Name: Galangkangin Gotera
Matric number: A0274760Y
Email: galangkangin@u.nus.edu

Codes

In this section, I will explain the code structure and how to build my codes.

Version of tools

Clang version: clang version 10.0.0-4ubuntu1
LLVM version: 10.0.0
C++ standard: C++14

Taint Analysis

All codes for taint analysis is located in folder **taint_analysis_task**. This folder includes the **taint_analysis.cpp** pass, and several tests* c codes to test the implementation.

A makefile is provided in the folder. Running

```
make
```

Will compile the llvm pass and all test codes into .ll files. To run a single test:

To print only taint variables

```
./taint_analysis test1.ll t
```

To print taint variables and taint path

```
./taint_analysis test2.ll p

// OR

./taint_analysis test2.ll
```

Very Busy Expression Analysis

All codes for taint analysis is located in folder **very_busy_analysis_task**. This folder includes the **very_busy_analysis.cpp** pass, and several tests* c codes to test the implementation.

Similar to taint analysis, a makefile is provided to compile everything. And running a test can use

```
./very_busy_analysis test_vb1.ll
```

Task 1

Type of analysis

For each user given input, we want to check in the future which operations use these input. Therefore, forward analysis is suitable here because of the forward flow.

We cannot guarantee that the user given input is definitely used, for example in the case of branches like these:

```
x = input()
if condition1:
  sink = x
else:
  sink = y

f(sink)
```

In this case, it is possible that sink might be tainted because it can copy x, but it is not always the case because sometimes it can take the value of y. Due to these challenges, the suitable type of analysis is “May”.

Therefore, I will be using **forward/may** analysis here.

Lattice

The lattice is defined as the set of tainted variables.

- $\top = \{x | \ x \ is \ a \ variable \ in \ the \ program\}$
- $\bot \ = \{\emptyset\}$
- $X \sqcup Y = X \cup Y$
- $X \sqcap Y \ = \ X \cap Y$

Data Structure Description

In order to implement task 2 and task 3 concurrently, I am using **map<Value*, set<vector<string>>>** to store the tainted variables. This implementation is a combination of two data structures.

- The set of keys (Value*) from the map represents all the tainted variables stored in the current context (task 2).
- The set of vectors<string> from **map[Value*]** represents all the abstract tainted data flow paths for tainted variable Value* in the current context (task 3). The contents of the vectors are the block names in sequential order.

Tainted Input Modelling

As recommended by the assignment description, I will be marking all assignments to “42” to be tainted variables. For example, in this program:

```
int source = 42, sink, x = 10, y = 20, z, a = 42;
```

Variable `source` and a are initially tainted.

Entry and Exit Equation

Assume X_1, X_2, … X_n are tainted variables set in the predecessors of instruction B.

Entry(B): $X_1 \cup X_2 \cup ... \cup X_n$

Exit(B): $(Entry(B) \setminus kill(B)) \cup gen(B))$

Gen and Kill function

Idea:

1. if an expression contains a tainted variable, then the variable assigned to it becomes tainted (gen)
2. If all variables in an expression are untainted, then the variable assigned to it becomes untained (kill)

Gen and Kill for each function:

1. **Alloca %a**: Alloca is always untainted as its just assigning a variable
    a. Gen: $\{\emptyset\}$
    b. Kill: $\{\emptyset\}$
2. **%x = binaryOp %a %b**: if %a or %b is tainted, then %x is tainted. Else if both are untainted then %x is untainted. Here binaryOP consists of add, sub, div, mul, rem
    a. Gen: $\{x \mid a \in tainted \ \vee \ b \in tainted\}$
    b. Kill: $\{x \mid a \notin tainted \ \wedge \ b \notin tainted\}$
3. **%x = load %a**: if %a is tainted, then %x is tainted
    a. Gen: $\{x \mid a \in tainted\}$
    b. Kill: $\{x \mid a \notin tainted\}$
4. **Store %a, %x**: if %a is tainted, then %x is tainted. Else %a is untainted
    a. Gen: $\{x \mid a \in tainted\}$
    b. Kill: $\{x \mid a \notin tainted\}$

Reaching Fixpoint in a simple while loop

Lets say I am processing basic block A where one of its successors is basic block B. If **exit(A) = entry(B)** then we have reached fixpoint and dont need to process B further.

My argument why this is always reached:

- If there A and B is not in a while loop, then we can process B without risk since it will never go back to A
- If A and B is in a while loop and exit(A) != entry(B), then we can process B again because the tainted variable set will get larger.
- If A and B is in a while loop and exit(A) = entry(B), it means that we have undergone one iteration from A to B where the tainted variables did not change. This means that running another iteration in the loop is useless.

Task 2

As described in task 1, I am storing the tainted variables in a **map<Value*, set<vector<string>>>**. However, for the purpose of explaining this task, I will be discussing the storage of tainted variables with **set<Value*>**, which can be derived by generating the set of keys from the map.

I am using a DFS algorithm using a stack which stores the current processed basic block and the exit tainted variables from its predecessors. Steps when processing a single basic block:

1. Union the set of tainted variables with the entry set of tainted variables in this block
2. Iterate over all instructions in this block and apply the corresponding gen and kill function to get the exit tainted variables of this block.
3. Iterate over all successors of this basic block and push them into the stack along with the exit tainted variables of this block

Experiments

More complete experiment results will be consolidated with task 3 as im using one program to generate them. However, the taint_analysis.cpp does provide an option to only print the tainted vars in each block as opposed printing it with the taint flow graph. We can run it with this option

Option t to print taint vars, option p (default) to print taint paths

```
./taint_analysis example1.ll t
```

I ran experiments on the two examples given in the assignment

Example1.c

```
TAINTED VARIABLES:
label entry:
i, source, %0

label if.else:
i, k, source, %0, %2

label if.end:
```

```
i, k, sink, source, %0, %3, %2

label if.then:
i, source, %0
```

Output is exactly the same as the assignment

Example2.c

```
TAINTED VARIABLES:
label entry:
source

label if.else:
source

label if.end:
j, source, %1

label if.then:
j, source, %1
```

Output is exactly the same as assignment

Task 3

This task asks us to generate all abstract taint paths to taint a variable in each block. To support that, I am using **map<Value*, set<vector<string>>>**. The idea is that every time variable x taints variable y, we copy all paths from map[x] to map[y]. And if x and y are in different blocks, then we append the block of “y” to the end of each of the vectors. Pseudocode below:

```
copyTaintedFlow(set<vector<string>> &from, set<vector<string>> &to,
string blockNameofTo) {

  for(taintedPath: from) {
    if(taintedPath.back() != blockNameOfTo)
taintedPath.append(blockNameOfTo);

    to.insert(taintedPath)
```

```
  }

}
```

Also note that if a variable **x** became untainted, then we erase the entry map[x] from the data structure.

Our block equations require us to union two map<Value*, set<vector<string>>> to calculate entry block of successor. The merging algorithm is straightforward. Create a new map with with union of keys from the two maps. If two keys are the same in the map, then union the two sets in these two keys.

Experiment results

In this part, we will detail the experiments we ran.

Test1.c

This was given in the assignment.

```
TAINTED VARIABLES:
label entry:
1. source:
tainted data flow paths:
- entry


label if.else:
1. source:
tainted data flow paths:
- entry


label if.end:
1. source:
tainted data flow paths:
- entry

2. sink:
tainted data flow paths:
- entry -> if.then
```

```
3. %2:
tainted data flow paths:
- entry -> if.then


label if.then:
1. source:
tainted data flow paths:
- entry

2. sink:
tainted data flow paths:
- entry -> if.then

3. %2:
tainted data flow paths:
- entry -> if.then
```

We can see that even though sink is only set to source in one if branch, it is still tainted in if.end

Test2_wipetaint.c

In this test, there is a source = a + c which wipes the taint in source. Therefore only D is tainted

```
TAINTED VARIABLES:
label entry:
1. source:
tainted data flow paths:
- entry


label if.else:
1. d:
tainted data flow paths:
- entry -> if.else

2. %3:
tainted data flow paths:
- entry -> if.else
```

```
label if.end:
1. d:
tainted data flow paths:
- entry -> if.else

2. %3:
tainted data flow paths:
- entry -> if.else


label if.then:
```

As expected, source became untainted because it was assigned to an untainted variable. As a replacement, d and %3 became tainted.

Test_tast3.c

This code was given in the assignment to test if taint path is working.

```
TAINTED VARIABLES:
label entry:
1. source:
tainted data flow paths:
- entry


label if.else:
1. k:
tainted data flow paths:
- entry -> if.else

2. source:
tainted data flow paths:
- entry

3. %2:
tainted data flow paths:
- entry -> if.else
```

```
label if.end:
1. i:
tainted data flow paths:
- entry -> if.then

2. k:
tainted data flow paths:
- entry -> if.else

3. sink:
tainted data flow paths:
- entry -> if.else -> if.end
- entry -> if.then -> if.end

4. source:
tainted data flow paths:
- entry

5. %1:
tainted data flow paths:
- entry -> if.then

6. %2:
tainted data flow paths:
- entry -> if.else

7. %3:
tainted data flow paths:
- entry -> if.then -> if.end

8. %4:
tainted data flow paths:
- entry -> if.else -> if.end

9. add:
tainted data flow paths:
- entry -> if.else -> if.end
- entry -> if.then -> if.end


label if.then:
1. i:
```

```
tainted data flow paths:
- entry -> if.then

2. source:
tainted data flow paths:
- entry

3. %1:
tainted data flow paths:
- entry -> if.then
```

Output is as expected. We can see that source to reach k, and source to reach sink in if.end is exactly as expected from the assignment document.

Test_task3_2.c

In this task, I want to see if my program can correctly capture taint paths that are not contiguous.

```
TAINTED VARIABLES:
label entry:
1. source:
tainted data flow paths:
- entry


label if.else:
1. source:
tainted data flow paths:
- entry


label if.else3:
1. i:
tainted data flow paths:
- entry -> if.else3

2. source:
tainted data flow paths:
- entry
```

3. %3:
tainted data flow paths:
- entry -> if.else3

label if.end:
1. source:
tainted data flow paths:
- entry

label if.end4:
1. i:
tainted data flow paths:
- entry -> if.else3

2. k:
tainted data flow paths:
- entry -> if.then2

3. sink:
tainted data flow paths:
- entry -> if.else3 -> if.end4
- entry -> if.then2 -> if.end4

4. source:
tainted data flow paths:
- entry

5. %2:
tainted data flow paths:
- entry -> if.then2

6. %3:
tainted data flow paths:
- entry -> if.else3

7. %4:
tainted data flow paths:
- entry -> if.then2 -> if.end4

8. %5:
tainted data flow paths:
- entry -> if.else3 -> if.end4

9. add:
tainted data flow paths:
- entry -> if.else3 -> if.end4
- entry -> if.then2 -> if.end4

```
label if.then:
1. source:
tainted data flow paths:
- entry


label if.then2:
1. k:
tainted data flow paths:
- entry -> if.then2

2. source:
tainted data flow paths:
- entry

3. %2:
tainted data flow paths:
- entry -> if.then2
```

Output is as expected. Even though there is block if.else. Because those block does not taint the variable, the data flow path goes directly to if.else3 and if.then2 where all the tainting happen.

Task 4

In this part, I am tasked to make the algorithm work for in while loops. To achieve this, I incorporated the technique to reach fixpoint described in task 1. When checking for successor, I check if the entry vars set is the same as the exit vars set of this block. Pseudocode:

```
// traversalStack is used to do the DFS
for(Succ: successors(BB)) {

  string sucName = getSimpleNodeLabel(Succ);

  if(exitTaintedVars != taintAnalysisMap[sucName])
      traversalStack.push({Succ, exitTaintedVars});
  }
}
```

Experiment results

We ran the experiment on the given while loop program

Test_task4.c

```
TAINTED VARIABLES:
label entry:
source

label if.else:
i, j, source, %3, %4

label if.end:
i, j, source, %3, %4

label if.then:
i, j, source, %3, %4

label while.body:
i, j, source, %3, %4

label while.cond:
i, j, source, %3, %4

label while.end:
i, j, sink, source, %6, %3, %4
```

Output is exactly as expected from the assignment files.

Test_task4_2.c

I made a similar test to the first one. Here even though the loop will taint j, I am untainting it in the body after the if. This means j will never be tainted when exiting the body.

```
TAINTED VARIABLES:
label entry:
source
```

```
label if.else:
i, j, source, %3, %4

label if.end:
i, source, %3, %4

label if.then:
i, source, %3, %4

label while.body:
i, source, %3, %4

label while.cond:
i, source, %3, %4

label while.end:
i, source, %3, %4
```

Output is as expected. Even though j is tainted at first by source in if.else, it is eventually untainted by assigning to a. Therefore j will always be untainted after moving from the loop body.

Task 5

In this task, I am assuming **only binary task can be very busy expressions**. Reasoning for excluding other instruction:

1. Alloca: each alloca uniquely allocates a variable
2. Store: not an expression
3. Load: Hoisting this is a not productive. If we hoist a load instruction above an if statement to a variable x, we would still need to copy x to the variable where the original load instruction is hoisted from.

There are two parts of the algorithm. First part is to figure out the actual variable name used in each binary operator, and then actually figuring out which expressions is always used.

Replacing Registers with Variable name

In test_vb1.ll, we have these two blocks:

```
// if.then
%2 = load i32, i32* %b, align 4
%3 = load i32, i32* %a, align 4
%sub = sub nsw i32 %2, %3
store i32 %sub, i32* %x, align 4
%4 = load i32, i32* %a, align 4
%5 = load i32, i32* %b, align 4
%add = add nsw i32 %4, %5
```

```
// if.then2
%8 = load i32, i32* %a, align 4
%9 = load i32, i32* %b, align 4
%add3 = add nsw i32 %8, %9
store i32 %add3, i32* %x, align 4
%10 = load i32, i32* %b, align 4
%11 = load i32, i32* %a, align 4
%sub4 = sub nsw i32 %10, %11
store i32 %sub4, i32* %y, align 4
br label %if.end
```

Notice that `%add = add nsw i32 %4, %5` and `%add3 = add nsw i32 %8, %9` are actually the same instruction. Because they load from %a and %b respectively. Unfortunately,

they are first loaded into different registers and so we cannot directly compare these two instructions. What I must do first is to change every binary instruction's register into a variable name. This is the purpose of **convertInstructionToVariable**

The algorithm is for every **%x = load %a** instruction, to make %x synonymous to %a. This will always be the case because registers can only be set once, and therefore its link will never change in the future. This will be the if.then block after running **convertInstructionToVariable**

```
// if.then
%2 = load i32, i32* %b, align 4
%3 = load i32, i32* %a, align 4
%sub = sub nsw i32 %b, %a
store i32 %sub, i32* %x, align 4
%4 = load i32, i32* %a, align 4
%5 = load i32, i32* %b, align 4
%add = add nsw i32 %a, %b
```

```
// if.then2
%8 = load i32, i32* %a, align 4
%9 = load i32, i32* %b, align 4
%add3 = add nsw i32 %a, %b
store i32 %add3, i32* %x, align 4
%10 = load i32, i32* %b, align 4
%11 = load i32, i32* %a, align 4
%sub4 = sub nsw i32 %b, %a
store i32 %sub4, i32* %y, align 4
br label %if.end
```

Now we can see that the expression **%sub** and **%sub4** are identical. So is **%add3** and **%sub4**

Note that a binary operation's result is considered a "variable" here. For example:

```
// if.then2
%8 = load i32, i32* %a, align 4
%9 = load i32, i32* %b, align 4
%add3 = add nsw i32 %a, %b
store i32 %add3, i32* %x, align 4
%sub4 = sub nsw i32 %add3, %a
store i32 %sub4, i32* %y, align 4
br label %if.end
```

In this case, on the %sub4 expression will not be decomposed back to %a and %b and will stay as is. Besides, this case does not matter because **sub %add3, %a** is never going be a very busy expression because we are re-setting the value of %add3 above.

Very Busy Analysis

After preprocessing done in the previous section. I did the very busy analysis. Since this is a backward analysis, I will be doing a DFS from each exit point of the program. The program's exit point can be found by iterating all basic blocks and finding basic blocks **with no successors**.

My algorithm is as follows:

1. Initialize DFS with all basic blocks with no successors
2. On a DFS instance, take the given very busy expressions, **intersect** with the current exit very busy expression, and run it in the basic block based on the gen and kill instructions given in the slides. Essentially
    a. **store** operations *kill* expressions (setting new values)
    b. binary operations *gen* operations.
3. After the entry very busy expressions is obtained, push all predecessors to the stack with the obtained entry very busy expression.

To store the very busy expressions, I am using a **Deque<Instruction*>.** The reason of using Deque over Set:

- Maintain relative expression ordering
- I cannot find a correct ordering function. For example, %a = add %x, %y and %b = add %x, %y are equivalent. However I dont know how to compare if they are not equal. Comparing by string representation does not work because for example:
    - %b = add %x, %y is in the set
    - %a = sub %y, %x is in the set
    - We want to find if %a = add %x, %y is in the set (exist, because %b = add %x, %y)
    - But due to the nature of sets (binary tree). Theres a chance it will compare with with "%a = sub %y, %x " first. Since it is lexicographically smaller, it will go left. Hence not finding "%b = sub %x, %y " which is in the right of the binary tree

Then to insert an expression in the deque, I just check if all the variables used in the operation are the same (ignoring where its going to be stored at).

I also add some **simple loop support**. Before pushing, if the entry very busy expression is the same as exit very busy of predecessors, then dont push to stack.

Experiments

To keep the report short, I will not write my code in this report. I have written comments on intended behaviour in the corresponding test code so their purpose is clear.

Test_vb1.c

In this test there are three blocks inside the if. The third block has a “x = b * a;” which is not present in other blocks and hence not very busy. The other two a + b and b - a are very busy.

Output:

```
VERY BUSY EXPRESSIONS:
===== entry =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

exit very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

===== if.then =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

exit very busy expressions:

===== if.else =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b
  %sub4 = sub nsw i32* %b, %a

exit very busy expressions:
  %add3 = add nsw i32* %a, %b
  %sub4 = sub nsw i32* %b, %a

===== if.then2 =====
entry very busy expressions:
```

```
  %add3 = add nsw i32* %a, %b
  %sub4 = sub nsw i32* %b, %a

exit very busy expressions:

===== if.else5 =====
entry very busy expressions:
  %add6 = add nsw i32* %a, %b
  %sub7 = sub nsw i32* %b, %a
  %mul = mul nsw i32* %b, %a

exit very busy expressions:

===== if.end =====
entry very busy expressions:

exit very busy expressions:

===== if.end8 =====
entry very busy expressions:

exit very busy expressions:
```

We can see that `%mul = mul nsw i32* %b, %a` is only very busy in its isolated block. But when moving from the ifs (block entry), only the %add and %sub are very busy.

Test_vb2.c

In this test, I am testing if the kill functions work correctly. There is c = x + b and d = y + b which should not be very busy because x was redefined above.

Output:

```
VERY BUSY EXPRESSIONS:
===== entry =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b
```

```
exit very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

===== if.then =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

exit very busy expressions:

===== if.else =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b
  %sub4 = sub nsw i32* %b, %a

exit very busy expressions:

===== if.end =====
entry very busy expressions:

exit very busy expressions:
```

As expected, **add %x, %b** and **add %y, %b** is not very busy.

Test_Vb3.c

In this test, there are three blocks inside the if. However in one block a + b is not present. Hence only b - a is very busy

```
VERY BUSY EXPRESSIONS:
===== entry =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a

exit very busy expressions:
  %sub = sub nsw i32* %b, %a

===== if.then =====
entry very busy expressions:
```

```
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

exit very busy expressions:

===== if.else =====
entry very busy expressions:
  %sub4 = sub nsw i32* %b, %a

exit very busy expressions:
  %sub4 = sub nsw i32* %b, %a

===== if.then2 =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b
  %sub4 = sub nsw i32* %b, %a

exit very busy expressions:

===== if.else5 =====
entry very busy expressions:
  %add6 = add nsw i32* %a, %c
  %sub7 = sub nsw i32* %b, %a

exit very busy expressions:

===== if.end =====
entry very busy expressions:

exit very busy expressions:

===== if.end8 =====
entry very busy expressions:

exit very busy expressions:
```

As expected, only %sub = sub nsw i32* %b, %a became very busy at the top because only that is present in all three if blocks

Test_Vb4.c

In this test, i am checking my logic for nested ifs. In this code, both a + b and b - a is present. However, b - a is present in a nested if on a block. This means that **b - a is not very busy** because its not always computed.

```
VERY BUSY EXPRESSIONS:
===== entry =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b

exit very busy expressions:
  %add3 = add nsw i32* %a, %b

===== if.then =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b
  %add13 = add nsw i32* %w, %z

exit very busy expressions:
  %add13 = add nsw i32* %w, %z

===== if.else =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b

exit very busy expressions:
  %add3 = add nsw i32* %a, %b

===== if.then2 =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b
  %mul = mul nsw i32* %a, i32 2

exit very busy expressions:

===== if.then5 =====
entry very busy expressions:
  %sub6 = sub nsw i32* %b, %a

exit very busy expressions:

===== if.end =====
entry very busy expressions:
  %add13 = add nsw i32* %w, %z

exit very busy expressions:
```

```
  %add13 = add nsw i32* %w, %z

===== if.else7 =====
entry very busy expressions:
  %add8 = add nsw i32* %a, %b
  %sub9 = sub nsw i32* %b, %a
  %mul10 = mul nsw i32* %b, %a

exit very busy expressions:
  %add13 = add nsw i32* %w, %z

===== if.end11 =====
entry very busy expressions:
  %add13 = add nsw i32* %w, %z

exit very busy expressions:
  %add13 = add nsw i32* %w, %z

===== if.end12 =====
entry very busy expressions:
  %add13 = add nsw i32* %w, %z

exit very busy expressions:

===== return =====
entry very busy expressions:

exit very busy expressions:
```

As we can see, only a + b is very busy at the top.

Test_vb5.c

In this test, I want to see if the kill function works correctly. In the third block, we have a = w + b, reinitializing a. Therefore, z = b - a on the following line is not very busy

```
VERY BUSY EXPRESSIONS:
===== entry =====
entry very busy expressions:
  %add = add nsw i32* %a, %b
```

```
exit very busy expressions:
  %add = add nsw i32* %a, %b

===== if.then =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

exit very busy expressions:

===== if.else =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b

exit very busy expressions:
  %add3 = add nsw i32* %a, %b

===== if.then2 =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b
  %sub4 = sub nsw i32* %b, %a

exit very busy expressions:

===== if.else5 =====
entry very busy expressions:
  %add6 = add nsw i32* %a, %b

exit very busy expressions:

===== if.end =====
entry very busy expressions:

exit very busy expressions:

===== if.end9 =====
entry very busy expressions:

exit very busy expressions:
```

As expected, on that if block (if.else5), only %add6 = add nsw i32* %a, %b is very busy. Therefore only add is very busy at the top.

Test_vb6.c

In this test, I am testing how my program works for a simple loop.

```
VERY BUSY EXPRESSIONS:
===== entry =====
entry very busy expressions:

exit very busy expressions:

===== while.cond =====
entry very busy expressions:

exit very busy expressions:

===== while.body =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

exit very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

===== if.then =====
entry very busy expressions:
  %sub = sub nsw i32* %b, %a
  %add = add nsw i32* %a, %b

exit very busy expressions:

===== if.else =====
entry very busy expressions:
  %add3 = add nsw i32* %a, %b
  %sub4 = sub nsw i32* %b, %a

exit very busy expressions:

===== if.end =====
entry very busy expressions:

exit very busy expressions:

===== while.end =====
entry very busy expressions:

exit very busy expressions:
```

From this experiment I got a very good observation:

1. No expressions from inside the loop body will be very busy outside the loop body, unless its also present outside the loop body. This is because its not certain whether we will enter the loop, therefore not guaranteeing evaluating this expression
2. Very busy inside the loop will be isolated and we can perform very busy analysis inside the loop treating it like a normal body.