COMP 161 - Lecture Notes - 19 - Evaluating Programs: Correctness and Simplicity

Spring 2014

In these notes we begin our exploration ration of the object evaluation of programs.

Measurable Quality

If the goal of programming is to develop correct, simple, and efficient programs, then we need solid, objective means of evaluating and measuring the correctness, simplicity, and efficiency of our finished product. Objectivity is critical because our notions of correct, simple, and efficient, must not be a moving target. All programmers should, in theory, understand what "this is an efficient program" means regardless of the application or language. Our ability to objectively measure the correctness, simplicity, and efficiency is critical if we wish to compare and contrast programs. To the degree that programmers are engineers, universally agreed upon, measures or quality allow us to understand and improve upon programs.

As a disciple rooted in mathematics and science, we must take on the mantle of mathematician and scientist. From this perspective, correctness, simplicity, and efficiency, are

Theory and Practice

Empirical observations and theoretical models of fundamental principles.

Programs exist in the real-world. They execute on specific devices with specific characteristics. Thus, we can write code, execute that code, and take measurements of the program's performance characteristics. This leaves open the questions of why we're seeing what we're seeing and what we can do improve performance?

Correctness

A program is correct if it produces the expected behavior for all possible inputs. Put another way, a program should be 100% free of errors and bugs for all of it's allowable inputs. If we can guarantee that a program is only run with an allowed input, then we can say that a correct program is 100% free of bugs and errors. This is a wonderfully flexible notion of correctness. It allows the programmer to identify expected behavior and possible inputs and thereby set the context and benchmark for correctness. At the same time, it makes it clear that programs that do not do what we say they'll do when we say they'll do it are emphatically *incorrect*. Unfortunately, programmers too often define expectation and the set of possible inputs in vague and ambiguous manner. When they do not set a clear target for correctness, then they never achieve it. Sometimes we just program, "until it works" with out clearly identifying what "works" is, or or definition of "work" is too vague of a target and is not something we can actually measure. If, for example, we say our hangman game is done when it plays hangman, then our target is not a program or computer systems notion of correctness, but something we draw from the problem. As programmers, we have to translate the problem we're solving into program terms. This includes what a complete solution to the problem looks like.

Measuring Correctness

In theory, we can simply run a program on all possible inputs and see if it performs as expected. In practice this is usually impossible. Either the set of inputs is infinite or it's so large that running the tests would take too long[1]. Absent of the use of such methods and tools, we can do certain things to obtain some weaker measures of correctness.

[1] a lifetime or more

- In languages that use annotated types, like C++, a program that compiles does not make calls to procedures that violate a procedures signature. This means that, as far as the computer is concerned, procedures are always run with inputs of the right type.
- We can use unit tests and other testing frameworks to show that our programs correctly function on some small subset of the possible inputs.
- We can use tools like Valgrind to verify the absences of certain run-time errors on a small subset of the possible inputs.

Simplicity

Efficiency