DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT KANPUR

CS639: Program Analysis Verification and Testing Report: Assignment 2

Program Synthesis using Symbolic Execution

Name: Prashant Mishra
Roll No: 231110036

Abstract

Symbolic execution of a program is used for exploring various paths taken by the program on different input sets. The purpose of the symbolic execution includes test case generation, verification of the program, finding loop invariants, security analysis, and finding bugs in the program/software.

Objective

The task in assignment 2 is to implement a **checkeq** function that returns the value of constant parameters that ensures logical equivalence of the 2 given programs.

Detailed understanding of the assignment

In this assignment, there are 2 programs one of them is a complete program and the other one is also syntactically correct but has some holes (Here holes are variables that are meant to be constants) the task is to find the value of these holes/variables such that both the programs are semantically equal (programs having a same output on same input for each possible input).

Assumptions

To implement the **checkeq** function for finding values of holes in the program, the assumptions are -

- Both programs should have the same number of inputs.
- The programs must be syntactically correct and executable.

- The program is written in a language supported by **Kachua** (i.e. Kachua syntax[1] is followed and the extension of the file is **.tl**)

Implementation

The steps followed in the implementation of **checkeq** function to find the value of holes are-

A. Writing programs:

At first, I wrote two programs such that Program 1 has some holes and Program 2 is complete and both the programs take taking same number of inputs.

B. testData.json file generation:

In the next step, I generate testData.json files by executing the symbolic execution engine for both programs. This file includes various features of the program such as potential test cases that cover almost all paths in the program, constrained of the program paths, parameters of the program, and symbolic values of all variables in the program.

The command always generates the testData.json file so I have manually renamed the testData file as testdata1.json and testData2.json for the programs 1 and 2 respectively.

Command: `python chiron.py -t 100 -se example/eqtest1.tl -d {':x':5,':y':100}`

C. Optimized.kw file generation:

Moreover other than testData.json, I generated optimized.kw file for the program having holes this file contains the binary representation of the program that can be executed by the Kachua framework. However, in my approach I have not used Optimized.kw data in **checkeq** function.

Command: `python chiron.py --dump_ir example/eqtest2.tl`

D. checkeq function implementation:

The tasks performed in **checkeq** function are as follows-

- Converted testData.json into a more readable format that removes unnecessary strings, and commas to make testData more process-friendly.
- Defined solvers to check the satisfiability of expressions.
- Now for each test case collected all input parameters and holes used in programs and defined symbolic variables for each.
- Further added all parameters assigned to their values in the solver and also added the corresponding constraints from another testData file.
- Now if these added constraints satisfy the current state of inputs then fetch symbolic encoding for this input from testData1.json and testData2.json and add a constraint of equality for both encodings in the solver.
- Repeat these steps for each test case.

- Check if the model is satisfiable for some constant parameters if the model is satisfiable then return the values of constant parameters for which the model is satisfied.

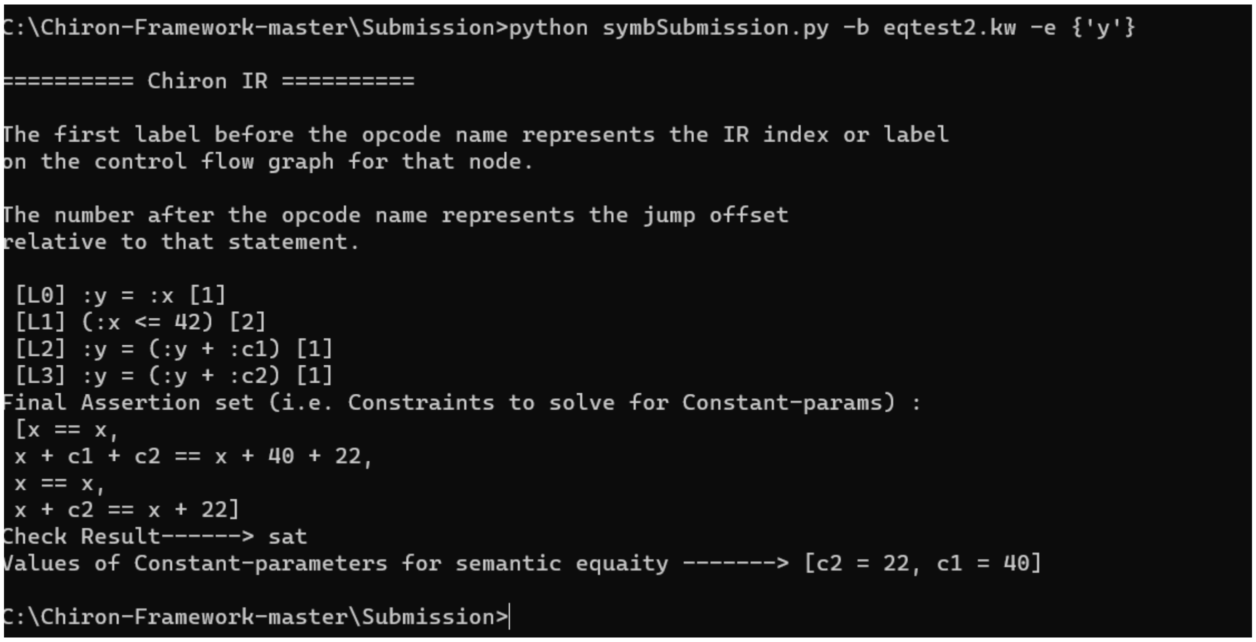
Run the **checkeq** function using this command to get the values of constant parameters.

command: python symbSubmission.py -b eqtest2.kw -e {'x','y'}

Results

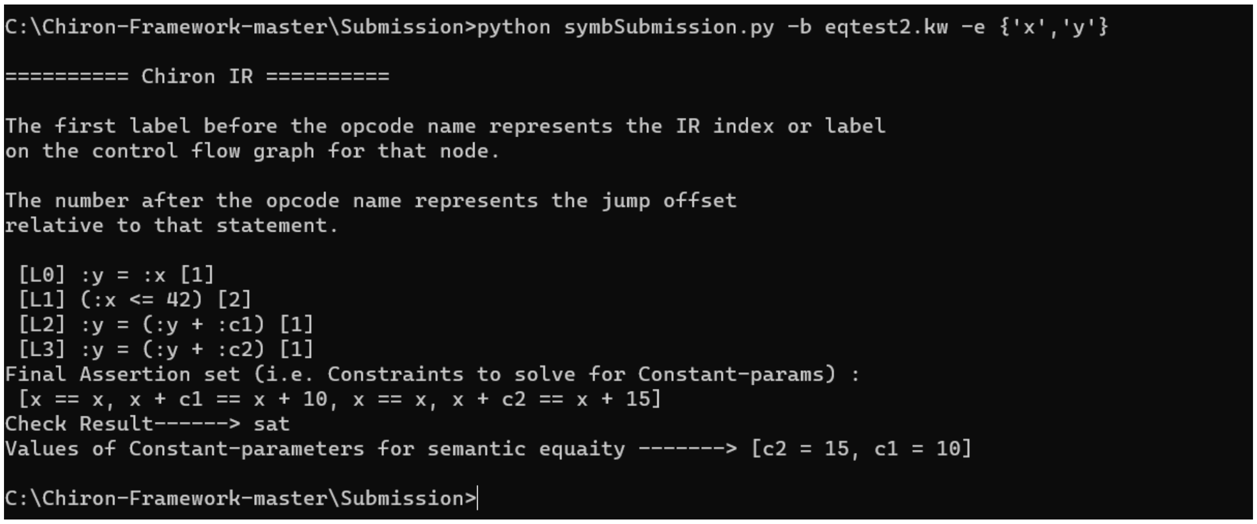
The results for each of 5 test cases are as follows-

1.



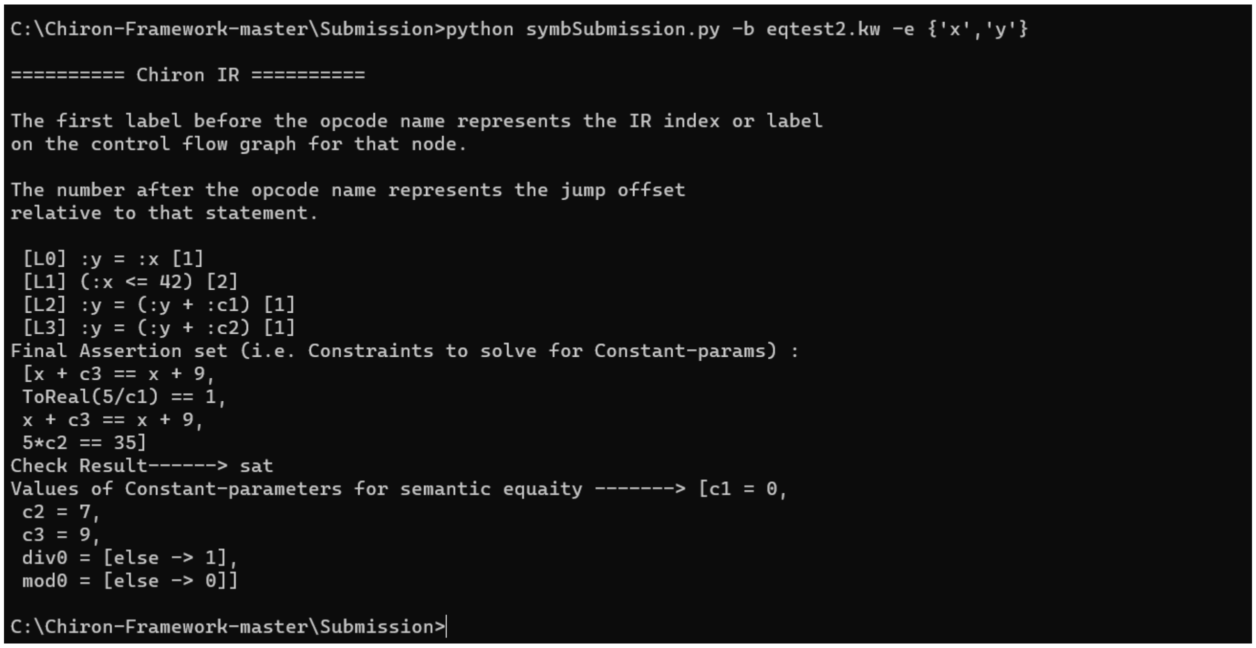
```
C:\Chiron-Framework-master\Submission>python symbSubmission.py -b eqtest2.kw -e {'y'}

========== Chiron IR ==========

The first label before the opcode name represents the IR index or label
on the control flow graph for that node.

The number after the opcode name represents the jump offset
relative to that statement.

 [L0] :y = :x [1]
 [L1] (:x <= 42) [2]
 [L2] :y = (:y + :c1) [1]
 [L3] :y = (:y + :c2) [1]
Final Assertion set (i.e. Constraints to solve for Constant-params) :
 [x == x,
 x + c1 + c2 == x + 40 + 22,
 x == x,
 x + c2 == x + 22]
Check Result------> sat
Values of Constant-parameters for semantic equaity -------> [c2 = 22, c1 = 40]

C:\Chiron-Framework-master\Submission>
```

2.



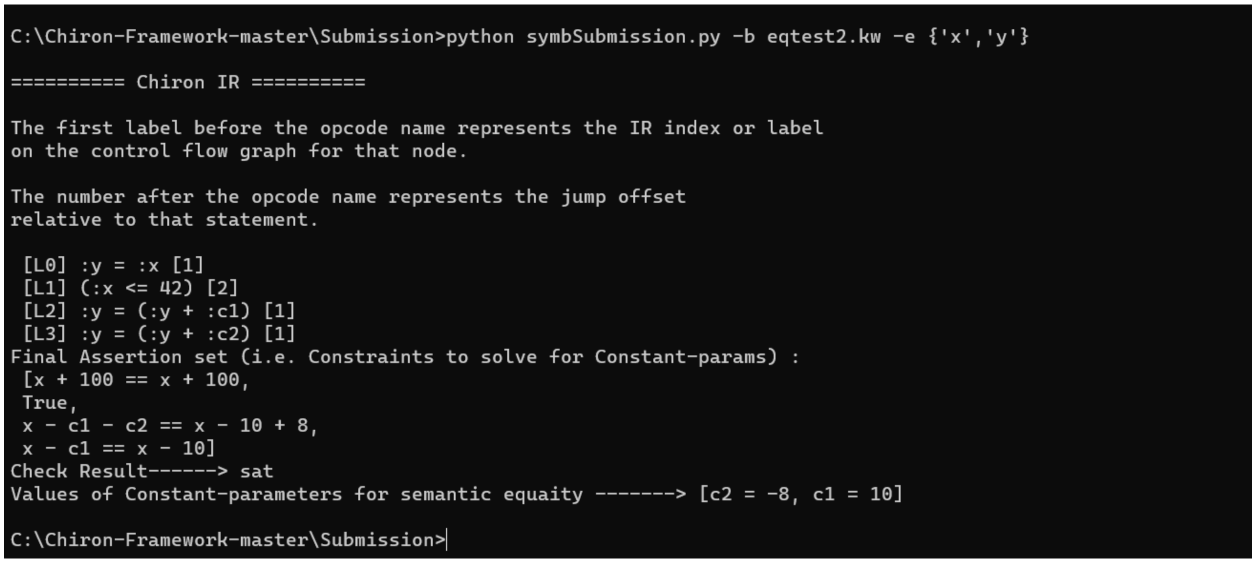
```
C:\Chiron-Framework-master\Submission>python symbSubmission.py -b eqtest2.kw -e {'x','y'}

========== Chiron IR ==========

The first label before the opcode name represents the IR index or label
on the control flow graph for that node.

The number after the opcode name represents the jump offset
relative to that statement.

 [L0] :y = :x [1]
 [L1] (:x <= 42) [2]
 [L2] :y = (:y + :c1) [1]
 [L3] :y = (:y + :c2) [1]
Final Assertion set (i.e. Constraints to solve for Constant-params) :
 [x == x, x + c1 == x + 10, x == x, x + c2 == x + 15]
Check Result------> sat
Values of Constant-parameters for semantic equaity -------> [c2 = 15, c1 = 10]

C:\Chiron-Framework-master\Submission>
```

3.



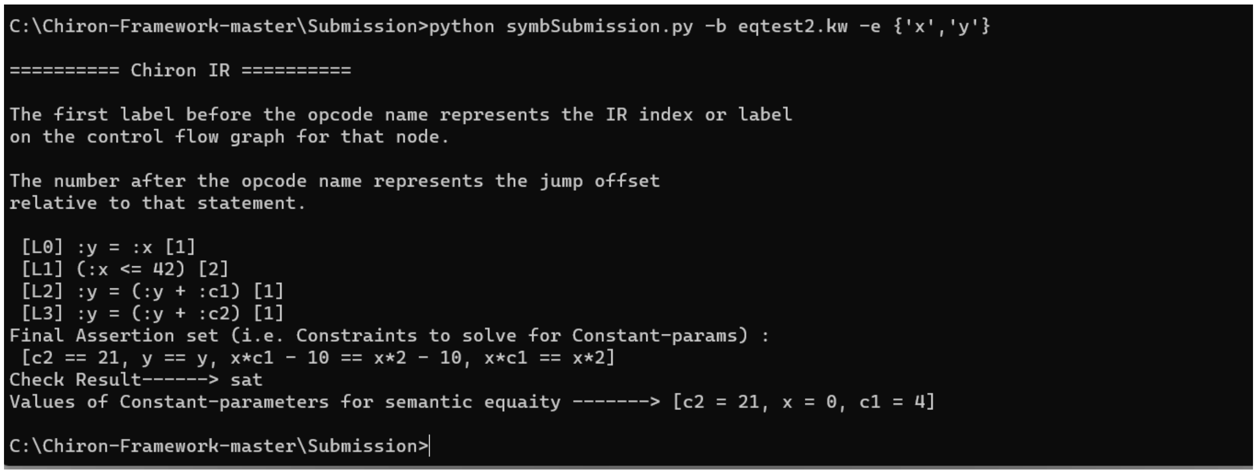
```
C:\Chiron-Framework-master\Submission>python symbSubmission.py -b eqtest2.kw -e {'x','y'}

========== Chiron IR ==========

The first label before the opcode name represents the IR index or label
on the control flow graph for that node.

The number after the opcode name represents the jump offset
relative to that statement.

 [L0] :y = :x [1]
 [L1] (:x <= 42) [2]
 [L2] :y = (:y + :c1) [1]
 [L3] :y = (:y + :c2) [1]
Final Assertion set (i.e. Constraints to solve for Constant-params) :
 [x + c3 == x + 9,
 ToReal(5/c1) == 1,
 x + c3 == x + 9,
 5*c2 == 35]
Check Result------> sat
Values of Constant-parameters for semantic equaity -------> [c1 = 0,
 c2 = 7,
 c3 = 9,
 div0 = [else -> 1],
 mod0 = [else -> 0]]

C:\Chiron-Framework-master\Submission>
```

4.



```
C:\Chiron-Framework-master\Submission>python symbSubmission.py -b eqtest2.kw -e {'x','y'}

========== Chiron IR ==========

The first label before the opcode name represents the IR index or label
on the control flow graph for that node.

The number after the opcode name represents the jump offset
relative to that statement.

 [L0] :y = :x [1]
 [L1] (:x <= 42) [2]
 [L2] :y = (:y + :c1) [1]
 [L3] :y = (:y + :c2) [1]
Final Assertion set (i.e. Constraints to solve for Constant-params) :
 [x + 100 == x + 100,
 True,
 x - c1 - c2 == x - 10 + 8,
 x - c1 == x - 10]
Check Result------> sat
Values of Constant-parameters for semantic equaity -------> [c2 = -8, c1 = 10]

C:\Chiron-Framework-master\Submission>
```

5.



```
C:\Chiron-Framework-master\Submission>python symbSubmission.py -b eqtest2.kw -e {'x','y'}

========== Chiron IR ==========

The first label before the opcode name represents the IR index or label
on the control flow graph for that node.

The number after the opcode name represents the jump offset
relative to that statement.

 [L0] :y = :x [1]
 [L1] (:x <= 42) [2]
 [L2] :y = (:y + :c1) [1]
 [L3] :y = (:y + :c2) [1]
Final Assertion set (i.e. Constraints to solve for Constant-params) :
 [c2 == 21, y == y, x*c1 - 10 == x*2 - 10, x*c1 == x*2]
Check Result------> sat
Values of Constant-parameters for semantic equaity -------> [c2 = 21, x = 0, c1 = 4]

C:\Chiron-Framework-master\Submission>
```