Concolic Execution for testing definitional interpreters

Philippos Boon Alexaki
Delft University of Technology
The Netherlands

Casper Bach Poulsen
Delft University of Technology
The Netherlands

Cas van der Rest
Delft University of Technology
The Netherlands

June 27, 2021

Abstract

Definitional interpreters are difficult to test with a pre-defined test suite. This paper tries to determine the effectiveness of automatic testing of definitional interpreters, using concolic execution. First we develop a model for concolic execution of a functional language. Then we identify different categories of common mistakes when writing interpreters and test which are caught by concolic execution. We find that while concolic execution is promising, as it can find counterexamples for most errors in a small language, the model developed in this paper is not sufficient to generate counterexamples for more complicated languages.

1 Introduction

The course "Concepts of Programming Languages" uses predefined test-cases to grade and provide feedback to students' submissions. Each year they manage to find missing test cases in the test-suite. This, of course, is not a desirable situation, because grading should be as complete and sound as possible. There are multiple approaches to solve this, and this paper will focus on testing these submissions using concolic execution.

Concolic execution is a form of symbolic execution where the program that is tested gets a random concrete input which evolves after each execution. During execution the tester records all the constraints on the input that are encountered, for example whether the input matches a specific pattern. After execution ends a new input is generated where one of these constraints is falsified. This new input is then fed back to the program, and given that one of the path constraints has been falsified this input is guaranteed to follow an unexplored path. Thereby automatically covering all possible execution paths.

The existing research on concolic execution does not focus on validating definitional interpreters or compilers. While an adjacent technique, generating constrained random data, has been used to verify the correctness of the Go compiler (Griesemer et al., 2020), concolic testing has not been researched for this use.

The **goal** of this paper is to investigate how effective concolic execution is for testing definitional interpreters.

To achieve this, we first define a simple concolic executor and evolution engine. This

model will be inspired by the arithmetic part of You, Findler, and Dimoulas (2021) and the LambdaPix language defined in Clune, Ramamurthy, Martins, and Acar (2020). The decision to base the concolic executor on LambdaPix is because this language was designed to be able to be converted to logical formulas, something that is very useful for an evolution engine. Furthermore the approach is very similar to the one presented in Giantsios, Papaspyrou, and Sagonas (2017) but has been developed independently.

We then determine the effectiveness of this model to find certain classes of mistakes often seen in students' submissions. To determine whether a "student submission" (in our case a correct/incorrect interpreter) is correct or incorrect, we compare its output with that of a known correct interpreter.

This paper contributes the following:

- A model of concolic execution of a functional language with recursive data structures in Section 2;
- A first look at the effectiveness of concolic execution as a method for verifying interpreters in Section 3.

Lastly Section 4 is a reflection on the model of concolic execution developed and the areas for improvement.

2 Concolic Execution

In this section we describe a model of concolic execution. To be able to do this we first define a language that we interpret. This language, called LambdaPix, and its behaviour are described in the Clune et al. (2020) paper.

In figure 1 the grammar for values and expressions in LambdaPix is described, together with a grammar for the representation of the execution path a value has taken.

To be able to record the constraints set on the input by the program that is run, extra information needs to be passed through an interpreter. With each value a representation of the execution path it has taken is associated. Figure 2 shows how this path is recorded by adding new information on top of the existing dynamic semantics of LambdaPix.

$\textsc{Conc}_1$ shows that a constant is evaluated as its value and the equivalent execution path is the constant itself and given that there is no constraint when evaluating a constant the constraints are empty. $\textsc{Conc}_2$ shows that to interpret a record we first need to interpret all the sub-expressions then construct the record value and equivalent execution path and finally concatenate all the constraints of the sub-paths followed. This is analogous for $\textsc{Conc}_3$ and $\textsc{Conc}_4$, where injections are evaluated.

Then we have $\textsc{Conc}_5$ where we evaluate primitive binary operations. The operation is applied on the values, if they are of the correct type, but for the respective execution path we record the fact that an operation has been applied to be able to reason about constraints on numbers.

$\textsc{Conc}_6$ regards the evaluation of function application. The constraints when interpreting the function and its argument are saved and concatenated with the constraints when the body is interpreted. Before interpreting the body the environment is extended by associating the variable name with the value and its respective execution path.

Finally there is $\textsc{Conc}_7$, when a case expression is evaluated we record all pairs of patterns and execution paths, until the pattern that matched the value, and whether the pattern matched with the corresponding value.

Pattern matching has been extended by running the pattern matching algorithm in parallel on the value and path. When a symbolic variable is found as part of the path being matched, its value is looked up in the environ-

base types	b	::=	$int \mid boolean \mid char$	
types	τ	::=	b	base type
		$\mid$	δ	data type
		$\mid$	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$	product type
		$\mid$	$\tau_1 \to \tau_2$	function type
injection labels	i	::=	$\text{label}_1 \mid \text{label}_2 \mid \ldots$	
patterns	p	::=	$\_$	wildcard pattern
		$\mid$	x	variable pattern
		$\mid$	$\{\ell_1 = p_1, \ldots, \ell_n = p_n\}$	record pattern
		$\mid$	x as p	alias pattern
		$\mid$	c	constant pattern
		$\mid$	$i \cdot p$	injection pattern (with argument)
		$\mid$	i	injection pattern (without argument)
primitive operations	o	::=	$+ \mid - \mid * \mid < \mid > \mid \leq \mid \geq$	
		$\mid$	$=_\tau \mid \neq_\tau \mid ++$ (concat)	
expressions	e	::=	c	constant
		$\mid$	x	variable
		$\mid$	$\{\ell_1 = e_1, \ldots, \ell_n = e_n\}$	record
		$\mid$	$i \cdot e$	injection (with argument)
		$\mid$	i	injection (without argument)
		$\mid$	case $e\{p_1.e_1 \mid \ldots \mid p_n.e_1\}$	case analysis
		$\mid$	$\lambda x.e$	abstraction
		$\mid$	$e_1\ e_2$	application
		$\mid$	`fix` x `is` e	fixed point
		$\mid$	o	primitive operation
paths	r	::=	c	constant
		$\mid$	x	symbolic input variable
		$\mid$	$\{\ell_1 = r_1, \ldots, \ell_n = r_n\}$	record path
		$\mid$	$i \cdot r$	injection path (with argument)
		$\mid$	i	injection path (without argument)
		$\mid$	$o\ r_1\ r_2$	application of a primitive operation

Figure 1: The syntax of the LambdaPix language

$$\frac{}{c \Rightarrow (c \text{ val}, c \text{ path}, [])}\ \textsc{Conc}_1$$

$$\frac{e_1 \Rightarrow (v_i \text{ val}, r_i \text{ path}, cs_i)}{\{\ell_1 = e_1, \ldots, \ell_n = e_n\} \Rightarrow (\{\ell_1 = v_1, \ldots, \ell_n = v_n\} \text{ val}, \{\ell_1 = r_1, \ldots, \ell_n = r_n\} \text{ path}, cs_1 +\!\!+ \ldots +\!\!+ cs_n)}\ \textsc{Conc}_2$$

$$\frac{}{i \Rightarrow (i \text{ val}, i \text{ path}, [])}\ \textsc{Conc}_3 \qquad \frac{e \Rightarrow (v \text{ val}, r \text{ path}, cs)}{i \cdot e \Rightarrow (i \cdot v \text{ val}, i \cdot r \text{ path}, cs)}\ \textsc{Conc}_4$$

$$\frac{e_1 \Rightarrow (v_1 \text{ val}, r_1 \text{ path}, cs_1) \quad e_2 \Rightarrow (v_2 \text{ val}, r_2, cs_2)}{o\ e_1\ e_2 \Rightarrow (v_1\ o\ v_2 \text{ val}, o\ r_1\ r_2 \text{ path}, cs_1 +\!\!+ cs_2)}\ \textsc{Conc}_5$$

$$\frac{e_1 \Rightarrow (\lambda x.e, , cs_1) \quad e_2 \Rightarrow (v_2 \text{ val}, r_2 \text{ path}, cs_2) \quad [(v_2, r_2)/x]\ e \Rightarrow (v \text{ val}, r \text{ val}, cs_3)}{e_1.e_2 \Rightarrow (v \text{ val}, r \text{ path}, cs_1 +\!\!+ cs_2 +\!\!+ cs_3)}\ \textsc{Conc}_6$$

$$\frac{e \Rightarrow (v \text{ val}, r \text{ path}, cs) \quad v \not\parallel p_1 \quad \ldots \quad v \not\parallel p_{i-1} \quad v \parallel p_i \quad e_i \Rightarrow (v_i \text{ val}, r_i \text{ path}, cs_i)}{\text{case } e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\} \Rightarrow (v_i \text{ val}, r_i \text{ val}, cs +\!\!+ [\texttt{false}.p_1.r, \ldots, \texttt{true}.p_i.r] +\!\!+ cs_i)}\ \textsc{Conc}_7$$

Figure 2: Adjustments to dynamic semantics of LambdaPix to record path constraints

ment and we record the value of that variable in the constraints, to link the variable with its substructure.

2.1 Evolution

These recorded path constraints are then passed to Z3[1] to generate a new value to pass to the program that is being tested. These values are generated through a three step process:

1. One of the path constraints is falsified, all the path constraints after that one are ignored;
2. The path constraints are then fed into Z3 in two ways:
   (a) First the matching patterns are asserted to be equal and non-matching patterns unequal to the constraint's respective path, including the variables of primitive type present in the pattern;
   (b) Second we generate a new value ignoring the primitive type variables by using the patterns to assert properties about the structure of the paths.
3. The top path constraint variable is then converted back from a Z3 value to a LambdaPix value.

Step 2 is split up because the naive method, the first one, will generally not generate more complex structures because Z3 will generate terms where the numbers are different. To be able to test an interpreter we want to generate structurally complex terms to create interesting cases, these are generated by then purely focusing on the structure of the symbolic values. The combination of these two methods

[1] `https://github.com/Z3Prover/z3`

creates terms that are both structurally complex, which leads to interesting test cases, and terms that follow all paths with conditionals on expressions with numbers.

3 Setup and results

We distinguish several categories of mistakes in definitional interpreters.

1. Structural matching
2. Constant values
3. Mistaken variables

Structural matching is when instead of interpreting the sub-elements of an expression, an interpreter matches on the structure of the input expression. So if an interpreter expects an expression of the form $a + b$ where a and b are numbers, an expression like $(a + b) + c$ should fail, as the left sub-expression would not be successfully matched to a number.

A constant value mistake is when, instead of using the expression provided, a constant value is returned.

An error is categorised as a mistaken variable when an interpreter performs an operation on incorrect variables. For example, in an expression of the form $(a + b)$ instead of adding a to b the following value is computed $a + a$. Another common example, that is not a typo but a conceptual misunderstanding, is specific to the behaviour of functions. This lies in the difference between a dynamic and static scope, where in the interpretation of a function instead of using the closure environment the dynamic environment is used.

To test each category we have created different variants of each mistake, for two languages and compared them with a correct interpreter. The first language is a language of arithmetic expressions, the second one an extension of that language with functions and closures.

The only possible mistakes for the arithmetic language are structural matching and mistaken variables. Given that the language is small all mistaken variable mistakes can be written out. These were all caught by the concolic executor. Structural matching mistakes, up until 2 levels deep, were also caught.

Structural matching mistakes in the functional language, even a simple one where an add expression only expects constants and can not have deeper structure, were not caught by the concolic tester. Some mistakes in category 3 were caught, such as the example provided above, were caught, but more complicated ones like a dynamic scope against static scope did not manage to generate a counterexample.

Mistakes regarding constant values were caught in both languages. The values generated that caught these mistakes were always generated by the first method (2a), which generates terms where the structure is the same but the numbers are randomised.

3.1 Threats to Validity

A limited amount of languages have been tested because the current implementation is limited in its capabilities. It is possible that these limitations are not specific to this case but concolic testing in general. The variants of the interpreters tested may not be representative of actual mistakes made in the wild.

4 Discussion

4.1 Limitations

There are several limitations with the current approach and implementation of concolic execution. There are several reasons for these limitations.

The first reason for the limitations of our approach is the representation of the paths that

have been followed and pattern matching applied to them. Because the symbolic values are also deconstructed, some workarounds were necessary to link sub-expressions with their parent. The current implementation is not complete which leads to insufficient functionality. This limitation can probably be lifted by following the approach of Giantsios et al. (2017) where substructures are referred to by applying accessors, that way no extra effort is necessary to refer to the full structure of a symbolic value.

Secondly there is the representation of values and their semantics in the SMT solver. The semantics of values in LambdaPix differ from the semantics in Z3, especially in the case of a pattern not matching a value. This fact should be taken into account and a proper way of translating the semantics of LambdaPix should be identified.

During the implementation of the concolic executor, the most difficulty was had implementing the generation of new terms. The type system of Z3 is pretty restrictive. It does not have mutually recursive parametric types. Therefore the current implementation does not have full support for all types possible in LambdaPix.

The approach in this paper does not have a sufficiently advanced heuristic for traversing the full tree of all possible execution paths. We traverse the execution tree with a naive breadth first search. This leads to exploration of paths that will not lead to a mistake being found, slowing down testing and maybe even ignoring paths that do lead to mistakes.

Lastly there is the issue of categorising errors. This method of automatic testing allows us to find mistakes in programs without too much manual labour. But it is only able to find these mistakes, it does not classify the type of mistake or suggest what should be done to fix it. If two programs are compared, where one is known or assumed to be correct, the difference in path constraints might be able to provide some hint as to where the mistake lies. This limitation should be kept in mind when using concolic execution to check the correctness of a program.

4.2 Process of Implementation

Concolic testing is popular because of the supposed ease of implementation on an existing language. The existing interpreters or compilers can be used, and easily instrumented to record the constraints on the test input. And given the abundance of powerful SMT solvers generating new values should not take much more effort than converting the constraints into logical formulas and feeding them to such a solver.

This opinion is for the greatest part confirmed by the experience of implementing a concolic executor for LambdaPix. While there are still limitations in the implementation, it is a working proof of concept for a functional language.

Most of the difficulty lay in adapting the methods developed for imperative languages, which rely on values with a simple structure, on structured values. Concolic execution for functional languages are only a recent development (Giantsios et al., 2017; Tikovsky, 2018).

5 Related works

5.1 Challenges and Limitations for concolic testing

Kannavara et al. (2015) write about the challenges facing concolic testing. The most relevant ones to this application of concolic testing is the idea of path explosion and the expensiveness of constraint solving. Given that the defined programming languages for such courses as CPL are well defined allow us to improve constraint solvers and concolic executors

specifically for these languages.

Qu and Robinson (2011) categorise the areas where current limitations lie for concolic testing of imperative languages and quantify the scalability of such tools for industrial use. They find that pointers and native calls are the most common issues for concolic testing and that the setup of such tools takes a considerable amount of effort for industrial size applications.

5.2 Concolic testing of functional programming languages

Giantsios et al. (2017) provide a concolic execution method for Erlang. Their approach is very close to the approach described in this paper, but is more sophisticated in its selection of future contraints to solve and in its representation of values. Given that Erlang is dynamically typed they had to approach the representation of values in Z3, and the way the type of a value is determined, in a manner more suitable for Erlang. Their approach shows that concolic testing is also suitable for functional languages that rely heavily on recursive structures, while previous research primarily shows the effectiveness of concolic testing for imperative languages and primitive types such as integers and booleans. In a similar vein is Tikovsky (2018), implementing a prototype for concolic execution of Curry, a functional logic language.

You et al. (2021) develop a method for testing higher-order functions. SMT solvers do not have theories for these, therefore a workaround in the creation of concolic functions is necessary. They prove that that the method is sound, complete and manages to find mistakes in benchmarks for concolic testing that previous methods have not been able to detect. Although the paper only describes an approach using numbers as the most basic value, it should generalize to recursive data types such as the ones seen in Giantsios et al. (2017).

Hallahan, Xue, Bland, Jhala, and Piskac (2019) describes a method for lazy symbolic execution. It allows verification of lazy languages such as Haskell. They also provide a new technique, counterfactual symbolic execution. This allows not only the verification of programs using pre- and post-conditions but also the verification of those pre- and post-conditions themselves by creating abstract counterexamples to functions that fulfil their specifications.

5.3 Improved heuristics

Jaffar, Murali, and Navas (2013) describe an algorithm that improves concolic testing by pruning the search tree. They mark branches that are, by being fully explored, known to contain no buggy paths with interpolants. When a concrete value satisfies such an interpolant execution is halted at that branch, as it is guaranteed that no bug will be found because it will only be able to follow non-buggy execution paths. The method provides a performance improvement of about 10 to 20 times.

Cha, Hong, Lee, and Oh (2018) present an algorithm that automatically generates search heuristics for concolic testing. They do this by defining a class of parameterised search heuristics and then give an algorithm that finds the parameters for the optimal search heuristic. They show that their method generates heuristics that outperform state of the art and manually crafted heuristics.

6 Responsible Research

This program for comparing interpreters is meant to be used in a course context where student's submissions are evaluated. This entails a few concerns that should be taken into consideration when using the program.

First of all is the fact that the implementation may not be correct, in that sense the re-

sults the program reports should not be taken at face value. For example a mistake in the way new values are constructed could prevent certain paths or constraints to be taken into account, such that some incorrect interpreters are erroneously reported as correct.

Then there is the issue of inherent limitations of the method. The current method, for example, does not allow automatic classification of errors. The system can detect whether an implementation is incorrect, by comparing it to a correct interpreter, but the class of error cannot detected. Additionally, some mistakes can be so complex that to create an input by iteration over the path can take quite a while such that an adequate input may not be generated before a set timeout.

Taking this into account the program should be mainly used to aid the teaching assistants and professors helping students and grading their submissions. Because the program cannot classify errors, or if the difference in results even constitute a mistake in the student's submission, basing a grade only on the programs output would be arbitrary. The technique described in this paper can be used to guide the attention of course staff to problematic locations in student's code, but should not be used without their supervision.

Lastly there is the question of reproducibility. This will be possible given that the code that implements the concepts in this paper is open-source.[2] The code, together with the paper, should provide enough material to be able to be used as a base for reproduction.

7 Conclusion

Concolic execution is a promising way to test variations of interpreters. Certain classes of errors, for which manually generating tests is cumbersome and prone to mistakes, could be caught automatically. This can be seen especially for errors such as structural matching and small typos which lead to a wrong branch being interpreted. Structural matching errors especially have the benefit of concolic or symbolic execution because these methods will generate expressions with the guidance of the constraints, while a predefined set of test will never be complete. More complicated errors, such as a difference in dynamic and static scope will have more difficulty being caught. However these limitations are suspected to lie in the specific implementation used in this paper, and not in the approach of concolic execution in general.

[2] `https://gitlab.ewi.tudelft.nl/cse3000-auto-test/concolic-execution`

References

Cha, S., Hong, S., Lee, J., & Oh, H. (2018). Automatically generating search heuristics for concolic testing. In *Proceedings of the 40th international conference on software engineering* (p. 1244–1254). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3180155.3180166` doi: 10.1145/3180155.3180166

Clune, J., Ramamurthy, V., Martins, R., & Acar, U. A. (2020, November). Program equivalence for assisted grading of functional programs. *Proc. ACM Program. Lang.*, *4*(OOPSLA). Retrieved from `https://doi.org/10.1145/3428239` doi: 10.1145/3428239

Giantsios, A., Papaspyrou, N., & Sagonas, K. (2017, nov). Concolic testing for functional languages. *Science of Computer Programming*, *147*, 109–134. Retrieved from `https://doi.org/10.1016%2Fj.scico.2017.04.008` doi: 10.1016/j.scico.2017.04.008

Griesemer, R., Hu, R., Kokke, W., Lange, J., Taylor, I. L., Toninho, B., … Yoshida, N. (2020, November). Featherweight go. *Proc. ACM Program. Lang.*, *4*(OOPSLA). Retrieved from https://doi.org/10.1145/3428217 doi: 10.1145/3428217

Hallahan, W. T., Xue, A., Bland, M. T., Jhala, R., & Piskac, R. (2019). Lazy counterfactual symbolic execution. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation* (p. 411–424). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3314221.3314618 doi: 10.1145/3314221.3314618

Jaffar, J., Murali, V., & Navas, J. A. (2013). Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering* (p. 48–58). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2491411.2491425 doi: 10.1145/2491411.2491425

Kannavara, R., Havlicek, C. J., Chen, B., Tuttle, M. R., Cong, K., Ray, S., & Xie, F. (2015, jun). Challenges and opportunities with concolic testing. In *2015 national aerospace and electronics conference (NAECON).* IEEE. Retrieved from https://doi.org/10.1109%2Fnaecon.2015.7443099 doi: 10.1109/naecon.2015.7443099

Qu, X., & Robinson, B. (2011, sep). A case study of concolic testing tools and their limitations. In *2011 international symposium on empirical software engineering and measurement* (p. 117-126). IEEE. Retrieved from https://doi.org/10.1109%2Fesem.2011.20 doi: 10.1109/esem.2011.20

Tikovsky, J. R. (2018). Concolic testing of functional logic programs. In D. Seipel, M. Hanus, & S. Abreu (Eds.), *Declarative programming and knowledge management* (pp. 169–186). Cham: Springer International Publishing.

You, S.-H., Findler, R. B., & Dimoulas, C. (2021). Sound and complete concolic testing for higher-order functions. In N. Yoshida (Ed.), *Programming languages and systems* (pp. 635–663). Springer International Publishing. Retrieved from https://doi.org/10.1007/978-3-030-72019-3_23 doi: 10.1007/978-3-030-72019-3_23