PIG Language: Interpreter, Testing and Dataflow Analysis

Alfonsus Rodriques Rendy

April 2024

1 Introduction

This project aims to implement testing and static analysis on PIG programming language interpreter. The testing includes (1) differential testing and (2) metamorphic testing. The dataflow analysis will analyze possible undefined variable in a PIG languages test case.

2 Differential Testing

Differential testing is done by comparing the output of an oracle which is guaranteed to behave similarly with the expected correct program. In this part two program is created: the oracle (PIG interpreter) and the random test case generator (fuzzer)

2.1 PIG Interpreter

The PIG interpreter takes a PIG code and generate the output based on the grammar of the language. The interpreter is designed to iterate the statement if the input code and terminates when one of the following happens

- An invalid statement is found (i.e., violates PIG grammar)
- Finish reading all lines (EOF)
- Reach 1001th line (e.g., invalid input, as PIG code must only have at max 1000 lines)
- Generate more than 5000 statements

The interpreter will have a global data structure to store the current live variable. The data structure is implemented as a dictionary storing both value (as a bit vector string) and the type of the variable, which is one of `{bv8, bv16, bv32, bv64}`. At every statement's execution, four instructions can occur

1. **D** : Declaring a new variable, where the correspond variable will be stored in the data structure (i.e. changing from `None` to a valid variable. Note that when such a variable already exist, the interpreter will raise an error, as the pig program is trying to declare a duplicate variable.
2. **A** : Assigning a variable, where the corresponding value evaluated from the expression will be stored
3. **B** : Branching based on the expression where the current line will be switched to the jump line when the expression evaluated is not 0
4. **O** : Output a variable value to the output file
5. **R** : Remove a variable from the data structure, where the value of the corresponding variable in the data structure will be changed to `None`

For each corresponding instruction, the statement will be parsed based on the syntax grammar of the PIG language.

To evaluate the expression, the interpreter will use two-stack mechanism where each stack will store the operand and operator. For each character(s) separated by space in the expression, the expression evaluator will determine the correct operation

- If operand is found, push it to the operand stack

- If operator except (is found, push it to operator stack
- If operator) is found, pop the last operand.
  - If the operand is a binary operator (ADD, SUB, AND, OR), then pop two operands, evaluate the expression, and push the result to the operand stack
  - If the operand is a unary operator (NOT), then pop one operand, evaluate the expression, and push the result to the operand stack
  - If the operand is open parenthesis '(', then pop the last operand, determine to either push it again (if it is a constant) or fetch the variable and push it to the operand stack (if it is a variable)

The operation evaluation is implemented using bitwise operation which is the most efficient in Python.

2.2 Testcase Generator

The test case generator is a random generator to generate every possible valid cases that the interpreter will encounter.

The generator is implemented in two modes to cover both cases. First is the use of complex nested branch, with constraint that no declaration and removal of variable in the middle of the code to avoid any undeclared variable or duplicate variable declaration. It is challenging to include the declaration and removal of variable due to complex branching behavior. The second is the use of simple sequential branching (no nested branch) but with declaring and removing variable in the middle of the branch. The generation mode is declared at the start of generation with flag `NO_INTER_VAR` being true for the first type generation.

First Type Generation: Nested Branch with No Intermediate Variable Declaration/Removal The first type is when nested branch is generated. Nested branch occurs when exists a branch inside another branch block that jumps to either inside or outside of the branch block creating a complex branching flows.

To avoid any invalid PIG generation, there will be no variable declaration and removal in the middle part of the code as finding a variable name that can be safely used or declared needs a behavior (static) analysis that is sound, but it can only be approximated using may/must analysis.

The nested branch is generated by first creating branch in the first layer. After the first layer branch is generated, it will generate the branch scope which then will have choice to generate `A, O, B` instructions in-between.

Second Type Generation: Sequential Branch with Intermediate Variable Declaration/Removal The second type is sequential branching that allows intermediate variable declaration and removal.

The branch is generated with similar mechanism. First, it will generate first layer branch which then will generate the branch scope. In the branch scope, the instruction allowed is different with the first type generation, which only allow `D, A, O, R` instructions.

In each branch scope, every newly declared variable will be removed before the scope ended to ensure the correct behavior of the generated code. The removal mechanism is done using a priority queue. For each declared variable, it will reserve a line to remove the variable, the line of which is stored in the priority queue for efficient check in every line iteration. If a line is already reserved, remove instruction will be added to such line.

Instruction Generation Instruction is randomly generated with dynamic update on the probability weights of each instruction. Generally branch will occupies 10% and declare will have decreasing probability as the generator reach the end of the lines (1000 lines).

For the first type generation, declaration and removal instruction will have 0 weights, meaning the generator will only generates branch, assignment, and output instructions. To make sure there exist live variable, there will be declaration instructions at the beginning of the PIG program generated with total variables ranging from 5 to 15.

Expression Generation Expression is generated in recursive manner using function `generateExpression()`. To avoid any line with $>$ 1000 characters, the expression generator will have a pruning which will return the pruned expression if the length of the expression reach the threshold of 950 characters.

In addition, there will be a linearly decreasing probability weights for both binary operation and not operation as the recursion reach deeper depth to control the balance between depth and operation combinations. The binary operation weight will decrease based on the recursion depth while not operation will decrease based on the number of previous leading not, to avoid deep reoccuring not operation (e.g. !(!(!(...(!(a))))))

Branch Generation Branch instruction will be generated with two scenario. For the first layer, a safeguard against infinite loop is created to decrease the number of cases with looping that will lower the quality of the test case.

The infinite loop safeguard is done by injecting a counter variable that increases at every loop branch iteration. The branch then will evaluate the condition based on the counter and some small constant. The evaluation check is done by checking whether subtraction evaluation leads to overflow. If overflow is detected then the evaluator will change it to 0 (termination condition) which leads to the branching termination

In the function `generateBranchLine()`, the jump line is generated using a bimodal distribution centered at $l + \Delta$ where l be the branch instruction line and Δ set to be 10, and standard deviation of 5. It also has a lower bound of number of fixed variables (for type 1 generation) and upper bound of number of maximum lines, 1000. The randomization will also reject the jump line if it is very near to l (set to be having minimum distance of 4 from the branch line l)

3 Dataflow Analysis

To analyze possible undefined variable a **forward must** analysis can be conducted. Forward analysis as we analyze the dependence to the prior lines, and must analysis as we analyze a branch to be taken or not taken, meaning a variable must be declared at both branch flow.

CFG construction The analysis will first construct the CFG of the PIG program. Constructing the CFG is done by

- Finding all basic block or BB leaders (entry statement, line after branch, branch target)
- Let all leaders be the starting statement of each BB and ends before next leaders.
- Construct the edge from the branching relations

The CFG is abstracted as a list of nodes representing the basic block and edge representing the branch flow. Each basic block is abstracted as a class which has starting line, ending line, incoming edge and outgoing edge representing the branching flow. In addition, entry and exit BB is also added to the CFG.

Gen/Kill Analysis The next step is to generate a bit vector representing generated variable and killed variable of each basic block (implemented in `killDeclaration()` and `genDeclaration()`, where each function will iterates each basic blocks in the CFG, and generate a bit vector representing whether a variable is declared (or killed) in each basic block. For declared variable, if it is killed after it is declared in the same basic block, then it is considered as not declared.

Dataflow Analysis (Forward Must) The next step is the forward must analysis. The algorithm for the analysis is as follows

- Every basic block is given an OUT bit vector initialized as 1, except the entry block
- While the iteration haven't converge, iterate each basic block and calculate the updated OUT using

$$OUT[B] = gen_B \cup (\cap_{I \in in(B)} IN[I] - kill_B)$$

Undeclared Variable Analysis After the OUT bit vector of every basic block is retreived from the analysis, the last step is to analyze the lines inside each basic block. First is to retreive the IN of each basic block by

$$IN[B] = \cap_{I \in in(B)} OUT[I]$$

After the IN bit vector is calculated, for each statement in the basic block, we will update the bit vector when we encounter (1) D, which will change 0 to 1 in the bit vector (2) R, which will change 1 to 0 in the bit vector.

In addition, for all statements, we will check the used variable. If the bit vector state of the corresponding variable is 0, meaning the variable may be undeclared, then we will point the line as a possible case of undeclared variable.

Result The result of the example test case are as follows

- Test 1 : 1 line
- Test 2 : 1 line
- Test 3 : 2 lines
- Test 4 : 2 lines
- Test 5 : 4 lines
- Test 6 : 13 lines
- Test 7 : 66 lines

4 Metamorphic Testing

Metamorphic testing is done by creating two inputs with some metamorphic relationship and check whether the pattern still holds in the output. For PIG interpreter, one metamorphic that we can exploit is equivalent coding, i.e. two PIG program that has the same output behavior.

Equivalent Expression One way to exploit equivalent coding for the metamorphic testing is to generate two expression that leads to the same result. Some possible equivalent expression are

1. **Addition**: usage of commutative and associative rule, i.e.
$$((a)+(b)) = ((b)+(a)) \qquad \text{and} \qquad (((a)+(b))+(c)) = ((a)+((b)+(c)))$$
2. **And, Or**: usage of De Morgan's rule:
$$a \vee b = \neg((\neg a) \wedge (\neg b)) \qquad \text{and} \qquad a \wedge b = \neg((\neg a) \vee (\neg b))$$
3. **And, Or**: usage of commutative rule
$$a(\vee,\wedge)b = b(\vee,\wedge)a$$
4. **Not**: even number of not cancels each other:
$$a = \neg(\neg a)$$

Equivalent Branching Another equivalent code that can be generated is by removing branch instruction where the branch is always branching or always not branching

- Branching to previous line: if the branch is always not branch, then the branch in the equivalent program can be replaced by other instruction that does not change the output behavior such as variable declaration
- Branching to next line: If the branch is always not branch, the branch can be replaced. If the branch is always branch, the branch can be replaced and the intermediate lines before the jump line can also be removed

The branch need to be replaced instead of being removed because it affect the number of executed lines which affect the output behavior for programs that executes more than 5000 lines.

Another way to generate equivalent branch jump line is to make a loop branch to be sequential execution. Using a deterministic counter similar to a for loop, the generator will generates one version with branching mechanism and another with sequential with number of time execution be the same with the branch mechanism.

Equivalent Variable The variable name used can also be replaced using one-to-one mapping from the original input to the modified input that has metamorphic equivalence.

The rest of the metamorphic generator implementation is the same as the differential generator. The difference is while the generator generates the original PIG program, it generate another equivalent code using equivalent variable, expression and branching method.

Checker The metamorphic checker is implemented to check whether the two outputs are the same. If the output differs by any character or even by the length of the output, the checker will output `1`. If the two outputs are completely the same, the checker will output `0`