Finding Bugs in Interpreters of Toy Languages

Jaemin Hong
jaemin.hong@kaist.ac.kr
KAIST
Daejeon, South Korea

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: interpreters, fuzzing, symbolic execution

ACM Reference Format:
Jaemin Hong. 2020. Finding Bugs in Interpreters of Toy Languages. In *Proceedings of IS893: Special Topics in Security and Privacy <Advanced Software Security>*. ACM, New York, NY, USA, 2 pages.

1 Motivation

The students of the CS320 course at KAIST implements interpreters of the Fiber langauge. Fiber [1] is a dynamically typed functional language that supports integers, booleans, tuples, lists, first-class functions, and recursive functions. While the instructors of the course provide the parser and the desugarer, the students have to implement the main logic of the interpreter by themselves. Their job is to define a function named `interp` that takes an expression and returns the result of evaluating the expression. As the language is tiny, the reference solution consists of only 96 lines of code. However, students often make mistakes in their implementations because they are not familar with programming language concepts and interpreters.

The instructors have tested the student's implementations with hand-written test cases. Even if the instructors carefully designed the test cases, they are not enough to cover every corner case in the interpreters. Therefore, some students could get perfect scores despite their incorrect implementations. We discuss one such example in the following paragraph.

Figure 1 shows a buggy implementation of the `App` case. (The example has been revised a bit for brevity.) The `App` case is reponsible for handling function applications. When a given expression is $e(e_1, \ldots, e_n)$, the expression matches the `App` case. Then, e is bound to `func`, and `List`$(e_1, \ldots, e_n)$ is bound to `args`. A student defined the function `makeEnv` in order to build an environment that is used during the evaluation of the function body. However, the function has a defect: it adds the value of an argument to the environment and passes the environment to the evaluation of the next argument. It means that the value of an ealier argument can be used for the later arguments. For example, `((x, y) => x + y)(1, x)` should result in a free identifier error, but the student's interpreter will yield 2 for the expression. In addition, `fenv ++ env` is the initial argument for `makeEnv`, so the interpreter goes wrong also when the environment for the function application and the environment captured by the closure contain variables of the same name. For instance, the following program should evaluate to 1, while the student's interpreter returns 2.

```
val f = { val x = 1; () => x };
val x = 2;
f()
```

It is nontrivial to expect such bugs and to add proper test cases intentionally. Thus, the student received a perfect score, which is frustrating for the instructors.

The above example clearly shows that test cases are not enough to grade students' submissions. Even though Fiber is a small toy language, manually designed test cases miss peculiar bugs. There must be a more systematic way to detect bugs in interpreters.

2 Approaches

In the literature, various program analysis techniques exist. Despite their improvements in last decades, there are still many obstacles to apply the techniques to large programs. Fortunately, they can be applied to interpreters of toy languages without being concerned by scalability issues. Since most students' implementations consist of less than 200 lines of code, many techniques can analyze the implementations in a reasonable amount of time. In this project, we focus on two techniques: fuzzing and symbolic execution.

2.1 Fuzzing

Fuzzing is a way of testing a target program by running the program with random inputs. Fuzzers can generate numerous inputs that can lead the target program to misbehave. Writing good test cases need human efforts. On the other hand, once a fuzzer is implemented, it can constantly produce new test cases while the time allows. Due to its advantages, fuzzing has succeeded in many areas.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IS893: Special Topics in Security and Privacy <Advanced Software Security>, November, 2020, Daejeon
© 2020 Association for Computing Machinery.

```
def interp(e: Expr, env: Env): Value = e match {
  case App(func, args) =>
    def makeEnv(params: List[String], args: List[Expr], env: Env): Env = (params, args) match {
      case (Nil, Nil) => env
      case (p :: pt, a :: at) => makeEnv(pt, at, env + (p -> interp(a, env)))
      case _ => error()
    }
    interp(func, env) match {
      case CloV(params, body, fenv) => interp(body, makeEnv(params, args, fenv ++ env))
      case _ => error()
    }
  ...
}
```

Figure 1. A buggy implementation of the App case

Fuzzers usually observe whether programs crash or not to find bugs. It allows them to detect bugs without having the exact knowledge about which behaviors are correct. However, many interesting semantic bugs cause incorrect outputs instead of crashes. Bugs in interpreters are typical examples. Interpreters can normally terminate but produce wrong results due to thier bugs.

In this project, the fuzzer will utilize the reference solution. Instead of waiting an interpreter to crash, the fuzzer will run both reference interpreter and student's one. If they behave differently, the fuzzer will judge that there is a bug in the student's implementation. This strategy will make the fuzzer find complex semantic bugs.

Another challenge in fuzzing is generation of syntactically valid inputs. Without enough domain knowledge, fuzzers often generate syntactically invalid inptus, which will be filtered out in the early phase of their target programs. Only a few inputs can reach the main logic of the target programs and get chances to discover semantic bugs. To overcome the limitation, fuzzers usually start with input seed corpus and mutate existing inputs to create new ones.

The fuzzer of this project aims only testing Fiber interpreters. Therefore, it will easily generate syntactically valid inputs by considering the syntax of the language. Every input can be parsed and reach the `interp` function, so the fuzzer will work efficiently.

One challenge of this project is to design an algorithm to generate inputs that can be successfully executed by interpreters. Without enough care, most inputs will cause errors, such as free identifier errors and type errors, though they do follow the syntax of the language. However, some semantic bugs can be found only when input programs are not erroneous. For example, the previous section shows the expression that evaluates to 1 in the reference interpreter but 2 in the student's one. To resolve the issue, the fuzzer will make simple expressions first and then build complex expressions by composing the simple expressions in a proper way. At the same time, expressions that do not make sense at all are also valuable. The previous section shows that an expression with a free identifier may incorrectly evaluate to an integer in the student's implementation. Therefore, the fuzzer will use both strategies: making completely random expressions and building sensible expressions by combining tiny expressions.

2.2 Symbolic execution

Symbolic execution executes a target program with symbolic inputs. It computes a constraint to reach a specific program point and unsatisfy a certain assertion. Presence of a solution for the constraint implies a bug becuase if the solution is used as an input, the program will reach the program point, and the assertion will fail. It can effectively detect bugs by constraint solving, while fuzzing requires lots of trials to find the same input. Alas, symbolic execution often suffers from scalability issues and cannot deal with loops well.

Song et al. [2] proposed automatic detection of semantic bugs in programming assignments. Symbolic execution on both reference solution and student's implementation is their key idea. After extracting results corresponding to constraints by symbolic execution, they use constraint solving to find an input that the reference implementation and the student's one behave differently.

We believe that a similar apporach can be used for finding bugs in Fiber interpreters. We will implement a symbolic execution framework for interpreters and check whether symbolic execution overperforms fuzzing.

References

[1] KAIST PLRG. 2020. FIBER: A Language with Functions, Integers, Booleans, Eagerness, and Recursion. https://github.com/kaist-plrg-cs320/fiber.g8/blob/master/src/main/g8/fiber-spec.pdf. (2020).

[2] Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and scalable detection of logical errors in functional programming assignments. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.