Functional Programming Error Triples for Automated Program Repair:

University of Central Florida

Adedamola R. Adebayo

Gary T. Leavens

Objective:

Automated program repair (APR) has always been a major area of interest for software developers. Having a grasp on how to design or improve automated program repair tools can help programmers rigorously test their code and optimize its usability. This study will focus on gathering a vital ingredient in verifying functional programs through APR, a corpus of common errors. Otherwise known as programming error triples. For purposes of this project, programming error triples are defined as a description of what the program is expected to do and two syntactically correct programs: one which implements the description and the other that does not implement the description.

Background and Literature Review:

Programmers have a penchant for developing programs that are almost correct. Its a rare achievement when a programmer to code up a perfectly working program from scratch. For that matter, it is common to see code that contains errors. When the amount of written code increases in size, it becomes hard to locate errors that distort the program’s goal. The compiler makes quick work of syntax (grammar) errors in the code, but semantic (meaning) errors can be harder to find since the compiler does not know what you want your code to do. APR can be thought of as a compiler for semantic errors. Programmers usually have an idea of the possible errors that might arise in their program. These thoughts can be used to build test cases that give the program different scenarios to load. The output of the program after going through a test case is checked for a match; it passes if the program correctly operates on the test case’s input. The test data (input) put into these cases drives the code through multiple paths to assert its durability. Dr. DeMillo and his partners used a similar approach when collecting data on the results of the mutation testing in Fortran programs. He suggested creating test data that separates different categories of errors in order to test your programs (DeMillo, 1978).

As of today, it is still an expensive endeavor to debug programs. During 2002 in the United States, software bugs cost $59.5 billion on its own. This led Dr. Bybro on his search to find tools that can fix programs. He stumbled upon mutation testing. Mutation testing addressed the coverage problem, how well a piece of software is tested and when to conclude the testing. Fortunately, mutation testing was built on different versions of the same program and an mutation score. Allowing the testing to cover a wide range of situations. Guru99 describes mutation testing as the process of mutating the original program to create mutants and testing all those mutants against an array of test cases (Guru99). Mutant programs differ from the original program by one or more syntactic changes. Such as the change of an operator from less than to greater than. Another instance can be just removing a segment of code. Software Testing Genius declares that mutation testing can be costly and extensive; it is much more ideal for short programs (Software Testing Genius). There has been a lack of mutation testing reports as its not being used much in the workforce due to the reasons mentioned above. Irrespective of that, Dr. Bybro wishes to see more results on how mutation testing will work with software projects so we can further evaluate its performance (Bybro, 2003).

One of the first stages of APR is fault localization. Dr. Le Goues explains this as locating the area in the code that contains the fault (Le Goues, 2017). Dr. Xin mentions the use of code searches such as semantic searches as efficient methods to find the faulty area (Xin, 2017). After the disruptive code has been identified, edits can be taken to redress it. This happens in the search space; the edits can be simple as replacing a function call or passing a different variable. Drawing from fault localization, Dr. Ghanbari worked with his partners to implement an APR tool named PraPr (Ghanbari, 2017). This approach integrated mutation testing and generate-and-validate techniques. Generate-and-validate techniques would alter the code and use test cases to verify the code. Those test cases can be used to create patches (code fix) that restore the program.

A corpus of commons errors (bugs) currently does not exist for functional programs. By collecting a corpus of common errors, the database of programming errors that can be used for APR can grow. The plan is to conduct a literature survey to observe the categories of common programming errors. We can then collect errors and validate the errors by comparing them to the errors in the literature. As well as write a paper detailing the programming errors in the database, demonstrating whether they can be discussed through a category of basic types of errors. The premise is that rudimentary errors such as the off by one or uninitialized variables can lead to the general faults in a program that fails to do its job. This project will test the hypothesis regarding mutation testing that states real errors programmers make are derived from basic errors. Basically, by truncating complex errors we are left with familiar basic errors we have fixes for, making it easier to debug programs.

Methods (300)

The design and analysis of these functional programming errors will be conducted in an online setting. The errors are observed in the functional programming language, Scheme. There are many distributions, but for the sake of simplicity work is done with the Dr. Racket version. Dr. Racket will be used to make the programs that will serve as functional programming error triples. LibreOffice Writer will act as the word processing program to analyze and record the data and a debugger provided by Dr. Racket will aid in rectifying incorrect programs, so they can be eventually fixed. The data collected will be functional programming error triples, so the working program, the program with the error and a specification of what the program must do.

Functional programming exercises from a textbook will be the source of the programs we will work to develop. These programs will form a database which will be evaluated on its size and documentation. The programs will also contain comments about what errors were present. These errors will be recorded in a document in which we will group similar types of errors. The errors will be divided into these groups based on what general solutions could be used to resolve specific sets of errors. We will proceed to compare these errors to the standard types of errors we encounter in Scheme. Then, a literature survey on common programming errors and automated program repair for functional programs will be produced. Along with a paper discussing the common programming errors in the database and whether they can be represented by a context-free grammar of standard types of errors. Also, the paper will seek to interpret the problem and express the database in words. The literature survey will be assessed on its scope, comprehension of literature and usefulness in analyzing the data.

Expected Results (414)

The results will be deduced by discerning how all the errors compare against the basic errors programmers are likely to face. We do that by appropriately classifying them based on the nature of the error and comparing them to the literature. We collected the following errors in Scheme: using an undeclared argument, using an argument as a procedure, missing an argument in a call, giving a procedure the incorrect number of arguments, applying the car procedure to an empty list, and forming a new list as a dotted pair. The errors mentioned above were more general, the next set of errors are more specific to a program’s specification. We also collected these errors: forming a dotted pair between the last two elements, forming a list that contains the first item in only the first sub-list, changing all elements in a list to the old item, and removing the first two occurrences of an item from the list. The expectation is to develop a database filled with lots of programming error triples featuring different errors such as control flow, type and logic errors. These different classifications of errors will lend themselves to code fixes that can solve almost any error in a specific classification.

The error triples could be used in a machine learning algorithm to calibrate or set up an APR tool so it has an idea of the possible errors it might encounter, how to solve them and where to find them. The APR tool will learn, grow and improve its brainstorming and decision-making capabilities by observing programs with errors and the differences they share with the correct version of said program. The data is meant to train the APR tool so it can eventually find and correct errors on its own. Errors root from different causes; if we take time to study them, we can help enhance APR to develop a good eye for detecting anomalies in code. Indeed, previous studies by Dr. DeMillo explain how error classifications is a must for scrutinizing code effectively (DeMillo). This data can be used to forge patches for faulty areas in the program as referenced by Dr. Le Goues (Le Goues). GenProg and PraPr would benefit from this; these APR tools could use some direction in locating the errors in code and determining the correct solution in hopes of restoring the program (Forest). The goal is to use error triples to create stronger APR tools that can be used in the workplace. With the help of APR, coders can spend more time problem solving and developing programs rather than error checking and verifying the code works. The endeavor of programming and the field of computer science will be much more enjoyable for newcomers because they will get to produce creative things instead of scratching their head to find an elusive error. Customers will have more faith that the software they purchase and use is working, free of errors. Companies will gain an economic advantage by releasing applications to the public at a quicker rate. Future work will be based on creating appropriate fixes for errors in a certain classification so APR can work even faster and smarter.

Timeline

* Month 1: Collect programming error triples, read literature on automated program repair and fault localization, half of the literature survey is completed
* Month 2: Collect more programming error triples, read literature on mutation testing and error classifications, the literature survey is completed
* Month 3: Collect more programming error triples, understand how the works apply to the development of a database of programming error triples, the paper is completed
* Months 4 - 8: Collect the last set of programming error triples and add to the database, classify data and finalize conclusions based on it

Budget

* Laptop Computer - Dell, XPS 13, Intel i5-8300H (8th generation), 8 GB RAM, 1 TB SSHD, 2666MHz - $1009.99

(A high-quality laptop will help to run Scheme programs in a quick manner and assist in work that involves several programs operating simultaneously)

References

Bybro, M. (2003). A Mutation Testing Tool for Java Programs. Retrieved from https://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2003/rapporter03/bybro\_mattias\_03083.pdf

Camilo-Junior, C. G., Goues, C. L. & Souza, E. F. (2018) A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints Retrieved from https://squareslab.github.io/materials/deSouzaFitness2018.pdf

Chen, L., Ji, T., Mao, X. & Yi, X. (2016) Automated Program Repair by Using Similar Code Containing Fix Ingredients Retrieved from http://lqchen.github.io/COMPSAC16\_repair.pdf

DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. Retrieved from file:///C:/Users/damol/AppData/Local/Packages/Microsoft.MicrosoftEdge\_8wekyb3d8bbwe/TempState/Downloads/DeMillo-Liption-Sayward78%20.pdf

Forest, S., Goues, C. L., Dewey-Vogt, M. & Weimer, W. (2012). A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. Retrieved from http://dijkstra.eecs.umich.edu/genprog/papers/weimer-icse2012-genprog-preprint.pdf

Ghanbari, A. & Zhang, L. (2017). Practical Program Repair via Bytecode Mutation. Retrieved from https://arxiv.org/pdf/1807.03512.pdf

Goues, C. L., Stepney, S. & Timperley, C.S. (2017) An Investigation into the Use of Mutation

Analysis for Automated Program Repair. Retrieved from https://www.cs.cmu.edu/~clegoues/docs/legoues-ssbse17.pdf

Guru99. (n.d.) Mutation Testing: Complete Guide. Retrieved from https://www.guru99.com/mutation-testing.html

Reiss, S. P. & Xin, Q. (2017) Leveraging Syntax-Related Code for Automated Program Repair Retrieved from http://cs.brown.edu/people/qxin/papers/repair\_ase17\_preprint.pdf

Software Testing Genius (n.d.) Mutation Testing and Error Seeding-White Box Testing Techniques. Retrieved from https://www.softwaretestinggenius.com/mutation-testing-and-error-seeding-white-box-testing-techniques/