Alex Davis: ahd212
Autumn Blackburn: blackburn37

Motivation

Computer programs have errors and as we know these errors can be expensive when they make it into a final version. Program verification is a way to prove that a program is free of some of these costly errors, however it is expensive to have trained programmers spend the time to evaluate every line of code looking for these errors. This is the way verification is handled today. There are existing tools that perform some amount of verification checking, but none that can for 100% of a program.

The main idea of Verification Games is to transform the task of code verification into a game that can be completed with no special training where a solution to a level provides valuable verification insight. This game can then be deployed to a large audience, crowdsourcing the laborious task of code verification to a population that doesn't need to be trained for it. Pipe Jam (fig. 1), a game developed by a team at the University of Washington, is one such game that uses a series of differently-sized pipes and balls to represent the variables and values, respectively, of a program. The goal is to make a configuration of pipes that allows the balls to flow through the whole system by manipulating the width of the pipes. If the player cannot find a solution, then they can use a "buzzsaw" to make the balls small enough to flow through the pipes, which identifies possible problem areas that need to be evaluated by a trained programmer. Currently, however, it is only capable of representing small programs because the number of variables, values, and dependencies quickly becomes unwieldy in meaningful programs. This limits the power of the tool considerably because the number of pipes would quickly become unmanageable for a player to deal with in larger programs.

Crowdsourcing the main tedious aspects of code verification, i.e. the reading and parsing of hundreds or thousands of lines of code, and identifying potential problem areas so that skilled engineers can examine them is the main motivation behind developing this sort of system. If this could be scaled up to be able to verify large programs, valuable developer time could be spent on other tasks.

Approach

Our approach to the problem of making Pipe Jam usable on large scale programs would be as the abstract of the project suggests: working on creating a way to either break up large programs so small parts can be evaluated for the constraint or to preprocess the programs in a way that makes them small enough to be handled by the current Pipe Jam program. Another approach to managing the complexity of the game would be to let users decide on the difficulty of the game they want to play: smaller programs would necessarily make for easier challenges while larger programs would be the opposite. Play testing would be needed to determine how complex the puzzles can become before becoming too cumbersome to be solved by even a patient player. The game needs to be fun and engaging as well as providing data about program verification.

We assume (given the previously mentioned paper) that much of the game's code and thus its architecture is written at this point and so we would just be extending that codebase. This includes some method of determining when an input program is too large to be a feasible single world and must be broken down further. Depending on the input, some classes may also need to be broken down further as well to create reasonable problems for the players to solve. This could be determined by some measure of the complexity of the examined code, for example by number of variables, values, or procedure calls.

Challenges and Risks

We believe that the most serious challenge to scaling up Pipe Jam is finding a way to simplify large programs so that the solution found by users in the smaller version guarantees that the original program fulfills the same specification. The risk of this is that the verification provided by Pipe Jam is not sound. Mitigating this will require in-depth knowledge of formal verification to prove our solution is still correct.