Symbolic Execution(Working title)

Aarhus Universitet

SOLIDUM PETIT IN PROFUNDIS · UNIVERSITAS ARHUSIENSIS ·

Søren Baadsgaard

February 5, 2019

Abstract

Contents

1 Introduction **2**

2 Summary of theory **3**

3 Basic symbolic execution for the *SImPL* language **4**

3.1 description 4

3.2 Introducing the *SImPL* language 4

3.2.1 Interpreting *SImPL* 5

4 Further extensions **6**

5 Conclusion **7**

A Source code **8**

B Figures **9**

Chapter 1

Introduction

Chapter 2

Summary of theory

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:

$$\langle \mathit{int} \rangle ::= \text{0} \mid \text{1} \mid \text{-1} \mid \text{2} \mid \text{-2} \mid \ldots$$

$$\langle \mathit{var} \rangle ::= \text{a} \mid \text{b} \mid \text{c} \mid \ldots$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle \mathit{exp} \rangle ::= \langle \mathit{int} \rangle} \\
\mid & \langle \mathit{var} \rangle \\
\mid & \langle \mathit{exp} \rangle + \langle \mathit{exp} \rangle \mid \langle \mathit{exp} \rangle - \langle \mathit{exp} \rangle \mid \langle \mathit{exp} \rangle * \langle \mathit{exp} \rangle \mid \langle \mathit{exp} \rangle \ / \ \langle \mathit{exp} \rangle \\
\mid & \langle \mathit{exp} \rangle > \langle \mathit{exp} \rangle \mid \langle \mathit{exp} \rangle == \langle \mathit{exp} \rangle
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle \mathit{stm} \rangle ::= \langle \mathit{exp} \rangle} \\
\mid & \langle \mathit{var} \rangle = \langle \mathit{exp} \rangle \\
\mid & \langle \mathit{stm} \rangle \ \langle \mathit{stm} \rangle \\
\mid & \text{if } \langle \mathit{exp} \rangle \text{ then } \langle \mathit{stm} \rangle \text{ else } \langle \mathit{stm} \rangle \\
\mid & \text{while } \langle \mathit{exp} \rangle \text{ do } \langle \mathit{stm} \rangle \\
\mid & \text{Print E}
\end{array}$$

$$\langle \mathit{prog} \rangle ::= \langle \mathit{stm} \rangle$$

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 $\langle \mathit{prog} \rangle$ 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