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. By combining statistical learning methods with elements of semantic and program analyses, I hope to construct algorithms that reduce the cost of developing reliable programs.

To this end, my work on the Automatic Program Repair (APR) project has already pushed forward the boundaries of automated software development. APR is a technique for debugging software which has generated repairs for over twenty real-world programs from a variety of application areas. Provided with a test suite, a known bug, and program source code, APR evolves these repairs automatically through an analog to Darwinian evolution called Genetic Programming (GP). Broadly, GP is a machine learning technique that optimizes a population of computer programs on a given metric, or fitness function. For APR, the fitness function is a weighted combination of regression tests, and it controls both scalability and correctness. Test cases validate repairs, ensuring that required program functionality is retained. Moreover, since the fitness of a candidate repair is evaluated on all test cases, they also limit performance, reducing the number of candidates you can explore. As genetic algorithms often have a high tolerance for noise, I believed 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 Weimer, I developed this new sampling-based fitness function. In line with my initial intuition, our technique lowers costs by 81%, repairing programs with hundreds of test cases in minutes. In addition, repair times grow more slowly than test suite size, suggesting that by applying test case sampling, large test suites will no longer serve as a bottleneck in the repair process.

Last July, I presented our paper on this work at the *Genetic and Evolutionary Computing Conference 2010* (GECCO), titled *Designing better fitness functions for automatic program repair*. As first author, I designed and ran each of our experiments concerning test suite sampling, and had a significant role in writing the paper's text. I chose our benchmark programs and test suites from a variety of application areas (scientific computing, image processing, web serving, ect.) and integrated each into a form usable by our repair method. In running these various experiments, I designed scripts to further automate our repair process. This decision proved very much to my advantage, as over the course of my research, I have collected data on over 1,000,000 variants. Further, while our demonstrated performance data was strong, I contributed to isolating the underlying causes of our results. For instance, I measured our repair metrics against the possible contributions of test case independence and change impact analysis, and weakened 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.

My investigations into APR continued last summer at the Santa Fe Institute (SFI). With Professor Forrest, I worked to quantify and explain the mutational robustness of software. 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. I expected that software would be somewhat robust — 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 quite surprised to find that mutational robustness is in fact quite high, exceeding 30% for all tested programs. This means that about a third of the mutations a program may undergo do not affect on its functionality. 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*.

As interdisciplinary research, my work has made use of concepts from programming languages, software engineering, and machine learning. In like manner, my interests are somewhat broad. [School Specifics]

From my work with Professors Weimer and Forrest, I have learned much about what it means to produce high quality research, and so I am certain that I will thrive at [School]. My contributions to APR occurred at every level of the research process, from high-level brainstorming and hypothesis formation, to the logistics of experimental design and academic writing. Moreover, I have published work contributing directly to the state of knowledge in programming languages and machine learning. Over the past few years, I have discovered a passion for the deep level of understanding uniquely enabled by research, and as a graduate student at [School], I would have the resources and mentorship necessary to pursue this passion fully.