Representative Example Generation for Cooperative Programming

Marvin Qian
Advisor: Ruzica Piskac

Background

Cooperative programming is a proposed programming environment that seeks to augment the live programming paradigm by allowing users to trigger code synthesis and repair by modifying the intended output of examples. The term live programming has been used to describe a variety of practices, but the way it is used in this proposal is to describe the practice of having immediate feedback on code changes. This is in contrast to how software engineers develop software today which usually involves writing the code, compiling it, and then finally testing it. The benefit of live programming is that programmers will be able to see how they are changing the execution of the code they are modifying through numerous example executions.

Cooperative programming takes this idea to the next level by making the code-example interaction bidirectional. In addition to modifying the code to fix the output of the examples, the programmer can also modify the output of the examples to fix the code. The newly synthesized code will conform with the modified examples and bring the programmer one step closer to creating a correct program. There are many components to creating a cooperative programming environment such as generating representative examples, error localization, error repair, code synthesis, tool development, and editor integration. My project intends to focus on the first step, generating representative examples.
Motivation

Cooperative programming benefits novice and experienced programmers alike. For experienced programmers, it improves the speed at which he or she can iterate and improve his or her code through immediate feedback. Another advantage of cooperative programming is that the outputs of examples can help programmers understand the behavior of a program which can help novices learn how to program or veterans comprehend obfuscated or poorly maintained code.

The project targets Haskell, a modern, functional programming language. While Haskell is not as widespread as established imperative languages like Java or Python, it is rapidly increasing in popularity and can serve as an excellent introduction to functional programming or programming in general. Due to their lack of side effects and immutable data structures, functional languages have recently made a resurgence in response to the growing need for concurrency when scaling applications. A cooperative programming environment for Haskell will lower the barrier to entry and promote the learning and usage of functional languages.

Overview

At the heart of live programming and cooperative programming are examples. The examples are the source of feedback as well as the initiator of code synthesis. Thus the quality of examples directly influences the quality of feedback as well as the potential avenues for code synthesis.

The examples presented to the user will be most effective if they are a representative of possible execution paths that the program might take instead of
completely random or redundant examples. Ideally, the provided examples will cover all branches of the control flow graph so that every portion of the program is tested. To explore all branches, one can utilize concolic execution. Concolic is a portmanteau of concrete and symbolic execution which involves trying a random concrete input while following the execution symbolically to obtain the constraints along that execution path. The last constraint is then negated and sent, along with the other constraints, to a solver to find a new input that will follow a different execution path. This process then continues until input examples are generated for all paths or all paths up to a certain depth. The end result will be a set of input examples and their outputs that comprehensively represents the program’s behavior.

While concolic execution would be ideal, the implementation is tricky and may not be the quickest way to get the project off the ground. Since the other portions of the cooperative programming project requires some set of examples to get started, my project will start by using randomly generated examples. I plan on using SmartCheck (an improvement on QuickCheck) to create random examples and HPC (Haskell program coverage) to analyze the coverage of the generated examples so only one or two examples are selected per branch. Once I have those results, I will iterate on the example generation process. If using random examples provides near sufficient coverage in practice, then I may focus on clever heuristics to optimize the random example generation. On the other hand, if random examples are not satisfactory, I will explore using concolic execution.

Concolic execution would require a symbolic execution engine, but no such package exists for Haskell. Possible alternatives are to use a symbolic execution engine
that targets an intermediate language used by the Haskell compiler such as C or LLVM (one possibility is Scher which uses KLEE for LLVM, but it's still very experimental). If that approach does not work, then I will investigate writing a symbolic execution engine for Haskell or its intermediate language, Core. This route will likely be based off of CutEr, a concolic testing tool built for another functional language, Erlang.

Some other challenges in generating useful examples include polymorphism, higher order functions, special data structures, and compatibility with synthesis. For example, polymorphic functions take many different types so one approach could be to select common concrete types and generate examples for each of those types for each branch. Another possible issue involves synthesis since having dependent examples could cause confusion during synthesis. Suppose that two examples follow the same execution path, but the user only changes the intended output of one example. The newly synthesized code will then have to branch between those two examples to conform with the specified output even though they should actually be on the same branch.

**Deliverables**

At the end of the semester, I plan to have implemented a Haskell module that can return a set of representative examples given a Haskell program as input. Time permitting, I will try to integrate my module into the programming environment being developed by other members of the project team so that we can have a simple live or cooperative programming demo.