Dakota Whelchel
Annie Chiou
Noah Marestaing

Phase 3 Document

Design Process:
For this phase of the project, we translate vapor code into vaporm code. The main difference between vapor and vaporm is that vaporm uses registers and stacks mapping local variables to registers and run-time stack elements. We use 23 registers, $s0-$s7 for callee-saved, $t0-$t8 for caller-saved, $a0-$a3 for argument passing, $v0 to return the result from a call,$v0 and $v1 can also be used for loading values from the stack, and $t9 is reserved for the next assignment. Our program does a linear scan to calculate liveness for each variable returning the lines where the variable is first use and last used. We used the formula found in the textbook to implement this top-down scan to calculate liveness. We also implemented a linear scan for register allocation, choosing when to spill variables to memory when all the other registers are taken. Registers are stored in a register pool implemented in the file RegisterPool.java and the Register type is implemented in the file Register.java.

How to Run:

1. Compile: javac -cp .:vapor-parser.jar <filename.vapor>
2. Choose which file you want to check: java -cp .:vapor-parser.jar V2VM < filename.vaporm
3. The program should then write to or create a new .vaporm file translated from the .vapor file input to the program.j

Example Code:
Liveness Calculation

```
public void translateFunction(VFunction function) {

    // for (VVarRef.Local param : function.params) {
    //     System.out.println(param.toString());
    // }
    //
    // for (String var : function.vars) {
    //     System.out.println(var.toString());
    // }

    graph = instrVisitor.createFlowGraph(function.body);
```

```
        Liveness liveness = graph.calcLiveness();

        Map<String, Interval> intervals = new HashMap<>(); // maps variables to its live interval
        List<Set<String>> active = new ArrayList<>();

        // live intervals (used by the linear scan algorithm) calculated using the active sets
        // by top-down scan of the instructions
        for (Node node : graph.getNodeList()) {
            // active[n] = def[n] U in[n]
            Set<String> act = node.getDefSet(); // def[n]
            act.addAll(liveness.in.get(node));  // in[n]
            active.add(act);
        }

        for (int i = 0; i < active.size(); ++i) {
            for (String variable : active.get(i)) {
                if (intervals.containsKey(variable)) {
                    intervals.get(variable).setEnd(i); // update end of interval
                } else {
                    intervals.put(variable, new Interval(variable, i, i)); // create new interval
                }
            }
        }

        // Pass live intervals and function parameters into register allocation
        registerAlloc(new ArrayList<>(intervals.values()), function.params);

        transVisitor.printFunc(function);

        System.out.println("");

    }
```

Register Pool

```
public class RegisterPool {

    private Set<Register> all = new LinkedHashSet<>();
    private Set<Register> use = new HashSet<>();

    // Add all registers to "all"
    public RegisterPool(Register[] registers) {
        Collections.addAll(all, registers);
```

```
}

public static RegisterPool CreateGlobalPool() {
   Register[] registers = {

      // Caller saved registers
      Register.t0, Register.t1, Register.t2, Register.t3,
      Register.t4, Register.t5, Register.t6, Register.t7,
      Register.t8,

      // Callee saved registers
      Register.s0, Register.s1, Register.s2, Register.s3,
      Register.s4, Register.s5, Register.s6, Register.s7

   };
   return new RegisterPool(registers);
}

public static RegisterPool CreateLocalPool() {
   Register[] registers = {

      Register.v0, Register.v1,
      Register.a0, Register.a1, Register.a2, Register.a3

   };
   return new RegisterPool(registers);
}
```