Program Analysis, University of Stuttgart, Winter 2023/24

Exercise 2: Data Flow Analysis

Deadline for uploading solutions via Ilias:
November 16, 2023, 11:59pm Stuttgart time

1 Available Expressions & Live Variables [70 points]

Consider the following program in a toy language with a syntax inspired by Python. Assume that all variables are integers and that operators have the obvious semantics.

```
1  left = n - m
2  right = n + m
3  if left == right:
4    return
5  temp = 0
6  while (temp != left + right - temp):
7    temp = temp + 1
8  mid = left + right - temp
9  if n == mid:
10   print("obvious")
11 if 2 * temp == 2 * n:
12   print("another obvious statement")
```

1.1 PART I: Available Expressions [25 points]

Your task is to perform the *Available Expressions* data flow analysis, as presented in the lecture. Complete the following subtasks.

1.1.1 Control-Flow Graph [5 points]

Draw the control-flow graph (CFG) of the given program. As in the lecture, nodes are individual statements. Label each statement/node in the graph with the line number corresponding to that statement in the program, to help with the following subtasks.

1.1.2 Transfer Function [10 points]

First, list all *non-trivial, arithmetic expressions* in the program, i.e., expressions involving the operators: `+`, `-`, and `*`. Only consider the full expressions, as they appear in the code, and not subexpressions of those expressions.

Expressions: {

}

Next, fill the following table with the *gen* and *kill* sets of each statement in the program. The statement numbers correspond to lines numbers in the program.

Statement s	$gen(s)$	$kill(s)$
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

1.1.3 Solving Data Flow Equations [10 points]

Now, use the iterative algorithm from the lecture to solve the data flow equations for each statement in the program. You can iteratively fill up the second and third column of the table below during solving.

Statement s	$AE_{entry}(s)$	$AE_{exit}(s)$
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

1.2 PART II: Live Variables [20]

Your task is to perform the *Live Variables* data flow analysis, as presented in the lecture. Complete the following subtasks.

1.2.1 Transfer Function [10 points]

First, write down the domain of the *Live Variables* analysis for the given program, i.e., complete the following set.

Domain: { }

Next, fill the following table with the *gen* and *kill* sets of each statement in the program.

Statement s	$gen(s)$	$kill(s)$
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

1.2.2 Solving Data Flow Equations [10 points]

Now, use the iterative algorithm from the lecture to solve the data flow equations for each statement in the program. You do not need to write down the data flow equations themselves here; you can iteratively fill up the second and third column of the table below during solving.

Statement s	$LV_{entry}(s)$	$LV_{exit}(s)$
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

1.3 PART III: Understanding & Application [25 points]

In this part, you will be asked to apply the results from Part I and Part II to optimize a machine code corresponding to the program given in this task. Suppose that the assembly-like code given below is the result of compiling the program given in this task. The compiler has no optimizations activated. Based on the results of the available expressions analysis and the live variables analysis, we can optimize the assembly code by:

- reusing registers containing available expressions or live variables,
- writing to a different register to avoid overwriting previously computed values that will be needed in the future, and
- deleting operations that recompute an already computed expression or unnecessarily load a variable from memory.

The assembly language has eight registers (`R0`, `R1`, ..., `R7`) with `R0` being the default register an instruction operates on. All the registers can be explicitly used for all operations. The set of instructions of the language is explained in the comments in the assembly code. To help you with this part, we provide a partial result of applying the above optimizations. Your task is to perform the following two actions whenever needed:

- Finish the missing parts of the optimized code. The missing parts that you should fill are marked with __.
- There are still unnecessary instructions that need to be removed, such as loading a variable from memory while it is still available in a register. You need to strike through those instruction (e.g, ~~LOAD [n]~~)

A template for your answer is given after the following, non-optimized version of the code.

Code without optimization:

```
1  LOAD [n]  // load value of variable n into the default register R0
2  MOVE R1, R0 // move the value in R0 to R1, meaning R1:=R0
3  LOAD [m]
4  SUB R1, R0 // subtract the value in R0 from R1, R1:=R1-R0
5  STORE R1, [left] // store the value in R1 into the variable left
6
7  LOAD [n]
8  MOVE R1, R0
9  LOAD [m]
10 ADD R1, R0 // R1:=R1+R0
11 STORE R1, [right]
12
13 LOAD [left]
14 MOVE R1, R0
15 LOAD [right]
16 CMP_EQ R1, R0 // R1 == R0
17 BRANCH_TRUE EXIT // exit the program if the comparison returns TRUE
18
19 LOAD 0
```

```
20 STORE [temp]
21
22 ENTER_LOOP:  // a label defining the point before the loop
23 LOAD [temp]
24 LOAD R1, [left]
25 LOAD R2, [right]
26 LOAD R3, [temp]
27 ADD R1, R2
28 SUB R1, R3
29
30 CMP_EQ R0, R1
31 BRANCH_TRUE EXIT_LOOP
32
33 LOAD [temp]
34 ADD 1
35 STORE [temp]
36 BRANCH ENTER_LOOP
37 EXIT_LOOP: // a label defining the point after the loop
38
39 LOAD [left]
40 LOAD R1, [right]
41 LOAD R2, [temp]
42 ADD R1
43 SUB R2
44 STORE [mid]
45
46 LOAD [n]
47 LOAD R1, [mid]
48 CMP_EQ R0, R1
49 BRANCH_TRUE PRINT1
50
51 SECOND_IF:
52 LOAD [temp]
53 LOAD R1, [n]
54 MULT R0, 2 // R0 = R0 * 2
55 MULT R1, 2
56 CMP_EQ R0, R1
57 BRANCH_TRUE PRINT2
58
59 PRINT1:
60 print "obvious"
61 BRANCH SECOND_IF
62
63 PRINT2:
64 print "another obvious statement"
65 EXIT
```

Code after optimization (your task is to finish the necessary changes):

```
1  LOAD R7, [n]
2  LOAD R6, [m]
3  MOVE R5, R7
4  SUB R5 , __
5  STORE __, [left]
6
7  LOAD [n]
8  LOAD [m]
9  ADD __, __
10 STORE R6, [right]
11
12 LOAD [left]
13 LOAD [right]
14 CMP_EQ __, __
15 BRANCH_TRUE EXIT // exit the program if the comparison returns TRUE
16
17 LOAD R4, 0
18 STORE R4, [temp]
19
20 ENTER_LOOP:  // a label defining the point before the loop
21
22 LOAD [temp]
23 MOVE R3, R5
24 ADD R3, __
25 SUB __, __
26
27 CMP_EQ __, __
28 BRANCH_TRUE EXIT_LOOP
29
30 LOAD [temp]
31 ADD __, 1
32 STORE __, [temp]
33 BRANCH ENTER_LOOP
34
35 EXIT_LOOP: // a label defining the point after the loop
36
37 LOAD [left]
38 LOAD R1, [right]
39 LOAD R2, [temp]
40 ADD R1
41 SUB R2
42 STORE __, [mid]
43
44 LOAD [n]
45 LOAD R2, [mid]
46 CMP_EQ __, __
47 BRANCH_TRUE PRINT1
```

```
48
49  SECOND_IF:
50  LOAD [temp]
51  LOAD R1, [n]
52  MULT __, 2
53  MULT __, 2
54  CMP_EQ __, __
55  BRANCH_TRUE PRINT2
56
57  PRINT1:
58  //print "obvious"
59  BRANCH SECOND_IF
60
61  PRINT2:
62  //print "another obvious statement"
63  EXIT
```

2 Copy Detect [30 points]

The following is about a data flow analysis that was *not* discussed in the lecture, which we call *Copy Detection*.

Copy Detection is an analysis that detects identical copies of variables at each point of a program. For example, right after executing the statement `x = y`, we say that `x` is a copy of `y`. The property is reflexive, meaning that `y` is also a copy of `x`. We consider a toy language with only integer variables. An assignment `x=y` makes `x` contains a copy of the value of `y`, but `x` and `y` point to two different memory locations. A variable `x` can be a copy of a variable `y` only through *assignment* and *transitivity*. "Transitivity" here means that if `x` is a copy of `y`, and `y` is a copy of `z`, then `x` is copy of `z` as well.

Consider the following example:

```
1  x = 1 // {}
2  y = x // {{y, x}}
3  z = x // {{y, x}, {z, x}, {z, y}, {y, x, z}}
4  x = 2 // {{z, y}}
```

In the example, the Copy Detection analysis computes all *copy sets*, i.e., sets of variables that contain the same value. The copy set after executing a line is shown in the comment at the end of the line. Initially, after line 1, the set is empty because `x` is assigned a constant value.

Line 2, where `y` is assigned the value of `x`, generates the copy set $\{x, y\}$. Subsequently, at line 3, when `z` is assigned the value of `x`, another copy set $\{z, y\}$ is formed. Note that our analysis encompasses all possible copy sets. Consequently, by the end of line 3, the set of copy sets contains not only $\{z, y\}$ but also $\{y, x, z\}$, as they are all valid and possible copy sets (as shown in the comment next to line 3).

Moving to line 4, the value of `x` is overwritten. The analysis then updates the set of copy sets by eliminating all sets containing `x`, which leaves us with one remaining copy set: $\{z, y\}$.

2.1 Defining a Data Flow Analysis [22 points]

Please describe the *Copy Detection* data flow analysis using the six properties discussed in the lecture. Explain the meaning of any formal symbols in your description.

1) Analysis domain:

2) Analysis direction:

3a) $gen(s)$ **function:**

3b) $kill(s)$ **function:**

3c) Transfer function:

4) Meet operator:

5) Boundary condition (i.e., value of the transfer function at the start/final node of the CFG):

6) Initial values (i.e., value of the transfer function of nodes at the start of iterative solving):

2.2 Application [8 points]

In this task, we ask you to perform the Copy Detection analysis that you just defined on the following program by filling the table below.

```
1  t = a
2  x = a
3
4  if x > 0:
5    y = x
6  else:
7    y = t
8
9  t = 0
10 z = t + x
11
12 if z == x:
13   print("obvious but we don't know")
14 if y == x:
15   print("obvious and we should know")
```

Statement s	$CD_{entry}(s)$	$CD_{exit}(s)$
1		
2		
4		
5		
6		
7		
9		
10		
12		
13		
14		
15		