PERSONAL STATEMENT

ETHAN FAST

How much of a software developer's work might be replaced by computation?

My research explores the boundaries of automated software development, investigating to what extent computers can assist humans in writing code. In particular, I am concerned with how techniques from machine learning might be applied to applications in programming languages. Along these lines, I am especially interested in working with **Professor Engler** on automatic bug detection, **Professor Aiken** on statistical debugging techniques, or **Professor Koller** on applications of machine learning. By combining statistical learning methods with elements of semantic and program analyses, I hope to construct algorithms that reduce the cost of developing reliable code.

As a member of the Automatic Program Repair (APR) research group, I have already taken initial steps toward these goals. APR is a technique for fixing program bugs. Provided with a test suite, a known bug, and program source code, it evolves repairs automatically using Genetic Programming (GP). An analog to Darwinian evolution, GP is a machine learning technique that optimizes a population of computer programs on the metric given by a fitness function. For APR, this function is a weighted combination of regression test cases. Over a number of generations, APR applies mutations to a population of candidate programs, and guided by the fitness function, it eventually converges on a repair. Upon joining the project, I was quite impressed by APR's success, as the technique has generated repairs for over twenty real-world programs and from a variety of application areas.

However, I soon noticed a weakness in APR. It tends to be slow, as often many test cases must be run before it finds a repair. These test cases guide evolution and validate repairs, but since each variant program is evaluated on all of them, test cases also reduce the number of candidate solutions you can explore. In several cases, APR took days to return results for the various experiments I was running. Motivated by this problem, and understanding that genetic algorithms often have a high tolerance for noisy fitness functions, I suspected that sampling test cases within each fitness evaluation might provide substantial savings in run time, with no cost to correctness so long as any final repair is vetted against the test suite in its entirety. Working with Professor Westley Weimer, I developed this new sampling-based fitness function, and in line with my initial intuition, our technique has increased performance by 81% on average, repairing programs with hundreds of test cases in minutes. Additionally, repair times grow more slowly than test suite size, suggesting that large test suites will no longer serve as a bottleneck in the repair process. In this way, we have demonstrated repairs for new programs that were previously intractable under reasonable time constraints.

Last July, I presented these results in our paper, ***Designing better fitness functions for automatic program repair***, at the *Genetic and Evolutionary Computing Conference 2010* (GECCO). As first author, I designed and ran each of our experiments concerning test suite sampling, and wrote the first draft of our text. I chose our benchmark programs and test suites from a variety of application areas (scientific computing, image processing, web serving, etc.) and integrated each into a form accessible to our repair method. In running our various experiments, I also designed scripts to automate the process. This has proven

much to my advantage, as I have collected data on over 1,000,000 variants over the course of my research. Further, while our performance data was strong, I contributed to isolating the underlying causes of our results. For instance, I measured our repair metrics against the contributions of test case independence and the results of a change impact analysis, weakening alternative explanations for the benefits of our technique. I planned out these experiments and others, writing various other scripts to aggregate and analyze our data.

The following summer, I continued investigating APR at the Santa Fe Institute (SFI). With Professor Stephanie Forrest, I worked to quantify and explain the mutational robustness of software. Broadly, mutational robustness describes how likely a phenotype (for APR, program behavior) is to remain constant in response to mutations applied within its genotype (an abstract syntax tree). In an evolutionary system, there is a tradeoff between robustness and evolvability, so for APR to work well some amount of robustness is necessary to avoid solutions stuck at local maxima. However, if individuals are fully robust, no evolution may take place at all. With this balance in mind, I asked myself a question: just how mutationally robust is software in general? I expected that software must be somewhat robust — as clearly APR works well for many program bugs — but that such robustness would be rather minimal. After all, making arbitrary mutations to program code would seem almost certain to affect program behavior. Running experiments across a wide variety of popular open source software, I was surprised to find high levels of mutational robustness for all tested programs. This means that many of the mutations a program may undergo do not appear to affect its functionality. Further, while it is possible that regression test suites are not an adequate means of characterizing program behavior, several experiments have suggested that code coverage and test suite size — metrics often used to evaluate the efficacy of a test suite — are not correlated with these robustness numbers. On the whole, I am still working to isolate the underlying causes of this phenomenon, but these initial findings were presented in a talk I gave to SFI, called *Mutational Robustness and Automatic Program Repair*.

In working with Professors Weimer and Forrest, I have learned much about what it means to produce high quality research, and I have come to realize that I will need a PhD to work on the cutting-edge problems with which I am interested. Moreover, I expect that I would thrive at Stanford. My contributions to APR occurred at many levels of the research process, from high-level brainstorming and hypothesis formation, to the logistics of experimental design and academic writing. Further, I have published work contributing directly to the state of knowledge in programming languages and machine learning. In short, I have discovered a passion for that deep level of understanding enabled uniquely by research, and as a graduate student at Stanford, I would have the resources and mentorship necessary to pursue this passion fully.