Implementing the Query Count Estimation from ‘Efficient State Merging in Symbolic Execution’

Bachelor Project - Spring 2023

Adrien Alain Bouquet

Definition of the problem

The central point of this report is the implementation of the heuristic query count estimation (QCE)[1] It is essential to explain the purpose of this heuristic in order to understand the implementation described below: In the context of symbolic execution, one of the objectives is to reduce the computational cost that the introduction of symbolic variables can generate so that we increase its efficiency. To do this, the paper proposes this QCE heuristic whose aim is to determine whether a variable v at a location l in the program is a sensitive variable (or a “hot variable” as denoted in the paper). As described in section 3.2, a variable v is sensible (or hot) at location l if the number of additional queries $Q_{add}(l, v)$ is greater than a fixed fraction α of the total number of queries $Q_t(l)$ (see §3.2). Consequently, the paper presents a formula to count recursively the number queries that are selected by a function c:

$$q(l', c) = \begin{cases} \beta q(succ(l'), c) + \beta q(l'', c) + c(l', v) & \text{instr(l') = if (e) go to l'' (1.a)} \\ 0 & \text{instr(l') = halt (1.b)} \\ q(succ(l', c)) & \text{otherwise (1.c)} \end{cases} \tag{1}$$

Where e is an expression, c is a function given in parameter that evaluates the expression e and β represents the probability that a branch is feasible. The paper fixes it to 0.6 as example. Therefore, it employs this formula to define the two numbers of queries defined above by:

$$Q_{add}(l, v) = q(l, \lambda(l', e).ite((l, v) \triangleleft (l', e), 1, 0)) \qquad (2)$$

$$Q_t(l, v) = q(l, \lambda(l', e).1) \qquad (3)$$

[1]Described in the first part of the paper

Presented in this way, the formulas do not allow us to understand clearly how to implement the LLVM pass and it is necessary to explain them. First of all, a program location l can be understood as an instruction, l' can be seen as the next instruction (the next line in terms of code) and $succ(l)$ as the successor of l. Then, (1) separates 3 cases which return a value. The first one $(1.a)$ is a branch condition with an expression (an if-else for instance), the second one $(1.b)$ is a program termination and the last one $(1.c)$ is all the other type of instructions. Hence, $(1.a)$ evaluates recursively the branch if e is true and the branch if e is false. It also evaluates the condition using the function c. The two last cases are pretty straight forward: repectively return 0 if the program ends and return the value of the next instruction. In summary, this formula can be represented as if we were walking through a tree. When reaching a condition, evaluate the condition using the given function and recursively evaluates the 2 branch. When reaching an other instruction, skip it, When reaching an end, it's a leaf.

As a result, Q_{add} becomes the evaluation of the formula when the function c check if the expression e depends on the variable v and hence will generate a solver query and Q_t becomes the evaluation of the formula when c returns always 1 as it simply counts the number of branch condition.

However, this explanation is not sufficient in itself, as certain instructions are problematic as explained in the paper. Firstly, loops and recursion prevent the formula from stopping because of the recursive nature of its definition. In the paper, a loop bound κ is set to limit the maximum loop iterations and recursion calls. In addition, functions calls are problematic because they must be evaluated too and their number of queries must be added to the total one. Therefore, the next part below will explain the implementation of the LLVM pass with respect to the formulas and constraints stated before.

Implementation

Before describing the code in concrete terms, it is important to define the different variables used in the theory in terms of code, and more specifically in terms of LLVM assembly or C++. The equivalent of an instruction l, as explained above, is the `Instruction` class, a variable v as the paper intended will be have for equivalent the `Value` class, the function q has for implementation the `compute_query` function described below and the function c has for implementation either the `returnTrueFunction` or the `matchValue` also decribed below.

Parameters and Data structure created

The value for β which represents the probability of the feasibility of a branch is fixed to 0.6 and the maximum number of iterations κ is separated into two variables `nloops`and `dcalls` both fixed to `1`. Furthermore, the α parameter which is used to compute if a variable is "hot" is fixed to 0.5. Theses values were chosen from the example at the end of page 5 to facilitate to facilitate comparison with the pass defined in the paper and the tests.

Besides, two data structures were created: `BranchCount` and `CallCount`. The former associates a branch instruction (i.e. `BranchInst` in LLVM assembly) to the number of time the instruction has been encountered, the latter is analog to the former one but it counts the number of times a function (i.e. `Function`) is encountered. Basically, they are used to keep track of the loop iterations and recursions.

Compute Query function

As stated before, the `compute_query` function is the equivalent of $q(l, c)$ which computes the number of query. It takes as argument a pointer to a llvm instruction `Inst` (i.e l), a pointer to a llvm value `researched` (i.e. v), the list of all the `BranchCount` of the backwards branch, in other words all the branch that jump to a label prior to the current one, the list of all the `CallCount` of all the `CallInst` that depends on the value `researched` and finally a function (i.e c) that evaluates if the condition of the branch instruction depends on the value `researched`. The function applies what has been previously defined and handles the constraint such as loops iterations or recursion by checking if the maximum count has been reached.

Match Value Function

The implementation of the function c that check if an expression depends on a particular variable is `matchValue` which takes as argument two llvm value: the first one correspond to the value we are testing and the second one is the value that we are searching (i.e. the variable v in the paper). Although its goal is pretty straightforward, its implementation is more complex. In fact, we have to backtrack in the code by going from the operands of an instruction, checking if they are the value we search and if not checking their operands until the variable searched is found or there is nore more instruction and hence the initial condition does not depends on the variable. However, this logic might causes `matchValue` to loop if they are loops or recursion. For instance:

```c
void loopFunc(int x){
    int index = x;

    while (index < 10){
        //...//
        index++;
    }
}
```

When reaching the condition of the while loop, the index has been redefined and thus there are two paths leading to it. In llvm, there exists `PHINode` which are instructions that are regroup the differents pahts of the variable, in other words the differents values and the label where they are defined. Therefore, we assume that we only check the paths for which the label is before the current one.

The pass

The pass computes twice the query count of a function for each its argument: once using the `matchValue` function described above and once using the `returnTrueFunction` to count the total number of queries. It then outputs the value using both function and if the variable is a hot variable.

Results

```
- case from the paper
- Does it work on Coreutils
    - Why coreutils ?
```

Basic cases

In order to evaluate the correctness on basic code, a set of cases[2] as been established to tests implementation and constraint handling, and we are going to demonstrate 2 of them.

Basic condition

```
void basicCondition(int a){
  int x = 0;
  int b = 19;
  if (a >= 5) {
    x = 5;
  } else {
    x = 10;
  }
  int y = x + a;
}
```

[2]https://github.com/abouquet27/BachelorProject/tree/main/test_project/cases

This case is pretty straightforward when applying the pass as the variable tested a is in the condition of the if and hence `matchValue` will return 1. Furthermore, the evaluation of the two branch will return 0 each as there are not any conditions left. Thus, the query count will be equals to 1.0 which is the value we theoretically expect. Here is the full expectation computation (line 1 correspond to the function header):

$$
\begin{aligned}
Q_{add}(1, a) = q(1, c) = q(4, c) \\
= \beta q(7, c) + \beta q(5, c) + c(a >= 5, a) \\
= \beta q(10, c) + \beta q(10, c) + 1 = 0 + 0 + 1 = 1
\end{aligned}
$$

Recursion

```
void recursionFunction1(int x){
    if (x > 0){
        printf("recursion needed");
        recursionFunction1(x-1);
    }
}
```

The interesting aspect of this case is recursion. When evaluated the first time, `matchValue` will return 1 on the first condition, the `printf` call will be skipped as it does the x variable and it will evaluate a second time the condition:

$$
\begin{aligned}
Q_{add}(1, x) = q(1, c) = q(2, c) = \beta q(5, c) + \beta q(3, c) + c(x > 0, x) \\
= \beta q(6, c) + \beta q(1, c) + 1 = \beta(\beta q(5, c) + \beta q(3, c) + c(x > 0, x)) + 1 \\
= \beta(0 + 0 + 1) + 1 = 0.6 + 1 = 1.6
\end{aligned}
$$

More generally, the different base cases have all been computed manually before being submitted to the pass and comparing the results. About twenty base cases have been established, computed and tested to make sure the pass correctly work on these cases.

Example from the paper

After creating cases to test the different issues the pass has to handle, the next step was testing it on the example from the paper[3]. The paper uses this simplified version of the `echo` program. However, the example has been truncated to make it work using the pass, mainly by adding the variable ‘arg’ in the arguments of the function. Consequently, the pass computes 1.6 which is the same value as the one computed by the paper and hence slightly strengthens the correctness of the pass

[3]Figure 1 page 5

Applying the pass Coreutils

An

Limitations

```
- Very theoritical pass
- Can be improved
    - Existing pass that can improve the code
    - More modularity
- Tested on some cases but is not garanteed
    - Still crash
    - Generic function
    - Function that are from an external library
    - Not properly proved
```

Personnal discoveries

This project has been a great learning experience on a large range of aspects. To start, it was my first big personal project in computer science and at EPFL. It required a lot of autonomy and a capacity of managing the hours for a result that won't be necessarily useful but that is closer to reality than exercices seen during courses.

Speaking of courses, this project and working with llvm was a good application of concepts seen during courses as Compilers or a good introduction to new concept such as Symbolic Executions. Another interesting point was the notion of research. Working on a project based on a paper from the laboratory in question is very motivating and stimulating because it requires you to do research and understand more than just what you need to implement.

Finally the main challenge was working with new languages with their particularities, more specific libraries with tougher documentation requiring more effort to understand how to use them or tools that requires a lot attempts to install and make them work. It took me a while to understand how does work LLVM and to use in my code to achieve the pass. The project required a lot of adaptation, but it developed cross-disciplinary skills that I'm sure will come in very useful later on.