Yisu Wang remywang 1723358

Issue 1: unit tests are helpful to ensure correctness of program and identify problems at an early stage. However, it is difficult to write unit tests for certain programs. One example is a program with side effects, e.g. printing a picture to the screen. It is much easier to manually run the program, look at the picture to see whether it is correct, than to write an oracle that describes what a correct picture looks like.

My solution: I would use a purely functional language and organize my program to separate pure functions from side-effecting functions as much as possible. The former can be easily tested in units, and there will be fewer effects to test. However, this does not solve the problem, if the specific software requires a lot of side effects.

Current approaches: There exists research on algebraic effects that aim to formally verify side-effecting programs. But the mathematics involved is too difficult for a regular programmer, as is the case of most current theorem provers.

There are also tools that automatically generate test oracles from examples. But they only generate oracles for whole-program testing instead of unit tests. The reason is, it is more difficult to demonstrate how a component of a system works than to demonstrate how the system as a whole works. For example, one can demonstrate a GUI by simply using it, but one cannot "simply use" part of the GUI code.

Proposed solution: building on current tools that generate whole program test oracles, we can build a tool that generate oracles and driver programs for unit tests as follows: 1. Run the generated whole program oracles and collect data that flow through each function. 2. For each function, automatically generate driver programs / test suites that call the function on the collected inputs. Furthermore, one can generalize the unit tests by iteratively running the test generation tool on the driver programs generated.

Issue 2: to understand a large system, the developer needs to know: 1. In the physical world, what the system is supposed to do, say rendering an image / align DNA sequences (domain knowledge) 2. From the digital world, how a command correspond to a physical action (knowledge about atomic code) 3. How a group of commands compose to perform a group of actions (knowledge about how atomic code compose). Usually, the developer can understand atomic pieces of code from the documentation. S/he can also run the whole system with different test inputs to understand the system's behavior as a monolithic piece. But it is difficult to understand a group of code that make up only parts of the system, because it is not documented and cannot be directly executed separately.

My solution: I would search for similar coding patterns in the code base, perhaps with grep and a regex that describes the pattern. Or perhaps I might even search for the pattern on the

internet, with the function names involved. I would also try to write a minimal program that uses that coding pattern and run it with different inputs.

Current approaches: in a debugger, the developer can direct the program execution to the desired part of code with special inputs via trial and error. But this can be less feasible and frustrating if the codebase is large. If the group of code spread across different functions or even different files, this becomes even more challenging.

Proposed solution: build an interface, perhaps a text editor / IDE plugin: the developer selects a fragment of code, upon which the interface brings up many other code with similar patterns from within the same repo or even from the internet. To implement the interface, we can generate regex (or some regex-like query language) from the code selection and pass the query to grep.

Issue 3: software that has existed for a long time can be difficult to use and maintain. First, heavy optimizations obscure implementation. Second, layers of features complicate the interface. C++ and linux are examples.

My solution: for heavily optimized code, I would try to write out a pseudocode that describes the program behavior in a simple yet structured way. For code with many features, I would go back its commit history to find the earliest implementation, and see how the features got added gradually.

Current approaches: there has long been deobfuscation tools. But they usually generate code in the same language as the obfuscated code, which can be less helpful if the language is foreign to the developer. E.g., a heavily optimized code could be in assembly code. There is also tools that assist in understanding multi-layered applications, but they are aimed at code with multiple layers each for a particular task (for example user interface, databases, business process).

Proposed solution: for the deobfuscation part, either develop or pick a language that can encode high-level implementation straightforwardly. Then try to compile the obfuscated code into that language. This can be a direct application of verified lifting, which uses program synthesis to search for a program in language A that implements another program in language B.

For the second part, I imagine a system that uses some program differencing algorithm to compare the source code between commits. Then the system maps the original, simpler program to part of the current code. Similarly, each layer of feature is mapped to a fragment of the current source. In the end, the developer can view each feature in separation, and opt to disable / enable certain features during trial runs of the program.