CSC4001 Project Report

Author: Ziyu XIE (谢子钰) **Student ID:** 121090642

I have finished all the parts including **bonus**.

1. Differential Testing

1.1. Differential Testing Overview

Differential testing is a **black-box technique** that works well when **systems implement the same behavior**.

The Project requires us to use differential testing to detect bugs of some buggy interpreters. If we give the same input (`input.pig`) into the interpreters, and the interpreters are of no bugs, then the outputs of all the interpreters should be the same. Then, since there are some bugs in the given interpreters, we can use our own correct interpreter (`pig.py`) to detect the bugs. If the buggy interpreters have the different output with our interpreter's result, we conclude that this interpreter contains bug.

In this testing, we need to firstly use `gen.py` to randomly generate non-empty PIG program, and we need to guarantee that this PIG program has no bug. Then we use `pig.py` to give the correct result `1.out` and use the buggy interpreter to give the result `2.out`, and finally we compare them to detect bug. This will go 100 times for one buggy interpreter.

Then, the problem is converted to how to implement `gen.py` and `pig.py`.

1.2. Implementing Generator

In order to generate a correct PIG code, I raise up an idea called **"Snowflake Block Generation Method"**. We let one basic block generate multiple blocks, and then let these blocks continue to generate more blocks, spreading like snowflakes until our maximum file length requirement (1000) is met.

Since the PIG language contains five kinds of statement "D", "A", "B", "O", "R", we can define five kinds of basic blocks (not exactly corresponding to statements but five basic functions), called "BASIC Block", "ASSIGN Block", "OUTPUT Block", "IF Block", and "FOR Block".

1.2.1. BASIC BLOCK Structure

The BASIC Block is structured as below:

```
BASIC_BLOCK (
    ''' declare many variables '''
    D TYPE VARx1
    # ... multiple lines
    D TYPE VARxn

    for _ in range(iter_num):
        # randomly choose one block to generate:
        ( BASIC_BLOCK or
```

```
            IF_BLOCK or
            FOR_BLOCK or
            ASSIGN_BLOCK or
            OUTPUT_BLOCK
        )
    ''' destroy these variables '''
    R TYPE VARx1
    # ... multiple lines
    D TYPE VARxn
)
```

The BASIC Block will know (passed by parameters):

1. how many lines have been generated (should not exceed maximum number),
2. which variables have been declared (should not redeclare),
3. which variables should not be used here (because some other blocks may not want some variables to be modified, e.g. FOR Block),
4. the iteration number of this function has been used (to guarantee that it can be stopped and will not reach to the maximum recursion times of Python).

If all the steps inside the BASIC Block are correct, then this block is guaranteed to not redeclare or undeclare variables before usage.

1.2.2. OUTPUT BLOCK Structure

The OUTPUT Block is structured as below:

```
OUTPUT_BLOCK (
    ''' output many variables '''
    O VARx1
    # ... multiple lines
    O VARxn
)
```

The OUTPUT Block will know (passed by parameters):

1. how many lines have been generated,
2. which variables have been declared.

1.2.3. ASSIGN BLOCK Structure

The ASSIGN Block is structured as below:

```
ASSIGN_BLOCK (
    ''' assign many variables '''
    A VARx1 Exp
    # ... multiple lines
    A VARxn Exp
)
```

The ASSIGN Block will know (passed by parameters):

1. how many lines have been generated,
2. which variables have been declared,
3. which variables should not be used here.

1.2.4. IF BLOCK Structure

The IF Block is structured as below:

```
IF_BLOCK (
    ''' creating conditonal variablen '''
    D TYPE VAR1
    A VAR1 Exp
    B tar_line VAR1
    ''' generate BASIC block inside the IF block '''
    BASIC_BLOCK
    ''' destroy conditional variable '''
    R VAR1
)
```

The IF Block will know (passed by parameters):

1. how many lines have been generated,
2. which variables have been declared,
3. which variables should not be used here.

1.2.5. FOR BLOCK Structure

The FOR Block is structured as below:

```
IF_BLOCK (
    ''' creating iteration variable '''
    D TYPE VAR1
    A VAR1 Exp
    ''' generate BASIC block inside the FOR block '''
    BASIC_BLOCK
    ''' next iteration '''
    A VAR1 ( VAR - 1 )
    B tar_line VAR1
    ''' destroy conditional variable '''
    R VAR1
)
```

The IF Block will know (passed by parameters):

1. how many lines have been generated,
2. which variables have been declared,
3. which variables should not be used here.

1.2.6. Expression Structure

Similar to the blocks, the expression can also be generated using recursion. The concept of this part is omitted here because it is simple and in fact it is just the inverse process of the calculating the expression values in the `pig.py`, which we will talk about later. (In my project, I firstly did the `pig.py`, and then did the `gen.py`)

1.2.7. Conclusion

Based on all the structures below, we just need to set some hyper-parameter such as max_line_num, max_iteration_num, block_iter_num, etc. Finally the `gen.py` can work properly.

1.3. Implementing Interpreter

Since the PIG language contains five kinds of statement "D", "A", "B", "O", "R", we need to firstly judge the statement type, then do interpreting. For "D" and "R", we simply use a dictionary in Python to store all the variables declared and destroyed. For "O", we just output the value of the variable based on corresponding variable types.

Then the remain things is to do "A" and "B" statement. For "A" statement, we need to give the variable the corresponding expression values, and for "B" statement, we need to decide whether to branch based on the expression given (branching is just changing the global pc value). Therefore, if we implement how to calculate the expression correctly, we can easily finish the interpreter.

1.3.1. Calculate Expression

Based on the given listing forms of the Exp:

```
Exp -> LP CONSTANT RP
    -> LP VAR RP
    -> LP Exp Bop Exp RP
    -> LP NOT Exp RP
```

We conclude that the expression is calculated in a recursive way, and the stopping criterion is encountering "CONSTANT" or "VAR".

In order to simplify our explanation, we simply say "CONSTANT" and "VAR" types have 3 tokens, "Bop" type has >= 5 tokens, and "NOT" type has >= 4 tokens.

Firstly, we judge whether the tokens number is equal to 3. If yes, then it must be a CONSTANT or VAR, this judgement is easy, and we return the value based on CONSTANT value or VAR value.

Secondly, if we detect that the second token is "NOT", we conclude that this is a NOT type, and hence do NOT with Exp, where Exp can be calculated recursively.

Thirdly, the type must be "Bop" if the PIG code is correct. Similarly, we calculate the value for Exp1 and Exp2 recursively and return the result.

The algorithm for calculating Exp can be:

```
expCalculation {
    if (len(tokens) == 3):
        if CONSTANT:
            do res = CONSTANT calculation
            return res
        else:
            do res = VAR calculation
            return res
    elif (judge NOT):
        do Exp calculation
        res = NOT Exp
        return res
    else:
        do Exp1 and Exp2 calculation
        res = Exp1 Bop Exp2
        return res
}
```

The remaining things are about the bit transforming and some bit operations, which are simple, hence omitted here.

1.3.2. Conclusion

After finishing all things above, the algorithm for the whole process can be given as:

```
while (new_line is ok) {
    if exceeding 5000:
        break
    else:
        get type of statement
        do corresponding things
        update pc value
}
```

The code can correctly read `input.pig` and output the result to `1.out`.

2. Metamorphic Testing

2.1. Metamorphic Testing Overview

The Project requires us to use differential testing to detect bugs of some buggy interpreters. We use `gen_meta.py` to generate two inputs (`input1.pig` and `input2.pig`) which can realize specific functions to the buggy interpreter and use `checker.py` to check whether there are some bugs.

2.2. Implementing the Generator and Checker

Since the PIG language contains five kinds of statement "D", "A", "B", "O", "R", we can check the functions of them separately. We firstly identify which kind of bug they may occur.

For "D" statement, the possible bugs are:

1. Do not declare the correct variable
2. Do not initialize the variable with zero
3. Do not declare with the correct type

For "R" statement, the possible bugs are:

1. Do not destroy the correct variable

For "O" statement, the possible bugs are:

1. Do not output
2. Output the wrong variable
3. Output the variable value but with wrong type

For "A" statement, the possible bugs are:

1. Wrong calculation of expressions
2. Assign to wrong variables

For "B" statement, the possible bugs are:

1. Do not do the determination of branching correctly
2. branch to the wrong lines

In order to test these bugs, I designed four checking blocks, including "checkDsRs", "checkOs", "checkAsExps", and "checkBs".

2.2.1. Idea of "checkDsRs"

This part is used to check the "D" and "R" statements. In order to check whether the variables are declared and destroyed correctly, we use some variables to do D and R multiple times. Do output directly and then do assign-output. So that these variables should be output 0 first, then be output many different values. The checker is designed consistent with this part. The structure of this part PIG code is:

```
DOUBLE TIME:
    D (many lines)
    O (many lines)
    A (many lines)
    O (many lines)
    R (many lines)

# two files have different var name but same value
```

2.2.2. Idea of "checkOs"

This part, we just assign variables with different kinds of values and types, then do the output. Each time we choose different types but the same value to assign, hence the output must be **different**. If there is something consistent, then there is a bug. The structure of this part PIG code is:

```
MANY TIMES:
    D type1 or type2 var
    A var values
    O var
    R var

# two files have different type1 or type2
```

2.2.3. Idea of "checkAsExps"

The idea is that we assign different expressions to the variable and check whether Bops and NOT works well, in addition with assigning. For example, we give `A var ( Exp1 + Exp2 )` to file 1, and give `A var ( {result of Exp1 + Exp2} )`, which should give the same answer. The structure of this part PIG code is:

```
# file one
D (many variables for usage)
MANY TIMES:
    D (testing variable)
    A var ( Exp1 + Exp2 )
    O var
    A var ( Exp1 - Exp2 )
    O var
    A var ( Exp1 & Exp2 )
    O var
    A var ( Exp1 | Exp2 )
    O var
    A var ( ! Exp1 )
    O var
R (many variables for usage)
```

```
# file two
```

```
D (many variables for usage)
MANY TIMES:
    D (testing variable)
    A var ( {result of Exp1 + Exp2} )
    O var
    A var (  {result of Exp1 - Exp2} )
    O var
    A var (  {result of Exp1 & Exp2} )
    O var
    A var (  {result of Exp1 | Exp2} )
    O var
    A var (  {result of ! Exp1} )
    O var
R (many variables for usage)
```

2.2.4. Idea of "checkBs"

We want to test whether the branch works well or not, we consider the structure below:

```
A var 0
B LINE:4 Exp1
A var Exp2
O var
```

And for the second file, we consider it has branched correctly:

```
A var 0
(A var Exp2) if Exp1 == 0
O var
```

If the branch is working well, then the output should be the same for var.

We do this for multiple times and get the result.

2.2.5. Conclusion

For the checker, the first, third, and fourth parts should be exactly the same, while the second part must be different. The checker use this logic to test. Finally, we store the result in `res.out` file.

3. Dataflow Analysis

3.1. Dataflow Analysis Overview

The dataflow analysis can be used for detecting some features of a code. Specifically, we can use **Reaching Definitions Analysis** to detect the potential undeclared variables in a program by adding the dummy variables in the beginning.

The algorithm of Reaching Definitions Analysis is shown below:

```
INPUT: CFG (kill_B and gen_B computed)
OUTPUT: IN[B] and OUT[B] for each basic block B
    OUT[entry] = empty set
    for each basic block B except entry:
        OUT[B] = empty set
    while (changes to any OUT occur):
        for each basic block B except entry:
            IN[B] = U_P a predecessor of B OUT[P]
            OUT[B] = gen_B U (IN[B] - kill_B)
```

Use this algorithm, we do reaching definitions analysis on the PIG code, and the result where dummy variables have not been killed should be the potential undeclared variables. And if the basic block uses these variables, the corresponding line will be considered as using of undefined variable in PIG code.

3.2. Implementing DA program

The program includes three main parts: 1. constructing the Control flow graph, 2. do reaching definition analysis, 3. do undeclared variable detection.

3.2.1. Implementing CFG Constructing

We need to determine the leaders in a sequence of three-address instructions of P and build the basic blocks.

For determining the leaders, we consider:

1. The first instruction in P is a leader
2. Any target instruction of a conditional or unconditional jump (in PIG code, is the branching statement) is a leader
3. Any instruction that immediately follows a conditional or unconditional jump is a leader

After determining all the leaders, we create basic blocks, including the entry and exit blocks (for efficient usage). Then we construct the edges between each block and finally make it a graph-like structure.

Here we just consider each blocks' end line. If this line is not a "B" statement, then it only goes to the next block. Otherwise, it goes to both the next block and the target block of the branching statement. We can also get all the **predecessors** of every block.

Finally, the CFG result should be structured like:

```
blocks_res = [
    [(start_line, end_line),
    prev_block (),
    OUT [0]^1000 concat [0]^declare_size (low to high)
    ],
    ...
] (idx is the block idx)

declare_res = [
    (line, var),
    ...
] (idx is the bit idx)
```

Note that the OUT integer is constructed by 1000 zeros and declare_size zeros from lower bit to higher bit. The declare_res is just used for determine the line and variables declared corresponding to a fixed index in the OUT.

3.2.2. Doing Reaching Definition Analysis

Then we can do reaching definitions analysis, we let the entry's OUT be 1000 ones bits so that the dummy variables are represented for from v000 to v999. After the reaching definitions analysis, the OUT for each block should be changed.

In details, it goes through each lines of one block and detect the declared variables (gen_B) from "D" statement, and detect the destroyed variables (kill_B) from "R" statement. At the end, we get the new OUTPUT of the block. We update until all OUTPUT remains the same.

3.2.3. Detecting Undeclared Variables

Finally, we execute each block again with known input. Similar to the analysis, but we additionally detect all variables to be used from "A", "B", "O", "R" statements. If the input has a dummy variable but we use that variable in our block, we conclude that line uses undefined variables. We store the information and finally output the number of lines.

4. Project Conclusion

This project is really wonderful. I learned how to use differential testing and metamorphic testing to detect bugs and how to use dataflow analysis to detect some code bugs. These things are useful.