Task3 - SCSR

Andrea Munarin (879607), Simone Jovon (882028)

March 2023

1 Exercise 1

A variable v is tainted by k if the value of v is affected by the value of k.

Given an initial set S of tainted expressions, the aim of the analysis is to find the set of the variables that are possibly tainted by S at each entry/exit point in the control flow graph.

SOLUTION

1. *What is the domain of the analysis?*

The domain of the analysis is the powerset of set of all variables in the program V. So the corresponding lattice can be formalized as:

$$Lattice = (V, \subseteq)$$

What is safe in the analysis:

- To assume that a variable is tainted even if it turns out not.
- The computed set of tainted variables in a point p will be a superset of the actual set of tainted variables at p.

The goal is to make the set of tainted variables as small as possible.

2. *Formalize the transfer functions associated to the program statements*

In order to formalize the transfer functions, we need to introduce some notation:

- `S` = {Initial set of tainted expression}
- `def[B]` = {Variable defined in B}
- `use[B]` = {Variable used in B}

Knowing this we can formalize the transfer function:

- The direction of the analysis is forward
- `out[B]` = {`def[B]|use[B]` $\cap$ (`S` $\cup$ `in[B]`)$\neq \emptyset$} $\cup$ (`in[B]-def[B]`)
- `in[P]` = $\cup$ `out[Q]` over the predecessors Q of P

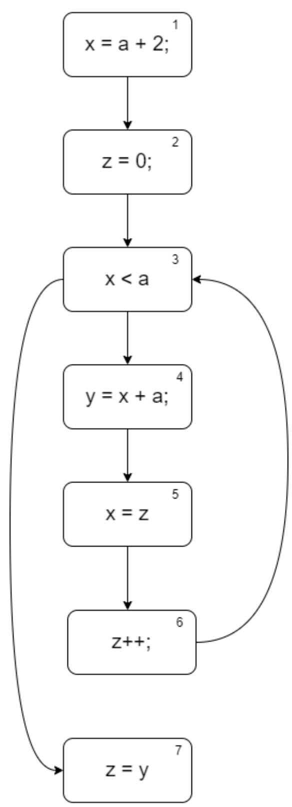
We initialize each block `out[B]` $= \emptyset$

3. *Formalize the two mutual recursive equations of the analysis:*
So we can define the mutual recursive equations TV_{in} and TV_{out}:

- $TV_{in}(p) = \emptyset$ if p is the initial point in the graph
- $TV_{in}(p) = \cup\{TV_{out}(q)|\ \textit{there is an arrow form q to p in the CFG}\}$
- $TV_{out}(p) = \{def_{TV}(p)|use_{TV}(p) \cap (S \cup TV_{in}(p)) \neq \emptyset\} \cup (TV_{in}(p) \setminus def_{TV}(p))$

4. *Show the execution steps of the analysis on a simple (but possibly non-trivial) code snippet containing at least one while loop.*
Given the initial set $S = \{a\}$ and the following code compute the TV analìysis:

```
x = a + 2;
z = 0;
while (x<a):
    y = x + a;
    x = z;
    z++;
z = y;
```



$TV_{in}(1) = \emptyset \rightarrow TV_{out}(1) = \{\text{x}\}$

$TV_{in}(2) = \{x\} \rightarrow TV_{out}(2) = \{\text{x}\}$

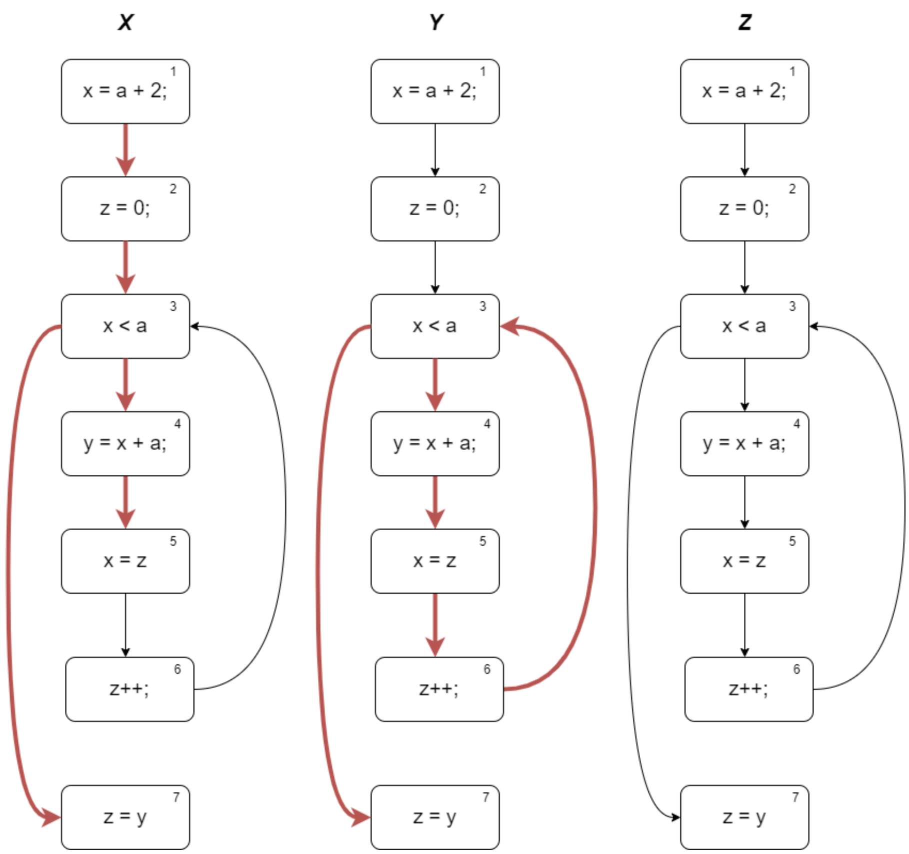
$TV_{in}(3) = \{x, y\}$[1] $\rightarrow TV_{out}(3) = \{x, y\}$

$TV_{in}(4) = \{x, y\} \rightarrow TV_{out}(4) = \{x, y\}$

$TV_{in}(5) = \{x, y\} \rightarrow TV_{out}(5) = \{y\}$

$TV_{in}(6) = \{y\} \rightarrow TV_{out}(6) = \{y\}$

$TV_{in}(7) = \{y\} \rightarrow TV_{out}(7) = \{y, z\}$

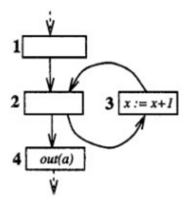


[1]Because we here we perform the union operation between out set of block 2 and block 6

2 Exercise 2

An assignment $x := a$ in a node n is faint if its left-hand variable x is such that either it is never used later on, or on every path from node n to the final nodes of the CFG every use of x is either preceded by an assignment to x or it belongs to a faint assignment.

Notice that faint analysis is not the same as liveness analysis: the following example shows a faint assignment where the variable x is not "dead".



1. *Specify the equations of a faint-analysis.* (Suggestion: define the analysis by using the results of a liveness analysis)

SOLUTION

For this type of analysis, we should start from the exit nodes and follow the CFG backward to see if a variable X is never used after his assignment or is used in a faint assignment. To do so, at the exit nodes we consider that all variables of the program are faint and if a variable is used on a statement that is not an assignment of a faint variable, then the used variable is no longer faint, instead if a variable is assigned in a statement it could be faint again.
We can define the **gen** and **kill** functions as:

- `def(B)` = {all variables defined in B}
- `use(B)` = {all variables used in B}

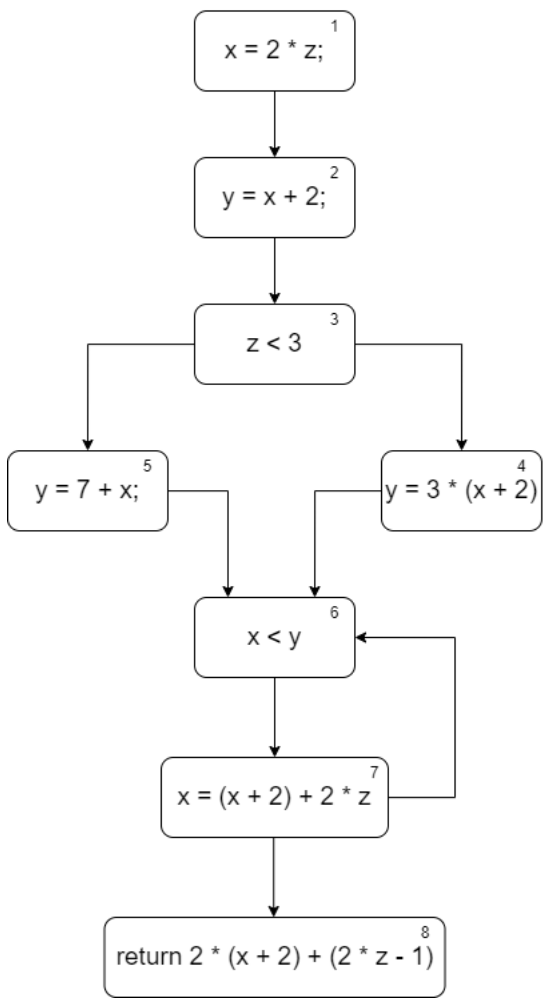
Having that $\mathbb{V}$ is the set of variable in the program, we can define the equations of a faint-analysis:

- $FA_{out}(p) = \{x | x \in \mathbb{V}\}$ if p is an exit node in the graph
- $FA_{out}(p) = \cap\{FA_{in}(q)|\ q\ follows\ p\ in\ the\ CFG\}$
- $FA_{in}(p) = def_{FA}(p) \cup (FA_{out}(p) \setminus (use_{FA}(p) | def_{FA}(p) \cap FA_{out}(p) = \emptyset))$

3 Exercise 3

```
x = 2 * z;
y= x + 2;
if(z < 3)
    y = 3 * (x + 2);
else
    y = 2 * z - 1;
while(x < y)
    x = (x + 2) + 2 * z;
return 2 * (x + 2) + (2 * z - 1)
```

SOLUTION
1. *Build the control flow graph:*



2. *Show the result of the Available Expression Analysis*

We remember that given the direction of analysis backward, and the confluence operator the union, we define how we populate the sets:

- `kill[B]` = {expressions whose operands are redefined in B without reevaluating the expression afterwards}
- `gen[B]` = {expressions evaluated in B without subsequently redefining its operands}
- `out[B]` = `gen[B]` $\cup$ (`in[B]-kill[B]`)
- `in[B]` = $\cap$ `out[P]` over the predecessor P of B

We initialize each block `in[B]` = $\emptyset$ So we can define and then provide for each Basic Block the AE_{entry} and the AE_{exit}:

- $AE_{entry}(p) = \emptyset$ if p is initial point in the graph
- $AE_{entry}(p) = \cap\{AE_{exit}(q)|$ *(q,p) in the CFG*$\}$
- $AE_{exit}(p) = gen_{AE}(p) \cup (AE_{entry}(p) \setminus kill_{AE}(p))$

$AE_{in}(1) = \emptyset$
$AE_{out}(1)$ = {2*z}

$AE_{in}(2)$ = {2*z}
$AE_{out}(2)$ = {2*z, x+2}

$AE_{in}(3)$ = {2*z, x+2}
$AE_{out}(3)$ = {2*z, x+2}

$AE_{in}(4)$ = {2*z, x+2}
$AE_{out}(4)$ = {2*z, x+2, 3*(x+2)}

$AE_{in}(5)$ = {2*z, x+2}
$AE_{out}(5)$ = {2*z, x+2, 2*z-1}

$AE_{in}(6)$ = {2*z}
$AE_{out}(6)$ = {2*z}

$AE_{in}(7)$ = {2*z}
$AE_{out}(7)$ = {2*z}

$AE_{in}(8)$ = {2*z}
$AE_{out}(8)$ = {2*z, x+2, 2*(x+2), 2*(x+2)+(2*z-1), 2*z-1}

On the basis of the results of the Available Expression Analysis, we can improve the performance of this piece of code by simply computing and store the value of $2 * z$ because we can easily see that is used in all the program

```
h = 2 * z
x = h;
y= x + 2;
if(z < 3)
    y = 3 * (x + 2);
else
    y = h - 1;
while(x < y)
    x = (x + 2) + h;
return 2 * (x + 2) + (h - 1)
```