CS179E: Project in Computer Science – Compilers
Document for Phase 3

Jiamin Pan, Yunqing Xiao

1 Introduction

In phase 2, we successfully translated MiniJava source file to Vapor source file, which is also called intermediate code generation. Now when we come to phase 3, we are required to allocate registers to each variables by data-flow analysis and algorithms for register allocation. We will translate Vapor source code to another one written by a new language Vapor-M, where each variable is replaced with its allocated register. As long as we can achieve the translation, we will be able to use this register map to apply the code logic to assembly languages. In other words, register allocation is essential for us to translate our code into assembly languages, no matter which instruction set architecture we will utilize.

2 Dependencies

1. JDK version = JDK-11.0.2.
2. Vapor parser and interpreter. [1]

3 Methods for Register Allocation

Vapor is an interpreted language. Unlike MiniJava, we just need to traverse each function top-down to record all useful information including variables, parameters, labels and statements. At the beginning, we will call the parsing function posted on homepage [1]. It will traverse the whole source code and return a `VaporProgram` object containing all information. The reconstruction of data segments is easy. We will make use of `dataSegment` array in the object and print it directly by the printer.

But that is not enough. The most essential part of this phase is the reconstruction of function blocks. Therefore, we will following the instructions on the homepage [1] and textbook [2]. We build control flow graph (CFG) for each function and compute live intervals of variables according to the CFG. Then we will use linear scan algorithm [3] to allocate registers to different variables. Finally, we will utilize the map we get from the allocation phase to complete the translation. We will state the details of each phase one by one.

3.1 Data-Flow Analysis

We regard each statement (not labels) as a node of CFG. For each node n, we maintain four set for it: $in[n]$, $out[n]$, $use[n]$, and $def[n]$. $in[n]$ contains all variables live-in in this node, while $out[n]$ contains all variables live-out. All variables

[1] They can be downloaded from the homepage.

defined in this node are included in $def[n]$, and all variables used in this node are included in $use[n]$. We also have

$$in[n] = use[n] \cup (out[n] - def[n]),$$
$$out[n] = \bigcup_{s \in succ[n]} in[s].$$

Before we compute $in[n]$ and $out[n]$, we should add all nodes and related edges to CFG. During the construction, we can directly set $use[n]$ and $def[n]$ by these statements we meet. Then we will iteratively compute $in[n]$ and $out[n]$ by the following algorithm:

```
1: repeat
2:     for each node $n$ of CFG do
3:         $in'[n] = in[n]$
4:         $out'[n] = out[n]$
5:         $in[n] = use[n] \cup (out[n] - def[n])$
6:         $out[n] = \bigcup_{s \in succ[n]} in[s]$
7: until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all node $n$
```

We can design the class `Node` like this. We set four `HashSet` objects representing these four sets we need. We also maintain two node lists to contain all the predecessor and successor nodes of current node. As long as we can set these field during the traverse, the computation for $in[n]$ and $out[n]$ will be very trivial.

```
public class Node {
    private int key;
    private NodeList succ;
    private NodeList pred;
    public HashSet<String> in;
    public HashSet<String> out;
    public HashSet<String> def;
    public HashSet<String> use;
    ...
}
```

After we successfully constructed CFG, we will compute the live intervals for each variable. Note that we will only need to do a simplified liveness analysis here. The start point of each variable is the first line of its definition, while the parameters' start points will be first line of the function. But the end points of variables are not so trivial. We can only determine it when we traverse the statements top-down (not graph traverse!). We will set the end point of the variable if it is in $in[n] - in[i]$ where n is the next statement of i. (n may not be contained in $succ[i]$!)

Since we need to record the start and end points of variables, we need to create another class to store these information. Hence, we create a new class `VarMap` for it. In the following steps, we will need to sort these intervals by start point or end point, so we implement the `Comparator` interface for this class, which provide a `Compare` method for object comparing.

```
public class VarMap {
    private String name;                // Name of variable
    private int startPoint;             // Start point of variable
    private int endPoint;               // End point of variable
    ...
    // Two comparators for compare two VarMap objects
    // By start point or end point
    public static final Comparator<VarMap> BY_START = new ByStart();
    public static final Comparator<VarMap> BY_END = new ByEnd();
    ...
    // Comparator class
    // Compare their start points
    private static class ByStart implements Comparator<VarMap> {
        @Override
        public int compare(VarMap vm_1, VarMap vm_2) {
            return Integer.compare(vm_1.startPoint, vm_2.startPoint);
        }
    }

    // Comparator class
    // Compare their end points
    private static class ByEnd implements Comparator<VarMap> {
        @Override
        public int compare(VarMap vm_1, VarMap vm_2) {
            return Integer.compare(vm_1.endPoint, vm_2.endPoint);
        }
    }
}
```

3.2 Register Allocation

Then we will use the live intervals of all variables we derived at last step. We first need to separate "caller-saved" variables and "callee-saved" variables before we conduct register allocation. The reason is that we will conduct different allocation for these two kinds of variables. For "caller-saved" variables, we allocate \$t0-\$t8 to them, while for "callee-saved", we allocate \$s0-\$s7 and \$v0-\$v1 (if needed) to them. Surprisingly, we can determine it at last step, since these "callee-saved" are contained in $out[c] - def[c]$ for every function call node c.

```
public class VarMap {
    ...
    private boolean isCalleeSaved;      // Mark for whether this is a callee-saved variable
    ...
}
...
// Deal with function call
if (func.body[i] instanceof VCall) {
    HashSet<String> saved = new HashSet<String>();
    // callee-saved = out - def
    saved.addAll(node.out); saved.removeAll(node.def);
    for (String var : saved) {
        liveIntervals.get(var).setCalleeSaved();
    }
}
```

Then we will implement the linear scan algorithm [3] according to these algorithms. We create a new class `LinearScanner` to complete the work automatically without calling the method out of the object. Since we treat these two kinds of variables seperately, we just pass an argument `type` to initialize different register pools. After that we will follow the algorithms using different register pools. Since we create a `ArrayDeque` object for the pool which can also work analogously like stacks, we will make the allocation the operations of stacks like "push" and "pop".

```
public class LinearScanner {
    // General use constants
    private final String[] tRegNames = {"t0", "t1", "t2", "t3", "t4", "t5", "t6", "t7", "t8"};
    private final String[] sRegNames = {"s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "v0", "v1"};
    private final int tRegNum = tRegNames.length;
    private final int sRegNum = sRegNames.length;
    // Useful fields
    private VarMap[] calleeSaved;                   // Callee-saved variables
    private VarMap[] callerSaved;                   // Caller-saved variables
    private HashMap<String, String> registerMap;    // The map of variables to registers
    private ArrayDeque<String> regPool;             // The pool of registers
    ...
    // Register allocation
    private void registerAlloc(VarMap[] vars, char type) {
        ArrayList<VarMap> active = new ArrayList<VarMap>();
        ArrayList<VarMap> spilled = new ArrayList<VarMap>();
        InitRegPool(type);
        // Set maximum size
        int maxsize = (type == 't')? this.tRegNum : this.sRegNum;
        for (VarMap vm : vars) {
            expireOld(active, vm);
            if (active.size() == maxsize)
                spillInterval(active, spilled, vm);
            else {
                String reg = getRegister();
                if (this.registerMap.get(vm.getName()) == null)
                    this.registerMap.put(vm.getName(), reg);
                insertVar(active, vm);
            }
        }
    }
    // Expire old intervals
    private void expireOld(ArrayList<VarMap> active, VarMap v) {
        ArrayList<VarMap> toRemove = new ArrayList<VarMap>();
        for (VarMap a : active) {
            if (a.getEndPoint() < v.getStartPoint()) {
                String reg = this.registerMap.get(a.getName());
                this.regPool.addFirst(reg);
                toRemove.add(a);
            }
        }
        active.removeAll(toRemove);
    }
    // Spill these nodes if needed
    private void spillInterval(ArrayList<VarMap> active, ArrayList<VarMap> spilled, VarMap v) {
        VarMap last = active.get(active.size() - 1);
        if (last.getEndPoint() > v.getEndPoint()) {
            String reg = this.registerMap.get(last.getName());
            this.registerMap.put(v.getName(), reg);
            spilled.add(last);
            active.remove(last);
            insertVar(active, v);
        } else spilled.add(v);
    }
    ...
}
```

3.3 Translations

After we have gained the register map for variables, we will begin to work on translation, which is not so difficult as the previous steps. Now we will also implement a `Visitor` object to deal with different type of statements separately. For each statement, we will scan the variables it used and replaced them by the allocated registers according to the maps. As for the integer literals, string literals and labels, we just need to print them directly without any modification. Here we provide the implementation of assign statement to which other statements are similar.

```
public void visit(VAssign a) throws IOException {
    if (this.regMap.get(a.dest.toString()) != null) {
        String s = "$" + this.regMap.get(a.dest.toString()) + "␣=␣";
        if (a.source instanceof VLitInt) {
            s += a.source.toString();
        } else if (a.source instanceof VLabelRef<?>) {
            s += a.source.toString();
        } else {
            s += "$" + this.regMap.get(a.source.toString());
        }
        printer.println(s);
    }
}
```

3.4 Potential Defect

However, there is still a potential defect in the program, which is the translation for $v0 and $v1 registers. If they are used as callee-saved registers, then they will need to communicate with local stacks to restore their values. Even though I made it right for $v0, but I am not very sure whether it will also be right for $v1. This is something we need to work on in the future.

4 Review for Phase 3

```
==== Results ====
Passed 18/18 test cases
- Submission Size = 28 kB
```

Fortunately, we eventually achieved our goal and passed all the provided test cases. The whole process was very interesting and exciting. It was very difficult for us to come up good ideas to solve the problems, but as long as we made it, everything will be easy. We perhaps will not work on the next optional phase, but we regard the time we spent on the first three phases very meaningful and unforgettable. This project fortified our programming skills and improved our knowledge on compiler design. Never would we get access to these details unless we dedicated ourselves to this project.

We feel very grateful to our professor, Mr. Mohsen Lesani, without whom we would never have achieved this. Specially, we would like to thank our teaching assistant, Miss Lian Gao. She was always ready to help us every time we had questions or problems. We derived numerous ideas from her suggestions and instructions.

References

[1] Course's Homepage: Compiler Project.

[2] Andrew W. Appel, Jens Palsberg, *Modern Compiler Implementation in Java*, Cambridge University Press, Second Edition, 2002.

[3] Massimiliano Poletto, Vivek Sarkar, *Linear Scan Register Allocation*, ACM TOPLAS 21(5):895-913, 1999.