Symbolic Execution(Working title)

Aarhus Universitet

Søren Baadsgaard

February 6, 2019

Abstract

Contents

Chapter 1

Introduction

Chapter 2

Summary of theory

In this chapter we will cover some of theory behind symbolic execution. We will start by describing what it means to *symbolically execute* a program and how we deal with potential branching. We will also describe the connection between a symbolic execution of a program, and a concrete one. Furthermore we will relate this technique to alternatives such as formally proving correctness.

2.1 Principles of symbolic execution

in [1], the author looks at a spectrum of ensuring that a program functions as expected. In one extreme we have program testing, where the programmer specifies a sample of input. The program is then run on this sample, and if the program runs as intended, the programmer can be sure that for this sample the program is correct. choosing a proper sample of inputs will then enable the programmer to have some level of confidence that the program runs correctly. But since most programs can take virtually an infinite number of different inputs, one can never be absolutely sure that the program is free of bugs. At the other end of the spectrum is formal program verification. This requires a proper mathematical specification of the program, and a proof procedure that will verify that this specification is correct w.r.t some formal requirements. This will give the programmer a complete confidence that the program is in fact correct. This comes at the cost of producing a proper and adequately strong specification, performing sound steps in the proof procedure as well as having a strong set of requirements. We find symbolic execution somewhere in the middle of this spectrum. While we do not build a formal specification and apply any sort of program verification, we do not restrict our self to specific samples of input. Instead, we try to test the program for whole classes of inputs (such as all integers that are a power of two), which is represented by symbolic values instead of actual ones.

2.1.1 symbolically executing a program

Symbolic execution is described in [**?**] by imagining a simple programming language which restricts all variables to signed integers. The language supports the usual arithmetic operations on these integers, as well as a conditional expression to decide whether a given value is ≤ 0. To execute such a program symbolically, we allow the variables to also take *symbolic* values, where these symbols represent some signed integer. We also allow arithmetic operations on these symbols, and for variables to hold such expressions. This means variables in fact contains polynomials over the integers. To illustrate this, we consider the following simple program, that takes parameters a, b, c and computes the sum:

```
Fun sum(a, b, c) {
var x = a + b
var y = b + c
var z = x + y - b
return z
}
```

If we run this program on concrete inputs, say $a = 2, b = 3, c = 4$, we would get the following execution:

1. x = 2 + 3 $\Rightarrow$ y = 5
2. y = 3 + 4 $\Rightarrow$ y = 7
3. z = 5 + 7 - 3 $\Rightarrow$ z = 9
4. return 9

So we see that on the specific input $a = 2, b = 3, c = 4$, the program returns the correct value.
Let us now run the program with symbols. Say that we the input the following symbolic values: $a = \alpha, b = \beta, c = \gamma$. The execution would then look like:

1. x = $\alpha + \beta$
2. y = $\beta + \gamma$
3. z = $(\alpha + \beta) + (\beta + \gamma) - \beta$
4. return $\alpha + \beta + \gamma$

Chapter 3

Basic symbolic execution for the *SImPL* language

3.1 description

In this chapter we will describe the process of implementing symbolic execution for a simple imperative language called ***SImPL*** .

3.2 Introducing the *SImPL* language

SImPL **(Simple Imperative Programming Language)** is a small imperative programming language, designed to highlight the interesting use cases of symbolic execution. The language supports only one type, namely the set integers $\mathbb{N}$. Furthermore we will interpret 0 as *false* and any other integer as *true*. ***SImPL*** supports basic variables that can be assigned the value of any expression, as well as basic branching functionality through an **If - Then - Else** statement. Furthermore it allows for looping through a **While - Do** statement.

We will describe the language formally, by the following Context Free Grammar:

$\langle int \rangle$ ::= 0 | 1 | -1 | 2 | -2 | ...

$\langle var \rangle$ ::= a | b | c | ...

$\langle exp \rangle$ ::= $\langle int \rangle$
| $\langle var \rangle$
| $\langle exp \rangle + \langle exp \rangle \mid \langle exp \rangle - \langle exp \rangle \mid \langle exp \rangle * \langle exp \rangle \mid \langle exp \rangle \ / \ \langle exp \rangle$
| $\langle exp \rangle > \langle exp \rangle \mid \langle exp \rangle == \langle exp \rangle$

$\langle stm \rangle$ ::= $\langle exp \rangle$
| $\langle var \rangle = \langle exp \rangle$
| $\langle stm \rangle \ \langle stm \rangle$
| if $\langle exp \rangle$ then $\langle stm \rangle$ else $\langle stm \rangle$
| while $\langle exp \rangle$ do $\langle stm \rangle$
| Print E

$\langle prog \rangle$ ::= $\langle stm \rangle$

where $+, *, -, /$ denotes the usual arithmatic operators on integers, and $>$, $==$ denotes the comparison-operators of *greater-than* and *equal-to* respectively. When interpreting a comparison-operator we will return 0 for *false* and 1 for *true*. Finally we see that a program $\langle prog \rangle$ is simply a sequence of one or more statements.

3.2.1 Interpreting *SImPL*

In order to work with ***SImPL*** , we have build a simple interpreter using the *Scala* programming language. We use a simple recursive strategy, where we recursively interpret statements from left to right, each time returning a potentially updated *environment* containing bindings of variable names to integer values.

Chapter 4

Further extensions

Chapter 5

Conclusion

Appendix A

Source code

Appendix B

Figures

Bibliography

[1] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`.