList<Integer> code) {
1457     ArrayList<NTuple> tuples = new ArrayList<NTuple>();
1458     for (int i = 0; i < code.size(); i++) {
1459         int pc = i;
1460         int opcode = code.get(i);
1461         int operandBytes =
1462             CLInstruction.instructionInfo[opcode].operandCount;
1463         short operandByte1, operandByte2, operandByte3, operandByte4;
1464         int pad, deflt;
1465         ArrayList<Short> operands = new ArrayList<Short>();
1466         switch (operandBytes) {
1467             case 0:
1468                 break;
1469             case 1:
1470                 operandByte1 = code.get(++i).shortValue();
1471                 operands.add(operandByte1);
1472                 break;
1473             case 2:
1474                 operandByte1 = code.get(++i).shortValue();
1475                 operandByte2 = code.get(++i).shortValue();
1476                 operands.add(operandByte1);
1477                 operands.add(operandByte2);
1478                 break;
1479             case 3:
1480                 operandByte1 = code.get(++i).shortValue();
1481                 operandByte2 = code.get(++i).shortValue();
1482                 operandByte3 = code.get(++i).shortValue();
1483                 operands.add(operandByte1);
1484                 operands.add(operandByte2);
1485                 operands.add(operandByte3);
1486                 break;
1487             case 4:
1488                 operandByte1 = code.get(++i).shortValue();
1489                 operandByte2 = code.get(++i).shortValue();
1490                 operandByte3 = code.get(++i).shortValue();
1491                 operandByte4 = code.get(++i).shortValue();
1492                 operands.add(operandByte1);
1493                 operands.add(operandByte2);
1494                 operands.add(operandByte3);
1495                 operands.add(operandByte4);
1496                 break;
1497             case DYNAMIC: // TBD
1498                 break;
1499         }
1500         tuples.add(new NTuple(pc, opcode, operands));
1501     }
1502     return tuples;
1503 }
1504
1505 /**
1506  * Construct and return a short integer from two unsigned bytes specified.
1507  *
1508  * @param a
1509  *         unsigned byte.

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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;

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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     // Extract types of arguments and the return type from
1617     // the method descriptor
1618     String argTypes = descriptor.substring(1, descriptor.lastIndexOf("("));
1619
1620     // Find number of arguments
1621     for (int j = 0; j < argTypes.length(); j++) {
1622         char c = argTypes.charAt(j);
1623         switch (c) {
1624             case 'B':
1625             case 'C':
1626             case 'I':
1627             case 'F':
1628             case 'S':
1629             case 'Z':
1630                 i += 1;
1631                 break;
1632             case '[':
1633                 break;
1634             case 'J':
1635             case 'D':
1636                 i += 2;
1637                 break;
1638             case 'L':
1639                 int k = argTypes.indexOf(";", j);
1640                 j = k;
1641                 i += 1;
1642                 break;
1643         }
1644     }
1645     return i;
1646 }

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```

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 'D':
1677             case 'J':
1678             case 'S':
1679                 args.add(type + String.valueOf(c));
1680                 type = "";
1681                 break;
1682             case '[':
1683                 type += c;
1684                 break;
1685             case 'L':
1686                 int k = argTypes.indexOf(";", j);
1687                 args.add(type + argTypes.substring(j, k));
1688                 type = "";
1689                 j = k;
1690                 break;
1691         }
1692     }
1693     return args;
1694 }
1695
1696 /**
1697  * Return the return type of a method given its descriptor.
1698  *
1699  * @param descriptor
1700  *        descriptor of the method.
1701  *
1702  * @return return type, "V" if void.
1703  */
1704
1705 private String returnType(String descriptor) {
1706     String returnType = descriptor
1707         .substring(descriptor.lastIndexOf("(") + 1);
1708     return returnType;
1709 }
1710
1711 }
1712 }
1713

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder