CS614: Advanced Compilers

Autumn 2023 (Due: October 21nd, 2023)

Assignment A3: From Spills to Thrills: Register Allocation in Action

1 Assignment Objective

Using graph coloring (kempe's heuristics) to perform register allocation.

2 Detailed Specification

Perform register allocation for variables in a function wherever possible and spill the rest to the memory using the help from liveness analysis pass provided in the `starter`. First line of every testcase contains a special comment of the form /*k*/ where k represents the maximum number of registers that can be used inside any given function. Your job is to replace all the variable declarations using a suitable register or memory reference. In order to do this you are given an implementation of liveness analysis which you must use to create an interference graph and color it using the specified number of registers allowed for that program. You are provided with the following files as a part of this assignment:

```
assignment/friendTJ.jj : The source grammar.
output-validator/friendTJMem.jj : The target grammar.
assignment/* : Java project using minijava.jj that provides an API for liveness info
output-validator/* : Java project using minijavareg.jj that parses the obtained output
public-testcase/* : Public test case input
testcase-output/* : Public test case output
```

2.1 Liveness API

This api provides the result of liveness analysis as a hashmap (See assn/Main.java) that contains a mapping from Node to Set<String>. The result contains the set of variables that are live at that node i.e. the IN set. All the Statement Nodes can be used to query the hashmap, namely, PrintStatement, VarDeclaration, AssignmentStatement, ArrayAssignmentStatement, FieldAssignmentStatement, IfthenStatement, IfthenElseStatement, WhileStatement and LivenessQueryStatement. Additionally, for getting the liveness information at the return statement of a function, MethodDeclaration node must be used.

```
# using MethodDeclaration node to access liveness information of return statement
public Integer visit(MethodDeclaration n) {
  if (resultMap.containsKey(n)) {
      System.out.println("Liveness: " + resultMap.get(n));
  }
  ...
}
# assn/Main.java prints the liveness for a given input program using the provided API.
java Main < ../testcase/TC02.java
```

2.2 Output Spec

The final output must follow the following spec:

- import static a3.Memory.*;[Static Import]: This static import must be declared immediately after the register limit comment.
- Object Rx;[Register declaration]: The registers must be declared as generic Object variables (instead of the regular VarDeclaration). You are free to choose any naming convention as long as it is supported by the grammar.
- Rx = Expression;[Register store]: The assignment to registers remains the same as normal assignment statement.
- ((TYPE) Rx)[Register load]: Here TYPE is the type of the variable and Rx is the register.
- alloca(SIZE);[MemoryAllocation] The number of spilled variables must be declared after all the register declarations; in case no variable was spilled the value of SIZE must be set to 0.
- store(INDEX,Expression);[Memory store]: INDEX represents the memory offset at which the expression must be stored.
- ((TYPE) load(INDEX))[Memory load]: Here TYPE is the type of the variable and INDEX is the memory offset of that variable.

2.3 Detailed Example - Public Testcase

Figure 1 shows an example code along with the IN sets of liveness analysis at that point. We create an interference graph for the same and perform simple graph coloring. Here variables a, e and t are allocated to R1, and variables b and c are allocated to R2. Variable d cannot be allocated, hence it is spilled to memory offset 0.

Figure 2 and Figure 3 show the given input and the expected output for a given testcase.

```
a = 5;          []
b = 6;          [a]
c = a + b;      [a,b]
d = c + a;      [a,c,d]
e = a - c;      [d,e]
t = d - e;      [t]
```

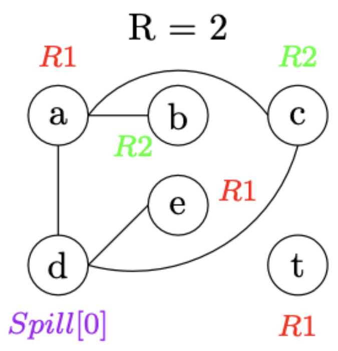

Figure 1: Graph coloring using 2 registers for function foo()

```java
/*2*/
class TC02 {
    public static void main
    (String[] args) {
        TestTC02 o;
        int res;
        o = new TestTC02();
        res = o.foo();
        System.out.println(res);
    }
}
class TestTC02 {
    public int foo() {
        int a;
        int b;
        int c;
        int d;
        int e;
        int t;
        a = 5;
        b = 6;
        c = a + b;
        d = c + a;
        e = a - c;
        t = d - e;
        return t;
    }
}
```

Figure 2: TC02.java

```java
/*2*/
import static a3.Memory.*;
class TC02 {
    public static void main(String[] args){
        Object r1;
        Object r2;
        alloca(0);
        r1 = new TestTC02();
        r1 = ((TestTC02)r1).foo();
        System.out.println(((int)r1));
    }
}

class TestTC02 {
    public int foo() {
        Object r1;
        Object r2;
        alloca(1);
        r1 = 5;
        r2 = 6;
        r2 = ((int) r1) + ((int) r2);
        store(0,((int) r2) + ((int) r1));
        r1 = ((int) r1) - ((int) r2);
        r1 = ((int) load(0)) - ((int) r1);
        return ((int)r1);
    }
}
```

Figure 3: Expected Output

2.4 Notes

There are some general notes and assumptions that you can make regarding the test cases.

- You cannot simply spill all the variables into the memory; we will use a **simple strategy** discussed in class to calculate the approximate number of spills needed; if your result is considerably higher than the simple solution then there will be a penalty for each additional spill.
- The type for the register must be Object.
- The output-validator checks for the parsing of the resultant program (You can check for parsing by supplying the generated output of your assignment).
- The **a3 folder** will always be present in the directory where the output program is run from.
- For formal function parameters **do not perform** register allocation; they can be used directly as it is.
- You can assume that no testcases will use any **fields or fields related operations.**
- You can assume that no testcases will use any **arrays or array related functions.**
- You can assume that no testcases will use the **this pointer**.
- You can assume that no testcases will have any **dead code.**

2.5 Evaluation

Your submission must be named `rollnum-a3.zip`, where `rollnum` is your roll-number in small letters. Upon unzipping the submission, we should get a directory named `rollnum-a3`. The main class inside this directory should be named `Main.java`. Your program should read from the standard input and print to the standard output. You can leave all the visitors and syntax-tree nodes as it is, but remember to remove all the `.class` files.

We would run the following commands in the evaluation script:

- `javac Main.java`
- `java Main <../public-testcase/TC02.java > out.java`

Check for parsing of the generated output:

- `cd output-validator`
- `javac Main.java`
- `java Main < out.java`

If the generated output parses successfully, generates output same as the original program and the number of spilling is same as expected, you would get marks for the corresponding testcase.

3 Plagiarism Warning

You are allowed to discuss publicly on the Slack channel, but are supposed to do the assignment completely individually. If plagiarism is found:

- First instance: 0 marks in the assignment
- Second instance: FR grade in the course
- Third instance: report to institutional committee

-*-*-*- Do the assignment honestly, enjoy learning the course. -*-*-*-