Functional Programming Error Triples for Automated Program Repair:

Literature Survey

Adedamola R. Adebayo

Gary T. Leavens

Programmers usually make various types of errors when creating a program. Automated program repair (APR) is meant to be a mechanism to correct these errors in order to produce functioning programs. Through the three-step process of localizing the fault, creating patches and verifying the quality of the patches, APR can successfully solve errors in programs. This is the ideal situation; however, the reality is APR research is currently in its early phase. More related data needs to be collected to see how we can improve the results of each individual step, so we can reach the objective soundly. This literature survey will provide insight on the common errors that programmers face, a categorization of the errors and how the errors help APR do its job.

In Practical Program Repair via Byetcode Mutation, Dr. Zhang and Dr. Ghanbari worked on implementing PraPr, a practical APR tool that uses bytecode mutation to solve real life errors in programs. PraPr was made specifically to correct Java programs. It is different from state-of-the-art APR tools, but it offers its own benefits such as not needing to verify patches by compiling the program. This can drastically reduce our search times in the search space, the domain that contains all the possible repairs we can try. With the aid of mutation operators, PraPr is able to produce several suitable patches for a program. PraPr is largely mutation based; the notion is that mutation operators can reflect programming errors, so mutating a faulty program can mend it too. A set of test cases are built to test the faulty program in order to assert if there is a bug. The program is mutated and tested again with test cases to find a viable solution. If the program manages to pass all the test cases, then it contains a possible solution and becomes a candidate patch. The tool would create many patches based on problematic segments of code.

After the mutations, they were left with programs that contained common errors programmers face. By applying a single or multiple mutation operators the following errors in a program can result negating a boolean expression causing an unintended effect, using the wrong arithmetic operator for a given equation or dereferencing a variable that holds NULL or an unknown address. Other errors exist such as calling the wrong function name, calling the wrong function under the same name otherwise known as overloading. As well as using the wrong variable to do math or check a condition or using a switch statement without the addition of break or return statements triggering the unexpected fall-through behavior. Lastly, the error could be enforcing the wrong condition inside an if or switch statement or a loop. These errors can subtly be divided into groups defined in the paper as arithmetic operator, conditional, dereference guard, method name, local variable and case breaker. These errors may be specific to Java, but they still provide us a context of the general errors programmers tackle on a daily basis to get their programs to work.

A Model-Based Approach to Software Debugging and Deductive Program Repair also introduced some common errors too such as leaving out an operator, missing a case in a switch or if statement, swapping arguments in a function or mixing up the order of an if and else structure. Some of these errors are definitely present in functional programming languages too. PraPR’s performance proved to be much more effective in terms of the number of valid patches it produced based on the given errors, outdoing state-of-the-art APR tools at their own game. There have been different approaches on how APR should correct errors in a efficient way. Generate and validate (G & V) seeks to find a version of the program that will pass certain sets of test cases. G & V starts by localizing the area in the code that contains the fault. It delegates suspicion values to specific statements in the code that cause it to fail test cases; these areas are most likely derailing the program from completing its goal. These statements are subjected to changes that are transformations taken from the search space in order to correct the code such as replacing a parameter or inserting a statement. APR tools can differ in how they create patches, some can build patches based on human fixes, others may play around with code by deleting or adding statements in order to get things to work. These simple changes can work for basic bugs, but for more complex bugs a more pragmatic method may need to be devised.

In An Investigation into the Use of Mutation Analysis for Automated Program Repair, Dr. Timperley, Dr. Stepney and Dr. Le Goues determined the effects mutation analysis had on APR. Corresponding suspicion values were assigned to mutants programs and their individual statements of code. Series of edit operations could be made to resolve an incorrect program; they could be in the form insertion, deletion or replacement of one or more statements depending on what the possible error is and how it hinders the program from following the specification. To apply these edits, a location must be found first. An area would be identified where the change would most likely make a significant impact; likely to be near or at a faulty area of code within the program.

In A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8, Dr. Le Goues, Dr. Forrest, Dr. Dewey-Vogt and Dr. Weimer investigated how they can improve GenProg to make it more productive while lowering the cost of its operations. GenProg is an APR tool that utilizes genetic programming to correct errors in legacy code without the need of a specification of the program’s goal. GenProg uses a search space that contains available operations or transformations it can apply to remedy the code. The search space increases in proportion to the lines of code, but at a higher rate. GenProg needs to be smart in how it locates fixing schemes in the search space to solve the problem. To find the appropriate transformations, GenProg narrows the search to focus on the faulty areas in the code and using preexisting code as a base to forge and pick adequate repairs. They sought to preserve the functionality of the programs being operated on while aiming repair efforts at statements that are affiliated with negative test cases when the program was not doing a task that was expected of it according to the specification. Mutations were made by swapping, deleting or inserting statements or conditions in a program. GenProg needs a set of test cases and developer approval of patches to function but is cost effective in terms of creating patches. GenProg makes swift fixes; giving developers time and guidance to figure out permanent solutions. GenProg’s approach to APR can be seen as behavioral as well; it employs methods that seek to tamper with code and state-based techniques that attempt to create a conducive environment for the code to run. GenProg is also capable of implementing a checkpoint algorithm that places debug statements throughout the program to track its execution path and the values contained in the variables. This will allow us to see how certain statements affect the values inside variables. This is vital if the program is misbehaving due to variables that are holding the wrong values.

In Automated Program Repair by Using Similar Code Containing Fix Ingredients, Ji, Chen, Mao and Yi observed that programming can sometimes follow the pattern of reusing such as copying and pasting code. This can cause the same errors to occur in perhaps similar segments of code. Thus, it would logical to recycle fix operations or code if there are no suitable fixes available to correct a program.

References:

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

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

Kneuss, E., Koukoutos, M. & Kuncak, V. (2015) Deductive Program Repair. Retrieved from https://link.springer.com/chapter/10.1007/978-3-319-21668-3\_13

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

Stumptner, M. & Wotawa, F. (1996) A Model-Based Approach to Software Debugging. Retrieved from https://pdfs.semanticscholar.org/b2d5/b87e8f727dfd5ee61c92c73fdb7f5f19e91f.pdf