Towards Automatic Test Suite Generation for Functional Programming Assignments using Budgeted Compositional Symbolic Execution

Wesley Jay Baartman

TU Delft

Towards Automatic Test Suite Generation for Functional Programming Assignments using Budgeted Compositional Symbolic Execution

by

W.J. Baartman

born in Hoofddorp, the Netherlands

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
to be defended publicly on Wednesday October 26, 2022 at 13:00.

Student number:	4603524	
Project duration:	October 25, 2021 – October 26, 2022	
Thesis committee:	Prof. dr. A. van Deursen,	TU Delft, supervisor
	Prof. dr. E. Visser[†],	TU Delft, supervisor
	Dr. C.B. Poulsen,	TU Delft, daily supervisor
	Dr. S. Dumančić,	TU Delft

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Preface

This thesis focuses on automatically generating test suites for functional programming language from a reference implementation and a set of potentially faulty student solutions in order to be used for grading. This project was conducted within the Programming Languages Group[1] at the Delft University of Technology. The daily supervision was provided by Assistant Professor Casper Bach Poulsen of the Programming Languages Group. The role of responsible supervisor of this project was originally fulfilled by Professor Eelco Visser of the Programming Languages Group. However, after his extremely unfortunate and unexpected passing, this role was taken over by Professor Arie van Deursen of the Software Engineering Research Group at the TU Delft. The third observing committee member is Assistant Professor Sebastijan Dumančić of the Algorithmics Group at the TU Delft.

The source code and evaluation scripts are digitally available[2].

[1]https://pl.ewi.tudelft.nl/
[2]https://github.com/CptWesley/AutomatedDifferentialTestingOfDefinitionalInterpreters

Acknowledgements

I would like to thank many people that have supported me over the years in order to reach this point in my career, and perhaps even more important, in my life. However, I would like to thank certain people in particular that have lead or have indirectly contributed to the work presented in this document.

First of all I want to thank Casper Bach Poulsen for igniting the spark for my interests in programming languages during his bachelor course "Concepts of Programming Languages". And who now many years later supervised this master thesis from start to end, even when I stopped believing in my myself and all communication felt difficult for me.

Second of all I would like to thank the late Eelco Visser, who originally shaped the aforementioned bachelor course later taught by Casper. And who further embolstered my passion for programming languages with his master courses on compiler construction. I will never forget that he took time to talk to me about my personal situation when times were hard during the covid lockdowns. Even though I have never been able to express this gratitude in person, it was very meaningful to me.

Thirdly I would like to thank my friends for supporting me throughout this journey. In particular I would like to thank Andra Popa, Paul van der Stel, Luc Everse, Taico Aerts and Caro Brandt for allowing me to bounce off ideas, and vent my frustrations, and also for proof reading (parts of) my thesis, and who believed in me after I stopped believing in myself. I would also like to thank all my other friends and acquaintances who have also been subject to either my pitching of ideas or frustration venting regarding my thesis over the last year. Furthermore, I would also like to thank Paul van der Stel again for creating the cover page of this thesis, after I told him I was jealous of his own thesis cover page.

W.J. Baartman
Delft, September 2022

Abstract

In this thesis, we have defined a symbolic execution technique to automatically generate test suites for programs written in functional programming languages that can find the behavioural differences between a reference implementation and a set of potentially different implementations. Our symbolic execution technique uses a constraint solver in order to find a model that satisfies all constraints that together represent an execution path through the program. Furthermore, our technique utilises manually defined budget constraints to guide the symbolic execution to more interesting areas of the programs. These budget constraints define an initial budget that dictates when the symbolic execution terminates, and a set of costs associated to certain operations such as calling (specific) functions or performing (specific) pattern matches. This allows the symbolic execution to deplete budgets faster when it explores functions or branches of a program that are deemed "unlikely to be interesting". This results in less system resources being used on exploring these uninteresting execution paths, and instead allows for exploration of deeper paths in more interesting areas of the programs. Our budget constraint optimisation strategy works alongside other well-known optimisation strategies for symbolic execution such as early branch pruning and reusing intermediate results, more commonly known in the field of symbolic execution as compositioning.

In order to perform our symbolic execution on functional programs, we have defined a variation of Control Flow Graphs (CFG) for functional programming languages which allow modelling the execution flow between the different expressions that form the body of functions in functional programming. Additionally, our technique uses a constraint solver in order to find what paths through the program are feasible and to generate inputs that will lead to the execution of said paths. To allow us to specify these constraints on a higher-level, we have defined an intermediate Domain Specific Language (DSL), which can be transformed into a lower-level language understood by the constraint solver. We refer to this DSL as our Intermediate Constraint Language (ICL). Our ICL supports working seamlessly with type systems that allow inheritance, rather than only supporting Algebraic Data Types (ADT).

Furthermore, we have developed a symbolic execution engine for programs written in a pure and functional subset of the Scala programming language. This symbolic execution engine implements our proposed symbolic execution technique alongside the other described optimisation techniques. We have used this implementation in order to measure the effectiveness of our symbolic execution technique. In order to do this, we have used our technique in a scenario to automatically generate test suites for real student submissions of introductory assignments to functional programming. We have performed a comparison between the obtained branch coverages and mutation scores with our technique and the manually written test suites that have been used for multiple years. Finally, we have reflected on the feasibility of using this approach in practice by looking at the execution time and the number of inquiries to the constraint solver, required to generate test suites that are able to find an adequate number of errors.

Contents

List of Figures

1 Introduction

In education, we like to evaluate the progress that our students have made using assignments. In our field, this is often done by means of programming exercises. When grading student submissions, we would ideally like to make sure that they have created an implementation that produces the correct output for any given input. Often, we create a *reference* solution for these exercises, which provides a correct implementation of the exercise. We can also say: We want to make sure the submissions are *behaviourally equivalent* to our reference implementation. However, it is known that solving this *equivalence problem* for programs is a variant of the *halting problem* and is thus undecidable in the general case [1]. Therefore, we have to resort to non-*complete* solutions. There are generally two approaches: manual grading and automated grading. Manual grading allows for more fine grained feedback that is relevant to the student, but is prone to human errors and is not scalable in terms of student numbers and assignment complexity. Automated solutions, on the other hand, are more scalable, systematic and allow for shorter feedback cycles.

We will focus on automated grading using automated (unit) testing. This entails running submissions against a set of test cases to detect incorrect behaviour in the implementation. Manually finding the right set of test cases, that can detect most of the student errors, can be quite tricky. Consider this **incorrect** *Haskell* implementation of an interpreter for a small function programming language:

```
interp :: Exp -> Environment -> Value
interp (Num n) _ = NumV n
interp (Add e1 e2) nv = case ((interp e1 nv), (interp e2 nv)) of
                        ((NumV n1), (NumV n2)) -> NumV (n1 + n2)
                        _ -> error "Operands not numbers"
interp (Var n) nv = find nv n
interp (Lambda n b) nv = Closure nv n b
interp (Apply e1 e2) nv = case interp e1 nv of
                          (Closure cnv n b) -> interp b (Entry n (interp e2 nv) nv)
                          _ -> error "Not a function"
```

Here, the error is that the current environment is being used to evaluate the body of the *closure* instead of the *environment* stored in the closure, causing *dynamic scoping* rather than the desired *lexical scoping*. Finding a test case that would catch this small error is probably even less evident:

```
((lambda x
  ((lambda z
    (x 3)
  ) 2)
) (lambda y z))
```

Our goal is to automatically generate a test suite for grading purely functional programs given a reference implementation. As a benchmark, we have focused on generating test suites to detect common errors made by students that we have observed in a second year bachelor's course on functional programming, including the scoping error shown above.

To this end, we explore *symbolic execution* [2] for programs written in a purely functional subset of Scala as an approach for automatically generating test cases. Symbolic execution is a technique, which, in contrast to the classical *concrete execution* of a program, uses symbolic inputs to execute all *feasible paths* in a program [3], rather than just a single path. However, this technique is infamous for suffering from the *path explosion* problem, due to the number of feasible paths through a program growing exponentially and in most real world scenarios being infinite, due to existence of unbounded loops and recursion [4].

Previously, techniques based on *compositional analysis* have been developed [5–9] to mitigate the path explosion problem. These *compositional symbolic execution* approaches achieve this by eliminating redundant constraint solver invocations. These approaches use so-called *summaries* to capture *pre-* and *post-conditions* of the symbolic values obtained by analysing smaller portions of the full program. These summaries can be reused when analysing a bigger portion of the program which contain these smaller portions.

We aim to further mitigate the path explosion problem by introducing budget constraints to compositional symbolic execution. These budget constraints indicate the total budget and the costs of taking certain paths through the program, which allow the user of the framework to direct the symbolic executor towards parts of the program in which we expect more errors to exist. This allows exploring fewer less interesting paths and thereby saving time.

We present the following technical contributions:

- We describe a *budgeted* symbolic execution-based framework for automatically generating a test suite for submissions of introductory functional programming assignments in chapters 4 to 7 based on some example programs described in chapter 2. This framework aims to efficiently explore all feasible paths through a program that do not violate a set of manually predefined budget constraints. The path explosion problem is further mitigated by employing optimisation strategies such as *pruning* infeasible branches, compositional analysis [5–9] and budget constraints to guide the symbolic execution. For each of the explored paths, it generates a set of inputs that would cause the execution of the program to follow that path. It then adds a test case to the test suite with given inputs and the expected output found by *concretely executing* the reference solution with the found inputs.

- We define a special annotated variant of *control flow graphs* [10] in chapter 4. This variant is used to model the execution flow within the expressions that form the function bodies of pure functional programs.

- In order to properly represent the type-relations present in the Scala language, we define our *intermediate constraint language*, which serves as a *domain specific language* which encodes the semantics of a pure functional language with a *type system* which supports *inheritance* as *algebraic data types* in order to ensure *type safety* in the *SMT solver* in chapter 5.

- We evaluate the effectiveness of the implemented framework (chapter 8) by looking at different aspects. Using a large set of student submissions and a collection of anecdotal programs of commonly occurring bugs, we evaluate the effects of different budget constraints on the run time of the symbolic execution and the quality of the generated test suite in terms of capability of finding the inserted bugs, the *structural code coverage* and the *mutation coverage*.

The remainder of this thesis is structured as follows: Chapter 2 further concretises the problem with more detailed examples and provides intuition for the proposed solution. The following sections will then provide a technical description of the implemented solution. Chapter 3 provides an overview of the Scala subset which is supported by the current implementation and provides suggestions of how support can be extended in the future. An explanation of how we transform *abstract syntax trees* into our special annotated variant of control flow graphs is provided in chapter 4. We then present our intermediate constraint language which we use to encode the Scala type system within the SMT solver in chapter 5. Next, chapter 6 then provides a detailed explanation of how our implementation performs symbolic execution, including how we defined our summaries, how we define budget constraints, and how we perform composition of those summaries, taking into account the budget constraints. Chapter 7 concludes the technical sections by explaining how the entire pipeline can be utilised to generate an entire test suite given a reference implementation and a set of input programs. We then evaluate and reflect on the effectiveness and limitations of our approach in chapter 8. We then discuss related work in chapter 9. Next, we provide suggestions for improvements that can be made in future work in chapter 10. Finally, we summarise this thesis in chapter 11.

2 Motivating Examples

In order to explain the workings of the proposed framework, we will explain the different steps using a number of examples. The examples provided throughout this thesis are written in the *Haskell* programming language, rather than our target language *Scala*, due to the more widespread familiarity of the Haskell language in the academic literature. The concepts we will discuss are applicable to pure functional programming concepts in general. We will first demonstrate an example with an evident error and will then demonstrate an example with a less evident error. Afterwards, we will explain how performing an analysis using *symbolic execution* can aid us in finding the aforementioned errors.

```haskell
data List = Nil
          | Cons Int List
```

(a) Type definitions used for all examples.

```haskell
len :: List -> Int
len Nil = 0
len (Cons _ t) = 1 + (len t)
```

(b) Correct implementation using recursion.

```haskell
len' :: List -> Int
len' xs = lenAcc xs 0

lenAcc :: List -> Int -> Int
lenAcc Nil n = n
lenAcc (Cons _ t) n = lenAcc t (n + 1)
```

(c) Correct implementation using tail recursion.

```haskell
len'' :: List -> Int
len'' (Cons _ Nil) = 1
len'' (Cons _ t) = 1 + (len'' t)
```

(d) Incorrect implementation where the empty list is not considered.

Figure 2.1: Example implementations of a function that computes the length of `List`s.

2.1 The Problem

For the first example (figure 2.1) we will consider a data type definition (figure 2.1a) of a *LISP*-like *cons list* named `List`. `List` has two possible constructors: `Nil`, representing the empty list and `Cons head tail`, where `head` represents an element in the list and `tail` represents the rest of the list. We describe three programs that were intended to count the length of the provided `List`. Figure 2.1b represents a straightforward and correct implementation using `Nil` as its base case. Figure 2.1c represents an implementation that uses a helper function `lenAcc` to employ an accumulator to compute the final value. This helper function also uses the empty list as its base case. The third program (figure 2.1d) however, does not use the empty list as its base case, but instead uses a list containing a single element as its base case. It is evident that this `len''` function will result in an error when an empty list is provided as input, but will result in the correct result for any other input. This means that we should at the very least expect `Nil` to be used as input in one of our generated test cases in order to catch this error. This is in line with the commonly used testing technique of *boundary value analysis*, where values for test cases are taken that are on either side of boundary values [11], where an empty list is a boundary value for the length of a list.

```
 1  data Exp = Num Int
 2           | Add Exp Exp
 3           | Lambda String Exp
 4           | Apply Exp Exp
 5           | Var String
 6
 7  data Value = NumV Int
 8             | Closure Environment String Exp
 9
10  data Environment = Empty
11                   | Entry String Value Environment
12
13  find :: Environment -> String -> Value
14  find Empty _ = error "Variable not found in environment"
15  find (Entry n1 v t) n2 = if n1 == n2 then v else find t n2
```

(a) Type definitions and helper functions used for all examples.

```
 1  interp :: Exp -> Environment -> Value
 2  interp (Num n) _ = NumV n
 3  interp (Add e1 e2) nv = case ((interp e1 nv), (interp e2 nv)) of
 4                          ((NumV n1), (NumV n2)) -> NumV (n1 + n2)
 5                          _ -> error "Operands not numbers"
 6  interp (Var n) nv = find nv n
 7  interp (Lambda n b) nv = Closure nv n b
 8  interp (Apply e1 e2) nv = case interp e1 nv of
 9                            (Closure cnv n b) -> interp b (Entry n (interp e2 nv) cnv)
10                            _ -> error "Not a function"
```

(b) Correct implementation.

```
 1  interp :: Exp -> Environment -> Value
 2  interp (Num n) _ = NumV n
 3  interp (Add e1 e2) nv = case ((interp e1 nv), (interp e2 nv)) of
 4                          ((NumV n1), (NumV n2)) -> NumV (n1 + n2)
 5                          _ -> error "Operands not numbers"
 6  interp (Var n) nv = find nv n
 7  interp (Lambda n b) nv = Closure nv n b
 8  interp (Apply e1 e2) nv = case interp e1 nv of
 9                            (Closure cnv n b) -> interp b (Entry n (interp e2 nv) nv)
10                            _ -> error "Not a function"
```

(c) Incorrect implementation where the current environment `nv` is used to evaluate the body of the closure instead of the environment of the closure `cnv` (line 9).

Figure 2.2: Example implementations of a simple interpreter.

```
1  ((lambda x
2    ((lambda z
3      (x 3)
4    ) 2)
5  ) (lambda y z))
```

Figure 2.3: Example of a test case that would reveal the error in figure 2.2c.

For the second example (figure 2.2) we will take a look at a more insidious programming error we find in student interpreters. We first define the used data types and some helper functions for them in figure 2.2a. We consider a very simple language that only supports integers, addition and functions. Figure 2.2b and figure 2.2c represent a correct and incorrect implementation of *closures* respectively. Closures are intended to ensure the *static scoping* of a program [12]. A test case that exposes this incorrect behaviour is more complex. To be precise, it requires a situation where a function application gets evaluated with an environment that holds a different value for a variable than the value that is stored inside the closure. One example of such an input is shown in figure 2.3. For the correct implementation `interp`, the result will be an error, because the variable `z` is not defined in the environment in the closure of `(lambda y z)`. While in the incorrect implementation `interp'`, the result will be `2`, due to the fact that `z` is bound to `2` in current environment when `(lambda y z)` gets applied.

As can be seen from the second example, it can be hard to spot small errors and find a suitable test case for them. This task is very time consuming, especially when trying to find suitable test cases for multiple of such errors. The error we just showed related to using the local environment rather than the closure environment when evaluating the body of the function, but there are many more ways in which students create undesired scoping behaviour, such as by using various combinations of the local and closure environments. However, scoping behaviour is not the only place where errors can be made. Other common errors are usually the result of implementing (incorrect) behaviour for cases that are either too specific, or not specific enough. An example of such an incorrect implementation is given in figure 2.4. Here the implementation for interpreting `Add` expressions matches a pattern which is too specific, which leads to valid inputs such as `(Add (Add (Num 1)(Num 2))(Num 3))` to result in an error. Thus, (partially) automating this task of finding all these test cases that can uncover all such errors could reduce the amount of time spent manually looking for these scenarios.

```haskell
interp :: Exp -> Environment -> Value
interp (Num n) _ = NumV n
interp (Add (Num e1) (Num e2)) nv = NumV (e1 + e2)
interp (Var n) nv = find nv n
interp (Lambda n b) nv = Closure nv n b
interp (Apply e1 e2) nv = case interp e1 nv of
                          (Closure cnv n b) -> interp b (Entry n (interp e2 nv) cnv)
                          _ -> error "Not a function"
```

Figure 2.4: Incorrect implementation of the same `interp` function as given in figure 2.2b, where the case for `Add` is too specific and does not allow for nested arithmetics.

2.2 The Solution

Our proposed solution is based on the intuition that if a program can exhibit some erroneous behaviour, we can make it exhibit this erroneous behaviour by executing every single path through the program with all possible inputs that guides the program through those paths and observing whether or not the output matches the output given by the reference program with the same inputs. In order to execute all these paths, we will utilise the technique of *symbolic execution* [2, 3].

Regular *concrete execution* uses concrete values as inputs to be given to a program, which then executes a single path through the program. Symbolic execution on the other hand, uses *symbolic variables* which do not yet have a particular value, but instead have (mathematical) constraints assigned to them along the different execution paths. Whenever during symbolic execution we reach a branching point, such as an *if-then-else* or a *match* expression, the execution forks and continues in all branching paths at the same time, while updating all symbolic values with the relevant constraints.

```haskell
abs :: Int -> Int
abs n =
    if
        n < 0    -- {}
    then
        -n       -- {n < 0}
    else
        n        -- {n >= 0}
```

Figure 2.5: Example of symbolic execution of a function which computes the absolute value of an integer. Constraints of the `n` variable are provided at different points in the execution.

In order to give an example of how symbolic execution works, we will first take a look at the very simple program given in figure 2.5. Here a simple function `abs` is given which computes the absolute value of an integer. In concrete execution, we would have to assign a concrete value to the variable `n` in order to execute `abs`. When we execute the function with `n` assigned to the concrete value `-5`, the program first evaluates the expression `n < 0`. This expression evaluates to `true`, since we know that `n` has the value `-5` and `-5 < 0` is true. This causes the execution to move to the *then* branch. Substituting the value of n into the expression -n results in --5, which evaluates to 5, which is consequently also the return value of the function.

With symbolic execution on the other hand, we do not yet assign any value to n when we start executing the function. The constraints that are known at each point of the evaluation are added as comments to the code in figure 2.5. When we execute the function, we first encounter the expression n < 0 in the condition of the *if-then-else* construct. The execution now forks into two different executions: one that continues in the *then* branch and one that continues in the *else* branch. Since the *then* branch can only be entered when n < 0 holds, it is added to the set of constraints. Similarly, when the *else* branch is entered, it cannot hold that n < 0, thus the negation of the condition (n >= 0) is added to the set of constraints. When the function has finished executing for all forks, we end up with two sets of constraints: {n < 0} and {n >= 0}. For each of these constraint sets we could now generate all possible values for n for which the constraints hold. This would give us the following two input sets: {n | n < 0} and {n | n >= 0} respectively.

We can use this approach to find the fault present in the example given by figure 2.1d. To keep things simple we will ignore the handling of function calls for now. To make things more clear, we will first rewrite the example to make the order of evaluation more explicit. Since the order of evaluation of pattern matching happens top-down, we convert the pattern matching expression into a chain of pattern matches. Furthermore, since the patterns in the example are not exhaustive, we will also insert an extra pattern which leads to aborting the program using the error function. The result of this rewrite can be found in figure 2.6.

```
1  len :: List -> Int
2  len xs = case xs of             -- {}
3    (Cons _ Nil) -> 1             -- {xs matches (Cons _ Nil)}
4    _ -> case xs of               -- {xs does not match (Cons _ Nil)}
5      (Cons _ t) -> 1 + (len t)  -- {xs does not match (Cons _ Nil), xs matches (Cons _ t)}
6      _ -> error "no match"       -- {xs does not match (Cons _ Nil), xs does not match (Cons _ t)}
```

Figure 2.6: Modified version of figure 2.1d that makes control flow more explicit.

As before, we included the set of constraints on the value of xs at different points in the program. The symbolic execution now forks twice: once after line 2 and once after line 4. We can observe that there will be three different sets of constraints when the analysis finishes:

```
{xs | xs matches (Cons _ Nil)}
{xs | xs does not match (Cons _ Nil) ∧ xs matches (Cons _ t)}
{xs | xs does not match (Cons _ Nil) ∧ xs does not match (Cons _ t)}
```

If we can find a value for xs which satisfies the third set of constraints, we have found an input which leads to the program aborting. The only value that satisfies these constraints is Nil. If we perform a concrete execution of the correct implementation in figure 2.1b with Nil as input, we find that the function returns 0. This means we have found difference in behaviour for the input Nil between the example in figure 2.1d and the example in figure 2.1b.

However, performing such an analysis in practice presents two major complications. The first one being that obtaining full *path coverage* by executing all possible paths is impossible for most real programs. This is due to the fact that there might be an infinite number of paths through a program, if the program contains loops or recursion [13]. This problem is also present in both of the examples we have provided. In the functions given in figure 2.6 a recursive call happens whenever a Cons with a non-Nil tail is encountered. Previously, we ignored this recursive call, but in order to make the symbolic execution *complete*, we would need to continue our symbolic execution in the called function. There are however no limitations on how many times a Cons can occur inside another Cons, therefore there can be an infinite number of different amounts of nested Cons values, causing an infinite number of possible paths through the programs. The same problem can be observed with the recursive handling of Add and Apply values in the functions given in figure 2.2. Therefore, we have to settle for only analysing a subset of all possible paths, in order to make the analysis feasible. As a result, we can no longer guarantee that all faulty behaviour will be uncovered. However, as can be seen from the analysis provided earlier, for some faults it is not necessary to analyse very deep paths to uncover them. Our evaluation in chapter 8 will provide some insight into how many errors can be found for different depths.

The second complication arises from the fact that, ideally, for each path we want to cover, we want to cover all possible values that guide the execution through this path. However, for many real programs, the number of possible inputs can be larger than is realistically feasible to test or can even be infinite. If we take another look at figure 2.5 and we want to find all inputs for the *then*-branch, we have to find all values of type `Int` which are smaller than `0`. If this was a Haskell program, identical to the example, then there would be 2^{29} such inputs. If this was a Scala program, this number would be even larger, namely 2^{31}. Similarly, we find an equal number of inputs for the *else* branch. We can rationalise that it is very likely that any value within these domains has the same observable behaviour as any other value within this domain, therefore it might be sufficient to test only a single or small number of inputs from each domain. If we only select a single input from each domain, we can reduce our required number of test cases from 2^{30} or 2^{32} (depending on the language) to 2. However, this does mean that the solution becomes subject to cases where the output is correct, but for the incorrect reasons [11]. Our solution makes this trade-off and only selects a single input set for each found execution path. In chapter 10 we discuss the possibility of extending the current implementation in order to select multiple input sets for each execution path.

Our proposed solution automatically generates a test suite for a given program using symbolic execution as described above. An example of the test cases that our solution finds for a program such as figure 2.5, are shown in figure 2.7.

```
assert ((abs -1) == 1)
assert ((abs  1) == 1)
```

Figure 2.7: Example of a potential test suite generated by our framework for the program given in figure 2.5.

In order to limit the number of explored paths, the user can provide the framework with *budget constraints* that control which paths the symbolic executor will explore. These budget constraints dictate the original starting budget and associate costs of performing certain actions during the symbolic execution. When the budget is depleted along a certain execution path, the symbolic execution along that path terminates. When no more branching paths are available where the budget has not yet been depleted, the overall symbolic execution will terminate. An example of such a cost association is assigning a cost to performing a function call. These costs can be assigned to all function calls or to calls to specific functions in order to give a lower priority to exploring certain functions. Additionally, costs can be assigned to matching specific patterns in pattern match expressions. More detailed explanations on how these budget constraints work are provided in chapter 6.4.

We have evaluated the feasibility and the effectiveness of the generated test suites when using different budget constraints for various scenarios. Additionally, we have also explored what budget constraints are necessary in order to detect the error present in figure 2.2c and various other errors. The findings of these evaluations can be found in chapter 8.3.

An example of a test suite generated by our framework for the program in figure 2.6, is given in figure 2.8. Here the only configured budget constraint is that each function deducts 1 from the remaining budget. The comments denote the minimum required initial budget for this test case to appear in the generated test suite. In the following sections we will provide a detailed description of each step in our solution.

```
assert ((len Nil)                                  == 0) -- budget >= 0
assert ((len (Cons 0 Nil))                         == 1) -- budget >= 0
assert ((len (Cons 0 (Cons 1 Nil)))                == 2) -- budget >= 1
assert ((len (Cons 0 (Cons 1 (Cons 2 Nil))))       == 3) -- budget >= 2
assert ((len (Cons 0 (Cons 1 (Cons 2 (Cons 1 Nil))))) == 4) -- budget >= 3
...
```

Figure 2.8: Example of a potential test suite generated by our framework for the program given in figure 2.6.

3 Supported Scala Subset

First, we will provide an overview of the Scala subset that is supported by our framework. We will do this, by first providing a brief description of the Scala language and how it differs from more traditionally used functional programming languages in chapter 3.1. We will then give an overview of the further limitations imposed by our current implementation, and suggestions on how these limitations can be resolved in chapter 3.2.

3.1 The Scala Language

The Scala programming language is a high-level general-purpose programming language which combines the concepts of *object-oriented programming* (OOP) and *functional programming* [14]. This, as a consequence, means that Scala is in many ways more similar to high-level OOP programming languages, such as *Java* [15] and *C#* [16], than it is to more classical examples of functional programming languages such as Haskell [17] and *Standard ML* [18].

However, as mentioned earlier, the subset of Scala which is supported by our framework is *pure* and functional. We define the purity of a language as lacking any *state* or *mutability* and therefore functions can not have any *side-effects* associated with them. We classify a language as being functional if it allows treating functions as values. Thus we have ignored most of the imperative and impure features of the Scala language for this study.

Nevertheless, there are still various OOP features present in this functional subset. The most important of those features relate to the type system: *classes*, *traits* and *objects*. OOP languages organise their source code using classes [19]. A class defines a *type* which encapsulates *data* as *fields*, similar to *structs* in languages such as *C*, and the functions that operate on those data as *methods*. A value of such a class type is referred to as an *instance*.

Furthermore, in order to promote reusability, it is possible for a class to extend another pre-existing class. This is referred to as *inheritance* between a *parent* class and a *child* class. Inheritance also allows for child classes to *override* methods that were previously defined in their parent class. The specific form of inheritance supported by the Scala language can be classified as *hierachical multilevel single inheritance*. This means that classes are only allowed to inherit from a single other class, which may also inherit from a single other class, but multiple classes may inherit from the same parent class [20]. An instance of a child class can always be used in places where an instance of the parent class is expected. Additionally, it is also possible for a class to be not fully defined yet. When this is the case, the class is considered *abstract*. This makes it impossible to create an instance of this class. Instead, instances can only be made of child classes.

Additionally, to bridge the gap between functional programming and OOP, Scala also offers *syntactic sugar* to define classes in a manner that is more akin to functional programming. This is done using so-called *case classes* [21], which are comparable to *record types* with immutable fields [22].

Furthermore, Scala allows class composition through the usage of one or more *mixins* in the form of *traits* [23]. Adding a trait to a class allows that class to expose the functionalities provided by said trait, similarly to how *interfaces* work in other Object Oriented languages such as Java or C#. Therefore, the Scala type system allows you to use a class that has a certain trait if the type signature expects that same trait. This means it is possible to simulate multiple-inheritance while partially mitigating the *diamond problem* [24], where ambiguity arises when a class inherits from two classes which share a common ancestor [25].

Moreover, due to the presence of inheritance and mixins, it is necessary to find the correct implementation of a method during run time, when the method of an instance is invoked. Since it might not always be possible to determine which implementation of the method to use during compilation time. This is also known as *dynamic dispatch* [26].

Finally, Scala also allows for the creation of classes which can only have a single instance, or *singleton*, by defining a class as an `object` [27]. This is similar to the `static` keyword present in Java and C#, where all fields, methods and classes marked with the `static` keyword exist independent of any instances of the class in which they are defined [19].

3.2 Implementation Limitations

We shall now discuss the further limitations which our implementation imposes on the proposed framework. It must be noted that these limitations are not theoretical limitations on the approach, but rather on the current implementation of the approach. The limitations serve as a way of maintaining the simplicity of the proof of concept of our approach. We will therefore also provide a possible way of resolving each of these limitations after describing the respective limitation.

Firstly, our analysis only limits itself to source code, meaning that any definitions provided by external libraries (including the standard library of both Scala and Java), are not usable by our analysis. However, this can be circumvented by including (equivalent) relevant source code sections of those definitions in the framework. As a consequence, this also means that many commonly used types, such as tuples and lists, are not supported. However, some Scala types such as `Any`, `AnyVal`, `Object`, `Int`, `Bool` and `String` have built-in support in the framework.

Secondly, in the current implementation the object-oriented programming concepts that are part of Scala are supported in limited capacity. Types have to be defined as either *abstract classes* or *case classes*. The primary reason for this is to ensure the purity of the supported subset. *Traits* [23] are not fully supported in the current implementation, however, the definitions provided in chapter 5 do consider the usage of traits. *Parametric polymorphism*, better known in Java and Scala as *generics* [23], is also not supported. Support for this can be extended in a fairly straightforward fashion if *type-inference* would be added as an extra step when performing the symbolic execution. All function definitions currently have to be defined in an `object`, meaning that they cannot depend on an instance of a class. However, it is possible to extend the support for this by modeling member functions in the same way as these static functions with an extra parameter. This extra parameter should have the type of the object and when the function is called, the provided argument to that parameter should be the object instance on which the function operates. Then, some logic should be introduced to decide which function implementation will be used, depending on the different possible input types.

Furthermore, the current implementation attempts to resolve all functions from a global scope. This means that any function which is passed around as a value can not be resolved back to its definition. Introducing an extra resolution step during the symbolic execution could allow finding the correct function definition along the execution path. This would mean that all static functions need to be inserted into the global scope and all local definitions or aliases need to be inserted into their respective scopes. Extra care should be taken during the resolution to ensure that local name definitions should take proper priority over names definitions in outer scopes. Thus, this would lift this limitation. Consequently, *lambdas*, better known in Scala as *anonymous functions* [28], are also not supported. However, it is possible to extend the aforementioned solution by allowing the resolution step to also resolve the local variables that are bound inside the anonymous function in order to be able to perform symbolic execution of anonymous functions. One way of achieving this could be to turn all anonymous functions into named functions that require an extra parameter for each variable referenced inside the body of the anonymous function that is defined outside of the anonymous function.

Finally, most of the remaining commonly used pure language features are supported. This means that while it is not possible to reassign a new value to a variable defined using `var` or use `for`- or `while`-loops, it is possible to use most other language features such as *pattern matching* (and *subtyping*), *recursion* and *exception throwing*. A *grammar* in *Backus-Naur Form*

(BNF) [29] of the supported Scala syntax is provided in figure 3.1.

```
<id>                        ::= <letter> | <id> <letter> | <id> <digit>
<type-name>                 ::= <id>
<natural-number>            ::= <digit> | <digit> <natural-number>

<args>                      ::= <exp> | <args> "," <exp>
<param>                     ::= <id> ":" <type-name>
<params>                    ::= <param> | <param> "," <params>

<type-def>                  ::= "abstract class" <id> "(" <params> ")"
                              | "case class" <id> "(" <args> ")"
<func-def>                  ::= "def" <id> "(" <params> "):" <type-name> "=" <exp>

<integer>                   ::= <natural-number> | "-" <natural-number>
<bool>                      ::= "true" | "false"
<val-ref>                   ::= <id>
<val-def>                   ::= "val" <id> "=" <exp>
                              | "val" <id> ":" <type-name> "=" <exp>
<func-call>                 ::= <exp> "(" <args> ")"

<add>                       ::= <exp> "+" <exp>
<sub>                       ::= <exp> "-" <exp>
<mul>                       ::= <exp> "*" <exp>
<div>                       ::= <exp> "/" <exp>

<if>                        ::= "if (" <exp> ")" <exp> "else" <exp>
                              | "if (" <exp> ")" <exp>
<exp-chain>                 ::= "" | <exp>
                              | <exp> <new-line-character> <exp-chain>
                              | <exp> ";" <exp-chain>
<block>                     ::= "{" <exp-chain> "}"

<patterns>                  ::= <pattern> | <pattern> "," <patterns>
<pattern>                   ::= <bool> | <integer> | "_"
                              | <type-name> "(" <patterns> ")"
<cases>                     ::= <case>
                              | <case> <new-line-character> <cases>
                              | <case> ";" <cases>
<case>                      ::= "case" <pattern> "=>" <exp>
                              | "case" <pattern> "if" <exp> "=>" <exp>
<match>                     ::= <exp> "match {" <cases> "}"

<exp>                       ::= "(" <exp> ")"
                              | <integer> | <bool> | <val-ref> | <val-def> | <func-call>
                              | <add> | <sub> | <mul> | <div>
                              | <if> | <block> | <match>
```

Figure 3.1: Backus-Naur Form of the Scala syntax which is supported in the concept implementation.

4 Building the Control-Flow Graph

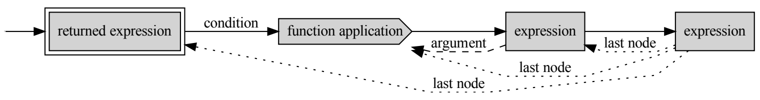
As the first step of our analysis, we use the *abstract syntax tree* (AST) of the program to construct an annotated *control-flow graph* (CFG) for each function in the program which is being analysed. These CFGs provide an intermediate abstraction of the program that allows us to more easily reason about the different execution paths through the functions. In chapter 6 we will use these CFGs to generate constraints in order to perform symbolic execution.

The CFGs which we construct are different in form from the CFGs originally described by [10]. The sole reason for this difference is that classic CFGs use graph nodes to represent basic blocks of instructions that are executed consecutively [10], while the language class that we are dealing with, namely functional languages, does not have a notion of instructions. Instead, our CFGs represent the evaluation order of the expressions inside their respective functions. Since pure functional programs do not contain any loop constructs such as `goto`, `for` or `while`, and because function calls have not been expanded, a constructed CFG will therefor always be a Directed Acyclic Graph (DAG).

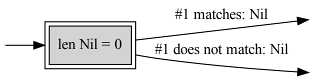
It is important to mention that when building the CFG, we consider pattern matching constructs as a sequence of nested *if-then-else* constructs where the matching happens as the *condition* and potential variables are bound in the *then* branch. We build the CFG in such a way that every *node* in the CFG represents an *expression* in the AST, taking into account the aforementioned implicit conversion of pattern matching constructs. Every *edge* in the CFG represents the evaluation going from one expression to the next and can be bound by conditions imposed by expressions that have diverting control flow, such as *if-then-else* expressions or pattern matching. Furthermore, we keep track of extra information such as which nodes correspond to return values, which nodes are function calls, which nodes are arguments to those function calls and which nodes are the last nodes required to evaluate a larger expression (e.g. the last operands of an operation). We will explain how this extra information is used for our symbolic execution technique in chapter 6.

The final CFGs of all example programs given in figure 2.1 are provided in figure 4.1. Here we let `#i` denote the i-th argument to the function. In order to make things more concrete, a step-by-step example of converting the program provided in figure 2.1b into its corresponding CFG follows:

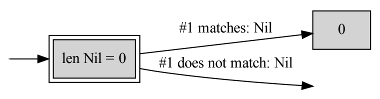
Legend:



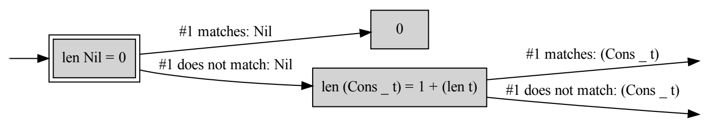
1. Since pattern matching happens top-to-bottom, we consider the pattern matching as a sequence of nested *if-then-else* constructs and thus we first try to match the input with the pattern `Nil` and we mark this initial node as the return value:



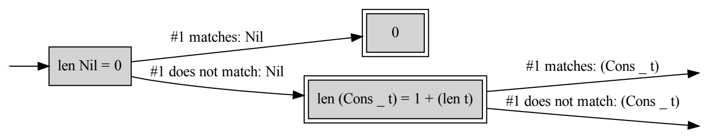
2. If the input matches the pattern `Nil`, the next evaluated expression is `0`, which also happens to be the last expression in this branch, so we are finished with this branch:



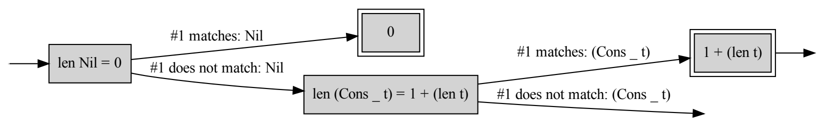
3. If the input does not match the pattern `Nil`, we then try to match the input with the pattern `(Cons _ t)`:



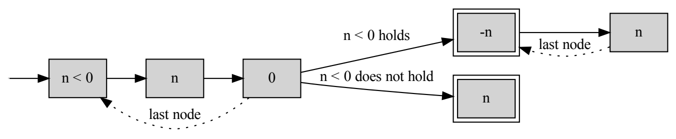
4. Since the evaluation of both expressions after the currently marked return node happen sequentially, in contrast to happening as part of evaluating the expression in the node, we *push* the return value marking down both branches:



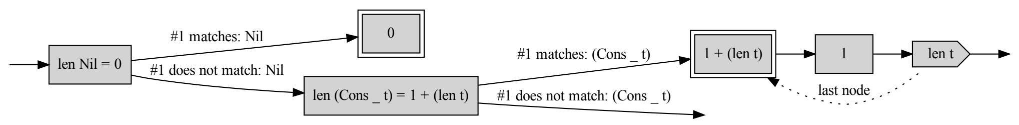
5. If the input matches the pattern `(Cons _ t)`, the program returns `1 + (len t)` and we push the return value marking into the branches (as described by step 4):



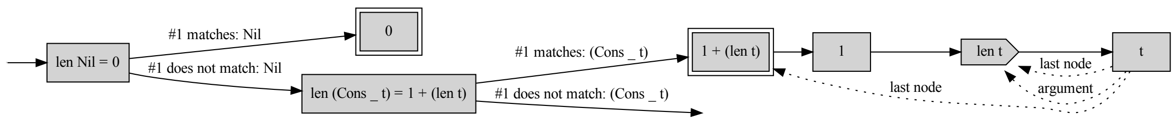
6. Evaluating `1 + (len t)` requires to first evaluate `1` and then evaluate `len t`. And since these evaluations happen as parts of evaluating `1 + (len t)`, we do not push the return value markings further into the branch. Furthermore, we observe that `len t` is a function application and thus mark it as such:



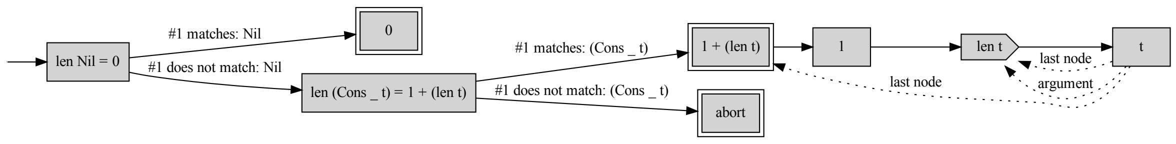
7. Since the evaluation of `1` doesn't require any further evaluations, we link the node to the evaluation of `len t`:



8. Evaluating `len t`, does require evaluating its argument `t` first, thus we append that evaluation to the branch, mark it as the argument to `len t` and we push the last node marking of `len t` deeper. We are now also done with this branch:

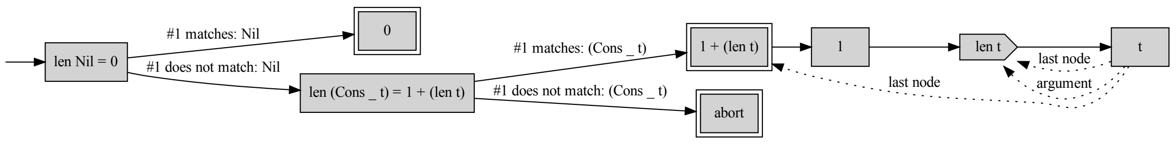


9. Finally if the input didn't match `(Cons _ t)` in step 3, no behaviour is specified in the program and the execution *aborts*:

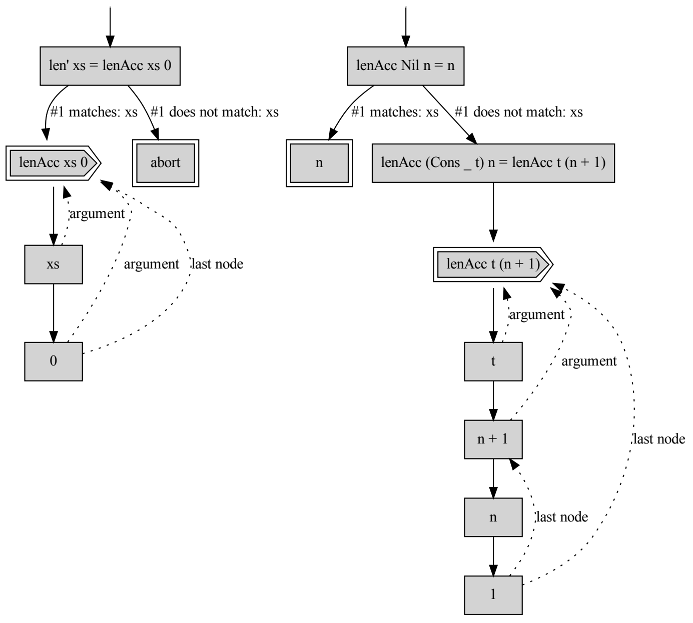


It should be noted that this procedure omits examples of handling certain language features such as sequential execution, real if-then-else expressions and pattern matching guards. Dealing with these features is relatively straight forward: For sequential execution, we simply connect the expressions being executed sequentially with an edge and push the return value marking or last node marking to the last sequential expression if the first expression was marked as either a return value or last node respectively. For if-then-else expressions, we simply use its condition and the negation of that condition as the edge conditions leaving the expression. For guards we create an extra node that represents the condition after the pattern is matched. The edge for which the condition holds connects to the node to which the pattern match originally pointed to. The edge for which the negation of the condition holds connects to the same node as the edge leaving the pattern match for which the pattern was not matched.

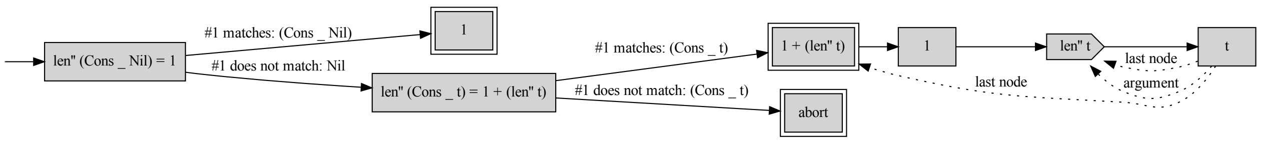
As we have seen in the provided examples, these CFGs encode all possible evaluation orders for all expressions in the AST of a function. This means that any path through a CFG encodes a full execution of that function. Finding all paths through a CFG means finding all possible execution paths through the function. In chapter 6 we will use the nodes and edges of the CFGs in order to generate a constraint set that represents an execution of the corresponding execution paths.



(a) Control flow graph for the correct `len` program in figure 2.1b.



(b) Control flow graph for the correct `len'` program in figure 2.1c.



(c) Control flow graph for the incorrect `len''` program in figure 2.1d.

Figure 4.1: Control flow graphs for the programs in figure 2.1 which compute the length of a `List`.

5 Intermediate Constraint Language

In order to perform symbolic execution of a program, we first need a way to transform an execution path through a program as a set of constraints that can be solved by a constraint solver. To that end we first define our *Intermediate Constraint Language* (ICL) used to encode execution paths as a set of constraints. This ICL forms an intermediate step between Scala, our source language, and Z3 [30], the SMT solver we have used. The ICL is an abstraction over the raw Z3 constraints, which has a larger similarity with the Scala language. It is therefore important that this ICL is capable of faithfully representing both the type relations and evaluation semantics that are present in the Scala language. Therefore, we will first describe the type systems of Z3 and Scala and provide an explanation of how Scala's type system can be encoded using Z3. Afterwards, we will explain the other language constructs present in our ICL.

5.1 Encoding the Type System

In order to perform our symbolic execution technique, it is necessary to use a constraint solver to solve constraint sets. For this study, we have used Microsoft's Z3 [30] SMT solver as our constraint solver. This particular choice should not affect the theory regarding our symbolic execution technique and could relatively easily be replaced by any other SMT solver, such as *cvc5* [31], given they provide similar ways of formulating constraints. We will now first describe Scala's type system and afterwards explain the problems that arise when trying to encode that type system in Z3's type system.

As we mentioned earlier in chapter 3.1, the Scala language has support for inheritance and mixins in the form of traits. Both inheritance and mixins are a form of *subtyping*. We first define a non-*reflexive* and non-*transitive* relation between two types: `A <| B`, which only holds if type `A` is a direct subtype of type `B`. This effectively means that either class `B` is the direct parent of class `A` **or** that class `A` has trait `B`. We also define a reflexive and transitive relation between two types: `A <: B`, which holds if type `A` and `B` are equal (reflexivity), if `A <| B` or if there exists a type `C` for which it holds that `A <| C` and `C <: B` (transitivity). This subtyping is both *nominal* and *inclusive*. Nominal meaning that two types are only considered subtypes if their definition specifies them to be [32]. And inclusive meaning that if `A <: B` holds, that any instance of type `A` is also an instance of type `B` and can therefore be used in contexts where an instance of type `B` is expected [33].

As an example, we will consider the *algebraic data type* (ADT) `Exp` provided in figure 2.2a. An equivalent representation in Scala would look similar to what is shown in figure 5.1. Here both `Exp` and `Add` are considered types, where `Add` is a sub-type of `Exp`. We can write this relation as: `Exp <| Add`. If no explicit parent class is provided in a class definition in Scala, the class implicitly inherits from the `Object` class, which inherits from the `Any` class. Thus the full subtype relation we find for the `Add` class is: `Add <| Exp <| Object <| Any`.

```scala
abstract class Exp();
case class Num(v: Int) extends Exp();
case class Add(e1: Exp, e2: Exp) extends Exp();
case class Lambda(n: String, b: Exp) extends Exp();
case class Apply(f: Exp, a: Exp) extends Exp();
case class Var(n: String) extends Exp();
```

Figure 5.1: Scala equivalent of the type definitions of the `Exp` type provided in figure 2.2a.

The first challenge appears as soon as we try to encode an object directly into Z3. Z3, along with most other SMT solvers, only support very basic ADTs. These ADTs are defined as a *data type* with multiple *constructors* [34], similar to Haskell's data types. Where each constructor represents a value of the given sort and where constructors can not be data types themselves. For example, lets take a look at a possible way of encoding `Num(1)`. The Simplest way would be to define a data type Exp with a constructor Num with a single field of sort Int. This would allow us to correctly encode a function which has a parameter of type Exp. However, this would not allow us to encode a function which has a parameter of type Int, since Int is not a data type. Furthermore, this would also not allow us to encode a function which has a parameter of type Object, because Exp is a data type and not a constructor. Both of these impossibilities are things that are possible in Scala. Therefore, this highlights the necessity for a way of encoding these functions.

One possible solution would be *type erasure*, where we simply get rid of the different types and their type relations and consider all values to be of the same type. This could be realised by defining a single over-arching data type and turning every type from our Scala program into a constructor for that type. However, this leaves room for the SMT solver to create models that violate the subtyping relations present in the original Scala program. For example, this would mean that Num(true) would become a valid value, because both Boolean and Int would be a constructor for Any. This would be impossible in Scala, because the relation Bool <: Int does not hold.

A better solution would require the present subtyping relations to not be violated. In order to achieve this, we opted for encoding both types and type-relationships as sorts in the Z3 type system. Similar to the approach of composing types using *coproducts* proposed by [35]. We do this by creating constructors for parent types that each wrap a subtype. For illustration purposes we will provide those definitions using Haskell syntax for defining data types. We start by manually defining the sorts for the primitive types:

```
data Int'     = ...
data Boolean' = ...
data String'  = ...
```

Here we use the suffix ' to differentiate between the sort representing the Scala type and the built-in primitive sorts in Z3 itself. We then define the constructors for the string and primitive types:

```
data Int'     = IntC Int
data Boolean' = BooleanC Bool
data String'  = StringC String
```

The syntax here means that IntC is a constructor for the type IntS which has a single argument which is of the built-in Z3 sort Int. We use the suffix C to differentiate between the constructor of a type and the corresponding type itself.

We then define the constructors for all other concrete types. Concrete types are all types of which instances can be created. In Scala this would be all classes that are not abstract. Let type S be the concrete type for which we wish to create a constructor. Let F be an ordered list of Z3 sorts which represent the types of the fields present in type S. Note that this also includes the fields of all types T for which it holds that S <: T. We now define a type and its corresponding constructor as:

```
data <S> = <S>C <F*>
```

Here we use the angle bracket (<...>) notation to indicate that the variable inside should be replaced with a specific value. Furthermore, since F is a list of types, we cannot use this directly in our definitions. Hence, it is necessary to first syntactically expand the list. Here F* represents the expansion of all types included in F. Doing this for all types defined in figure 5.1, gives us the following definitions:

```
data Num     = NumC Int'
data Add     = AddC Exp Exp
data Lambda  = LambdaC String' Exp
data Apply   = ApplyC Exp Exp
data Var     = VarC String'
```

So far we have not defined any of our parent types. Most notably, the Exp type which we have used above. We will now start defining the (abstract) parent types:

```
data Any    = ...
data AnyVal = ...
data Object = ...
data Exp    = ...
```

As we can see, these abstract types do not directly have their own constructors. Instead, we define constructors that represent subtyping relationships, which wrap subtypes in their respective super types. This means that for any subtype relation S <| T, we introduce the following definitions:

```
data <T> = ... | <T>$<S> <S> | ...
```

Doing this again for the types defined in figure 5.1 and the types used by these types from the Scala standard library:

```
data Any    = Any$AnyVal AnyVal
            | Any$Object Object
data AnyVal = AnyVal$Int Int'
            | AnyVal$Boolean Boolean'
            | AnyVal$String String'
data Object = Object$Exp Exp
data Exp    = Exp$Num Num
            | Exp$Add Add
            | Exp$Lambda Lambda
            | Exp$Apply Apply
            | Exp$Var Var
```

We can now combine these definitions into a full set of type definitions that encode both concrete values and subtyping relationships. These combined definitions are shown in figure 5.2.

```
data Any      = Any$AnyVal AnyVal
              | Any$Object Object
data AnyVal   = AnyVal$Int Int'
              | AnyVal$Boolean Boolean'
              | AnyVal$String String'
data Object   = Object$Exp Exp
data Exp      = Exp$Num Num
              | Exp$Add Add
              | Exp$Lambda Lambda
              | Exp$Apply Apply
              | Exp$Var Var
data Int'     = IntC Int
data Boolean' = BooleanC Bool
data String'  = StringC String
data Num      = NumC Int'
data Add      = AddC Exp Exp
data Lambda   = LambdaC String' Exp
data Apply    = ApplyC Exp Exp
data Var      = VarC String'
```

Figure 5.2: Type definitions which represent concrete values and subtyping relationships for the programs shown in figure 2.2.

If we take another look at the problematic encoding example of the value Num(1), we now find multiple different encoding possibilities. The simplest of these encodings is: NumC(Int'(1)). The sort of this encoding is Num. However, we can also wrap this term into its parent type to obtain a term of the Exp sort: Exp$Num(NumC(Int'(1))). We can continue doing this by further wrapping it in the Object$Exp and then Any$Object constructors to obtain representations of sorts Object and Any respectively.

5.2 Encoding the Evaluation Semantics

In order to encode the proper evaluation semantics of Scala, we defined numerous high-level constraints that can later be converted into constraints that can be solved by an SMT solver. We categorise these higher-level constraints into two groups: *Extended Intermediate Constraint Language* (EICL) and *Core Intermediate Constraint Language* (CICL). The EICL serves as the first layer to encode the semantics into a set of constraints that is still very similar to the Scala language. The CICL is a *proper subset* of the EICL and is more similar to the Z3 constraints, while still encoding the type system described in chapter 5.1. The reason for this distinction is that the EICL allows for some specialized optimizations that are no longer possible with the CICL. However, the CICL still provides a level of abstraction that is easier to work. *Backus-Naur Form grammars* [29] for the expressions in the CICL and the EICL are provided in figure 5.3.

```
<cicl-expression>           ::= "(" <expression> ")"
                              | <primitive> | <variable>
                              | <logic> | <arithmetic>
                              | <casting> | <objects>
<expression>                ::= <cicl-expression>

<arguments>                 ::= "" | <expression> | <expression> ", " <arguments>
<id>                        ::= <letter> | <id> <letter> | <id> <digit>
<type-name>                 ::= <id>

<primitive>                 ::= <boolean> | <integer> | <string>
<boolean>                   ::= "true" | "false"
<integer>                   ::= <digit> | <digit> <integer> | "-" <integer>
<string>                    ::= """ <string-body> """
<string-body>               ::= <character> | <character> <string-body>

<variable>                  ::= <id>

<logic>                     ::= <not> | <equals> | <lesser-than>
                              | <lesser-than-or-equals> | <greater-than>
                              | <greater-than-or-equals> | <and> | <or>
<not>                       ::= "!" <expression>
<equals>                    ::= <expression> "==" <expression>
<lesser-than>               ::= <expression> "<" <expression>
<lesser-than-or-equals>     ::= <expression> "<=" <expression>
<greater-than>              ::= <expression> ">" <expression>
<greater-than-or-equals>    ::= <expression> ">=" <expression>
<and>                       ::= <expression> "&&" <expression>
<or>                        ::= <expression> "||" <expression>

<arithmetic>                ::= <minus> | <add> | <subtract> | <multiply> | <divide>
<minus>                     ::= "-" <expression>
<add>                       ::= <expression> "+" <expression>
<subtract>                  ::= <expression> "-" <expression>
<multiply>                  ::= <expression> "*" <expression>
<divide>                    ::= <expression> "/" <expression>

<casting>                   ::= <cast-up> | <cast-down> | <can-cast-down>
<cast-up>                   ::= <expression> "=<:" <type-name>
<cast-down>                 ::= <expression> "=:>" <type-name>
<can-cast-up>               ::= <expression> "<:"  <type-name>
<can-cast-down>             ::= <expression> ":>"  <type-name>

<objects>                   ::= <value> | <field-accessor>
<value>                     ::= <type-name> "(" <arguments> ")"
<field-accessor>            ::= <expression> "[" <integer> "]"
```

(a) Grammar for the Core Intermediate Constraint Language.

```
<eicl-expression>        ::= <cicl-expression> | <matching>
                           | <not-equals> | <cast> | <can-cast>
<expression>             ::= <eicl-expression>

<matching>               ::= <match> | <no-match>
<pattern>                ::= <type-name> "(" <arguments> ")"
<match>                  ::= <expression> "?" <pattern>
<no-match>               ::= <expression> "!?" <pattern>

<not-equals>             ::= <expression> "!=" <expression>
<cast>                   ::= <expression> ":"  <type-name>
<can-cast>               ::= <expression> "=:" <type-name>
```

(b) Grammar for the additional constructs in the Extended Intermediate Constraint Language.

Figure 5.3: Backus-Naur Form grammar for the Intermediate Constraint Language.

We define transformation rules for transforming the EICL to CICL in figure 5.4. Here the function signature `desugar :: EExp -> CExp` indicates that the `desugar` function takes as input an expression from the extended language `EExp` and returns an expression in the core language `CExp`. We have left out the desugaring of the pattern matching operator `?` due to the complexity of representing it without comprimising the comprehensivity of desugaring other language constructs. The actual strategy used for desugaring this pattern matching is similar to a naive interpretation strategy of pattern matching. This strategy (tries to) cast the expression to the type being matched, after which the same strategy is applied recursively to all fields on the matched type. Along the way, any encountered name bindings are inserted into the current scope.

```
desugar              ::  EExp -> CExp
desugar (e1 +  e2)    =  ((desugar e1) =:> Int)  +  ((desugar e2) =:> Int)
desugar (e1 -  e2)    =  ((desugar e1) =:> Int)  -  ((desugar e2) =:> Int)
desugar (e1 *  e2)    =  ((desugar e1) =:> Int)  *  ((desugar e2) =:> Int)
desugar (e1 /  e2)    =  ((desugar e1) =:> Int)  /  ((desugar e2) =:> Int)
desugar (e1 <  e2)    =  ((desugar e1) =:> Int)  <  ((desugar e2) =:> Int)
desugar (e1 <= e2)    =  ((desugar e1) =:> Int)  <= ((desugar e2) =:> Int)
desugar (e1 >  e2)    =  ((desugar e1) =:> Int)  >  ((desugar e2) =:> Int)
desugar (e1 >= e2)    =  ((desugar e1) =:> Int)  >= ((desugar e2) =:> Int)
desugar (e1 && e2)    =  ((desugar e1) =:> Bool) && ((desugar e2) =:> Bool)
desugar (e1 || e2)    =  ((desugar e1) =:> Bool) || ((desugar e2) =:> Bool)
desugar (!e)          = !((desugar e)  =:> Bool)
desugar (-e)          = -((desugar e)  =:> Int)
desugar (e1 == e2)    =  ((desugar e1) =<: T)    == ((desugar e2) =<: T)
  where
    T1 = (typeOf e1)
    T2 = (typeOf e2)
    T  = (nearestCommonAncestor T1 T2)
desugar (e1 != e2)    = !(((desugar e1) =<: T)   == ((desugar e2) =<: T))
  where
    T1 = (typeOf e1)
    T2 = (typeOf e2)
    T  = (nearestCommonAncestor T1 T2)
desugar (e  ?  p)     = ...
desugar (e  !? p)     = !((desugar (e ? p)))
desugar e             = e
```

Figure 5.4: Transformation rules to transform expressions of the EICL to equivalent expressions in the CICL.

As an example, let us take a look at the constraint `e1 == (e2 + 12)` in our EICL. The first desugaring rule we encounter is the rule regarding the `==` operator. Here we first have to find the types of both operands `e1` and `(e2 + 12)`. Since we have no information about the type of `e1`, we assign it type `Any`. For `(e2 + 12)` on the other hand, we can observe that the `+` operator must indicate an `Int` type. We then find the nearest common ancestor for both `Any` and `Int`. Since we know that `Int <| AnyVal <| Any`, we find that the nearest common ancestor is `Any`. Thus we find the result of this desugaring step:

```
((desugar e1) =:> Any) == ((desugar (e2 + 12)) =:> Any)
```

First solving (desugar e1) results in e1, since no desugaring rules can be applied. Now solving (desugar (e2 + 12)) we can apply the desugaring rule for the + operator. We find that both sides of an addition are required to be an integer. Thus the result will be:

```
((desugar e2) =:> Int) + ((desugar 12) =:> Int)
```

For both remaining desugaring operations we can not find any desugaring rules, thus bringing everything back together gives us the following final CICL constraint:

```
(e1 =:> Any) == ((e2 =:> Int) + (12 =:> Int) =:> Any)
```

To make the semantics complete, we also have to define a (possibly empty) set of implicit constraints to each expression in the CICL. These implicit constraints serve to ensure the correctness of the types in the expressions. These implicit constraints will also need to be solved by the SMT solver whenever a constraint that includes this expression needs to hold. Unfortunately this means we can not simply find all the implicit constraints in our expression and add them to our set of constraints, due to the possible inclusion of the disjunction connective (||). A disjunction expression holds when the constraint and its implicit constraints of one branch holds. This means that it is possible that in the one of the branches the implicit constraints do not hold. Instead, we must *hoist* these implicit constraints to the nearest top-level logical connective constraint (&& or ||). Pseudocode for this hoisting operation is provided in figure 5.5. Here the getImplicits function finds the implicit constraints for expressions. Pseudocode for this function is given in figure 5.6.

```
hoist           :: CExp -> CExp
hoist (e1 && e2) =  ((hoist e1) && (getImplicits e1)) && ((hoist e2)  && (getImplicits e2))
hoist (e1 || e2) =  ((hoist e1) && (getImplicits e1)) || ((hoist e2)  && (getImplicits e2))
hoist (!e)       = !((hoist e)  && (getImplicits e))
hoist (e1 == e2) =  ((hoist e1) == (hoist e2))  &&  (getImplicits e1) &&  (getImplicits e2)
hoist (e1 >  e2) =  ((hoist e1) >  (hoist e2))  &&  (getImplicits e1) &&  (getImplicits e2)
hoist (e1 >= e2) =  ((hoist e1) >= (hoist e2))  &&  (getImplicits e1) &&  (getImplicits e2)
hoist (e1 <  e2) =  ((hoist e1) <  (hoist e2))  &&  (getImplicits e1) &&  (getImplicits e2)
hoist (e1 <= e2) =  ((hoist e1) <= (hoist e2))  &&  (getImplicits e1) &&  (getImplicits e2)
hoist e          =  e
```

Figure 5.5: Hoisting implicit constraints.

```
getImplicits            :: CExp -> CExp
getImplicits (e1 +  e2) = (getImplicits e1) && (getImplicits e2)
getImplicits (e1 -  e2) = (getImplicits e1) && (getImplicits e2)
getImplicits (e1 *  e2) = (getImplicits e1) && (getImplicits e2)
getImplicits (e1 /  e2) = (getImplicits e1) && (getImplicits e2)
getImplicits (-e)       = (getImplicits e)
getImplicits (e =<: t)  = (e <: t)          && (getImplicits e)
getImplicits (e =:> t)  = (e :> t)          && (getImplicits e)
getImplicits e          = true
```

Figure 5.6: Implicit constraint generation for constructs defined by figure 5.3a.

Using these functions we are now able to convert any constraint (boolean expression) written in the EICL into a constraint in the CICL. We can do this by applying the desugar and hoist functions sequentially as follows: (hoist (desugar e)), where e is a boolean expression in the EICL. We will apply this combination of functions on each constraint in our constraint sets before sending them to the SMT solver. In order to make things more clear, we will now take another look at the result obtained in our desugaring example:

```
(e1 =:> Any) == ((e2 =:> Int) + (12 =:> Int) =:> Any)
```

Applying the hoist function to this constraint results in the following constraint:

```
(e1 =:> Any) == ((e2 =:> Int) + (12 =:> Int) =:> Any) &&
(e1 :> Any && true) &&
(e2 :> Int && true) &&
(12 :> Int && true) &&
(((e2 =:> Int) + (12 =:> Int)) :> Any && true)
```

Note that all true constraints are redundant and the constraint could be simplified. Furthermore, We briefly describe this simplification step in chapter 6.5. The resulting set of constraints tells us a number of things, most notably: e1 has a value of e2 + 12, where e2 has to be a value of type Int. The result of the addition will also be of type Int. However, because we are not certain what the type of e1 is, both e1 and the result of the addition e2 + 12 are cast to their nearest common ancestral type Any before being compared for equivalence.

6 Symbolic Execution

Now that we have discussed the building blocks that are required for our analysis, we will explain how our analysis works exactly. Our analysis uses so-called *summaries* in order to represent (intermediate) results of the exploration phase of the symbolic execution. We will explain the intuition behind these summaries using an example. And we will provide a definition of the type of summaries that we have used for our analysis in chapter 6.1. Secondly, we will provide a detailed explanation of how we can combine these summaries in order to perform a more efficient form of symbolic execution in chapter 6.2. We will continue by explaining how we further reduce the number of explored execution paths by using early branch pruning in chapter 6.3. Furthermore, in chapter 6.4 we will refine our definition of summaries by introducing budget constraints and explain how they affect the paths taken by the symbolic execution. Finally, we give an overview of various other optimisation tactics that were applied on our analysis in chapter 6.5.

6.1 Summaries

There are multiple reasons why it is useful for us to be able to have a data structure that allows us to represent a (partially) explored path during and after performing the symbolic execution. The most obvious reason is to simply have a set of constraints that can be solved by an SMT solver in order to find a set of inputs that would satisfy all these constraints. These inputs can then be used in concrete execution to concretely execute the same path through the program. However, a more interesting reason is that it allows us to use these results for other purposes, such as further optimisations as described in chapter 6.5, or reusing intermediate results in a *dynamic programming* [36] approach in order to obtain more complicated results as described in chapter 6.2.

Existing literature [5, 7, 8] uses summaries to encode these (intermediate) results. Originally these summaries were used to represent a collection of pre- and their respective postconditions for a function [5]. Here, the preconditions are a set of constraints that must hold for the input values, while the postconditions are the constraints that will hold after executing the function. Later works defined these summaries to only represent a single path through a (partial) function, rather than all possible paths [7, 8]. We follow an approach similar to the latter definitions, where we let a summary specify the pre- and postconditions of a single path through a complete function. Since we are dealing with pure functions, meaning that calling a function can only impact the return value, our postcondition will always be the equivalence of the return value to some expressions, therefore we have decided to only keep track of the return value, rather than a set of postconditions. We will start by defining our summaries as a pair of the preconditions and the return value: $\langle \rho_{\mathrm{pre}}, \mathrm{R} \rangle$, where ρ_{pre} represents the preconditions, a set of constraints written in our EICL that we defined earlier in chapter 5, and R represents the return value, also an expression in our EICL. We will further refine this definition in chapters 6.2 and 6.4. If we now revisit our example in figure 2.5, we can rewrite the found input sets as summaries in the following way:

- $\langle \{\mathrm{n} < 0\}, -\mathrm{n} \rangle$
- $\langle \{!(\mathrm{n} < 0)\}, \mathrm{n} \rangle$

However, in order to programmatically derive these summaries, we need to specify some extra procedures. As a basis for finding these summaries, we use the annotated CFGs that were described in chapter 4. The annotated CFG of the `abs` function is given in figure 6.1. Here we can see that there exist two paths through the function and therefore we expect to find two different summaries. We start our exploration with the set of constraints that contain the constraints that each input can be cast to the type specified by the signature. Since our type signature is

`Int -> Int` and our parameter name is `n`, we start with the set $\{n : \text{Int}\}$. Since each node in the graph represents an expression, we can convert each node that we walk through into a constraint which says something about the expression. If we do this for the three nodes prior to the branching point, we obtain the constraint set consisting of:
$\{n : \text{Int},\ e_0 == (e_1 < e_2),\ e_1 == n,\ e_2 == 0\}$.

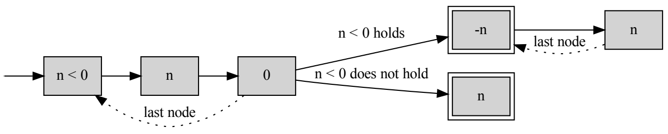


Figure 6.1: Annotated CFG for the `abs` function given in figure 2.5.

When we reach the branching point in our program, we fork our exploration and continue exploring both branches with distinct constraint sets. Both of these branch explorations initially start with the constraint set that was obtained thus far. Since the condition `n < 0` corresponds to e_0 in our constraint set, we can simply include e_0 as a constraint in the constraint set of the *then* branch and $!e_0$ in the constraint set of the *else* branch. When we continue exploring the *then* branch, we can convert the remaining two nodes into the following constraints: $e_3 == -e_4$ and $e_4 == n$. We also find that in this branch, e_3 is marked as the return value. Similarly, when we continue along the *else* branch, we convert the remaining node into the constraint: $e_5 == n$ and find that e_5 is the returned expression. Thus we have found the following two summaries:

- $\langle \{n : \text{Int},\ e_0 == (e_1 < e_2),\ e_1 == n,\ e_2 == 0,\ e_0,\ e_3 == -e_4,\ e_4 == n\},\ e_3 \rangle$
- $\langle \{n : \text{Int},\ e_0 == (e_1 < e_2),\ e_1 == n,\ e_2 == 0\ !e_0,\ e_5 == n\},\ e_5 \rangle$

Applying a simple unification algorithm to these summaries results in the same summaries that we have found by simply rewriting the input sets. Note that the $n : \text{Int}$ is redundant, due to the implicit constraints introduced by the $n < 0$ constraint.

We can apply the same strategy to more complicated programs as well. If we do the same with the example in figure 2.6, whose CFG is given by figure 4.1c, we find the following summaries:

- $\langle \{xs : \text{List},\ e_0 == xs,\ e_0\ ?\ (\text{Cons}\_\text{Nil}),\ e_1 == 1\},\ e_1 \rangle$
- $\langle \{xs : \text{List},\ e_0 == xs,\ e_0\ !?\ (\text{Cons}\_\text{Nil}),\ e_0\ ?\ \_,\ e_2 == xs,\ e_2\ ?\ (\text{Cons}\_t),$
  $e_3 == e_4 + e_5,\ e_4 == 1,\ e_5 == (\text{len}\ e_6),\ e_6 == t\},\ e_3 \rangle$
- $\langle \{xs : \text{List},\ e_0 == xs,\ e_0\ !?\ (\text{Cons}\_\text{Nil}),\ e_0\ ?\ \_,\ e_2 == xs,\ e_2\ !?\ (\text{Cons}\_t),$
  $e_2\ ?\ \_,\ e_7 == \text{"nomatch"}\},\ e_7 \rangle$

Or the equivalent summaries after applying unification:

- $\langle \{xs : \text{List},\ xs\ ?\ (\text{Cons}\_\text{Nil})\},\ 1 \rangle$
- $\langle \{xs : \text{List},\ xs\ !?\ (\text{Cons}\_\text{Nil}),\ xs\ ?\ (\text{Cons}\_t)\},\ 1 + (\text{len}\ t) \rangle$
- $\langle \{xs : \text{List},\ xs\ !?\ (\text{Cons}\_\text{Nil}),\ xs\ !?\ (\text{Cons}\_t)\},\ \text{"no match"} \rangle$

We can make two observations: the second summary contains the unexpanded function call $(\text{len}\ e_6)$ and the third summary does not actually return anything, but instead, it throws an exception. These cases will require special treatment before sending their constraints to the constraint solver. From this point on, we will refer to summaries that contain unexpanded function calls as *partial summaries* and we will refer to summaries that do not have any unexpanded function calls as *complete summaries*. Thus in the previous example the first and last summaries are complete summaries, while the second summary is a partial summary. For our test suite generation, we will only consider complete summaries. We will further explain how we transform partial summaries into complete summaries in the next section.

6.2 Compositional Analysis

As mentioned in the previous section, our found summaries might not be complete. This means that we have constraints in the summary that have some form of reliance on one or more function calls that are not expanded. This does not pose a direct problem if the result of this function call only influences the return value of the function, but it does pose a problem if the outcome of this call affects the execution path, either directly or indirectly.

An example program where this problem can be seen is shown in figure 2.2. Here there are three such cases: two recursive calls to `interp` when matching on an `Add` constructor and one recursive call to `interp` when matching on the `Apply` constructor. We find that the partial summary for the path that successfully performs an addition looks similar to:

- $\langle\{..., e_{x_1} == (\text{interp}\ e_{y_1}\ e_{z_1}),\ e_{x_2} == (\text{interp}\ e_{y_2}\ e_{z_2}),\ ...\},\ e_r\rangle$

For brevity we have omitted details that are not essential to our explanation. Here e_{y_1} and e_{y_2} would be unified with `e1` and `e2` respectively and both e_{z_1} and e_{z_2} would be unified with `nv`. Since we have chosen the path which successfully performs the addition, we also know that both e_{x_1} and e_{x_2} should be some value of the `NumV` constructor. Furthermore, e_r is an expression which indirectly depends on the integer values stored in those `NumV` constructors. If we take the same approach as we initially took in chapter 2.2 and simply ignore these constraints, there will no longer be any constraint that relates the input parameters to e_{x_1}, e_{x_2} and therefore also e_r. As a consequence, when we use the constraint set to generate input values, it is entirely possible that the SMT solver will decide on input values that do not cause the recursive calls to return values of type `NumV`, thus causing a different execution path to be taken.

It is clear that this problem needs to be addressed. The most naive solution would be to continue the exploration inside the called function whenever a function call is encountered. Let p_{u_1} and p_{v_1} represent the two parameters of the first recursive call and let r_1 represent the return value. When we continue the exploration of this recursive call, we insert the constraints that say the parameters are equal to the provided arguments: $p_{u_1} == e_{y_1}$ and $p_{v_1} == e_{z_1}$. As the exploration continues in the called function, the exploration may fork multiple times as well. When each fork of the call finishes exploring, we insert another constraint which says that the return value of the fork is equal to the expression that represented the call: $e_{x_1} == r_1$. This approach would solve the aforementioned problem. However, this solution poses two new issues:

The first problem is this exploration might go on indefinitely when a function is called recursively. In order to prevent this, we need some form of stopping condition which ends the exploration when some condition has been met. For now, we will ignore this problem. How our implementation deals with this problem is described in chapter 6.4.

The second issue is that the number of paths which we need to explore will grow very rapidly. This problem is also known as the path explosion problem [4]. We can observe that we are able to make seven summaries for the different implementations of `interp` in figure 2.2. One of these summaries, more specifically the summary which does not match any of the input expressions, is infeasible since the all possible patterns of the `Exp` type are covered. Furthermore we have two complete and four partial summaries. If we were to only perform this further exploration of function calls in the addition case, each function call would further split the summary into seven summaries. Since the calls happen sequentially, this branching factor is multiplied, since there can be 7×7 different combinations of paths taken in the called function. Out of these 49 summaries, only 9 of them are complete, while 40 of them are still partial. In general we can observe that expanding these function calls results in a *cartesian product*-like growth of potential paths through the program. When we take this another step further we end up with 961 summaries, out of which the vast majority is again partial. It is clear that this exponential growth is problematic for the feasibility of using this approach.

However, it is possible to partially resolve this issue. We can consider exploring these function calls as solving a sub-problem of solving our main problem. Furthermore, we can observe that the exploration of each of these recursive calls is mostly identical. The seven paths found while exploring each of these calls are the same, the only differences are the constraints that relate the expressions of the calling function to those of the callees. Thus if we are able to reuse the solutions found for these sub-problems, we can apply a dynamic programming approach [36] in order to solve the main problem more efficiently. Various studies [5–9] use earlier obtained summaries as solutions to these sub-problems and recombine, or "compose" them in order to find solutions for the main problem. We have taken the same approach in our implementation.

To make things more concrete, we will once again consider our interpreter programs. Considering we are interested in finding complete summaries, we will start with our three complete summaries of the `interp` function as our base case. We then iteratively expand this set by considering the partial summaries. For each function call, instead of exploring the function being called again, we simply reuse the set of complete summaries that we have previously obtained for that function. It must be noted that this does not necessarily decrease the total number of paths that we are considering in our analysis, but rather obtains the same collection of paths in a more efficient manner. For the addition case that we looked at earlier, this effectively means that each recursive call expands into the three complete summaries that we started with. Thus our set of complete summaries, now contains twelve summaries: the original three summaries and the cartesian product of those three paths in the addition case. We can now further enlarge this set of summaries by repeating this process for the same and the other partial summaries until we have reached the entry point of the execution or reached our stopping condition.

The example we just explained performs this composition in a *bottom-up* [36] fashion, where we start with the smallest sub-problems, which are the complete summaries, and then find solutions for the partial summaries exclusively using the solutions we found earlier. However, as described by [5], this bottom-up approach is not ideal, because it builds summaries for functions for all potential function arguments. The reality, however, is that not all of those arguments can actually occur in functions if we start executing the program from a fixed entry point. This approach can therefore lead to the computation of summaries that will be discarded later on in the symbolic execution, effectively "wasting" computation time.

A solution to this issue is a *top-down with memoization* approach [36], which is also referred to as a *demand-driven top-down* approach by [5]. Instead of starting at the complete summaries, we now start at the function which serves as the entry point of the program and work our way through functions when they are being called, much akin to our initial naive approach. Within the context of performing the compositional step in our symbolic execution we will refer to a combination of a function ϕ and the progress towards a stopping condition when that function was called ψ as the *calling context* $\langle\phi, \psi\rangle$. Thus we can say that we start our analysis with the calling context consisting of the entry function and the initial progress. For example, if the depth of the *call stack* is used as a stopping condition, the initial progress might be `0`. During our analysis we maintain a *summary cache* which holds the set of previously obtained summaries for different calling contexts.

We start our analysis by finding the initial sets of partial and complete summaries for each function in the program that is being symbolically executed. We then continue with our compositioning by expanding the calling context of our entry function with our initial progress. Whenever we expand a calling context, we consider all summaries that we have obtained in our initial step. All of those summaries that are complete and do not satisfy the stopping condition can be returned immediately. However, all function calls in the partial summaries need to be expanded first. For each function call in the partial summaries, we define the calling context using the function that is being called and the current progress towards the stopping condition. We check if the calling context exists in the summary cache. If it does, we simply retrieve the set of summaries from the cache and compose them in the same way as we described in the bottom-up approach, discarding those that satisfy the stopping condition. If the calling context does not have an entry in the cache, we first continue the expansion for that calling context and await its results before performing the same composition with its returned summaries. Whenever we are done expanding a calling context, we store the resulting set of summaries in the summary cache.

To make things more concrete again, we will consider the same example. As our stopping condition we will use a call stack depth greater than 1. We start with our initial calling context: $\langle$interp, 0$\rangle$. Since our cache is empty, we have to start expanding this calling context. We find that the `interp` function has seven summaries, as mentioned earlier. We will once again focus on the addition case summary. This summary has two recursive calls. Firstly, we will consider the first call and establish its calling context: $\langle$interp, 1$\rangle$. We check our cache and find it has no entry for this calling context, thus we will continue by expanding this calling context first. We once again find the seven summaries. However, we now have additional information: we know that any partial summary will satisfy the stopping condition, since it would increase the call stack depth above 1. Thus, we do not have to consider expanding the function calls in the partial summaries. Instead, we can simply return the three complete summaries. Now that the expansion of $\langle$interp, 1$\rangle$ has finished, we can add an entry with its results to the cache. If we then continue in the original expansion of $\langle$interp, 0$\rangle$, we look at the second function call and find the same calling context as for the first function call: $\langle$interp, 1$\rangle$. When we try to find this calling context in the cache the second time, we do find its summaries, thus we can simply use those directly. We then proceed by combining the constraint sets and relating the arguments and return values as described in the bottom-up example.

Now that we have explained how we can use composition to obtain the same summaries that we would be able to find in the naive approach with normal execution flow, these is still one unresolved problem that remains: *exceptions*. Exceptions are usually raised in order to indicate some abnormality in the execution, such as invalid arguments being provided in a function call [37]. Examples of this can be seen in our `interp` functions. When the input program attempts to add two values which are not both numbers, an exception (or an `error` in the Haskell code) is raised. When an exception is raised, it causes the control flow to abruptly stop and return to the nearest location where this exception is handled [37], or, if left unhandled, cause the program to crash. In our initial naive approach, dealing with this would have been easy, we can simply stop the exploration and start tracing back to an exception handler whenever we encounter a point in the program which raises an exception. However, in our top-down demand-driven approach we can not directly apply the same strategy. The main issue occurs during composition of a function call that can raise an exception. Without any special care, we might incorrectly assign the exception value to the expression corresponding to the function call and even worse, we might continue our symbolic execution after a function call raises an exception that is left unhandled by the current function.

In order to solve these issues, we must further refine our definitions of summaries. Rather than storing a set of constraints ρ_{pre}, we store an ordered sequence σ_{pre} of CFG nodes and its imposing constraints **and** the constraints imposed by its incoming edge $\langle \nu, \rho_{\text{node}} \rangle$, where ν represents the CFG node and ρ_{node} represents the constraint set of the node and its incoming edge. Furthermore, we add an additional value ε to our summaries to indicate whether or not the summary raises an unhandled exception. The resulting summaries can be written as $\langle \sigma_{\text{pre}}, \text{R}, \varepsilon \rangle$.

We can now alter our compositioning to instead of taking the union of both constraint sets, insert its constraint sequence of the callee into the constraint sequence of the caller at the right location. When an unhandled exception was raised in the callee and it is not handled by the caller, we change the return value to be equal to the return value of the callee that raised the exception, furthermore we drop the remainder of the sequence of the caller that appeared after the function call and set the ε flag to true. This results in the symbolic execution being properly terminated when it encounters an unhandled exception and the exception being propagated to any other function that might call the current function. However, when the current function does provide logic to handle the exception, we instead have to insert a constraint which says that the exception value is equal to the expression that binds the exception in the exception handler. In this situation, we do not have to drop the remainder of the sequence either, because the normal execution flow will continue.

6.3 Early Branch Pruning

We can further reduce the number of explored paths significantly based on two observations: The first observation is that many execution paths we find are in reality impossible to reach. The second observation is that adding additional constraints to a constraint set that is already unsatisfiable, can never make it satisfiable.

An example of the former can be found in our interpreter examples in figure 2.2. In the previous section we have established that initially we can find 7 different summaries for the `interp` function. The implicit execution path corresponding to the summary which raises an exception because the given expression does not match any of the patterns in the function can never actually be executed, because all of the possible patterns of the `Exp` type are covered by the function. This, as a consequence, implies that the corresponding constraint set of that summary is actually unsatisfiable. This is, however, not the only example of an occurrence of an impossible path in this program. Another example can be found in successful case of interpreting the `Add` expression. When we symbolically execute this path, we assume that the results of the recursive calls were some object of type `NumV`. However, due to the way we previously described our composition, it is entirely possible that the actual result type is something completely different, such as `Closure`.

Given the second observation, we can also conclude that any summary containing a sub-path with an unsatisfiable set of constraints, will also have an unsatisfiable set of constraints. Thus we can safely stop exploring and discard the current summary whenever the set of constraints has become unsatisfiable, without losing any satisfiable summaries in our final exploration results. Doing so can drastically reduce the number of summaries that require solving in the end. As an example, we can consider the *Add* case in the interpreters again. Previously, we determined that there are 7×7 or 49 possible unique combinations of summaries for this case, without fully expanding all partial summaries. Since we know that only 6 of those summaries are satisfiable, we now only have to consider 6×6 or 36 different combinations.

However, checking whether or not a set of constraints is satisfiable, requires an expensive call to the SMT solver. Thus, we should limit the places where we check this satisfiability. Since the primary purpose of pruning unsatisfiable summaries is to reduce the branching factor of the symbolic execution, it is natural that we perform this satisfiability check right after each branching point in the function. Furthermore, since the compositioning of various summaries in a function call can also introduce unsatisfiable summaries, we also perform this satisfiability check after expanding function calls. Whenever the satisfiability check fails, we simply discard the summary.

This idea of pruning branches as early as possible can also be found in [38]. There each branching condition is checked whether or not it is always true, always false or can be either along the current execution path. Based on this result, the *then* or *else* branch can be disregarded for further exploration.

6.4 Budget Constraints

As we have hinted at before, without a stopping condition symbolic execution would continue indefinitely in many real world examples. Thus it is necessary to define a stopping condition. A very simple example of such a stopping condition is setting a time limit for the symbolic execution [4]. This allows the user to make sure that the symbolic execution does not take longer than the provided time. However, it is also entirely possible that this strategy does not yield the intended results, thus different stopping conditions can be considered as well. We previously used a different, yet still simple stopping condition when explaining our compositional analysis in chapter 6.2. There we used the call stack depth as a stopping condition. This ensured that we explored all paths that only made a maximum number of nested function calls. However, as a trade-off, this did not put a limit on the maximum amount of time spent on the symbolic execution.

Furthermore, while we previously showed how we can reduce the amount of necessary computations by reusing summaries via composition, we have not reduced the total number of considered paths. While this might be a desirable trait, it is not purely beneficial. Having a large number of summaries still comes at a high cost in terms of execution time and memory usage when trying to generate input values and storing these summaries in memory. Thus reducing this number of summaries in order to improve feasibility, at the cost of potentially missing some bugs, can be beneficial.

When we reduce the number of summaries, we want to minimise the potential loss in error-detecting capabilities. Thus we wish to only eliminate the summaries which are the least likely to detect any errors. Doing this requires some insight regarding the problems that the students are solving with their implementations. An example of this can be observed in the interpreter examples in figure 2.2. Here the interpreting of `Add` expressions leads to the largest number of execution branches, while sufficiently testing the correctness of the implementation is not likely to require many different test cases. On the other hand, properly interpreting `Apply` expressions are far more likely to be implemented incorrectly and require far more different test cases in order guarantee the correctness of the implementation. However, simply using the call stack depth as a stopping condition, will result in the vast majority of generated test cases being some form of nested `Add` expressions.

Thus we propose the technique of using *budgeted symbolic execution* in order to eliminate "uninteresting" parts of a program from being thoroughly explored. For this, we will introduce a new stopping condition: the *budget*. We also introduce the concept of *budget constraints* which associate *costs* with performing certain operations in our symbolic execution. An example of such a budget constraint is the cost of calling functions. We can consider this budgeted symbolic execution approach as a generalisation of the previously used example of maximum call stack depth, where the budget is set to the maximum call stack depth and the costs associated with all function calls is equal to `1`.

However, we may also associate costs with performing other operations, such as matching patterns. Thus, if we go back to our example, it is now possible to associate a cost larger than zero with matching an `Add` expression. This would mean that matching `Add` expressions depletes the budget more rapidly than matching other expressions. Thus whenever an `Add` expression is matched, the exploration will terminate earlier, and consequently will result in a smaller number of summaries that follow execution paths through the interpretation of (nested) `Add` expressions. Furthermore, it might also be beneficial to associate an extra cost with matching on `Num` expressions, in order to reduce the number of simple summaries that do not mix many different expression types instead of nesting more interesting different types of expressions.

Additionally, we may also associate different costs for calling specific functions. An example of utilising this could be to decrease the cost of calling the `find` function. Doing so would allow for larger environments to be considered while searching for variables, while not allowing the interpreter to consider deeper nested expressions. This allows for more fine-grained control over the way that the symbolic execution will explore the presented programs.

6.5 Further Optimisations

Besides the aforementioned improvements to the feasibility of our approach, we have also implemented a number of linear improvements. The most significant improvement here is *multi-threading* of as many parallel operations as possible at any time. For example, during the compositioning of various summaries, we perform the budget constraints check and the satisfiability check of each of those resulting compositions in parallel.

Additionally we also perform an adaption of the *unification* algorithm proposed by [39] on the constraint sets before being sent to the SMT solver. Here all input expressions are considered as constants, in order to prevent them from completely disappearing during unification. This reduces the number of constraints which the SMT solver will need to solve, slightly improving its solving speed.

Furthermore, we also eliminate any trivial *tautologies* and tautologies caused by redundancy in order to further reduce the size of the constraints set. Examples of such a trivial tautologies are `A && A` and `A && true`, where `A` is any arbitrary constraint. Here the second `A` and the `true` do not add any new constraints that were not already implied by the first `A` and can thus be omitted. An example of a redundancy is `e :> Int && e :> Any`, where `e` is any arbitrary expression. Here, `e :> Any` does not add any new information that is not already implied by `e :> Int`. This is the case, because we know that `Int <: Any`, thus any value of type `Int`, must also be of type `Any`.

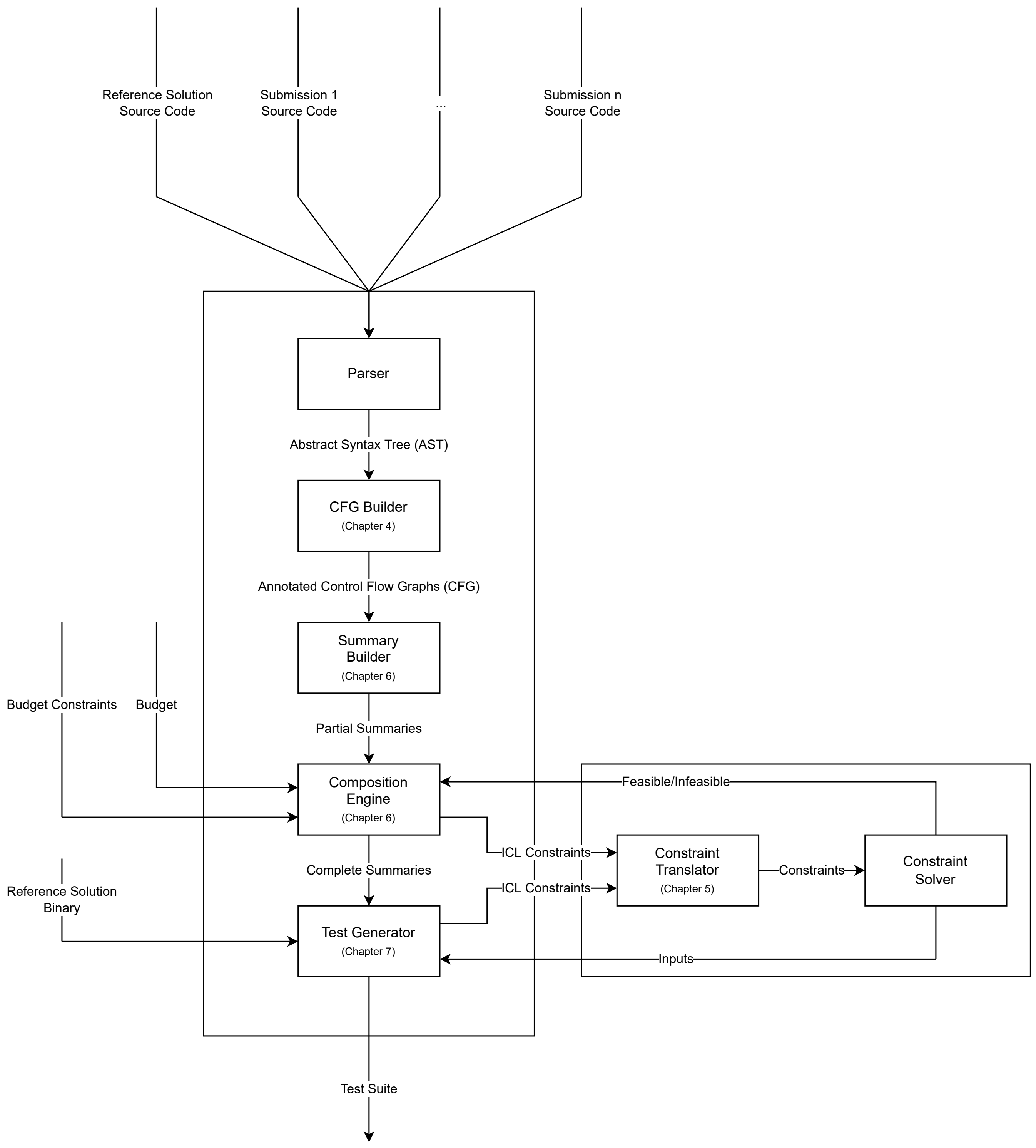
7 Test Suite Generation

Now that we have described all the necessary steps to perform our symbolic execution, we can put everything together and generate a test suite. This chapter describes how our framework generates a test suite for a reference solution and a set of potentially faulty programs.

First the user of our framework establishes a set of budget constraints, as described in chapter 6.4. This budget should steer the symbolic execution towards the more interesting cases. For each program, including the reference solution, we perform all the steps of our pipeline: We parse the source code of the program into its respective abstract syntax tree. Then, we convert the abstract syntax tree of each function in the program into a special control flow graph, as described in chapter 4. We proceed by using those control flow graphs to generate summaries for each function, as described in chapter 6.1, by using constraints defined in our intermediate constraint language, which is described in chapter 5. We then combine these summaries to obtain summaries that represent deeper execution paths in the program, as described in chapter 6.2. We keep performing this composition step until we can no longer create new summaries that do not violate the set budget constraints. Finally, for each of the obtained summaries that are complete, meaning they don't have any unexpanded function calls inside their path, we use the constraint solver to solve the path constraints. The constraint solver then provides us with inputs for each summary. Having obtained these inputs, we then feed these inputs into the reference solution to find out the expected output values.

Each of these combinations of inputs and their expected output form a test case [40]. Having found all test cases of each program, we can eliminate any duplicate test cases that have been found in order to reduce the total number of test cases. This results in a test suite which executes all feasible paths in any of the given programs, within the provided budget constraints, at least once. Finally, these test cases are then converted into source code, so that they can be used for testing or grading, similarly to the resulting test suites obtained in figures 2.7 and 2.8.

A diagrammatic representation of our full test suite generation pipeline is shown in figure 7.1. This diagram also shows the steps of our symbolic execution pipeline and their respective inputs and outputs.



Reference Solution Source Code
Submission 1 Source Code
...
Submission n Source Code
Parser
Abstract Syntax Tree (AST)
CFG Builder (Chapter 4)
Annotated Control Flow Graphs (CFG)
Summary Builder (Chapter 6)
Partial Summaries
Budget Constraints
Budget
Composition Engine (Chapter 6)
Feasible/Infeasible
ICL Constraints
ICL Constraints
Constraint Translator (Chapter 5)
Constraints
Constraint Solver
Complete Summaries
Reference Solution Binary
Test Generator (Chapter 7)
Inputs
Test Suite

Figure 7.1: A diagrammatic representation of our test suite generation pipeline. Each node indicates what chapter explains the step represented by the node. Each edge indicates the inputs and outputs used by the various steps in the process.

8 Evaluation

In order to evaluate the effectiveness of our symbolic execution technique, we applied the tool on a number of different data sets, consisting of different implementations of a programming problem. The metrics used for this evaluation are described in chapter 8.1. The software and hardware used to collect these metrics are described in chapter 8.2, alongside the description of the different data sets. For this evaluation we were interested in answering a number of research questions (RQ). These questions can be divided in two categories:

- **RQ1:** (*Quantitative*) How does the effectiveness of automatically generated test suites compare to hand written test suites?
  - **a)** Can the technique be applied within a realistic amount of time?
  - **b)** Can a higher branch coverage be achieved?
  - **c)** Can a higher mutation score be achieved?
  - **d)** Can more behavioural differences be found?
- **RQ2:** (*Qualitative*) Can automatically generated test suites detect a certain set of planted bugs?
  - **a)** Can the technique be applied within a realistic amount of time?
  - **b)** What budget constraints are necessary to find them?

The results of this evaluation are discussed in chapter 8.3. Finally we discuss the limitations of our symbolic execution technique in chapter 8.4.

8.1 Metrics

When evaluating our tool we are concerned with the following things: how effective the generated tests are at findings errors and whether we can run the tool in a realistic amount of time. We use a variety of metrics to evaluate the results obtained with our tool.

In order to evaluate the effectiveness of the generated test suite we looked at two classes of metrics: *branch coverage* and *mutation score*. Branch coverage is a form of *code coverage* which measures the percentage of *branches* in a program which have been executed by a test suite [13]. Branches are the places in a program where the control-flow of the program can conditionally divert. An example of such branching points are if-then-else expressions, which create two branches: one where the condition is true and one where the condition is false. However, numerous studies have shown that this metric is not a good indicator for the error-detection capabilities of a test suite [41, 42]. Therefore we strictly interpret this metric as an indication of how much of the program is covered by our generated tests.

A better metric for estimating the quality of a test suite is the *mutation score* obtained through *strong mutation testing*, as shown by various studies [43, 44]. Strong mutation testing involves inserting faults into the program and verifying if these faults cause a test in the test suite to fail [45]. A program which contains one of these introduced faults is called a *mutant*. If one of the test cases in a test suite fails when running the suite on a mutant, the mutant is said to have been *killed*, otherwise it *survived*. The mutation score is then computed by taking the percentage of killed mutants. These faults can be inserted automatically using so-called *mutation operators*. Mutation operators are transformation rules that are applied to the original program in order to automatically create a mutant [45]. An example of such a mutation operator would be a mutation operator that replaces additions ($+$) with subtractions ($-$). For each place in the program where such a mutation operator can be applied, it creates a new mutant, leaving all other parts of the program unchanged.

Furthermore, we measured the total amount of time spent on performing the analysis of each program in each data set, in order to get an impression of the feasibility of using this technique in practise. Because the time spent is partially bound by the hardware which is used to perform the analysis, we have also measured the number of inquiries to the SMT solver. This number is interesting to look at, because as mentioned earlier, these inquiries are relatively long operations, thus giving us a measurement that will be the same, regardless of the hardware used to perform the analysis.

8.2 Setup

In order to collect the aforementioned metrics, we have used various pieces of software. We have chosen to use version 1.8.0 of *Pitest* [46] as our mutation testing framework. Pitest is the industry standard mutation testing framework for the Java programming language. However, both Scala and Java are executed on the *Java Virtual Machine* and Pitest is therefore also able to perform analysis on Scala projects with some minor workarounds. This choice was made, because at the time of writing, Pitest is more mature and supports a larger number of mutation operators [47] than *Stryker4s* [48], the most popular mutation framework for Scala. The group of used mutation operators was configured to `ALL`, which applies all mutation operators that are available in the framework. Because the workaround to use Pitest for Scala code required a Java test suite project, we have also opted to use a Java library for measuring code coverage. More specifically, we have used version 0.8.8 of *JaCoCo* [49] to measure branch coverage. Furthermore, the specific Z3 version used for the constraint solving is 4.8.17. Finally, all evaluations are performed using Scala version 2.13.2 and the OpenJDK runtime version 1.8.0_322.

The system which was used for performing the evaluation has an AMD Ryzen 2700X processor with a clock speed of 3.7 GHz, 8 physical cores and 16 logical processors. The memory of the used system is 32 GB with a clock speed of 2800 MHz. The operating system of the system is Microsoft Windows 10.

Our evaluation cab be divided in two different parts: A quantitative part corresponding to RQ1 and a qualitative part corresponding to RQ2 . The first part of the evaluation (RQ1) uses real anonymised student submissions of a second year Computer Science course which introduces students to the concepts of functional programming. This set consists of roughly 550 submissions for two introductory assignments. For each of these submissions which only use the supported Scala syntax, we have generated a test suite using various budget constraints. Note that because of the simplicity of the submissions, we have primarily tweaked the total available budget and not the costs of matching certain patterns. We have collected the code coverage and mutation score metrics for different combinations of test suites: using only the test suite generated for the reference solution (*Reference*), using only the test suite generated for that specific submission (*Self*), using both the test suites for the reference implementation and the solution (*Ref + Self*) and using the combined test suite of using all submissions including the reference solution (*All*). A description of the assignments and the used reference implementations can be found in appendix A. We compared these results for each submission to the code coverage and mutation score that was obtained by the manually written tests that were used to grade students during the course, in order to compare the thoroughness of the generated test suites to the thoroughness of the manually written test cases.

The second part of our evaluation (RQ1) comprises of a qualitative evaluation where we have manually seeded common errors that we have observed students make into a reference solution. For this analysis, we have used a simplified version of a more complex assignment provided in a later stage of the same second year course. This assignment required students to implement a definitional interpreter for a simple functional programming language. This interpreter has to be able to deal with integers, additions, lambda functions binding a single variable and their applications. The interpreters have to be implemented using environments and closures, rather than substitution.

During the course we have observed that the largest struggle for most students is understanding when to use what environment in situations where there are multiple environments. Using the wrong environment in the wrong place, causes the scoping of variables to be incorrect. An example of such an error was previously given in figure 2.2.

Another observed obstacle for many students is properly dealing with recursive pattern matching. When interpreting expressions such as additions, it is paramount that both arguments of the addition are first interpreted, before being validated as being integers and added together. If this does not happen, but instead the arguments are validated immediately, this will not allow the nesting of addition inside additions.

Thus we have created a number of implementations in which we seeded various variations of the aforementioned errors. These implementations can be found in appendix B. For each implementation we have generated test suites using different budget constraints and observed whether or not the faults were discovered.

8.3 Results

The results of the evaluation on the student submission data sets are presented in figure 8.1 (corresponding to the assignment described in appendix A.1) and figure 8.2 (corresponding to appendix A.2).

The first table in both figures (figures 8.1a and 8.2a) gives an overview of the total number of student submissions for that assignment, how many of those submissions were written using only the allowed Scala subset. Furthermore the table also provides the number of submissions using the supported Scala subset for which at least one test failed in the manually written test suite.

The following pair of boxplots (figures 8.1b and 8.1c and figures 8.2b and 8.2c) show the distributions of the execution time and the number of Z3 inquiries of using the framework to generate a test suite for each individual submission in the test set that uses the supported language subset. These results provide an answer to RQ1a . We observe that both figures increase in a similar manner when the budget of the test suite generation is increased. We also observe that in figure 8.1b while the majority of submissions, between the first and third quartile, seem to increase their execution time almost linearly, there are a few submissions for which either their execution time increases much more rapidly or does not increase at all. The former can be explained due to some students potentially implementing solutions with a higher branching factor than the majority of other students did. A higher branching factor leads to a larger number of summary combinations that can occur in the composition stage, as described in chapter 6.2. This in turn leads to more computations, including inquiries to the constraint solver, and can also lead to a larger set of summaries being available for the next composition step. Which in turn leads to a more exponentially shaped increase of run time. The observation that the minimum execution time and minimum number of constraint solver inquiries stays constant can be explained by the fact that it is possible that students submitted (more than likely) incorrect code which does not call any other functions from the assignment functions. Because the budgets constraints are only configured with costs for making function calls, the initial budget does not matter for such submissions. Furthermore, in figure 8.2b we can see that for the second assignment, the relation between the execution time and the budget increases in an exponential fashion for all submissions. However, we can still observe that the minimum execution time remains constant.

The next pair of boxplots (figures 8.1d and 8.1e and figures 8.2d and 8.2e) represent the distributions of the branch coverage and mutation score for the various test suite configurations. These boxplots provide answers to RQ1b and RQ1c respectively. It must be noted that while the previous boxplots consist of data points of all supported submissions, these boxplots do not. This is due to the fact that Pitest and JaCoCo are unable to collect their metrics when a unit test in the test suite fails. Therefore, each boxplot only consists of the data points of the submissions that passed all unit tests in the test suite. Overall, we can observe that the branch coverage in each test suite is higher than the respective mutation scores. We do see that there is some correlation between the measured branch coverage and mutation score. However, we can also observe that while the branch coverage in figure 8.1d seems to not be significantly influenced by the budget, the mutation score in figure 8.1e does experience a noticeable increase going from a budget of one to a budget of two for the *Reference*, *Self* and *Ref + Self* test suites.

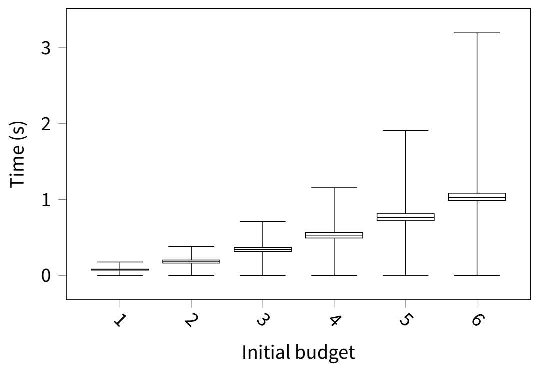
Furthermore, we can observe that there appears to be an upperbound for both the branch coverage and the mutation score, which is notably lower than a 100%. A possible explanation for this is the that it is possible that there are a number of branches and mutations that are unable to be covered under any circumstances. The number of these impossible to cover branches and mutations may be inflated due to the way that the Scala source code is converted into Java bytecode by the Scala compiler. The tools we used to obtain metrics are not aware of the underlying Scala source code and solely consider the Java bytecode. Examination has shown that there are instances where the Scala compiler emits a function call to an empty function without any return values, while one of the mutators used by PITest attempts to remove function calls without return values. This means that each of those empty function calls results in a mutation where that call is removed. However, due to the fact that the body of that function is empty, removing the call does not actually change the behaviour of the program, therefore this mutation will always remain undetectable.

The final table (figures 8.1f and 8.2f) in each figure gives an overview of comparing the measurements of individual submissions between the manually written tests and the generated test suites. This table provides an answer to RQ1d . The *Higher Branch Coverage* and *Lower Branch Coverage* columns respectively indicate the number of submissions for which a higher branch coverage and lower branch coverage was measured compared to the branch coverage measured for the manually written test suite. For the same reason as mentioned earlier, only the submissions which passed all tests in both the manually written test suite and the respective test suite are taken into consideration. Similarly, the *Higher Mutation Score* and *Lower Mutation Score* columns represent the the differences in measured mutation scores. Finally, the *More Differences Found* and *Fewer Differences Found* column compare the behavioural-difference finding capabilities of the various test suite configurations to the manually written test suite. Thus, a values larger than zero in the *More Differences Found* column indicate the number of submissions for which at least one unit test failed, while the submission passed all tests in the manually written test suite. Similarly, a value larger than zero in the *Fewer Differences Found* column indicate the number of submissions that failed at least one unit test in the manually written test suite, but passed all tests in the respective generated test suite.

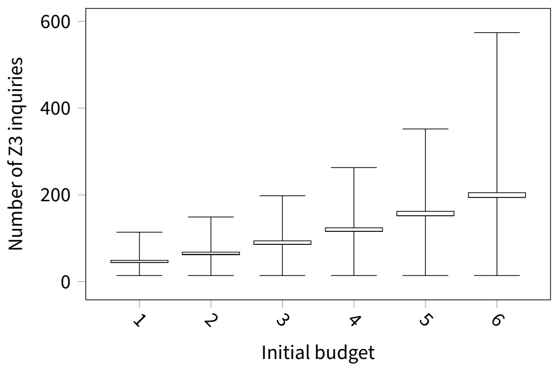
Overall, we can conclude that for these particular data sets, increasing the initial budget rapidly leads to diminishing returns. We do not expect that further increasing the budgets will significantly influence the measurements. Ideally, when looking for a test suite which is most accurate at checking behavioural equivalence, we would opt for using a configuration which has the highest number of improvements and the fewest number of deteriorations compared to the manually written test suite as shown in (figures 8.1f and 8.2f). However, one also has to take into consideration the availability of the input programs and the time required to generate such a test suite. It is quite clear overall that using the *Self* configuration by itself is not enough to guarantee that a submission is tested adequately enough, due to the very low minimum branch coverage and mutation score. We observe that *Reference* configuration performs a lot better than the *Self* configuration, however, as is visible in figure 8.1f, there are a few instance of incorrect submission that were caught by our manually written tests, but not with the test suites that were generated using our *Reference* configuration. On the other hand, we can observe that in figure 8.2f, for any budget larger than our generated test suite performs better than the manually written one. We found that using the *All* configuration with any budget has the best prospects. However, generating this test suite requires access to a large number of submissions and therefore also takes a longer time to generate, and might therefore be less suited for providing instantaneous feedback to students when they submit their code, for example using a platform such as described by [50]. This problem can be mitigated if a set of submissions is available beforehand, such as when a course has used the same assignments previously. If this instantaneous feedback is necessary and no data sets exists prior to the grading, we can observe that using the *Ref* + *Self* configuration with budget 2 either improves or obtains the same measurements as the manually written test suite for all submissions, while only taking a few seconds to generate. This is especially the case for the results shown in figure 8.2f, where the effectiveness of *Ref* + *Self* are near identical to those of *All*, while only having a small fraction of the required computation time. This makes this configuration more suitable for providing instantaneous feedback to students.

All submissions	561
Submissions in supported syntax	388
Behavioural differences detected by manually written tests	69

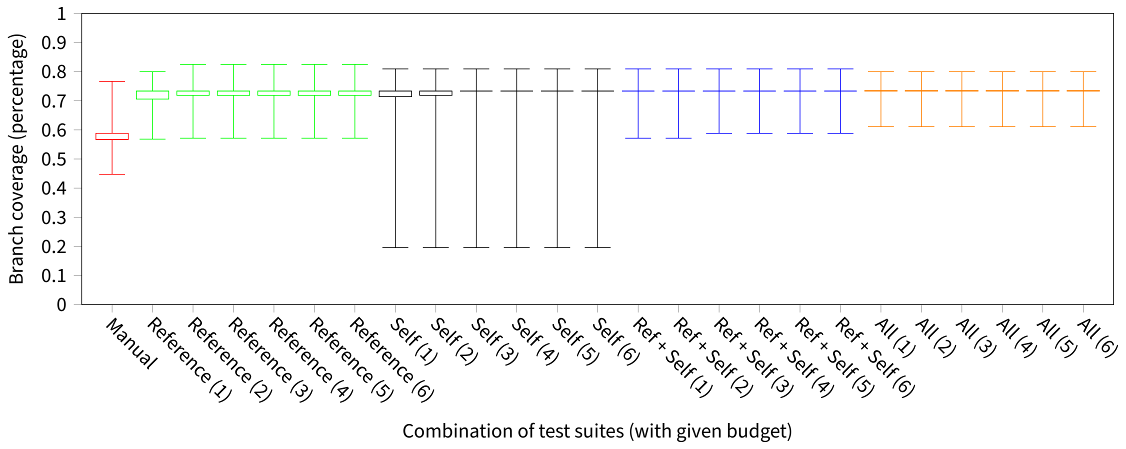
(a) Specifications of submission numbers.



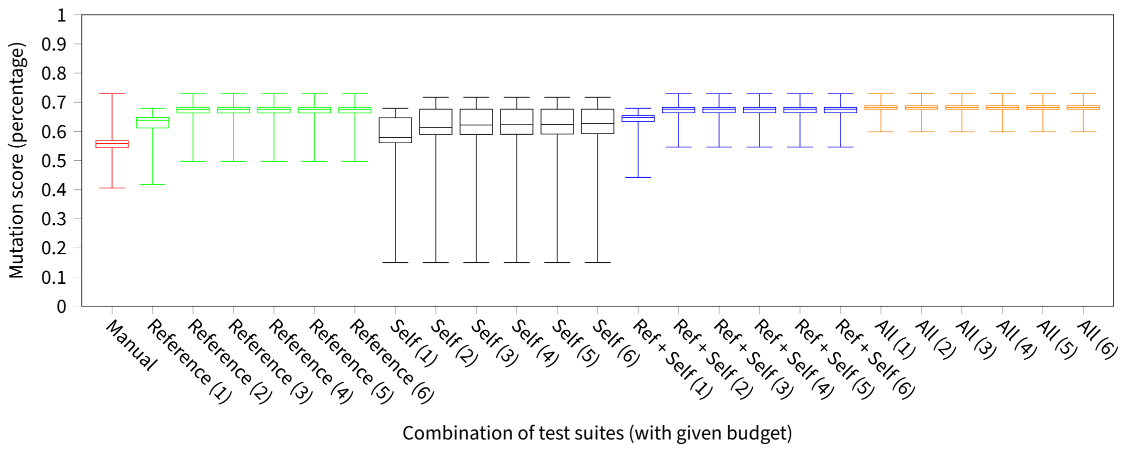
(b) Distributions of the time in seconds required to generate a test suite for a single solution for different budgets. (RQ1a)



(c) Distributions of the number of Z3 inquiries required to generate a test suite for a single solution for different budgets. (RQ1a)



(d) Distributions of the branch coverage of different test suites and different budgets in parenthesis. (RQ1b)



(e) Distributions of the mutation score of different test suites and different budgets in parenthesis. (RQ1c)

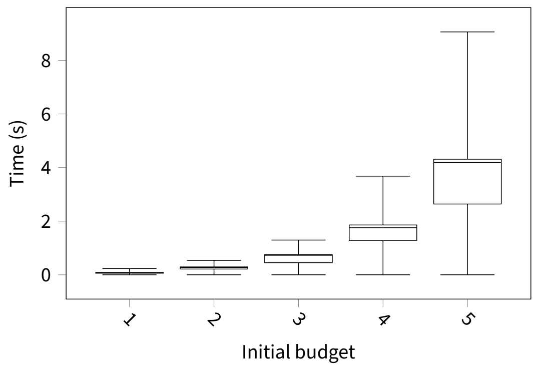
Test Suite	Budget	Higher Branch Coverage	Lower Branch Coverage	Higher Mutation Score	Lower Mutation Score	More Differences Found	Fewer Differences Found
Reference	1	295	4	257	4	16	1
	2	300	0	289	0	17	1
	3	300	0	289	0	17	1
	4	300	0	289	0	17	1
	5	300	0	289	0	17	1
	6	300	0	289	0	17	1
Self	1	255	23	204	23	39	28
	2	255	16	245	16	40	20
	3	262	14	247	14	40	19
	4	262	13	248	13	40	19
	5	264	13	249	13	40	19
	6	265	12	248	12	40	19
Ref + Self	1	275	2	264	2	40	1
	2	276	0	268	0	41	0
	3	277	0	268	0	41	0
	4	277	0	268	0	41	0
	5	277	0	268	0	41	0
	6	277	0	268	0	41	0
All	1	116	0	116	0	203	0
	2	116	0	116	0	203	0
	3	116	0	116	0	203	0
	4	116	0	116	0	203	0
	5	116	0	116	0	203	0
	6	116	0	116	0	203	0

(f) Comparison of branch coverage, mutation score and behavioural equivalence verdicts of different test suites compared to the manually written test suite. Each cell indicates the number of solutions for which the criteria apply. (RQ1d)

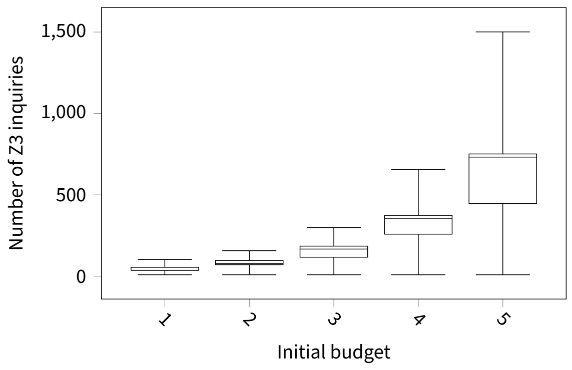
Figure 8.1: Evaluation results of the *Lists* data set.

All submissions	558
Submissions in supported syntax	479
Behavioural differences detected by manually written tests	58

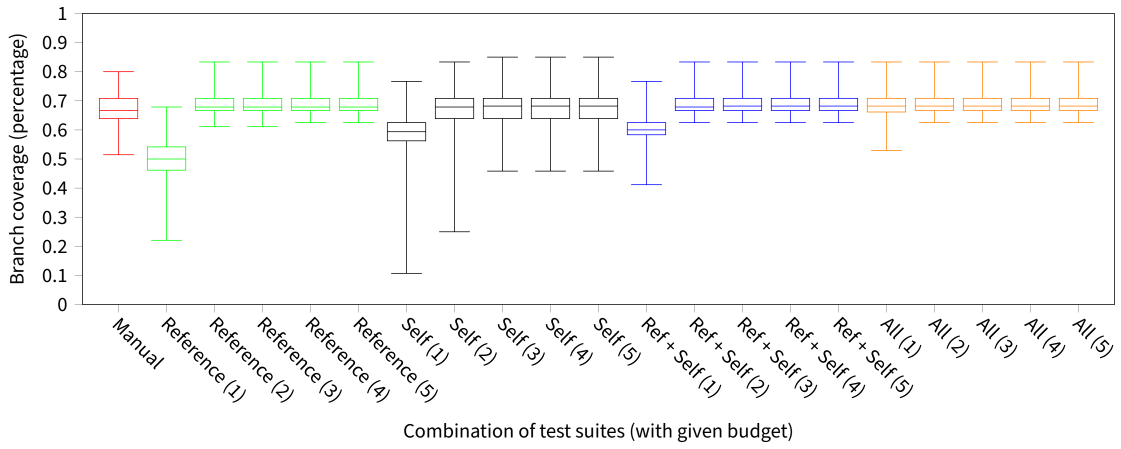
(a) Specifications of submission numbers.



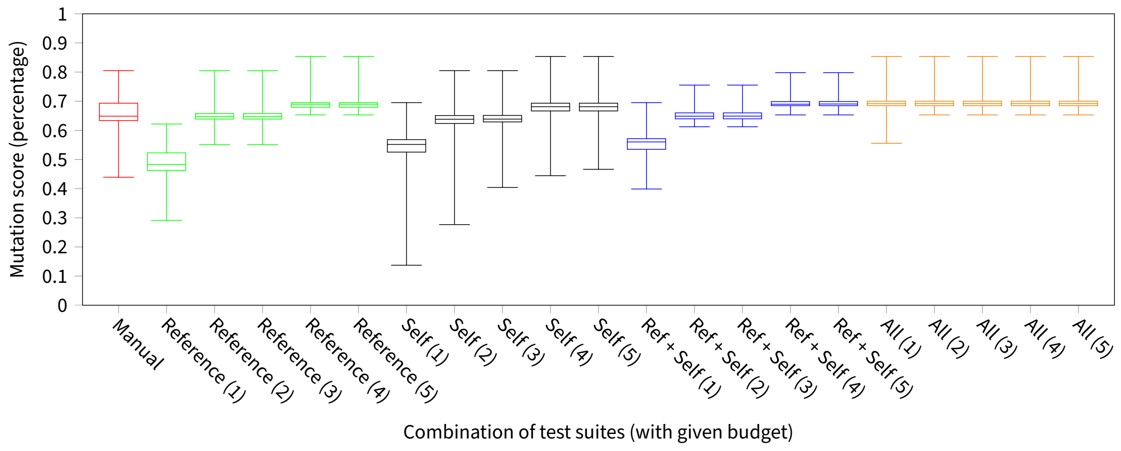
(b) Distributions of the time in seconds required to generate a test suite for a single solution for different budgets. (RQ1a)



(c) Distributions of the number of Z3 inquiries required to generate a test suite for a single solution for different budgets. (RQ1a)



(d) Distributions of the branch coverage of different test suites and different budgets in parenthesis. (RQ1b)



(e) Distributions of the mutation score of different test suites and different budgets in parenthesis. (RQ1c)

Test Suite	Budget	Higher Branch Coverage	Lower Branch Coverage	Higher Mutation Score	Lower Mutation Score	More Differences Found	Fewer Differences Found
Reference	1	0	422	0	422	0	6
	2	66	1	12	1	330	0
	3	66	1	12	1	330	0
	4	67	0	92	0	330	0
	5	68	0	92	0	330	0
Self	1	2	414	3	414	2	18
	2	63	35	46	35	7	15
	3	63	25	46	25	16	7
	4	65	23	277	23	16	7
	5	65	20	277	20	17	7
Ref + Self	1	2	413	5	413	3	6
	2	66	0	13	0	332	0
	3	61	0	13	0	341	0
	4	61	0	81	0	341	0
	5	61	0	80	0	342	0
All	1	60	0	79	0	343	0
	2	59	0	78	0	344	0
	3	59	0	78	0	344	0
	4	59	0	78	0	344	0
	5	59	0	78	0	344	0

(f) Comparison of branch coverage, mutation score and behavioural equivalence verdicts of different test suites compared to the manually written test suite. Each cell indicates the number of solutions for which the criteria apply. (RQ1d)

Figure 8.2: Evaluation results of the *Binary Search Tree* data set.

The results of our second evaluation using manually seeded errors are presented in figure 8.3. For this evaluation we have selected a number of different configurations in order to try and detect manually inserted bugs. We have given each configuration two hours to finish. Figure 8.3a provides an overview of the time it took to generate the respective test cases and the number of times the constraint solver was called. When test suite generation for a particular configuration was unable to complete within the allotted time, its entries in the table are marked as 'Timeout'. This table provides an answer to RQ2a . Additionally, we have provided similar tables that show the number of constraint solver inquiries and total number of generated test cases in figure C.1 and figure C.2 respectively. Furthermore, figure 8.3b gives an overview of which faults were detected by the test suites that were generated using the various budget constraints configurations and different combinations of test suites. When an error was detected, the cell in the table contains an 'X' marking. When an error was not detected, the mutation score is given in order to give an idea of how well the remainder of the program is covered. We have chosen to only consider the *Ref + Self* and *All* configurations, due to the fact that the previous data sets showed that the other combinations were significantly less effective overall. This table provides an answer to RQ2b .

	Reference B.2	Body Local B.3	Body Local + Closure B.4	Body Closure + Local B.5	Body Empty B.6	Argument Closure B.7	Argument Closure + Local B.8	Argument Local + Closure B.9	Argument Empty B.10	Structural Matching 1 B.11	Structural Matching 2 B.12	Extra Case B.13
Budget: 1 Call cost: 1	74	33	30	31	31	40	35	36	35	33	67	83
Budget: 2 Call cost: 1	214	149	134	138	127	145	139	136	165	113	237	183
Budget: 3 Call cost: 1	660	512	518	521	533	534	557	542	537	271	425	1050
Budget: 4 Call cost: 1	2108	1881	1757	1716	1895	1904	1698	1719	1856	889	3139	4246
Budget: 5 Call cost: 1	6406	6253	5362	5277	6220	6172	5268	5255	6396	2744	11424	15266
Budget: 6 Call cost: 1	20999	20313	15797	15659	20505	21143	15981	15745	20090	6878	28778	60182
Budget: 7 Call cost: 1	79595	82810	49817	49994	81680	81574	50515	50359	89462	21043	270314	971885
Budget: 8 Call cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 7 Call cost: 1 Match "AddC" cost: 1	19833	19781	12184	12163	20210	20461	12200	11929	19119	12320	16980	26307
Budget: 8 Call cost: 1 Match "AddC" cost: 1	82397	90600	33870	33527	87551	85864	34225	31997	81493	45333	65708	147801
Budget: 9 Call cost: 1 Match "AddC" cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 8 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	36362	38738	21865	22362	38092	37771	21228	20999	35165	17716	19180	48493
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	212624	291639	64227	63926	278336	249159	64137	77872	203051	79048	84659	384767
Budget: 10 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 2	135543	174796	55640	58534	148255	167681	59553	54408	110084	53077	49183	192182
Budget: 10 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 2	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout

(a) Execution times in milliseconds of different budget constraint configurations for different interpreter implementations. (RQ2a)

		Body Local B.3	Body Local + Closure B.4	Body Closure + Local B.5	Body Empty B.6	Argument Closure B.7	Argument Closure + Local B.8	Argument Local + Closure B.9	Argument Empty B.10	Structural Matching 1 B.11	Structural Matching 2 B.12	Extra Case B.13
Budget: 1 Call cost: 1	Ref + Self	0.28	0.22	0.22	0.27	0.27	0.22	0.22	0.27	0.44	0.69	X
	All	0.44	0.35	0.35	0.43	0.43	0.35	0.35	0.43	X	0.69	X
Budget: 2 Call cost: 1	Ref + Self	0.29	0.23	0.23	0.28	0.28	0.23	0.23	0.28	0.45	0.69	X
	All	0.45	0.36	0.36	0.44	0.44	0.36	0.36	0.44	X	0.69	X
Budget: 3 Call cost: 1	Ref + Self	0.45	0.36	0.36	0.44	0.44	0.36	0.36	0.44	0.45	0.69	X
	All	0.45	0.36	0.36	0.44	0.44	0.36	0.36	0.44	X	0.69	X
Budget: 4 Call cost: 1	Ref + Self	0.6	0.56	0.56	0.6	0.6	0.56	0.56	0.61	0.63	0.77	X
	All	0.6	0.56	0.56	0.6	0.6	0.56	0.56	0.61	X	0.77	X
Budget: 5 Call cost: 1	Ref + Self	0.76	0.69	0.69	0.77	0.77	0.69	0.69	0.77	X	0.85	X
	All	0.76	0.69	0.69	0.77	0.77	0.69	0.69	0.77	X	X	X
Budget: 6 Call cost: 1	Ref + Self	0.78	0.7	0.7	0.78	0.78	0.7	0.7	0.78	X	X	X
	All	0.78	0.7	0.7	0.78	0.78	0.7	0.7	0.78	X	X	X
Budget: 7 Call cost: 1	Ref + Self	X	X	X	X	X	X	X	X	X	X	X
	All	X	X	X	X	X	X	X	X	X	X	X
Budget: 7 Call cost: 1 Match "AddC" cost: 1	Ref + Self	X	X	X	X	X	X	X	X	X	X	X
	All	X	X	X	X	X	X	X	X	X	X	X
Budget: 8 Call cost: 1 Match "AddC" cost: 1	Ref + Self	X	X	X	X	X	X	X	X	X	X	X
	All	X	X	X	X	X	X	X	X	X	X	X
Budget: 8 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	Ref + Self	X	X	X	X	X	X	X	X	0.81	0.86	X
	All	X	X	X	X	X	X	X	X	X	0.86	X
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	Ref + Self	X	X	X	X	X	X	X	X	X	X	X
	All	X	X	X	X	X	X	X	X	X	X	X
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 2	Ref + Self	X	X	X	X	X	X	X	X	0.83	0.69	X
	All	X	X	X	X	X	X	X	X	X	0.69	X

(b) Mutation score of generated test suites of finding seeded errors in different faulty interpreters. An 'X' indicates that the inserted bug was found.

Figure 8.3: Evaluation of test suit generation for simple interpreters with seeded errors. (RQ2b)

We can observe that low starting budgets can be effective at finding relatively simple errors, such as those provided by figures B.11 to B.13. However, finding more complicated errors, such as figures B.3 to B.10 requires a higher budget. Consequently, this requires significantly more time to compute if budget constraints are not added to help guide the symbolic execution towards these cases.

We can observe this when we look at the configuration with a budget of 7 and no additional budget constraints. This is the first configuration where all errors are detected, as is shown in figure 8.3b. When we introduce an extra budget constraint that makes exploring additions more expensive, we can observe that we were still able to detect all errors, while the time required to generate the test suites was reduced by a factor 4, as shown by figure 8.3a. However, we can observe that if we keep increasing the costs associated to various pattern matches, we are no longer able to detect the simpler errors in figures B.11 and B.12 without further increasing the total budget.

Furthermore, we can also see that for a number of implementations, we are unable to detect certain errors using the *Ref + Self* combination at lower depths, while we were able to detect them using the *All* combination. This can be explained by the fact that the test cases necessary to detect these errors are present in the test suites generated for other implementations. Thus, when all combined test suites are used in the *All* configuration, the error is still detected.

From these observations we can conclude that this technique is capable of generating test suites for introductory functional programming exercises that are capable of competing with manually written test suites. This approach produces the best results when the set of input programs is varied, due to the significant improvements that the *All* configuration has in figures 8.1 to 8.3. However, this improvement comes at the cost of longer execution times.

8.4 Limitations

There are some limitations that could prevent us from applying this technique in an actual course setup. The first and foremost concern is the resource cost. While we have seen that the technique is effective at generating a test suite for relatively simple assignments, we have seen that while the branching factor of solutions increases linearly, the required time to generate a test suite which finds all "interesting" errors increases exponentially. This problem can be partially mitigated by using a well-configured set of budget constraints. However, carefully configuring such a set of budget constraints could take a lot of time, especially since there is no clear strategy yet of how to effectively define budget constraints in such a way that it finds the test cases we are interested in. As of yet, there is no way of estimating the projected execution time or coverage with a given set of budget constraints, prior to performing the symbolic execution. This means a user might make a configuration which will either not terminate or not lead to meaningful results. Conversely, if the budget is depleted too rapidly along certain paths, errors might not be found. This does not fully rule out the usefulness of this approach however. Our technique can still be used to generate a basic set of test cases that cover the simple errors, allowing for more time to manually write more interesting test cases. Additionally, the technique is also capable of generating more complex test suites if the right budget constraints are provided.

Furthermore, as we mentioned earlier in chapter 3, not all language features are currently supported. Even though the assignments we considered for the evaluation were relatively simple, there were still a significant number of students for which our tool did not work, as can be seen in figures 8.1a and 8.2a. Assignments with a higher degree of complexity will also likely require using more language features, increasing the chance that student submissions will not be able to be processed by our tool. This means that in order for the tool to be able to be used on more complex assignments, more language features need to be supported. We provide suggestions of how this can be done in chapter 3.2.

Finally, upon further inspection, we found that a large number of differences found in student submissions were caused by an underspecification of certain behaviour in the assignment description. An example of this in the *Binary Search Trees* assignment is how the program should deal with malformed trees, that is, trees that are not binary search trees. This primarily leads to implementation differences surrounding the `contains` function. Here the reference implementation always searches both branches for the value, while many student implementations only searches the left branch if the value is lower than the value of the current node and only search the right branch if the value is higher. This leads to different behaviour when a malformed tree for example has a value which is lower inserted into its right branch, rather than the left branch. Since the assignment did not specify how to treat malformed search trees, one could argue that the student implementations are also correct, while they are still behaviourally different from our reference implementation. These instances must therefore be taken into account when using this technique.

9 Related Work

There are various frequently applied techniques for automatically testing the correctness of programs. Most of these techniques can be classified as a form of one of the following: program equivalence, *random testing*, *test suite generation*, or symbolic execution. We will now briefly discuss these techniques and how they relate to our approach.

9.1 Program Equivalence

Ideally we would like to determine whether two programs are equivalent. However, this problem is equivalent to the halting problem, since solving this would allow us to determine whether or not a program is equivalent to one that halts. This means that in the general case, determining the equivalence of two programs is undecidable [1].

However, less strict notions of equivalence can be (automatically) proven for two arbitrary programs. One such notion is *partial equivalence*, which states that two programs are equivalent if, when given the same inputs, the programs produce the same outputs if they terminate [51]. One study looked into developing an automatic partial equivalence prover for student submissions of Java assignments which did not suffer from limitations imposed by previous approaches [52]. Other studies have shown that it is possible to automatically prove a stronger notion of equivalence known as *full equivalence*, in which two programs are fully equivalent if their execution paths either both diverge or end up in equivalent states [51].

Different studies have also used heuristics to perform different forms of equivalence checking. One recent study employed aligning program semantics in order to simplify the equivalence proof [53]. A different recent study used recursion structure similarity and behavioural equivalence in functional programs to cluster student submissions for manual feedback. This reduced the number of submissions that needed to be provided with feedback, since the feedback given to a single submission, could be given to all students within the same cluster [54].

However, many of these techniques come with very strict limitations imposed on the programs that are being compared. Furthermore these techniques generally do not provide information about how the programs are different if they are found to be different. However, our approach has no theoretical limitations on the type of functional programs that are being compared and is able to provide counter examples in the form of test cases to disprove equivalence.

9.2 Random Testing

Random testing is a form of automated testing in which input data is randomly generated and supplied to the function under test, after which the correctness of the resulting output is verified. Over the years, many different forms of random testing have been developed. The most prominent ones will now be discussed.

Fuzzing [55] is perhaps the oldest, but still commonly applied [56] practice of supplying random input to a program and seeing if it crashes. More advanced forms of fuzzing have been developed, such as *coverage guided fuzzing*, where code coverage information is obtained during fuzzing and used when generating new inputs, in order to improve the effectiveness of finding bugs [57]. A popular approach to this is using *evolutionary algorithms* to modify existing test cases to find new interesting test cases that increase the coverage. An example of this is the popular *AFL* framework [58]. Fuzzing has also successfully been applied to Roslyn, Microsoft's .NET Compiler [59], finding behavioural differences between the binaries produced for different target platforms [60]. More recently, fuzzing has also been used in order to automatically provide feedback regarding implementation correctness of student submissions [61].

More advanced approaches of random testing exist. One of these techniques is *Adaptive Random Testing* (ART), where the next tried test case will be the one furthest away from the already executed test cases, which causes a better distribution of input data. This was shown to significantly speed up bug-finding [62]. The primary challenges in this technique lie in finding an efficient algorithm to properly spread the input data [63] and dealing with non-numeric inputs [64]. As such, many different techniques have been developed that deal with these challenges in various ways [63].

Another more advanced form of random testing is *Property Based Testing* (PBT). This technique was originally introduced by the random testing framework *QuickCheck* [65] for the Haskell programming language. Using this technique the test engineer has to specify properties that must hold for a function under test [65], rather than specifying input and their expected output values. Similarly to ART, PBT attempts to uniformly sample the input space in order to more effectively uncover errors in a program [65]. After having found an errorous input, *shrinking* can be applied to the input to create a simpler input which exposes the same errorous behaviour [66]. More recently, the technique has been combined with other testing techniques such as *coverage guided fuzzing* [67] and *combinatorial testing* [68]. Additionally, the technique has also been applied to various different domains, such as *multi-agent systems* [69] and *quantum computing* [70]. Furthermore, PBT with shrinking has also recently been utilised to provide automatic feedback to students working on C programming exercises [71].

These random testing approaches are easy to employ, but provide little guarantees due to their random nature. Additionally, applying these techniques directly to student submissions, rather than generating a test suite that validates the correctness of the submissions, could lead to unfairness in grading if different students are graded using a different set of tests. On the other hand, our approach guarantees that all feasible paths that do not violate the budget constraints are tested. Furthermore, since our approach does not rely on randomness, different students with equivalent solutions will not be assigned different grades.

9.3 Test Suite Generation

Besides our symbolic execution based technique to generate test suites, various other techniques exist to generate test suites that can be used for reproducible testing. The most common approach is *search-based software testing*, where search heuristics are used to generate a test suite that optimises some heuristic. The most commonly used heuristics are coverage metrics such as line coverage, branch coverage or mutation coverage [72]. Initially most studies used evolutionary algorithms as their search technique in order to incrementally improve their produced test suite [73]. The most notable example of this approach is the *EvoSuite* [73] test suite generator.

More recently, several studies have been conducted to evaluate the effectiveness of different variations of this approach. Most notably a study looking into the effectiveness of generating multiple test suites using different optimisation goals compared to generating a single test suite optimizing for multiple goals at the same time [72]. Another study looked into the effectiveness of different search algorithms that have been proposed in more recent years [74].

9.4 Symbolic Execution

In our work we focused on applying the optimisation techniques of early branch pruning, compositional analysis [5] and budget constraints to a relatively simple symbolic executor for functional programs in order to generate a test suite. However, various other symbolic execution-based techniques and optimisations have also been developed.

A technique called *differential symbolic execution* [75] has been developed, which uses symbolic execution combined with equivalence checking techniques to find which areas of two programs are equivalent. When two programs are considered nonequivalent, it provides inputs that show a difference in behaviour. This approach is mainly aimed at regression testing throughout different versions of the same software and therefore relies on the notion that both programs are mostly identical.

A different notable technique which combines symbolic execution with concrete execution is *concolic execution*. Concolic execution, popularised by the *CUTE* [76] framework, is one such technique. The *CUTE* framework performs its analysis by initially performing a concrete execution with concrete inputs. During this execution, path constraints are collected along the concretely executed path. When a symbolic value cannot be determined, the concrete value is used in its place. One of the path constraints is then negated and the system of constraints is solved in order to obtain a new concrete input that will lead to the execution of a different path. This process is then repeated. This technique has also been used to improve the effectiveness of search-based test suite generation. In particular, it has been used to improve the effectiveness of *EvoSuite* [77]. A recent study has also combined concolic execution with fuzzing [78, 79] in order to improve the performance of fuzzing.

A slightly different approach which combines symbolic execution and concrete execution is *Execution-Generated Testing* (EGT), popularised by the *KLEE* [38] framework. The *KLEE* framework performs its analysis by performing symbolic execution and determining along the way what symbolic values are effectively concrete (i.e. the symbolic value can only have a single possible value). Those values are then computed and marked as concrete. When all arguments to a function execution are marked as concrete, the function will be executed concretely, rather than symbolically [80]. Different studies have also combined this approach together with fuzzing (the *AFL* framework in particular) in order to improve effectiveness [81, 82].

Furthermore, recently efforts have been made to improve the performance of symbolic execution by compiling the symbolic execution rather than interpreting it from some intermediate language [83, 84].

10 Future Work

In order to improve the effectiveness of our tool, we make a number of suggestions for improvements for future research. These improvements aim to improve various aspects of the tool.

The first potential improvement is increasing the supported language subset. The specific areas of the language which can be improved upon and suggestions of how to add support for each particular feature has been previously described in figure 3.1. Solving these issues would allow the tool to be used for a broader range of different assignments and allow for considering more student submissions for the test suite generation.

```haskell
subtract_correct :: Int -> Int -> Int
subtract_correct a b = a - b

subtract_incorrect :: Int -> Int -> Int
subtract_incorrect a b = a + b
```

Figure 10.1: Example programs implementing a subtraction function.

Secondly, in chapter 2.2 we described that ideally we wanted to cover all inputs through each path. However, in our solution we have sacrificed this ideal by only finding a single set of inputs for each explored path. This can however lead to test cases being generated that do not actually properly validate the correctness of the program. Figure 10.1 shows a small example of a correct and incorrect program that would lead to the generation of a single input set. If this input set has the value of `b` set to `0`, these programs will produce the same output, even though the `subtract_incorrect` is trivially incorrect. These problems could be prevented by generating multiple different input sets for each summary. This operation can be done relatively easily by adding constraints to the resulting summaries that prevent the input types and values to be equal to the types and values of all previous inputs that were generated for that summary.

Thirdly, in chapter 8.4 we noted that there were instances of student submissions being behaviourally different from our reference implementation, while they were in fact correct given the assignment specification. One way this problem can be mitigated is by adding the possibility of selecting multiple solutions as reference solutions. This would require either implementing multiple reference solutions or using correct student submissions as reference solutions. Instead of comparing the output value against the output of the one reference solution, it could be compared to the output of multiple reference solutions and the test should pass if the output matches one of them.

Additionally, we have observed that, while the generated test cases can detect errors, it can be relatively hard to understand what behaviour the test cases are testing. Therefore, it would be useful if the generated test cases could be automatically classified or explained in human language. This could provide valuable insights for both the people responsible for the grading and for the students receiving the grades.

Furthermore, different pre-existing improvements to the usability of symbolic execution, such as those described in chapter 9.4, could be incorporated with our technique. An example of such a technique that could be incorporated into our tool which could improve the run time of our analysis is EGT. This technique only serves to replace symbolic execution with concrete execution whenever possible [80] and does therefore not sacrifice the completeness of our analysis. Another potential technique that could be incorporated to reach deeper paths is some form of (adaptive) random testing to replace some or all summaries of function calls. This has the benefit of allowing errors in very deep execution paths to be uncovered [81], without exponentially increasing the number of paths that need to be explored. This has a disadvantage that the generated test suite might not be consistent between different executions of the tool, due to the introduction of randomness to the system.

Moreover, it would be valuable to perform evaluations to determine the effectiveness of our symbolic execution technique in other related problem domains. For this study, we have only considered grading student submissions as the use case of our technique. However, it is also theoretically possible to apply the same technique to other problems where it is necessary to establish the equivalence of two programs with a certain degree of confidence. One example of such a problem is determining whether a code change due to refactoring has resulted in a behavioural difference. Another example would be determining whether multiple implementations of the same algorithm in different programming languages are behaviourally equivalent.

Finally, improvements could be made in the way that budget constraints are configured. Adding support for associating costs with a more varied set of operations than function calls and pattern matches alone could allow for more advanced configurations. However, as mentioned in chapter 8.4, the problem also remains that there is currently no obvious way of configuring the budget constraints. Assisting the user with configuring these budget constraints could be very helpful. A potential way of doing this would be to provide a way of estimating the run time or coverage metrics for a given program and a given set of budget constraints. Another way the system of budget constraints could be improved is by finding a way to dynamically fine-tune the budget constraints during the symbolic execution. Knowledge obtained by performing the symbolic execution on other submissions for the same assignment could be used for this.

11 Conclusion

To summarise, in this thesis we have presented a technique to automatically generate test suites for student submissions of programming assignments in pure functional programming languages. This technique utilises symbolic execution in order to build summaries for all paths through a program that are bounded by a set of budget constraints. Each of these summaries consists of a set of constraints that can be solved in order to obtain a set of inputs which cause the path corresponding to the summary to be executed.

In order perform this symbolic execution, we have shown a way of converting abstract syntax trees of functional programs into a special type of control-flow graphs that can be used to more easily rationalise about the execution flow of the program. Additionally, we have provided a definition of a language which serves as an intermediate representation for constraints, which allows us to more easily define constraints in a way that respects the semantics of the Scala type system.

Finally, we have shown that the test suites generated by this technique are more effective at finding errors than our manually written test suites for simple introductory assignments. We have also shown that the technique can be used to generate a basic test suite for more complex assignments which is capable enough to catch some errors. However, we have discovered that this technique as-is is not suitable for fully replacing manually written test suites for more complex assignments, due to the high resource costs of generating an adequate enough test suite that can catch all major errors. We have provided a number of ways that the technique can be improved upon in order to make it more suitable for complex assignments.

References

[1] V. A. Zakharov, "The equivalence problem for computational models: Decidable and undecidable cases," in *Machines, Computations, and Universality*, M. Margenstern and Y. Rogozhin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 133–152, ISBN: 978-3-540-45132-7. DOI: 10.1007/3-540-45132-3_8. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45132-3_8.

[2] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: https://dl.acm.org/doi/10.1145/360248.360252.

[3] ——, "A new approach to program testing," in *Proceedings of the International Conference on Reliable Software*, Los Angeles, California: Association for Computing Machinery, 1975, pp. 228–233, ISBN: 9781450373852. DOI: 10.1145/800027.808444. [Online]. Available: https://dl.acm.org/doi/10.1145/800027.808444.

[4] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, pp. 1–39, 2018. DOI: 10.1145/3182657. [Online]. Available: https://dl.acm.org/doi/10.1145/3182657.

[5] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07, Nice, France: Association for Computing Machinery, 2007, pp. 47–54, ISBN: 1595935754. DOI: 10.1145/1190216.1190226. [Online]. Available: https://dl.acm.org/doi/10.1145/1190215.1190226.

[6] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381, ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_28. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-78800-3_28.

[7] Y. Lin, T. Miller, and H. Søndergaard, "Compositional symbolic execution using fine-grained summaries," in *2015 24th Australasian Software Engineering Conference*, 2015, pp. 213–222. DOI: 10.1109/ASWEC.2015.32. [Online]. Available: https://ieeexplore.ieee.org/document/7365810.

[8] Y. Lin, T. Miller, and H. Søndergaard, "Compositional symbolic execution: Incremental solving revisited," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, 2016, pp. 273–280. DOI: 10.1109/APSEC.2016.046. [Online]. Available: https://ieeexplore.ieee.org/document/7890598.

[9] J. F. Santos, P. Maksimovic, G. Sampaio, and P. Gardner, "Javert 2.0: Compositional symbolic execution for javascript," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–31, 2019. DOI: 10.1145/3290379. [Online]. Available: https://dl.acm.org/doi/10.1145/3290379.

[10] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970, ISSN: 0362-1340. DOI: 10.1145/390013.808479. [Online]. Available: https://dl.acm.org/doi/10.1145/390013.808479.

[11] R. M. Hierons, "Avoiding coincidental correctness in boundary value analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, pp. 227–241, Jul. 2006, ISSN: 1049-331X. DOI: 10.1145/1151695.1151696. [Online]. Available: https://dl.acm.org/doi/10.1145/1151695.1151696.

[12] S. Krishnamurthi, "7 functions anywhere," in *Programming Languages: Application and Interpretation*. 2012. [Online]. Available: http://cs.brown.edu/courses/cs173/2012/book/.

[13] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997, ISSN: 0360-0300. DOI: 10.1145/267580.267590. [Online]. Available: https://dl.acm.org/doi/10.1145/267580.267590.

[14] École Polytechnique Fédérale Lausanne (EPFL) Lausanne, *The scala programming language*. [Online]. Available: https://www.scala-lang.org/.

[15] Oracle, *Java software*. [Online]. Available: https://www.oracle.com/java/.

[16] Microsoft, *.NET*. [Online]. Available: https://dotnet.microsoft.com/en-us/.

[17] Haskell Organization, *Haskell language*. [Online]. Available: https://www.haskell.org/.

[18] SMLFamily, *Standard ml family*. [Online]. Available: https://smlfamily.github.io/.

[19] A. K. Quentin Charatan, *Java In Two Semesters*, 3rd ed. McGraw-Hill Education, 2009, ISBN: 13 978-0-07-712267-6.

[20] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '86, Portland, Oregon, USA: Association for Computing Machinery, 1986, pp. 38–45, ISBN: 0897912047. DOI: 10.1145/28697.28702. [Online]. Available: https://dl.acm.org/doi/10.1145/28697.28702.

[21] École Polytechnique Fédérale Lausanne (EPFL) Lausanne, *Case classes*. [Online]. Available: https://docs.scala-lang.org/tour/case-classes.html.

[22] T. Pape, V. Kirilichev, C. F. Bolz, and R. Hirschfeld, "Record data structures in racket: Usage analysis and optimization," *SIGAPP Appl. Comput. Rev.*, vol. 16, no. 4, pp. 25–37, Jan. 2017, ISSN: 1559-6915. DOI: 10.1145/3040575.3040578. [Online]. Available: https://dl.acm.org/doi/10.1145/3040575.3040578.

[23] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger, "An overview of the scala programming language second edition," École Polytechnique Fédérale Lausanne (EPFL) Lausanne, 2006. [Online]. Available: http://scala-lang.org/docu/files/ScalaOverview.pdf.

[24] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black, "Traits: A mechanism for fine-grained reuse," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 331–388, Mar. 2006, ISSN: 0164-0925. DOI: 10.1145/1119479.1119483. [Online]. Available: https://dl.acm.org/doi/10.1145/1119479.1119483.

[25] H. Bretthauer, T. Christaller, and J. Kopp, "Multiple vs. single inheritance in object-oriented programming languages," *Microprocessing and Microprogramming*, vol. 28, no. 1, pp. 197–200, 1990, ISSN: 0165-6074. DOI: 10.1016/0165-6074(90)90173-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0165607490901737.

[26] S. B. Lippman, *Inside the C++ Object Model*, 3rd ed. Addison-Wesley, 1996, ISBN: 0-201-83454-5.

[27] École Polytechnique Fédérale Lausanne (EPFL) Lausanne, *Singleton objects*. [Online]. Available: https://docs.scala-lang.org/tour/singleton-objects.html.

[28] ——, *Anonymous functions*. [Online]. Available: https://docs.scala-lang.org/scala3/book/fun-anonymous-functions.html.

[29] J. W. Backus, “The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference,” in *IFIP Congress*, 1959.

[30] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[31] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “Cvc5: A versatile and industrial-strength SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, D. Fisman and G. Rosu, Eds., ser. Lecture Notes in Computer Science, vol. 13243, Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-99524-9_24.

[32] A. Kennedy and B. C. Pierce, “On decidability of nominal subtyping with variance,” in *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, Jan. 2007. [Online]. Available: https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/.

[33] K. Crary, “Typed compilation of inclusive subtyping,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’00, New York, NY, USA: Association for Computing Machinery, 2000, pp. 68–81, ISBN: 1581132026. DOI: 10.1145/351240.351247. [Online]. Available: https://dl.acm.org/doi/10.1145/351240.351247.

[34] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, “A history of haskell: Being lazy with class,” in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III, San Diego, California: Association for Computing Machinery, 2007, 12–1–12–55, ISBN: 9781595937667. DOI: 10.1145/1238844.1238856. [Online]. Available: https://dl.acm.org/doi/10.1145/1238844.1238856.

[35] W. Swierstra, “Data types à la carte,” *Journal of Functional Programming*, vol. 18, no. 4, pp. 423–436, 2008. DOI: 10.1017/S0956796808006758. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/data-types-a-la-carte/14416CB20C4637164EA9F7709790940

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press Ltd., 2009, ISBN: 978-0-262-03384-8.

[37] J. B. Goodenough, “Exception handling: Issues and a proposed notation,” *Commun. ACM*, vol. 18, no. 12, pp. 683–696, Dec. 1975, ISSN: 0001-0782. DOI: 10.1145/361227.361230. [Online]. Available: https://dl.acm.org/doi/10.1145/361227.361230.

[38] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08, San Diego, California: USENIX Association, 2008, pp. 209–224. DOI: 10.5555/1855741.1855756. [Online]. Available: https://dl.acm.org/doi/10.5555/1855741.1855756.

[39] A. Martelli and U. Montanari, “An efficient unification algorithm,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, pp. 258–282, Apr. 1982, ISSN: 0164-0925. DOI: 10.1145/357162.357169. [Online]. Available: https://dl.acm.org/doi/10.1145/357162.357169.

[40] “Iso/iec/ieee international standard - systems and software engineering–vocabulary,” *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, 2017. DOI: 10.1109/IEEESTD.2017.8016712. [Online]. Available: https://ieeexplore.ieee.org/document/8016712.

[41] Y. Wei, B. Meyer, and M. Oriol, “Is branch coverage a good measure of testing effectiveness?” In *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, B. Meyer and M. Nordio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 194–212, ISBN: 978-3-642-25231-0. DOI: 10.1007/978-3-642-25231-0_5. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-25231-0_5.

[42] P. S. Kochhar, F. Thung, and D. Lo, “Code coverage and test suite effectiveness: Empirical study with real bugs in large systems,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 560–564. DOI: 10.1109/SANER.2015.7081877. [Online]. Available: https://ieeexplore.ieee.org/document/7081877.

[43] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, “An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption,” *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 597–608, 2017. DOI: 10.1109/ICSE.2017.61. [Online]. Available: https://ieeexplore.ieee.org/document/7985697.

[44] A. Parsai and S. Demeyer, “Comparing mutation coverage against branch coverage in an industrial setting,” *International Journal on Software Tools for Technology Transfer*, vol. 22, pp. 365–388, 2020. DOI: 10.1007/s10009-020-00567-y. [Online]. Available: https://link.springer.com/article/10.1007/s10009-020-00567-y.

[45] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. DOI: 10.1109/TSE.2010.62. [Online]. Available: https://ieeexplore.ieee.org/document/5487526.

[46] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “Pit: A practical mutation testing tool for java (demo),” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 449–452, ISBN: 9781450343909. DOI: 10.1145/2931037.2948707. [Online]. Available: https://dl.acm.org/doi/10.1145/2931037.2948707.

[47] Pitest Team, *Mutation operators*. [Online]. Available: https://pitest.org/quickstart/mutators/.

[48] Stryker Team, *Supported mutators*. [Online]. Available: https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/.

[49] Eclemma, *JaCoCo Java Code Coverage Library*. [Online]. Available: https://www.eclemma.org/jacoco/.

[50] T. v. d. Lippe, T. Smith, D. Pelsmaeker, and E. Visser, "A scalable infrastructure for teaching concepts of programming languages in scala with weblab: An experience report," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, ser. SCALA 2016, Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 65–74, ISBN: 9781450346481. DOI: 10.1145/2998392.2998402. [Online]. Available: http://resolver.tudelft.nl/uuid:bab45902-284f-4f6b-9362-be5b3df7ff20.

[51] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, "A language-independent proof system for full program equivalence," *Form. Asp. Comput.*, vol. 28, no. 3, pp. 469–497, May 2016, ISSN: 0934-5043. DOI: 10.1007/s00165-016-0361-7. [Online]. Available: https://dl.acm.org/doi/10.1007/s00165-016-0361-7.

[52] Q. Zhou, D. Heath, and W. Harris, *Completely automated equivalence proofs*, 2017. eprint: arXiv:1705.03110. [Online]. Available: https://arxiv.org/pdf/1705.03110.pdf.

[53] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1027–1040, ISBN: 9781450367127. DOI: 10.1145/3314221.3314596. [Online]. Available: https://dl.acm.org/doi/10.1145/3314221.3314596.

[54] J. Clune, V. Ramamurthy, R. Martins, and U. A. Acar, "Program equivalence for assisted grading of functional programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428239. [Online]. Available: https://dl.acm.org/doi/10.1145/3428239.

[55] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://dl.acm.org/doi/10.1145/96267.96279.

[56] R. Guo, "Mongodb's javascript fuzzer: The fuzzer is for those edge cases that your testing didn't catch.," *Queue*, vol. 15, no. 1, pp. 38–56, Feb. 2017, ISSN: 1542-7730. DOI: 10.1145/3055301.3059007. [Online]. Available: https://dl.acm.org/doi/10.1145/3055301.3059007.

[57] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, *The art, science, and engineering of fuzzing: A survey*, 2018. DOI: 10.48550/ARXIV.1812.00140. [Online]. Available: https://arxiv.org/abs/1812.00140.

[58] M. Zalewski, *American fuzzy lop*. [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[59] .NET Foundation, *Roslyn The .NET Compiler Platform*. [Online]. Available: https://github.com/dotnet/roslyn.

[60] J. B. Nielsen, C. Schmidt, and J. Larsen, *Fuzzlyn*, Jun. 2018. [Online]. Available: https://github.com/jakobbotsch/Fuzzlyn.

[61] P. Ortegat, B. Vanderose, and X. Devroey, "Towards automated testing for simple programming exercises," in *Proceedings of the 4th International Workshop on Education through Advanced Software Engineering and Artificial Intelligence (EASEAI '22)*, 2022. DOI: 10.1145/3548660.3561334. [Online]. Available: https://xdevroey.be/publication/ortegat-2022/ortegat-2022.pdf.

[62] T. Y. Chen, H. Leung, and I. K. Mak, “Adaptive random testing,” in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, M. J. Maher, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 320–329, ISBN: 978-3-540-30502-6. DOI: 10.1007/978-3-540-30502-6_23. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-30502-6_23.

[63] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, “A survey on adaptive random testing,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2052–2083, 2021. DOI: 10.1109/TSE.2019.2942921. [Online]. Available: https://ieeexplore.ieee.org/document/8846002.

[64] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, R. Merkel, and G. Rothermel, “A cost-effective random testing method for programs with non-numeric inputs,” *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3509–3523, 2016. DOI: 10.1109/TC.2016.2547380. [Online]. Available: https://ieeexplore.ieee.org/document/7442567.

[65] K. Claessen and J. Hughes, “Quickcheck: A lightweight tool for random testing of haskell programs,” *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000, ISSN: 0362-1340. DOI: 10.1145/357766.351266. [Online]. Available: https://dl.acm.org/doi/10.1145/351240.351266.

[66] J. Hughes, “Experiences with quickcheck: Testing the hard stuff and staying sane,” in *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, S. Lindley, C. McBride, P. Trinder, and D. Sannella, Eds. Cham: Springer International Publishing, 2016, pp. 169–186, ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_9. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-30936-1_9.

[67] L. Lampropoulos, M. Hicks, and B. C. Pierce, “Coverage guided, property based testing,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360607. [Online]. Available: https://dl.acm.org/doi/10.1145/3360607.

[68] H. Goldstein, J. Hughes, L. Lampropoulos, and B. C. Pierce, “Do judge a test by its cover,” in *Programming Languages and Systems*, N. Yoshida, Ed., Cham: Springer International Publishing, 2021, pp. 264–291, ISBN: 978-3-030-72019-3. DOI: 10.1007/978-3-030-72019-3_10. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-72019-3_10.

[69] C. Benac Earle and L.-Å. Fredlund, “A property-based testing framework for multi-agent systems,” in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS ’19, Montreal QC, Canada: International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 1823–1825, ISBN: 9781450363099. DOI: 10.5555/3306127.3331931. [Online]. Available: https://dl.acm.org/doi/10.5555/3306127.3331931.

[70] S. Honarvar, M. R. Mousavi, and R. Nagarajan, “Property-based testing of quantum programs in q#,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 430–435, ISBN: 9781450379632. DOI: 10.1145/3387940.3391459. [Online]. Available: https://dl.acm.org/doi/10.1145/3387940.3391459.

[71] P. Vasconcelos and R. P. Ribeiro, “Using Property-Based Testing to Generate Feedback for C Programming Exercises,” in *First International Computer Programming Education Conference (ICPEC 2020)*, R. Queirós, F. Portela, M. Pinto, and A. Simões, Eds., ser. OpenAccess Series in Informatics (OASIcs), vol. 81, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 28:1–28:10, ISBN: 978-3-95977-153-5. DOI: 10.4230/OASIcs.ICPEC.2020.28. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12315.

[72] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, “A detailed investigation of the effectiveness of whole test suite generation,” *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, Apr. 2017, ISSN: 1573-7616. DOI: 10.1007/s10664-015-9424-2. [Online]. Available: https://link.springer.com/article/10.1007/s10664-015-9424-2.

[73] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE ’11, Szeged, Hungary: Association for Computing Machinery, 2011, pp. 416–419, ISBN: 9781450304436. DOI: 10.1145/2025113.2025179. [Online]. Available: https://dl.acm.org/doi/10.1145/2025113.2025179.

[74] M. Khari, A. Sinha, E. Verdú, and R. G. Crespo, “Performance analysis of six meta-heuristic algorithms over automated test suite generation for path coverage-based optimization,” *Soft Computing*, vol. 24, no. 12, pp. 9143–9160, Jun. 2020, ISSN: 1433-7479. DOI: 10.1007/s00500-019-04444-y. [Online]. Available: https://link.springer.com/article/10.1007/s00500-019-04444-y.

[75] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pundefinedsundefinedreanu, “Differential symbolic execution,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’08/FSE-16, Atlanta, Georgia: Association for Computing Machinery, 2008, pp. 226–237, ISBN: 9781595939951. DOI: 10.1145/1453101.1453131. [Online]. Available: https://dl.acm.org/doi/10.1145/1453101.1453131.

[76] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005, ISSN: 0163-5948. DOI: 10.1145/1095430.1081750. [Online]. Available: https://dl.acm.org/doi/10.1145/1081706.1081750.

[77] J. P. Galeotti, G. Fraser, and A. Arcuri, “Improving search-based test suite generation with dynamic symbolic execution,” in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 360–369. DOI: 10.1109/ISSRE.2013.6698889. [Online]. Available: https://ieeexplore.ieee.org/document/6698889.

[78] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” *Network and Distributed System Security Symposium*, 2016. DOI: 10.14722/ndss.2016.23368. [Online]. Available: https://doi.org/10.14722/ndss.2021.24118.

[79] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “QSYM : A practical concolic execution engine tailored for hybrid fuzzing,” in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[80] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. [Online]. Available: https://dl.acm.org/doi/10.1145/2408776.2408795.

[81] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, “Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC ’18, Pau, France: Association for Computing Machinery, 2018, pp. 1475–1482, ISBN: 9781450351911. DOI: 10.1145/3167132.3167289. [Online]. Available: https://dl.acm.org/doi/10.1145/3167132.3167289.

[82] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, “Safl: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE ’18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 61–64, ISBN: 9781450356633. DOI: 10.1145/3183440.3183494. [Online]. Available: https://dl.acm.org/doi/10.1145/3183440.3183494.

[83] S. Poeplau and A. Francillon, “Symbolic execution with SymCC: Don’t interpret, compile!” In *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 181–198, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[84] ——, “Symqemu: Compilation-based symbolic execution for binaries,” *Network and Distributed System Security Symposium*, 2021. DOI: 10.14722/ndss.2021.24118. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/.

A Student Data Set

A.1 Lists

```scala
object Solution {
  /**
    * We will now define a slightly less trivial case class,
    * representing a list structure containing integers.
    *
    * Any list can be one of two types:
    * it can be empty, or it can be an element joined to the remaining list.
    *
    * Notice that the structure of these case classes is recursive.
    */
  sealed abstract class IntList
  case class Empty()                           extends IntList // The empty list, often called Nil
  case class Element(n: Int, tail: IntList) extends IntList // Element is usually called Cons

  /**
    * As an example, let's create a function that lists descending integers from n to 1.
    */
  def listFrom(n: Int): IntList = {
    if (n == 0) Empty()
    else Element(n, listFrom(n-1))
  }

  /**
    * EXERCISE:
    * Implement the function sumIntList(xs).
    * It should take an IntList, and return the sum of all it's elements.
    * Use pattern matching!
    */
  def sumIntList(xs: IntList): Int = throw new RuntimeException("Not yet implemented")

  /**
    * EXERCISE:
    * Implement the function head(xs).
    * It should return the first element in a list.
    * If the list is empty, throw a NoSuchElementException.
    */
  def head(xs: IntList): Int = throw new RuntimeException("Not yet implemented")

  /**
    * EXERCISE:
    * Define the function tail(xs).
    * It should accept an IntList and return the same IntList, but without the first element.
    * If the list is empty, throw a NoSuchElementException.
    */
  def tail(xs: IntList): IntList = throw new RuntimeException("Not yet implemented")

  /**
    * EXERCISE:
    * Define the function concat(xs, ys).
    * It should concatenate two IntLists.
    */
  def concat(xs: IntList, ys: IntList): IntList = throw new RuntimeException("Not yet
       implemented")

  /**
    * EXERCISE:
    * Define the function take(n, xs).
    * It should return the first n elements of xs.
    * This function should never throw an exception.
    */
  def take(n: Int, xs: IntList): IntList = throw new RuntimeException("Not yet implemented")

  /**
    * EXERCISE:
    * Define the function drop(n, xs).
    * It should return the list xs, without the first n elements.
    * This function should never throw an exception.
    */
  def drop(n: Int, xs: IntList): IntList = throw new RuntimeException("Not yet implemented")
}
```

Figure A.1: *Lists* assignment template and description.

```scala
object Solution {
  sealed abstract class IntList
  case class Empty()                        extends IntList // The empty list, often called Nils
  case class Element(n: Int, tail: IntList) extends IntList // Element is usually called Cons

  def listFrom(n: Int): IntList = {
    if (n == 0) Empty()
    else Element(n, listFrom(n-1))
  }

  def sumIntList(xs: IntList): Int = xs match {
    case Empty()          => 0
    case Element(n, tail) => n + sumIntList(tail)
  }


  def head(xs: IntList): Int = xs match {
    case Empty()       => throw new NoSuchElementException
    case Element(n, _) => n
  }

  def tail(xs: IntList): IntList = xs match {
    case Empty()       => throw new NoSuchElementException
    case Element(_, t) => t
  }

  def concat(xs: IntList, ys: IntList): IntList = xs match {
    case Empty()        => ys
    case Element(n, ts) => Element(n, concat(ts, ys))
  }

  def take(n: Int, xs: IntList): IntList = xs match {
    case Element(e, ts) if n > 0 => Element(e, take(n-1, ts))
    case _                       => Empty()
  }

  def drop(n: Int, xs: IntList): IntList = xs match {
    case Element(_, ts) if n > 0 => drop(n-1, ts)
    case _                       => xs
  }
}
```

Figure A.2: *Lists* assignment reference implementation.

```scala
import org.scalatest.FunSuite
import Solution._

class SpecTest extends FunSuite {
  test("Sum IntList") {
    def sumIntList(xs: IntList): Int = xs match {
      case Empty() => 0
      case Element(n, tail) => n + sumIntList(tail)
    }

    val list = listFrom((Math.random * 40 + 10).asInstanceOf[Int])
    assert(Solution.sumIntList(Empty()) == 0)
    assert(sumIntList(list) == Solution.sumIntList(list))
  }

  test("Concat empty lists") {
    assertResult(Empty()) {
      concat(Empty(), Empty())
    }
  }

  test("Concat with one empty list") {
    assertResult(Element(3, Empty())) {
      concat(Element(3, Empty()), Empty())
    }
  }

  test("Concat") {
    assertResult(Element(3, Element(6, Empty()))) {
      concat(Element(3, Empty()), Element(6, Empty()))
    }
  }

  test("Concat more elements") {
    assertResult(Element(34, Element(3, Element(6, Empty())))) {
      concat(Element(34, Element(3, Empty())), Element(6, Empty()))
    }
  }

  test("Head empty") {
    intercept[NoSuchElementException] {
      head(Empty())
    }
  }

  // tail(xs): return the tail of the list

  test("Tail") {
    assertResult(Element(3, Empty())) {
      tail(Element(34, Element(3, Empty())))
    }
  }

  // take(n, xs): take the first n elements

  test("take 10") {
    assertResult(Element(34, Element(3, Element(6, Empty())))) {
      take(10, Element(34, Element(3, Element(6, Empty()))))
    }
  }

  test("take 2") {
    assertResult(Element(34, Element(3, Empty()))) {
      take(2, Element(34, Element(3, Empty())))
    }
  }

  test("take 1") {
    assertResult(Element(3, Empty())) {
      take(1, Element(3, Element(34, Element(6, Empty()))))
    }
  }

  test("take 0") {
    assertResult(Empty()) {
      take(0, Element(34, Element(3, Empty())))
    }
  }

  // drop(n, xs): drop the first n elements

  test("drop 0") {
    assertResult(Element(34, Element(3, Element(6, Empty())))) {
      drop(0, Element(34, Element(3, Element(6, Empty()))))
    }
  }

  test("drop 1") {
    assertResult(Element(3, Element(6, Empty()))) {
      drop(1, Element(34, Element(3, Element(6, Empty()))))
    }
  }

  test("drop 10") {
    assertResult(Empty()) {
      drop(10, Element(34, Element(3, Element(6, Empty()))))
    }
  }
}
```

Figure A.3: *Lists* assignment manually written test suite.

A.2 Binary Search Trees

```scala
object Solution {
  /**
   * EXERCISE:
   * Define the case classes for Tree.
   * A tree can be a Leaf, or a Node with a value and two child trees.
   * The height function serves as an example for how the tree structure should be organized
   */

  def height(tree: Tree): Int = tree match {
    case Leaf() => 0
    case Node(elem, left, right) => 1 + Math.max(height(left), height(right))
  }

  /**
   * EXERCISE:
   * Define the following functions.
   *
   * Do not worry about rebalancing the tree.
   *
   * Hint:
   * Pattern matches can be made more specific with arbitrary conditions:
   *     `case A(n) if n>3 => print("n is greater than 3!")`
   */
  def insert(e: Int, t: Tree): Tree = throw new RuntimeException("Not yet implemented")

  def contains(e: Int, t: Tree): Boolean = throw new RuntimeException("Not yet implemented")

  def size(t: Tree): Int = throw new RuntimeException("Not yet implemented")
}
```

Figure A.4: *Binary Search Trees* assignment template and description.

```scala
object Solution {
  sealed abstract class Tree
  case class Leaf() extends Tree
  case class Node(elem: Int, left: Tree, right: Tree) extends Tree

  def insert(i: Int, t: Tree): Tree = t match {
    case Leaf()                  => Node(i, Leaf(), Leaf())
    case Node(e, l, r) if i > e => Node(e, l, insert(i, r))
    case Node(e, l, r) if i < e => Node(e, insert(i, l), r)
    case _ => t
  }

  def contains(i: Int, t: Tree): Boolean = t match {
    case Leaf()                   => false
    case Node(e, _, _) if e == i  => true
    case Node(_, l, r)            => contains(i, l) || contains(i, r)
  }

  def size(t: Tree): Int = t match {
    case Leaf()         => 0
    case Node(_, l, r) => 1 + size(l) + size(r)
  }
}
```

Figure A.5: *Binary Search Trees* assignment reference implementation.

```
import Solution._
import org.scalatest.FunSuite

class SpecTest extends FunSuite {
  test("testSizeSkel") {
    assertResult(3) {
      size(Node(1, Node(0, Leaf(), Leaf()), Node(2, Leaf(), Leaf())))
    }
  }

  test("testSize1") {
    assertResult(0) {
      size(Leaf())
    }
  }

  test("testSize2") {
    assertResult(1) {
      size(Node(0, Leaf(), Leaf()))
    }
  }

  test("testSize3") {
    assertResult(3) {
      size(Node(0, Node(1, Leaf(), Leaf()), Node(2, Leaf(), Leaf())))
    }
  }

  test("testSize7") {
    assertResult(7) {
      size(Node(4,
        Node(2, Node(1, Leaf(), Leaf()), Node(3, Leaf(), Leaf())),
        Node(6, Node(5, Leaf(), Leaf()), Node(7, Leaf(), Leaf()))))
    }
  }

  test("testInsert1") {
    assertResult (Node(5,Node(4,Node(3,Leaf(),Leaf()),Leaf()),Leaf())) {
      insert(3, insert(4, insert(5, Leaf()))) }
  }

  test("testInsert2") {
    assertResult (Node(5,Node(4,Leaf(),Leaf()),Node(52,Leaf(),Leaf()))) {
      insert(52, insert(5, insert(4, insert(5, Leaf()))))
    }
  }

  test("testInsert3") {
    assertResult (Node(10,Node(4,Node(3,Leaf(), Leaf()),Node(5,Leaf(),Leaf())),Node(12,Node(11,
        Leaf(),Leaf()),Leaf()))) {
      insert(3, insert(5, insert(11, insert(12, insert(4, insert(10, Leaf()))))))
    }
  }

  test("testFind0") {
    assertResult(false) {
      contains(3, Leaf())
    }
  }

  test("testFind1") {
    assertResult(true) {
      contains(3, insert(5, insert(9, insert(1, insert(3, Leaf())))))
    }
  }

  test("testFind2") {
    assertResult(false) {
      contains(7, insert(5, insert(9, insert(1, insert(3, Leaf())))))
    }
  }
}
```

Figure A.6: *Binary Search Trees* assignment manually written test suite.

B Interpreter Data Set

B.1 Type Definitions

```scala
abstract class ExprC
case class NumC(value: Int) extends ExprC
case class AddC(e1: ExprC, e2: ExprC) extends ExprC
case class FdC(param: Int, body: ExprC) extends ExprC
case class IdC(id: Int) extends ExprC
case class AppC(f: ExprC, arg: ExprC) extends ExprC

abstract class Environment
case class EmptyEnvironment() extends Environment
case class EnvironmentNode(name: Int, value: Value, tail: Environment) extends Environment

abstract class Value
case class NumV(value: Int) extends Value
case class ClosV(env: Environment, param: Int, body: ExprC) extends Value
```

Figure B.1: Type definitions shared for all interpreters.

B.2 Reference Solution

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = {
    e match {
      case NumC(v) => NumV(v)
      case AddC(e1, e2) => interp(e1, nv) match {
        case NumV(v1) => interp(e2, nv) match {
          case NumV(v2) => NumV(v1 + v2)
          case _ => throw InterpretException()
        }
        case _ => throw InterpretException()
      }
      case IdC(id) => find(nv, id)
      case FdC(n, b) => ClosV(nv, n, b)
      case AppC(f, a) => interp(f, nv) match {
        case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv), cnv))
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
  }
}
```

Figure B.2: Reference implementation of a simple interpreter.

B.3 Faulty Solutions

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = e match {
    case NumC(v) => NumV(v)
    case AddC(e1, e2) => interp(e1, nv) match {
      case NumV(v1) => interp(e2, nv) match {
        case NumV(v2) => NumV(v1 + v2)
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
    case IdC(id) => find(nv, id)
    case FdC(n, b) => ClosV(nv, n, b)
    case AppC(f, a) => interp(f, nv) match {
      case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv), nv))
      case _ => throw InterpretException()
    }
    case _ => throw InterpretException()
  }
}
```

Figure B.3: Faulty implementation which uses the local environment to interpret the body of a closure.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def concat(front: Environment, back: Environment): Environment = front match {
    case EmptyEnvironment() => back
    case EnvironmentNode(name, value, tail) => EnvironmentNode(name, value, concat(tail, back))
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = e match {
    case NumC(v) => NumV(v)
    case AddC(e1, e2) => interp(e1, nv) match {
      case NumV(v1) => interp(e2, nv) match {
        case NumV(v2) => NumV(v1 + v2)
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
    case IdC(id) => find(nv, id)
    case FdC(n, b) => ClosV(nv, n, b)
    case AppC(f, a) => interp(f, nv) match {
      case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv), concat(
          nv, cnv)))
      case _ => throw InterpretException()
    }
    case _ => throw InterpretException()
  }
}
```

Figure B.4: Faulty implementation which first uses the local environment, then the closure environment to interpret the body of a closure.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def concat(front: Environment, back: Environment): Environment = front match {
    case EmptyEnvironment() => back
    case EnvironmentNode(name, value, tail) => EnvironmentNode(name, value, concat(tail, back))
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = e match {
    case NumC(v) => NumV(v)
    case AddC(e1, e2) => interp(e1, nv) match {
      case NumV(v1) => interp(e2, nv) match {
        case NumV(v2) => NumV(v1 + v2)
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
    case IdC(id) => find(nv, id)
    case FdC(n, b) => ClosV(nv, n, b)
    case AppC(f, a) => interp(f, nv) match {
      case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv), concat(
          cnv, nv)))
      case _ => throw InterpretException()
    }
    case _ => throw InterpretException()
  }
}
```

Figure B.5: Faulty implementation which first uses the closure environment, then the local environment to interpret the body of a closure.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = {
    e match {
      case NumC(v) => NumV(v)
      case AddC(e1, e2) => interp(e1, nv) match {
        case NumV(v1) => interp(e2, nv) match {
          case NumV(v2) => NumV(v1 + v2)
          case _ => throw InterpretException()
        }
        case _ => throw InterpretException()
      }
      case IdC(id) => find(nv, id)
      case FdC(n, b) => ClosV(nv, n, b)
      case AppC(f, a) => interp(f, nv) match {
        case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv),
            EmptyEnvironment()))
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
  }
}
```

Figure B.6: Faulty implementation which uses the empty environment to interpret the body of a closure.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = e match {
    case NumC(v) => NumV(v)
    case AddC(e1, e2) => interp(e1, nv) match {
      case NumV(v1) => interp(e2, nv) match {
        case NumV(v2) => NumV(v1 + v2)
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
    case IdC(id) => find(nv, id)
    case FdC(n, b) => ClosV(nv, n, b)
    case AppC(f, a) => interp(f, nv) match {
      case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, cnv), cnv))
      case _ => throw InterpretException()
    }
    case _ => throw InterpretException()
  }
}
```

Figure B.7: Faulty implementation which uses the closure environment to interpret the argument of a function application.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def concat(front: Environment, back: Environment): Environment = front match {
    case EmptyEnvironment() => back
    case EnvironmentNode(name, value, tail) => EnvironmentNode(name, value, concat(tail, back))
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = e match {
    case NumC(v) => NumV(v)
    case AddC(e1, e2) => interp(e1, nv) match {
      case NumV(v1) => interp(e2, nv) match {
        case NumV(v2) => NumV(v1 + v2)
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
    case IdC(id) => find(nv, id)
    case FdC(n, b) => ClosV(nv, n, b)
    case AppC(f, a) => interp(f, nv) match {
      case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, concat(cnv,
          nv)), cnv))
      case _ => throw InterpretException()
    }
    case _ => throw InterpretException()
  }
}
```

Figure B.8: Faulty implementation which first uses the closure environment, then the local environment to interpret the argument of a function application.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def concat(front: Environment, back: Environment): Environment = front match {
    case EmptyEnvironment() => back
    case EnvironmentNode(name, value, tail) => EnvironmentNode(name, value, concat(tail, back))
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = e match {
    case NumC(v) => NumV(v)
    case AddC(e1, e2) => interp(e1, nv) match {
      case NumV(v1) => interp(e2, nv) match {
        case NumV(v2) => NumV(v1 + v2)
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
    case IdC(id) => find(nv, id)
    case FdC(n, b) => ClosV(nv, n, b)
    case AppC(f, a) => interp(f, nv) match {
      case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, concat(nv,
           cnv)), cnv))
      case _ => throw InterpretException()
    }
    case _ => throw InterpretException()
  }
}
```

Figure B.9: Faulty implementation which first uses the local environment, then the closure environment to interpret the argument of a function application.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = e match {
    case NumC(v) => NumV(v)
    case AddC(e1, e2) => interp(e1, nv) match {
      case NumV(v1) => interp(e2, nv) match {
        case NumV(v2) => NumV(v1 + v2)
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
    case IdC(id) => find(nv, id)
    case FdC(n, b) => ClosV(nv, n, b)
    case AppC(f, a) => interp(f, nv) match {
      case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a,
           EmptyEnvironment()), cnv))
      case _ => throw InterpretException()
    }
    case _ => throw InterpretException()
  }
}
```

Figure B.10: Faulty implementation which uses the empty environment to interpret the argument of a function application.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = {
    e match {
      case NumC(v) => NumV(v)
      case AddC(NumC(v1), NumC(v2)) => NumV(v1 + v2)
      case IdC(id) => find(nv, id)
      case FdC(n, b) => ClosV(nv, n, b)
      case AppC(f, a) => interp(f, nv) match {
        case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv), cnv))
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
  }
}
```

Figure B.11: Faulty implementation which does not interpret the arguments of addition before adding them.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = {
    e match {
      case NumC(v) => NumV(v)
      case AddC(NumC(v1), NumC(v2)) => NumV(v1 + v2)
      case AddC(AddC(NumC(v1), NumC(v2)), NumC(v3)) => NumV(v1 + v2 + v3)
      case AddC(NumC(v1), AddC(NumC(v2), NumC(v3))) => NumV(v1 + v2 + v3)
      case AddC(AddC(NumC(v1), NumC(v2)), AddC(NumC(v3), NumC(v4))) => NumV(v1 + v2 + v3 + v4)
      case IdC(id) => find(nv, id)
      case FdC(n, b) => ClosV(nv, n, b)
      case AppC(f, a) => interp(f, nv) match {
        case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv), cnv))
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
  }
}
```

Figure B.12: Another faulty implementation which does not interpret the arguments of addition before adding them.

```scala
object Interp {
  case class InterpretException(msg: String = null) extends Exception
  def find(nv: Environment, n: Int): Value = nv match {
    case EmptyEnvironment() => throw InterpretException()
    case EnvironmentNode(name, value, tail) => if (name == n) value else find(tail, n)
  }
  def interpEntry(e: ExprC): Value = interp(e, EmptyEnvironment())
  def interp(e: ExprC, nv: Environment): Value = {
    e match {
      case NumC(v) => NumV(v)
      case AddC(NumC(i1), NumC(i2)) if (i2 < 0) => NumV(i1 - i2)
      case AddC(e1, e2) => interp(e1, nv) match {
        case NumV(v1) => interp(e2, nv) match {
          case NumV(v2) => NumV(v1 + v2)
          case _ => throw InterpretException()
        }
        case _ => throw InterpretException()
      }
      case IdC(id) => find(nv, id)
      case FdC(n, b) => ClosV(nv, n, b)
      case AppC(f, a) => interp(f, nv) match {
        case ClosV(cnv, param, body) => interp(body, EnvironmentNode(param, interp(a, nv), cnv))
        case _ => throw InterpretException()
      }
      case _ => throw InterpretException()
    }
  }
}
```

Figure B.13: Faulty implementation which has a malicious extra case for additions.

C Extra Results

	Reference B.2	Body Local B.3	Body Local + Closure B.4	Body Closure + Local B.5	Body Empty B.6	Argument Closure B.7	Argument Closure + Local B.8	Argument Local + Closure B.9	Argument Empty B.10	Structural Matching 1 B.11	Structural Matching 2 B.12	Extra Case B.13
Budget: 1 Call cost: 1	24	24	27	27	24	24	27	27	24	22	31	39
Budget: 2 Call cost: 1	43	43	46	46	43	43	46	46	43	37	55	66
Budget: 3 Call cost: 1	111	111	114	114	111	111	114	114	111	63	93	201
Budget: 4 Call cost: 1	331	331	305	305	331	331	305	305	331	168	492	672
Budget: 5 Call cost: 1	999	999	861	861	999	999	861	861	996	430	1510	2285
Budget: 6 Call cost: 1	3051	3051	2438	2444	3049	3052	2438	2431	3025	977	3521	7910
Budget: 7 Call cost: 1	9432	9430	6980	7025	9412	9441	6985	6930	9254	2588	13226	27684
Budget: 8 Call cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 7 Call cost: 1 Match "AddC" cost: 1	2696	2694	1776	1807	2682	2702	1801	1764	2623	1675	2017	3524
Budget: 8 Call cost: 1 Match "AddC" cost: 1	7135	7120	4164	4283	7072	7161	4267	4123	6762	4192	5170	9656
Budget: 9 Call cost: 1 Match "AddC" cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 8 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	4129	4120	2790	2850	4089	4153	2804	2722	3896	1934	1957	4334
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	10166	10126	6236	6449	9990	10249	6310	6024	9250	4411	4471	10730
Budget: 10 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 2	8156	8122	5139	5303	8003	8238	5203	4971	7343	3312	3319	8217
Budget: 10 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 2	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout

Figure C.1: Number of Z3 inquiries of different budget constraint configurations for different interpreter implementations.

	Reference B.2	Body Local B.3	Body Local + Closure B.4	Body Closure + Local B.5	Body Empty B.6	Argument Closure B.7	Argument Closure + Local B.8	Argument Local + Closure B.9	Argument Empty B.10	Structural Matching 1 B.11	Structural Matching 2 B.12	Extra Case B.13
Budget: 1 Call cost: 1	3	3	3	3	3	3	3	3	3	4	7	4
Budget: 2 Call cost: 1	8	8	8	8	8	8	8	8	8	8	14	10
Budget: 3 Call cost: 1	21	21	21	21	21	21	21	21	21	12	21	30
Budget: 4 Call cost: 1	58	58	51	51	58	58	51	51	58	29	71	91
Budget: 5 Call cost: 1	159	159	134	134	159	159	134	134	159	59	164	284
Budget: 6 Call cost: 1	446	446	344	344	446	446	344	344	446	106	310	913
Budget: 7 Call cost: 1	1286	1286	911	911	1286	1286	911	911	1286	289	1285	3039
Budget: 8 Call cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 7 Call cost: 1 Match "AddC" cost: 1	335	335	197	197	335	335	197	197	335	182	235	407
Budget: 8 Call cost: 1 Match "AddC" cost: 1	798	798	415	415	798	802	415	415	794	432	546	1056
Budget: 9 Call cost: 1 Match "AddC" cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 8 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	430	430	258	258	430	431	257	257	429	190	197	454
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	967	967	516	514	967	971	505	508	961	385	401	1023
Budget: 10 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 1	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
Budget: 9 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 2	723	723	401	399	723	725	394	397	719	266	268	731
Budget: 10 Call cost: 1 Match "AddC" cost: 1 Match "NumC" cost: 2	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout

Figure C.2: Number of generated tests of different budget constraint configurations for different interpreter implementations.