Symbolic Execution(Working title)

Aarhus Universitet

Søren Baadsgaard

February 10, 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 [1] 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 $\alpha + \beta + \gamma$

From this execution we can actually conclude that the program will return the correct result for any three integers that we give as input.

2.1.2 Handling branching in a program

In the previous section we gave an example of a symbolic execution of a simple program that computes the sum of three integers. The program is an extremely simple case, in which the program will behave exactly the same for any possible input values. In reality however, most program languages have some way of allowing branching to happen, and with this feature, we cannot guarantee that the program will behave exactly the same for all inputs. If our program contains an expression like *If* $x > 2$ *Then ... Else ...* , the program will behave differently depending on whether or nor $x > 2$. To encapsulate this, we introduce a *path-constraint (PC)*, which is a boolean expression that will contain all properties that the input must satisfy to follow the given path. At the beginning of the execution, the PC will be initialized with the value $true$ as no assumptions have been made yet (If we introduce pre-conditions on the input, the PC will of course contain these). At any *if-expression* with condition q, during the execution we will look at the following to expressions

1. $PC \supset q$
2. $PC \supset \neg q$

where expression 1 tells us that q is contained in the PC, and expression 2 tells us that the opposite of q is contained in the PC. It is clear that at most one of these expressions can be true at any given time, which leaves us with three possible outcomes. If expression 1 holds, we simply follow the then-branch, and if expression 2 holds, we follow the else branch. In both these outcomes, PC is not updated since it already contains the property that leads to the given branch that was chosen. The third outcome happens when neither expression holds. At this point we cannot simply follow one of the branches, so we must fork our execution into two parallel executions, one where we assume that q holds and one where we assume $\neg q$ holds. This will produce two new PCs, namely

$$PC' \leftarrow PC \wedge q$$
$$PC'' \leftarrow PC \wedge \neg q$$

and from here we will have two seperate executions, one with PC' that will follow the then-branch, and one with PC'' that will follow the else-branch. It is important to note that the PC can never become identically $false$. To see this we first note that PC will always start with the value $true$, and all other updates will be of the form $PC \leftarrow PC \wedge r$ where $r \in \{q, \neg q\}$. But these updates only happen whenever we cannot infer q or $\neg q$ from the existing value of PC and it will only receive exactly of q or $\neg q$. So it is never possible to have a state of PC in which we have $\ldots \wedge q \wedge \ldots \wedge \neg q \ldots$.

As in [1], we can illustrate this with the following program that computes a^b.

```
Fun pow(a, b) {
var r = 1
var i = 0
while (i < b) {
r = r*a
i = i + 1
}
return r
```

If we assign $a = \alpha$ and $b = \beta$, we get the following execution:

- PC is initialized to $true$
- $r \leftarrow 1$
- $i \leftarrow 0$
- We hit a branching point, so we check if $true \supset (0 < \beta)$ or $true \supset \neg(0 < \beta)$. Since neither of these hold, we must fork:
- **Case** $\neg(0 < \beta)$: $PC \leftarrow true \wedge \neg(0 < \beta)$. The program returns 1. So we can conclude that the program returns 1 when $\beta \leq 0$.
- **Case** $(0 < \beta)$: $PC \leftarrow true \wedge (0 < \beta)$.

  $r \leftarrow 1 \cdot a$

  $i \leftarrow 0 + 1$
- We hit a branching point again, so we check if $true \wedge (0 < \beta) \supset 1 < \beta$ or $true \wedge (0 < \beta) \supset \neg(1 < \beta)$. Since neither of these holds, we fork again:
- **Case** $\neg(1 < \beta)$: $PC \leftarrow true \wedge (0 < \beta) \wedge \neg(1 < \beta)$. The program returns α. So we can conclude that the program returns α when $\beta = 1$.
- **Case** $1 < \beta$: $PC \leftarrow true \wedge (0 < \beta) \wedge (1 < \beta)$.

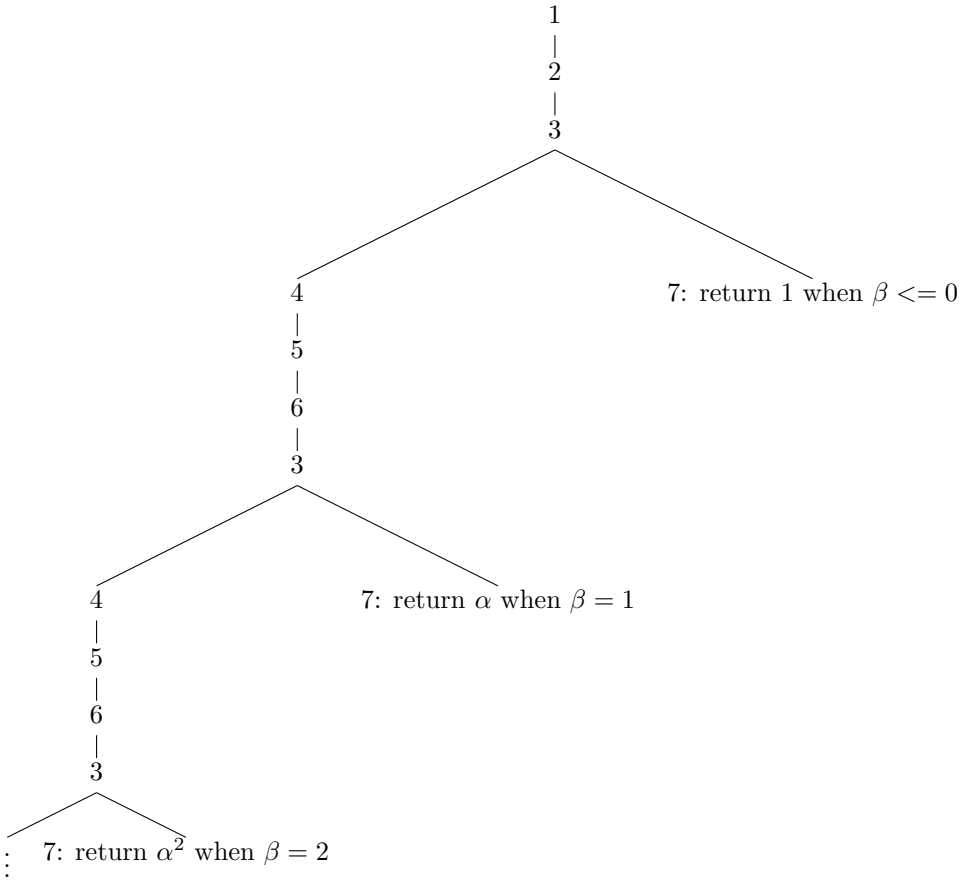
  $r \leftarrow a * a$

  $i \leftarrow 1 + 1$
- We hit a branching point ...

2.1.3 Limitations due to infinite execution trees

As demonstrated in the last example in the previous sections, a symbolic execution can easily become infinite as soon as we introduce branching and some looping structure. This is further illustrated if we consider an execution tree for a program. In [1], they are describes by enumerating each statement, and let each node in the tree, correspond to an execution of one of the command. The edges going out from a node corresponds to the transition from one statement to the next. From this we can see that non forking statements will only have a

single edge outgoing, while forking statements will have two. The forking edges will then correspond to splitting up the execution and following a different path, with a different *path-constraint*. As an example, we can look at the execution tree for the program *pow*, that computes a^b.



As we can see, this tree is infinite, since $b = \beta$ and β can be arbitrarily large. In this case, our symbolic execution would run forever if we insisted on exhaustively exploring all possible paths. This illustrates one of the limitations of symbolic execution. For programs with infinite execution trees, we simply cannot exhaust all possible inputs, so we have to restrict our testing to exploring only a finite number of paths.

For each branch that we do explore however, we will know the behavior of any input that satisfies the *path-constraint* which is still a large number of test runs, resolved with a single execution. An important point here, is that since the *path-constraint* can never become identically false, there must exist input that would follow exactly that path. For this reason there exists a commutative property between the symbolic values, and concrete values that satisfy the particular *path-constraint*. We get the same result if we simply run the program on these concrete values, or if we run it symbolically, and then substitute the symbolic

values with concrete values. For this reason, we should be able to always ask for a set of concrete values that will follow the same path as our symbolic execution. This, way we can generate test cases that represents a potentially infinite class of input values.

(Maybe write something about induction over trees, and finite trees allowing for exhaustive search)

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 | ...

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

$\langle exp \rangle$ ::= $\langle aexp \rangle$ | $\langle bexp \rangle$ | $\langle nil \rangle$

$\langle nil \rangle$ ::= ()

$\langle bexp \rangle$ ::= True | False
| $\langle aexp \rangle > \langle aexp \rangle$
| $\langle aexp \rangle == \langle aexp \rangle$

$\langle aexp \rangle$ ::= $\langle int \rangle$ | $\langle id \rangle$
| $\langle aexp \rangle + \langle aexp \rangle$ | $\langle aexp \rangle$ - $\langle aexp \rangle$
| $\langle aexp \rangle \cdot \langle aexp \rangle$ | $\langle aexp \rangle$ / $\langle aexp \rangle$
| $\langle cexp \rangle$

$\langle cexp \rangle$ ::= $\langle Id \rangle$ ($\langle aexp \rangle$*) # Call expression

$\langle stm \rangle$ ::= $\langle exp \rangle$
| $\langle Id \rangle = \langle exp \rangle$
| $\langle stm \rangle$ $\langle stm \rangle$
| if $\langle exp \rangle$ then $\langle stm \rangle$ else $\langle stm \rangle$
| while $\langle exp \rangle$ do $\langle stm \rangle$

$\langle fdecl \rangle$ ::= $\langle Id \rangle$ ($\langle Id \rangle$*) $\langle fbody \rangle$

$\langle fbody \rangle$ ::= $\langle stm \rangle$
| $\langle fdecl \rangle$ $\langle fbody \rangle$

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

Expressions

expressions comes in three different types, arithmetic, boolean and a *nil* expressions.

arithmetic expressions consists of integers, variables referencing integers, or the usual binary operations on these two. We also consider function calls an arithmetic expression, but assigning anything other than integers to a variable will result in a runtime error. **boolean expressions** consists of the boolean values $true$ and $false$, as well as comparisons of arithmetic expressions.

nil nil expression is a special case which is only returned from *while* statements.

Statements

Statements consists of assigning integer values to variables, *if-then-else* statements for branching and a *while-do* statement for looping. Finally a statement can simply an expression, as well as a compound statement to allow for more than one statement to be executed.

Function declarations

Function declarations consists of an identifier, followed by a list of zero or more identifiers for parameters, and finally a function body which is simply a statement. Functions does not have any side effects, so any variables declared in the function body will be considered local. Furthermore, any mutations of globally defined variables will only exist in the scope of that particular function.

programs

We consider a program to be zero of more top-level function declarations, as well a statement

3.2.1 Interpreting *SImPL*

In order to work with ***SImPL*** , we have build a simple interpreter using the *Scala* programming language.

We represent our environment as a map

$$env : \langle Id \rangle \to \mathbb{N} \tag{3.1}$$

and each time we interpret a statement we return an instance of this map that reflects the current state of the environment. Note that we restrict both variables and function arguments to integer values.

A program is represented as an object $Prog$ which carries a map

$$funcs : \langle Id \rangle \to \langle fdecl \rangle$$

of top-level function declarations, as well as a root statement. To interpret the program we simply traverse the tree starting with the root statement. Interpreting function calls consists of looking up the function in the $Prog$ object, add the function parameters to the environment and then interpreting the statement in the function body. In order to keep functions from having side effects, we simply return the original environment as it was before adding function parameters and interpreting the body.

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`.