Deriving a Symbolic Executor for Definitional Interpreters Suitable for the Study of Heuristics

Laura-Ana Pîrcălăboiu Casper Bach Poulsen Cas van der Rest

Delft University of Technology
Research Project - Q4 2020 / 2021

Abstract

Recent years have seen a surge of interest for dynamic testing techniques, one of which is symbolic execution. It is the main point of interest of this research paper, in which we give an overview of a framework for symbolically executing definitional interpreters. We will also discuss techniques that we made use of in developing the symbolic execution framework. The context of this project is the automated grading and validation of student submissions, and the results and performance of our approach will also be reviewed and criticized.

1 Introduction

Most Computer Science courses rely on practical assignments as a means of making sure that students achieve the learning objectives of various courses. According to the U.S. Bureau of Labor Statistics, the number of job openings in the field is set to grow by 32% by 2029 [1]. This statistic is also reflected in the growing number of Computer Science study programs and students who are enrolling in them worldwide. Therefore, it can be inferred that the manual checking of the aforementioned assignments is quickly becoming more and more inefficient and error prone.

This paper addresses the concern of automatic testing of student submissions. In the present research we will discuss how efficient symbolic execution is at finding differences between definitional interpreters. These interpreters are similar to what students would be writing in a course about programming languages, and are similar to what is discussed in the textbook *Programming Languages: Application and Interpretation* [2]. To this end, we will:

- Define an intermediate representation (Section 2.4) for definitional interpreters. This is a step that facilitates the other contribution outlined in this list.
- Define a simple small-step transition function (Section 2.5) that given such an intermediate representation returns the next possible step, which enables us to explore the branches of an intermediate representation, or IR, recursively. This results in an approach that, as seen in Section 3, effectively distinguishes between interpreters.

The topic of dynamic test generation has seen a marked increase in interest in recent years as an effective technique for generating sound and complete test suites for

gradually more complex software. Symbolic execution in particular is a technique that is widely employed in the industry [3]–[5].

At a high level, the idea of symbolic execution is that instead of a given program being ran on ***concrete*** input (numeric values for example), the input variables are allowed to be ***symbolic*** - in other words, placeholders for their concrete counterparts.

2 Method

The idea of this paper is to explore a simpler alternative to the approach to symbolic execution detailed in Mensing et al's work [6] to serve as a basis for heuristic research. Ideally, we would like to get to a point where we can run two interpreters in parallel and compare them as they run. We have adopted the usage of the small step transition function as it was proposed by Mensing et al. [6] to accomplish that, albeit simplifying it, as will be detailed in a further section. We conjecture that this is an important part of the symbolic executor that we are deriving since it allows us to unfold an expression once (as opposed to until termination of the program).

2.1 Intuition

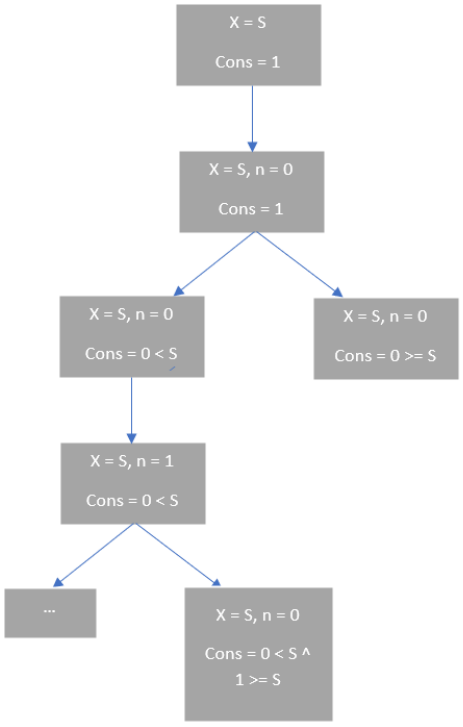
The result that we would like to arrive at is an execution tree, where all paths (and the respective constraints that guard them) that can be taken in the program are explored. Take for example a dummy program in a C-like language, such as the one in Listing 1.

Listing 1: A Simple Program.

```
void test(int x){
    int n = 0;
    while (n < x){
        n ++;
    }
}
```

Intuitively, if we were to take pen to paper and draw a tree of all the possible paths one might take through this program, it would look similar to what can be seen in Figure 1.

Figure 1: Execution Tree for a Simple Program.

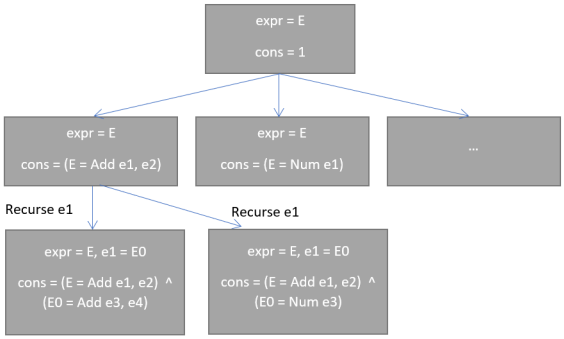


A similar approach can also be taken with testing a defitional interpreter such as the one in Listing 2 ending up with a result like the one in Figure 2.

Listing 2: A Simple Definitional Interpreter.

```
data Expr = Num Int
          | Add Expr Expr
eval :: Expr -> Int
eval (Num i) = i
eval (Add e1 e2) =
    eval e1 + eval e2
```

Figure 2: Execution Tree for a Simple Symbolic Executor.



An intuitive way of defining the small step transition function method that will be used in this research paper is to picture a finite state automaton. In a more general sense, take the concept of a transition function: it takes as input one state, and returns the logical continuation of this state as a result of an action being taken as an output. For example, in *tic-tac-toe*, the logical continuation of a player making the move *Draw an X on the lower left corner of the board* is a new game state where there is an X on said spot.

2.2 Background into the Methodology Exhibited by Mensing et al.

The symbolic executor described by Mensing et al. [6] provides an approach to symbolic execution where each path is explored in a breadth-first search manner. The two main particularities of the methodology are the small-step transition function and the usage of free monads, which are a way of encoding an execution state [7]. The general idea is to use a family of free monads to encode a state in a symbolic execution of the program in question, and the authors define a transition function that encodes a *next step* following from a given free monad. In a way, this is analogous to the transition function in finite state automata. In the paper, the authors also provide a framework for generating test cases for faulty definitional interpreters and generating terms that prove certain constraints incorrect. These constrints are defined in a way similar to what we will define later in this paper.

2.3 Definition of the Language Under Test

The language we will be running our symbolic executor will be defined in Listing 3. We opted for a lisp-like language that has basic support for functions, with function application, lambdas and identifiers. It also features numbers and the possibility to add them. A definitional interpreter for this language might feature a top level pattern match, and recursive calls for the evaluation of nested expressions.

Listing 3: Structure of the programming language to be used.

```
data Expr = Add Expr Expr
          | Num Int
          | Lam String Expr
          | App Expr Expr
          | Ident String
```

2.4 An Intermediate Representation for Definitional Interpreters

It proves difficult to transform code directly into execution trees. Having an uni-

fied format to encode these interpreters into would provide convenience and extensibility. Therefore, it is useful to define an alternative representation for definitional interpreters as defined in the textbook *Programming Languages: Application and Interpretation* [2]. This intermediate representation, or IR as it will be called in the implementation, has to prove amiable for comparing and checking interpreters for equivalence. It makes therefore sense to think of interpreters in terms of choices, guards and recursive calls. An element that sets the current implementation apart from the old one is the distinction between regular function calls and recursive calls in this intermediate representation.

Therefore, as stated beforehand, the three elements that will define the behavior of a given interpreter are the following:

- *Choice.* We need the option to specify different branches for, for example, the top level pattern matches that occur in definitional interpreters. We will represent them by means of the + symbol.
- *Guards.* Branches are most often guarded by constraints, which for the purposes of this paper will be defined by means of *equality* or *inequality* between the input given to the interpreter and a certain input shape. We will represent them as constraints written between square brackets (for example, [e = Num(i)]).
- *Recursive function calls.*The interpreter must be able to recursively call itself so that it can evaluate nested expressions. We will simply pass a list of arguments and a string that denotes the variable we will be using to refer to the result of the recursive call in the continuation. For example, recurse i as i1. return +(i1, 3).

In Listing 4 the grammar of the intermediate representation language is defined.

In listing 5 an example of a simple interpreter in the syntax defined above is laid out.

Listing 5: Definitional Interpreter.

```
eval = ([e ≡ Num(i)]. return t)
    + ([e ≢ Num(i),
        e ≡ Add(e1, e2)]
            .recurse e1 as i1
            .recurse e2 as i2.
            return +(i1, i2))
```

2.5 Implementation of The Symbolic Executor

In the following sub-sections we will discuss implementation details of the symbolic executor that we have developed.

2.5.1 Definition of a Small-Step Transition Function

The small step transition function is the metaphorical heart and soul of the symbolic executor presented in this paper. It takes as input an intermediate representation of a symbolic executor and returns an array of intermediate representations symbolizing the next possible steps that can be taken by our definitional interpreter.

One of the trivial cases for the step function is the choice case, which will by definition simply return an array of all the choices that can be made by the interpreter. Two other trivial cases are Return, which simply returns the variable in question, and Raise, which simply returns an error.

There are two notable cases for the small-step transition function which will be detailed in the following sections. The first problem that arises in the development of this function is that of making a recursive call to the interpreter through the transition

Listing 4: Grammar of the IR.

```
t ∈ Term ::= f(t) | i | x (terms)
T ∈ Tree ::= T + T | [(t = t)]. T | recurse t as x.
           | return t (trees)
f ∈ TermCon = {Num/1, Add/2, ...} (term constructor)
x, y, z ∈ Var = {x, y, z, ...} (variables)
i ∈ Integer = {0, −1, 1, ...} (integers)
```

function. This proves difficult since the latter is ignorant of what the top-level interpreter looks like (it only knows what the current step it is performing looks like).

The second notable case that we will discuss is that of performing a next step on a guarded branch. To this end, we would like to know whether there exist variable assignments that can satisfy the guards of the branch.

2.5.2 Constraint Resolution and Unification

For the purposes of our symbolic executor, we often work with branches that are guarded by constraints [8]. Consider, for example, a constraint such as the one in Listing 6.

Listing 6: Constraint.

```
[e0 = Num(i)]
```

which tells us that variable `e` can match the pattern Num(i). Therefore, for the sake of consistency we would like to be able to treat the symbolic variable `e0` as a Num(i) from now on.

In computer science, unification is the process of solving equations between symbolic variables. Usually, the solution to an unification problem is a set of substitutions that we can make to make the two sides of the equation equal. In this case for example we would like to substitute e with a Num(i), so our unifier would be represented by the array [(e, Num(i))]. An empty unifier ([]) means that the unification problem has no solutions (so that no suitable substitution has been found).

The unification algorithm we will use is based on the one from Mensing et al., which in turn is based on the one by Martelli and Montanari [9].

We implemented several auxillary functions to implement the unification algorithm, the most important of which are the occurs checker and the function that performs the substitution after successfully obtaining an unifier by means of the unify function. The unify function performs a fold over Con variables and simply checks for occurrence in the case of 2 regular symbolic variables.

2.5.3 Recursive Calls

The main problem of the recursive step is that infinite loops must be avoided. Additionally, we also would like to be aware of the result that has been returned by the recursive call in the *continuation* IR. To this end, we have defined a notion of sequencing which takes two IR arguments (the IR to be evaluated into a value, and the expression that we would like to substitute this value into).

The step function for *recurse* generates a list of sequential statements. For the purposes of this paper, which is supposed to act like a proof-of-concept, is limited to

recurring on symbolic variables, although this can be expanded on in future work. Following this, the variable resulting from each of the first steps is substituted in its respective continuation.

2.5.4 Driver Loop

The driver loop which can be seen in Listing 7 is relatively simple. It takes a set of intermediate representations of definitional interpreters, and returns the "next step" using the transition function for each of them in a round-robin fashion. This solution to run multiple interpreters at the same time, albeit simple, fulfils its intended purpose.

Listing 7: Driver function.

```
driver :: [IR] -> [IR]
driver [] = []
driver (x:xs) = (step x)
        ++ (driver xs)
```

2.5.5 Case study: Step by Step Symbolic Execution of An Example

In this section we will detail how an interpreter is explored by our symbolic execution engine. Take, as an example, the interpreter in Listing 5. Performing one step will yield an array which contains the branches where the Num and Add pattern matches are performed, as can be seen in Listing 8.

Listing 8: Stepped Once.

```
[
    [[Symbolic [] "a"
        matches Num]]
        return Symbolic [] "a",
    [[Symbolic [] "a"
        matches Add "i1" "i2"]]
        recurse "i1" as "c".
        recurse"i2" as "d".
        return applying
            "+" to "c" and "d"
]
```

If we keep stepping each of the respective guarded statements, we will eventually get to the full execution tree of the sample definitional interpreter, which is shown in Listing 9.

3 Experimental Setup and Results

The code described in this paper was developed and ran on a machine running Windows 10, in the Haskell programming language using Stack 2.7.

The test setup consists of multiple functions which simply call the driver loop on several interpreters that have been transcribed into the intermediate representation defined in Section 2.4, which an interested reader should be able to run. Our symbolic executor will therefore generate an execution tree for each of these interpreters (as well as possibly synthesizing terms that would exercise certain paths in the program).

In this section, we will detail how effective the program is at finding bugs in interpreters by transcribing the interpreters from the Mensing paper, for example, running them two at a time and comparing the outputs against each other. This is, to some

Listing 9: Stepped Multiple Times.

```
[
    (return
    Symbolic [Symbolic [] "a" matches Num] "a"),
    (seq [[Symbolic [] "a" matches Num]]
        return
            Symbolic [] "a" as "c"
    then
        recurse"i2" as "d".
        return applying
            "+" to "c" and "d"),
    (seq [[Symbolic [] "a" matches Add "i1" "i2"]]
        recurse "i1" as "c".
        recurse "i2" as "d".
        return applying "+" to "c" and "d"
            as "c"
    then recurse "i2" as "d".
        return applying "+" to "c" and "d")
]
```

Listing 10: Compare Function and Higher Level Driver Loop.

```
runTimes :: Int -> Term -> [IR] -> [IR]
runTimes 1 e eval = driver e eval eval
runTimes n e eval =  driver e eval (runTimes (n - 1) e eval)

compareInterp :: Int -> Term -> [IR] -> [IR] -> Bool
compareInterp n e a b = (runTimes n e a == runTimes n e b)
```

extent, an exercise in imagination in which one of them is the *template* solution, the one which defines the expected behavior, and the other is a potentially faulty student submission.

This has been achieved by means of a driver loop that compares the two interpreters that run against each other, namely that which can be seen in Listing 10.

Comparing the approach detailed in this paper to the Mensing approach qualitatively is difficult, but our approach has the advantage of being able to be ran in a step-by-step manner (i.e.: for a specific amount of steps, up to a certain depth in the tree). The kinds of bugs we are aiming to find are interpreters with buggy variable orders, for example. We have opted for a simple binary equivalence system: deciding whether interpreters are or not equivalent. We have tested this approach with 8 interpreters out of which there are three equivalence classes. The results of the tests that have been ran against the interpreters are reported in the following confusion matrix in Figure 3. For the results below, we have decided to use the compareInterp function and unfold each interpreter three

times. The *Predicted* label is indicative of whether any two given interpreters are or not equivalent, and the *Actual* label indicates whether the symbolic executor classifies them as equal or not.

Figure 3: Confusion Matrix of Results.

		Predicted: True	Predicted: False
Actual	True	4	0
	False	10	20

The table makes it clear that while there are no false positives, a number of interpreters are classified as non-equivalent even though they are. We conjecture that this effect happens because the symbolic executor is agnostic of branch order, and does not recognize interpreters that should be equivalent but have a different branch order (for example, *Choice [a, b]* and *Choice [b, a]*). We think this is a relevant point to be added in future work.

4 Responsible Research

4.1 Reproducibility of Results

The code of the symbolic executor will be made available online on Github under the CC-BY 4.0 license, as well as the test suite used for evaluation, so that interested readers can reproduce the results stated in this paper.

4.2 Threats to Validity

Since this research project was developed over the course of 10 weeks, it is not out of the question that there will be bugs or other issues in the code. What is more, the test data is likely incomplete and not large, and the project would likely benefit from additions to the test dataset and more thorough testing before being deployed into practice.

Another threat to the external validity of the research in question is how well the approach in question generalizes [10]. We have tested it on a simple language with a relatively straight forward grammar, but we do not know how well it will work for more complex languages.

4.3 Ethical Considerations of the Research

Given the fact that the software in question can be used for assistance in grading student submissions in university level courses, some ethical considerations arise. This piece of software should not be used without the supervision of a responsible teacher.

Kramer et al. consider in their work the relation between false positives and false negatives in scientific research [11]. They conjecture that false positives (in our case, declaring two interpreters equivalent when they are not) are good for the individual, while false negatives (in our case, declaring two interpreters not equivalent when they are) bring benefits in the long run by encouraging scientific advancements. In our case, the balance between false positives and false negatives is important as well, because it concerns the choice between having students receive higher grades, or having to check more solutions manually.

5 Discussion and Future Developments

In this paper we discussed a simplified framework for symbolic execution. In the light of the features discussed above, we believe that the ideas that we exhibited in this paper can be used in future development.

First of all, the usage of an intermediate representation facilitates the development

of the small-step transition function. This is currently being used to evaluate programs in a round-robin fashion as seen in the driver loop, which brings the advantage of being able to run and compare multiple programs at the same time. This naive driver loop can be further optimized and improved.

Extending the group of languages for which this approach works, while not trivial, is made easier by transcribing current interpreters into the IR encoding. This is a development that can be looked into in future work.

The framework in question could benefit from an alternative way of searching or pruning next states in future developments of this work. Further reducing the number of false negatives would also be ideal.

6 Related Work

In this section, an overview of existing tools and techniques concerning symbolic execution will be provided. It is useful to remark that the contributions outlined in this paper are based on the symbolic executor outlined in the paper by Mensing et al. [6], more specifically the online Haskell version of the aforementioned paper. The authors of this paper employ free monads and a small-step execution strategy to evaluate and compare definitional interpreters, similar to the ones a student could write in an university course on programming languages (such as the ones in the textbook Programming Languages: Application and Interpretation [2]).

The previous lines of work exhibited by the MiniKanren family of relational programming languages follow a similar vein as the symbolic executor developed by Mensing et al [12]. MiniKanren makes no distinction between input and output variables and performs a "backwards search" with the purpose of finding an input that produces a given output, provided a piece of software. It does so by exhaustively searching the execution tree of said program, even though that might imply it running infinitely.

Another interesting work for the purposes of the project being discussed in this paper is the research of Cadar and Sen on the KLEE framework and testing of COREUTILS in Unix systems [13]. In it, several ways of improving a naive symbolic executor such as the one described in this paper. This might prove interesting for future research, as it does delve deeper into heuristic research. These heuristics focus primarily on achieving high statement and branch coverage, but they could also be employed to optimize other desired criteria. Some ideas explored in the paper which we might want to employ in the future are, for example, path pruning, which takes place when a symbolic executor reaches a path that it is equivalent to one it has explored before (same constraints on the same symbolic variables). Other techniques based, for example, on random exploration and mutation testing combined with symbolic execution are also used, although these might be less interesting for the scope of this paper.

In conclusion, the symbolic executor described in this paper builds on the foundations established by Mensing et al., adopting the concept of the small-step transition function and losing some information related to the current execution state by not making use of monads. Our approach is easily extensible to be able to evaluate other similar languages as well, due to its simpler notion of state. Additionally, because of this and the flexibility of the driver functions, the authors reckon that the symbolic executor is open to the addition and study of heuristics.

7 Conclusion

In conclusion, our paper proposes a simple framework for evaluating definitional interpreters in university courses. Our work builds upon the solid foundations of Mensing et al. We make use of a small-step transition function and a simple constraint resolution language to build a symbolic executor that evaluates all possible paths through a definitional interpreter.

All things considered, we believe that the framework can and should be further expanded and explored before any kind of didactic use. In future work, heuristics to improve the efficiency of our symbolic executor and refine our approach to running multiple definitional interpreters at the same time can also be explored.

Acknowledgements

We thank the Academic Communication Skills and Responsible Computer Science lecturers for their time and valuable feedback, as well as the anonymous reviewers and members of the peer group for their timely comments and suggestions going forwards. We'd also like to thank Casper Bach Poulsen and Cas van der Rest for their invaluable support during the project.

References

[1] *Information Security Analysts : Occupational Outlook Handbook: : U.S. Bureau of Labor Statistics*, en. [Online]. Available: https://www.bls.gov/ooh/computer-and-information-technology/information-security-analysts.htm#tab-6. (visited on 16/06/2021).

[2] S. Krishnamurthi, "Programming Languages: Application and Interpretation," en, p. 207,

[3] P. Godefroid, M. Y. Levin and D. Molnar, "Automated Whitebox Fuzz Testing," en, p. 16,

[4] N. Tillmann and J. de Halleux, "Pex–White Box Test Generation for .NET," en, in *Tests and Proofs*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, B. Beckert and R. Hähnle, Eds., vol. 4966, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 134–153, ISBN: 978-3-540-79123-2 978-3-540-79124-9. DOI: 10.1007/978-3-540-79124-9_10. [Online]. Available: http://link.springer.com/10.1007/978-3-540-79124-9_10 (visited on 26/06/2021).

[5] S. Anand, C. S. Păsăreanu and W. Visser, "JPF–SE: A Symbolic Execution Extension to Java PathFinder," en, in *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, Eds., vol. 4424, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 134–138, ISBN: 978-3-540-71208-4 978-3-540-71209-1. DOI: 10.1007/978-3-540-71209-1_12. [Online]. Available: http://link.springer.com/10.1007/978-3-540-71209-1_12 (visited on 26/06/2021).

[6] A. D. Mensing, "From Definitional Interpreter to Symbolic Executor," en, p. 10, 2019.

[7] O. Kiselyov and H. Ishii, "Freer monads, more extensible effects," in *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, ser. Haskell '15, New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 94–105, ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804319. [Online]. Available: https://doi.org/10.1145/2804302.2804319 (visited on 27/06/2021).

[8] F. Fages and G. Huet, "Complete sets of unifiers and matchers in equational theories," en, *Theoretical Computer Science*, vol. 43, pp. 189–200, Jan. 1986, ISSN: 0304-3975. DOI: 10.1016/0304-3975(86)90175-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397586901751 (visited on 26/06/2021).

[9] A. Martelli and U. Montanari, "An Efficient Unification Algorithm," en, *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 2, pp. 258–282, Apr. 1982, ISSN: 0164-0925, 1558-4593. DOI: 10.1145/357162.357169. [Online]. Available: https://dl.acm.org/doi/10.1145/357162.357169 (visited on 02/06/2021).

[10] *Threats to validity of Research Design*. [Online]. Available: https://web.pdx.edu/~stipakb/download/PA555/ResearchDesign.html (visited on 25/06/2021).

[11] 262588213843476, *A* (A star) search in haskell*, en. [Online]. Available: https://gist.github.com/abhin4v/8172534 (visited on 13/05/2021).

[12] W. E. Byrd, *Relational Programming in MiniKanren: Techniques, . . .* 2009.

[13] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," en, p. 16,