The Ph.D. Grind

A Ph.D. Student Memoir

Philip J. Guo

philip@pgbovine.net

www.pgbovine.net

June 29, 2012

To everyone who aspires to create.

Contents

Preface

This book chronicles my six years of working towards a Ph.D. in computer science at Stanford University from 2006 to 2012. A diverse variety of people can benefit from reading it, including:

- undergraduates who might be interested in pursuing a Ph.D.,
- current Ph.D. students who are seeking guidance or inspiration,
- professors who want to better understand Ph.D. students,
- employers who hire and manage people with Ph.D. degrees,
- professionals working in any creative or competitive field where self-driven initiative is crucial,
- and educated adults (or precocious kids) who are curious about how academic research is produced.

The Ph.D. Grind differs from existing Ph.D.-related writings due to its unique format, timeliness, and tone:

Format – *The Ph.D. Grind* is a memoir for a general educated audience, not a "how-to guide" for current Ph.D. students. Although Ph.D. students can glean lessons from my experiences, my goal is not to explicitly provide advice. There are plenty of how-to guides and advice columns for Ph.D. students, and I am not interested in contributing to the fray. These articles are filled with generalities

such as "*be persistent*" and "*make some progress every day,*" but an advantage of the memoir format is that I can be concrete and detailed when telling my own story.

Timeliness – I wrote *The Ph.D. Grind* immediately after finishing my Ph.D., which is the ideal time for such a memoir. In contrast, current Ph.D. students cannot reflect on the entirety of their experiences like I can, and senior researchers who attempt to reflect back on their Ph.D. years might suffer from selective hindsight.

Tone – Although it's impossible to be unbiased, I try to maintain a balanced tone throughout *The Ph.D. Grind*. In contrast, many people who write Ph.D.-related articles, books, or comics are either:

- successful professors or research scientists who pontificate stately advice, adopting the tone of "*grad school is tough, but it's a delectable intellectual journey that you should enjoy and make the most of . . . because I sure did!*"
- or bitter Ph.D. graduates/dropouts who have been traumatized by their experiences, adopting a melodramatic, disillusioned, self-loating tone of "*ahhh my world was a living hell, what did I do with my life?!?*"

Stately advice can motivate some students, and bitter whining might help distressed students to commiserate, but a general audience will probably not be receptive to either extreme.

Finally, before I begin my story, I want to emphasize that there is a great deal of diversity in Ph.D. student experiences depending on one's school, department, field of study, and funding situation. I feel very fortunate that I have been granted so much freedom and autonomy throughout my Ph.D. years; I know students who have experienced far more restrictions. My story is only a single data point, so what I present might not generalize. However, I will try my best to avoid being overly specific. Happy reading!

Philip Guo, June 2012

Prologue

Since I majored in Electrical Engineering and Computer Science in college, the majority of my classmates started working in engineering jobs immediately after graduating with either a bachelor's or master's degree. I chose to pursue a Ph.D. instead due to a combination of subliminal parental influences and my own negative experiences with engineering internships throughout college.

My parents never pressured me to pursue a Ph.D., but I could tell that the job they respected the most was that of a tenured university professor, and a Ph.D. was required for that job. Why was being a professor regarded as their golden ideal? It wasn't due to some lofty reverence for the purity of scholarly pursuits. Although my parents respected intellectuals, they were highly pragmatic immigrants who were more captivated by the lifetime job security offered by a tenured professorship.

Many of my parents' friends were Chinese immigrants who worked in corporate engineering jobs. Due to their weak English language skills and lack of American cultural literacy, they mostly had negative experiences throughout their engineering careers, especially as they grew older. At holiday parties, I would constantly hear jaded-sounding stories of people suffering under oppressive managers, encountering age discrimination and "glass ceiling" effects, and facing massive rounds of layoffs followed by prolonged unemployment. Although my father was not an engineer, he worked in the high-tech sector and had similar

tales of struggling with management and bureaucracy, culminating in his final corporate layoff at the relatively young age of 45.

My mother was the only exception to this dismal trend. She loved her job as a tenured professor of sociology at UCLA. Unlike most of her Chinese immigrant friends, she enjoyed lifetime job security, never needed to report to a boss, could pursue her own intellectual interests with nearly full freedom, and was famous within her academic field. Seeing the stark contrast between my mother's successful career trajectory and the professional downward spirals of my father and many of their friends made a lasting impression on me throughout my high school and college years.

Of course, it would be foolish to pursue a Ph.D. solely out of irrational childhood fears. To get a preview of corporate working life, I did internships at engineering companies every summer during college. Since I happened to work in offices where I was the only intern, I was given the full responsibilities of a junior engineer, which was a rare privilege. Although I learned a lot of technical skills, I found the day-to-day work to be mind-numbingly dull. My coworkers were also unenthusiastic about their jobs, and there were few appealing prospects for career advancement. Of course, I'm not claiming that *all* engineering jobs are mind-numbingly dull; it just happened that the companies I worked for were not first-rate. Many of my college friends who interned at first-rate companies such as Microsoft and Google loved their experiences and signed on to work at those companies full-time after graduation.

Since I felt bored by my engineering internships and somewhat enjoyed my time as an undergraduate teaching and research assistant back in college, I set my sights on university-level teaching and academic research as future career goals. By the middle of my third year of college at MIT, I had made up my mind to pursue a Ph.D. degree since it was required for those kinds of jobs. I planned to stay at MIT for a five-year combined bachelor's and master's program, since that

would give me more research experience before applying to Ph.D. programs and hopefully increase my chances of admissions into top-ranked departments.

I found a master's thesis advisor and, like any ambitious kid, began proposing my own half-baked quasi-research project ideas to him. My advisor patiently humored me but ultimately persuaded me to work on more mainstream kinds of research that fit both his academic interests and, more importantly, the conditions of his grant funding. Since my master's program tuition was partially paid for by a *research grant* that my advisor had won from the U.S. government, I was obliged to work on projects within the scope of that grant. Thus, I followed his suggestions and spent two and a half years creating new kinds of prototype tools to analyze the run-time behavior of computer programs written in the C and C++ languages.

Although I wasn't passionately in love with my master's thesis project, it turned out that aligning with my advisor's research interests was a wise decision: Under his strong guidance, I was able to publish two papers—one where I was listed as the first (lead) author and the other a latter author—and write a master's thesis that won the annual department *Best Thesis Award*. These accomplishments, along with my advisor's help in crafting my application essays, won me admissions into several top-ranked computer science Ph.D. programs. Since Stanford was my top choice, I felt ecstatic and could barely sleep during the night when I received my admissions notice.

I was also lucky enough to win the prestigious NSF and NDSEG graduate research fellowships, each of which was awarded to only around five percent of all applicants. These two fellowships fully paid for five out of the six years of my Ph.D. studies and freed me from the obligations of working on specific grant-funded projects. In contrast, most Ph.D. students in my field are funded by a combination of professor-provided grants and by serving as teaching assistants for their department. Funding for Ph.D. students pays for university tu-

ition and also provides a monthly stipend of around $1,800 to cover living expenses. (Almost nobody in my field pays their own money to pursue a Ph.D. degree, since it's not financially worthwhile to do so.)

Since I had a decent amount of research and paper writing experience, I felt well-prepared to handle the rigors of Ph.D.-level research when I came to Stanford in September 2006. However, at the time, I had absolutely no idea that my first year of Ph.D. would be the most demoralizing and emotionally distressing period of my life thus far.

Year One: Downfall

In the summer of 2006, several months prior to starting my Ph.D. at Stanford, I thought about ideas for research topics that I felt motivated to pursue. In general, I wanted to create innovative tools to help people become more productive when doing computer programming (i.e., improving *programmer productivity*). This area of interest arose from my own programming experiences during summer internships: Since my assigned day-to-day work wasn't mentally stimulating, I spent a lot of time in my cubicle reflecting on the inefficiencies in the computer programming process at the companies where I worked. I thought it would be neat to work on research that helps alleviate some of those inefficiencies. More broadly, I was interested in research that could help other types of computer users—not only professional programmers—become more productive. For example, I wanted to design new tools to assist scientists who are analyzing and graphing data, system administrators who are customizing server configurations, or novices who are learning to use new pieces of software.

Although I had these vague high-level interests back then, I was still many years away from being able to turn them into legitimate publishable research projects that could form a *dissertation*. To graduate with a Ph.D. from the Stanford Computer Science Department, students are expected to publish two to four related papers as the first (lead) author and then combine those papers together into a book-length technical document called a dissertation. A student is allowed

to graduate as soon as a three-professor *thesis committee* approves their dissertation. Most students in my department take between four to eight years to graduate, depending on how quickly they can publish.

At new student orientation in September 2006, professors in my department encouraged all incoming Ph.D. students to find an *advisor* as soon as possible, so my classmates and I spent the first few months chatting with professors to try to find a match. The advisor is the most important member of a student's thesis committee and has the final say in approving a student to graduate. In my field, advisors are responsible for providing funding for their students (usually via research grants) and working with them to develop ideas and to write papers. I met with a few professors, and the one whose research interests and style seemed most closely related to mine was Dawson, so I chose him as my advisor.

When I arrived on campus, Dawson was a recently-tenured professor who had been at Stanford for the past eight years; professors usually earn *tenure* (a lifetime employment guarantee) if they have published enough notable papers in their first seven years on the job. Dawson's main research interest was in building innovative tools that could automatically find *bugs* (errors in software code) in complex pieces of real-world software. Over the past decade, Dawson and his students built several tools that were able to find far more bugs than any of their competitors. Their research techniques were so effective that they created a successful startup company to sell software bug-finding services based on those techniques. Although I somewhat liked Dawson's projects, what appealed more to me was that his research philosophy matched my own: He was an ardent pragmatist who cared more about achieving compelling results than demonstrating theoretical "interestingness" for the sake of appearing scholarly.

During my first meeting with Dawson, he seemed vaguely interested in my broader goals of making computer usage and programming more productive. However, he made it very clear that he wanted to

recruit new students to work on an automatic bug-finding tool called *Klee* that his grant money was currently funding. (The tool has had several names, but I will call it "Klee" for simplicity.) From talking with other professors and senior Ph.D. students in my department, I realized it was the norm for new students to join an existing grant-funded research project rather than to try creating their own original project right away. I convinced myself that automatically finding software bugs was an indirect way to make programmers more productive, so I decided to join the Klee project.

When I started working on Klee in December 2006, Dawson was supervising five other students who were already working on it. The project leader, Cristi, was a third-year Ph.D. student who, together with Dawson, built the original version of Klee. Dawson, Cristi, and a few other colleagues had recently coauthored and published their first paper describing the basic Klee system and demonstrating its effectiveness at finding new kinds of bugs. That paper was well-received by the academic community, and Dawson wanted to keep up the momentum by publishing a few follow-up papers. Note that it's possible to publish more than one paper on a particular research project (i.e., follow-up papers), as long as each paper contains new ideas, improvements, and results that are different enough from the previous ones. The paper submission deadline for the next relevant *top-tier conference* was in March 2007, so the Klee team had four months to create enough innovations beyond the original paper to warrant a new submission.

~

Before I continue my story, I want to briefly introduce how academic papers are peer-reviewed and published. In computer science, the most prestigious venues for publishing papers are *conferences*. Note that in many other academic disciplines, *journals* are the most prestigious venues, and the word "conference" means something quite

different. The computer science conference publication process works roughly as follows:

1. Each conference issues a call for papers with a list of topics of interest and a specific submission deadline.

2. Researchers submit their papers by that deadline. Each conference typically receives 100 to 300 paper submissions, and each paper contains the equivalent of 30 to 40 pages of double-spaced text.

3. The conference *program committee* (PC), consisting of around 20 expert researchers, splits up the submitted papers and reviews them. Each paper is reviewed by three to five people, who are either PC members or volunteer external reviewers solicited by PC members. The review process takes about three months.

4. After everyone on the PC is done with their reviews, the PC meets and decides which papers to accept and which to reject based on reviewer preferences.

5. The PC emails all authors to notify them of whether their papers have been accepted or rejected and attaches the written reviews to the notification emails.

6. Authors of accepted papers attend the conference to give a 30-minute talk on their paper. All accepted papers are then archived online in a digital library.

A prestigious *top-tier* conference accepts 8 to 16 percent of submitted papers, and a *second-tier* conference accepts 20 to 30 percent. Due to these relatively low acceptance rates, it's not uncommon for a paper to be rejected, revised, and resubmitted several times before being accepted for publication—a process that might take several years. (A paper can be submitted to only one conference at a time.)

~

After Dawson made it clear that he wanted to aim for that particular March 2007 top-tier conference submission deadline, he told me what the other five students were currently working on and gave options for tasks that I could attempt. I chose to use Klee to find new bugs in *Linux device drivers*. A *device driver* is a piece of software code that allows the operating system to communicate with a hardware peripheral such as a mouse or keyboard. The *Linux* operating system (similar to Microsoft Windows or Apple Mac OS) contains thousands of device drivers to connect it with many different kinds of hardware peripherals. Bugs in device driver code are both hard to find using traditional means and also potentially dangerous, because they can cause the operating system to freeze up or crash.

Dawson believed that Klee could find new bugs that no automated tool or human being had previously found within the code of thousands of Linux device drivers. I remember thinking that although new Linux device driver bugs could be cool to present in a paper, it wasn't clear to me how these results constituted a real research contribution. From my understanding, I was going to use Klee to find new bugs—an application of existing research—rather than improving Klee in an innovative way. Furthermore, I couldn't see how my project would fit together with the other five students' projects into one coherent paper submission in March. However, I trusted that Dawson had the high-level paper writing strategy in his head. I had just joined the project, so I didn't want to immediately question these sorts of professor-level decisions. I was given a specific task, so I wanted to accomplish it to the best of my abilities.

~

I spent the first four months of my Ph.D. career painstakingly getting Klee to analyze the code of thousands of Linux device drivers in an attempt to find new bugs. Although my task was conceptually straightforward, I was overwhelmed by the sheer amount of grimy details involved in getting Klee to work on device driver code. I would often spend hours setting up the delicate experimental conditions required for Klee to analyze a particular device driver only to watch helplessly as Klee crashed and failed due to bugs in its own code. When I reported bugs in Klee to Cristi, he would try his best to address them, but the sheer complexity of Klee made it hard to diagnose and fix its multitude of bugs. I'm not trying to pick on Klee specifically: Any piece of prototype software developed for research purposes will have lots of unforeseen bugs. My job was to use Klee to find bugs in Linux device driver code, but ironically, all I ended up doing in the first month was finding bugs in Klee itself. (Too bad Klee couldn't automatically find bugs in its own code!) As the days passed, I grew more and more frustrated doing what I felt was pure manual labor—just trying to get Klee to work—without any intellectual content.

This was the first time in my life that I had ever felt hopelessly overwhelmed by a work assignment. In the past, my summer internship projects were always manageable, and although lots of schoolwork in college was challenging, there was always a correct answer waiting to be discovered. If I didn't understand something in class, then teaching assistants and more advanced students would be available to assist. Even when doing research as an undergraduate, I could always ask my mentor (who was then a fourth-year Ph.D. student) to help me, since I worked on relatively simple problems that he usually knew how to solve. The stakes were also lower as an undergraduate research assistant, since research was only a small fraction of my daily schedule. If I was stuck on a research task, then I could instead focus on classwork or hang out with friends. My college graduation didn't depend

on excelling in research. However, now that I was a Ph.D. student, research was my only job, and I wouldn't be able to earn a degree unless I succeeded at it. My mood was inextricably tied to how well I was progressing every day, and during those months, progress was painfully slow.

I was now treading in unfamiliar territory, so it was much harder to seek help than during my undergraduate years when answers were clear-cut. Since I was the only person trying to use Klee on device driver code, my colleagues were unable to provide any guidance. Dawson gave high-level strategic advice from time to time, but like all tenured professors, his role was not to be "fighting in the trenches" alongside his students. It was our job to figure out all of the intricate details required to produce results—in my case, to find new bugs in Linux device drivers that nobody had previously found. Professors love to repeat the refrain, "If it's already been done before, then it wouldn't be research!" For the first time, I viscerally felt the meaning of those words.

Despite my daily feelings of hopelessness, I kept on telling myself: *I'm just getting started here, so I should be patient.* I didn't want to appear weak in front of my advisor or colleagues, especially because I was the youngest student in Dawson's group. So I trudged forward day after day for over 100 consecutive days, fixing Klee-related problems as they arose and then inevitably encountering newer and nastier obstacles in my quest to find those coveted Linux device driver bugs.

During almost every waking moment, I was either working, thinking about work, or agonizing over how I was stuck on obscure technical problems at work. Unlike a regular nine-to-five job (e.g., my summer internships) where I could leave my work at the office and chill every night in front of the television, research was emotionally and mentally all-consuming. I found it almost impossible to shut off my brain and relax in the evenings, which I later discovered was a common ailment afflicting Ph.D. students. Sometimes I even had trouble sleeping due

to stressing about how my assigned task was unbelievably daunting. There was no way to even fathom taking a break because there was so much work to do before the paper submission deadline in March.

In the midst of all of this manual labor, I tried to come up with some semi-automated ways to make my daily grind less painful. I discussed a few preliminary ideas with Dawson, but we ultimately concluded that there was no way to avoid such time-consuming grinding if we wanted Klee to find bugs in Linux device drivers. I had to tough it out for a few more months until we submitted the paper.

My rational brain understood that experimental research in science and engineering fields often involves a tremendous amount of unglamorous, grungy labor to produce results. Ph.D. students, especially first- and second-years, are the ones who must bear the brunt of the most tedious labor; it's what we are paid to do. In a typical research group, the professor and senior Ph.D. students create the high-level project plans and then assign the junior students to grind on making all of the details work in practice. First- and second-year students are rarely able to affect the overall direction of the group's project. Even though I fully accepted my lowest rank on the pecking order, my emotional brain still took a huge beating during those first few months because the work was so damn hard and unrewarding.

~

After two months of grinding, I began to win some small victories. I got Klee working well enough to find my first few bugs in the smallest device drivers. To confirm whether those bugs were real (as opposed to false positives due to limitations of Klee), I sent emails describing each potential bug to the Linux programmers who created those drivers. Several driver creators confirmed that I had indeed found real bugs in their code. I was very excited when I received those email confirmations, since they were my first small nuggets of external validation. Even though I wasn't doing groundbreaking new research, I

still felt some satisfaction knowing that my efforts led to the discovery of new bugs that were difficult to find without a tool such as Klee.

My morale improved a bit after those first few bug confirmations on the smallest device drivers, so I set my sights on trying to get Klee to work on larger, more complex drivers. However, the new technical problems that arose in subsequent weeks became unbearable and almost drove me to the point of burnout. Here is a summary of the difficulties: Klee can effectively find bugs only in software that contains less than approximately 3,000 lines of code (written in the C language). The smallest Linux device drivers contain about 100 lines of code, so they are well within Klee's capabilities. Larger drivers have about 1,000 lines of code but are intricately connected to 10,000 to 20,000 lines of code in other parts of the Linux operating system. The resulting combination is far beyond Klee's capabilities to analyze, since it's impossible for Klee to "surgically extract" the code of each device driver and analyze its 1,000 lines in isolation. I made various attempts to reduce the number of these external connections (called *dependencies*), but doing so required several days of intricate customized manual effort for each driver.

I met with Dawson to express my exasperation at the daunting task that I was now facing. It seemed absurd to have to spend several days to get Klee working with each new device driver. Not only was it physically wearing me out, but it wasn't even research! What would I write about in our paper—that I had spent nearly 1,000 hours of manual labor getting Klee to work on device drivers without obtaining any real insights? That wasn't a research contribution; it just sounded foolish. I also began to panic because there were only five weeks left until the paper submission deadline, and Dawson had not yet mentioned anything about our group's paper writing strategy. It usually takes at least four weeks to write up a respectable paper submission, especially when there are six students involved in the project who need to coordinate their efforts.

Several days after our meeting, Dawson came up with a plan for improving Klee to overcome the dependency problems I was facing. The new technique that he invented, called *underconstrained execution* (abbreviated "UC"), might allow Klee to "surgically extract" the Linux device driver code from the 10,000 to 20,000 lines of surrounding external code and thus analyze the drivers in isolation. He immediately set out to work with a senior student to incorporate the UC technique into Klee; they called the improved version *Klee-UC*. Even though I was exhausted and almost burned-out, I was glad that my struggles at least motivated Dawson to invent a brand-new idea with the potential to become a worthy research contribution.

Dawson and the other student spent the next few weeks working on Klee-UC. In the meantime, they told me to keep trying to find Linux device driver bugs the old-fashioned manual way. They planned to show the effectiveness of Klee-UC by re-finding the bugs that I had found manually using regular Klee. The argument they wanted to make in the paper submission was that instead of having a Ph.D. student (me!) tediously spend a few days setting up Klee to find each bug, Klee-UC could automatically find all of those bugs in a matter of minutes without any setup effort.

After grinding furiously for a few more weeks, I was ultimately able to get the original Klee to analyze 937 Linux device drivers and discover 55 new bugs (32 of which were confirmed by each respective driver's creator via email). I then had to set up the fledgling Klee-UC tool to analyze those same 937 drivers, which was even more tricky because Dawson and the other student were in the process of implementing (programming) Klee-UC while I was trying to analyze drivers with it. Thankfully, Klee-UC was indeed able to re-find most of those bugs, so at least we had some research contribution and results to write up for our paper submission.

There was one huge problem, though. By the time we got those favorable results, there were only *three days* left until the paper sub-

mission deadline, and nobody had even begun writing the paper yet. In that tiny amount of time, it's physically impossible to write, edit, and polish a paper submission that stands any chance of being accepted to a top-tier computer science conference. But we still tried. In the final 72 hours before the deadline, Dawson and five of us students (one had dropped out of the project by now) camped out at the office for two consecutive all-nighters to finish up the experiments and to write the paper. All of us students knew in the back of our minds that there was absolutely no way that this paper would get accepted, but we followed Dawson's lead and obediently marched forward.

We ended up submitting an embarrassing jumble of text filled with typos, nonsensical sentence fragments, graphics without any explanations, and no concluding paragraphs. It was a horrid mess. At that moment, I had no idea how I would ever complete a Ph.D. if it meant working in this terrible and disorganized manner. As expected, three months later our paper reviews came back overwhelmingly negative, filled with scathing remarks such as, "The [program committee] feels that this paper is blatantly too sloppy to merit acceptance; please do not submit papers that are nowhere near ready for review."

~

Right after this ordeal, I applied for and accepted a summer internship at Google, since I desperately longed for a change of environment. The internship wasn't at all relevant to my research interests, but I didn't care. I just wanted to get away from Stanford for a few months.

It was now April 2007, and there were still ten weeks left until my internship started in June. I had no idea what I could work on, but I wanted to get as far away as possible from my previous four months of dealing with Klee and Linux drivers. I didn't care if we ended up never revising and resubmitting our failed paper (we didn't); I just wanted to escape from those memories. But since I had accumulated nearly 1,000 hours of experience using Klee and it was the only project

Dawson cared about, I figured that it was a wise starting point for developing new project ideas. Thus, I talked to Dawson about ideas for using Klee in unconventional ways beyond simply finding bugs.

However, I quickly realized that I didn't need to be bound by Klee at all since I was funded by the NDSEG fellowship, not by Dawson's grants. In contrast, all of Dawson's other students had no choice but to continue working on Klee since they were funded by his Klee-related grants. So I kept Dawson as my advisor, but I left the Klee project and set out to create my own research project from scratch.

Why didn't I "go solo" sooner? Because even though my fellowship theoretically gave me the financial freedom to pursue whatever research direction I wanted, I knew that I still needed the support of some advisor in order to eventually graduate. Dawson was clearly interested in having all of his new students work on Klee, so I spent four months as a "good soldier" grinding on Klee rather than arrogantly demanding to do my own project from the beginning. Besides, if I had chosen another advisor, I would still need to prove myself by initially working on their projects. There was no way to avoid paying my dues.

I spent the next ten weeks daydreaming of my own research ideas in a complete vacuum without talking to anyone. Since I had such a negative initial experience working in a research group for the past few months, I now wanted to be left alone to think for myself. Dawson was fine with my absence, since he wasn't funding me through his grants.

I lived in complete isolation, mentally burned-out yet still trying to make some gradual progress. Every single day, I tried reading several computer science research papers and taking notes to get inspired to think of my own creative ideas. But without proper guidance or context, I ended up wasting a lot of time and not extracting any meaningful insights from my readings. I also rode my bicycle aimlessly in the neighborhoods around campus in futile attempts to think of new research ideas. Finally, I procrastinated more than I had ever done in my life thus far: I watched lots of TV shows, took many naps, and

wasted countless hours messing around online. Unlike my friends with nine-to-five jobs, there was no boss to look over my shoulder day to day, so I let my mind roam free without any structure in my life.

Although my research brainstorming was largely unfocused, my thoughts slowly gravitated towards ideas related to the following question: *How can we empirically measure the quality of software?* This was one of my broad research interests prior to starting my Ph.D., inspired by my encounters with low-quality software during engineering internships. However, the problem with dreaming up ideas in a vacuum back then was that I lacked the experience necessary to turn those ideas into real research projects. Having full intellectual freedom was actually a curse, since I was not yet prepared to handle it.

Although I was interested in developing new ways to measure software quality, I acknowledged that it was only a fuzzy dream with no grounding in formal research methodologies that the academic community would deem acceptable. If I tried to pursue this project on my own, then I would be yet another quack outsider spouting nonsense. There was no way I could possibly get those ideas published in a top-tier or even second-tier conference, and if I couldn't get my work published, then I wouldn't be able to graduate. I no longer had lofty dreams of becoming a tenured professor: I just wanted to figure out some way to eventually graduate.

I hardly talked to anybody during those ten solitary weeks—not even friends or family. There was no point in complaining, since nobody could understand what I was going through at the time. My friends who were not in Ph.D. programs thought that I was merely "in school" and taking classes like a regular student. And the few friends I had made in my department were equally depressed with their own first-year Ph.D. struggles—most notably, the shock of being thrown head-first into challenging, open-ended research problems without the power to affect the high-level direction of their assigned projects. Here we were, talented young computer scientists voluntarily

working on tasks that were both excruciatingly difficult and seemingly pointless, all while earning one-fourth as much salary as our friends in the corporate working world. It was so sad that it was perversely funny. However, I didn't feel that group whining would be productive, so I kept silent. I avoided coming to the Computer Science Department building to work, since I dreaded running into colleagues. I was afraid that they would inevitably ask me what I was working on, and I didn't have a respectable answer to give. Instead, I preferred hiding out in libraries and coffee shops.

In retrospect, going solo so early during grad school was a terrible decision. Contrary to romanticized notions of a lone scholar sitting outside sipping a latte and doodling on blank sheets of notebook paper, real research is never done in a vacuum. There needs to be solid intellectual, historical, and sometimes even physical foundations (e.g., laboratory equipment) for developing one's innovations. The wiser course of action during those weeks would have been to talk to Dawson more frequently, and to actively seek out collaborations with other professors or senior students. But back then, I was so burned-out and frustrated with the traditional pecking order of group-based research—putting new Ph.D. students through the meat grinder on the most unglamorous work—that I recoiled and went off on my own.

~

Towards the end of my ten weeks of isolation—right before I started my summer internship at Google—I emailed Dawson a blurb from a technical blog post I had recently read and reflected upon. That blog post made me think about measuring software quality by analyzing patterns in how programmers edit code throughout the lifetimes of software projects. To my pleasant surprise, Dawson shot back a quick reply saying that he had a side interest in these sorts of measurement techniques, especially in how they might assist automatic bug-finding tools such as Klee.

I grew hopeful when I learned about Dawson's interest in an area that I also found interesting, since he might be able to help make my ideas more substantive. I jotted down some notes on my newly-proposed *empirical software measurement* project with the intention of returning to it after my summer internship. Thus, I ended my first year of Ph.D. on a somewhat optimistic note after four months of traumatic Klee grinding followed by ten weeks of aimless meandering.

Year Two: Inception

My summer internship at Google was a welcome break from research. The work was low-stress, and I had fun socializing with fellow interns. By the end of the summer, I had recuperated enough from my previous year's burnout to make a fresh start at being a Ph.D. student again.

At the end of the summer, I wrote an email to Dawson reaffirming my desire to pursue my personal interests while simultaneously acknowledging the need to do legitimate publishable research: "I've realized from this summer and my previous work experiences that it's going to be really hard for me to push ahead with a Ph.D. project unless I feel a strong sense of ownership and enthusiasm about it, so I really want to work to find the intersection of what I feel passionate about and what is actually deemed 'research-worthy' by professors and the greater academic community."

I planned to continue working with Dawson on the empirical software measurement project that we had discussed at the end of my first year. However, I had a sense that this project might be risky because it was not within his main areas of expertise or interest; Klee was still his top priority. Therefore, I wanted to hedge my bets by looking for another project to concurrently work on and hoping that at least one of them would succeed. I'll describe my main project with Dawson later in this chapter, but first I'll talk about my other project.

~

Right before starting my second year of Ph.D. in September 2007, I took a one-week vacation to Boston to visit college friends. Since I was in the area, I emailed a few MIT professors whom I knew from my undergraduate days to ask for their guidance. When they met with me, they all told me roughly the same thing: *Be proactive in talking with professors to find research topics that are mutually interesting, and no matter what, don't just hole up in isolation.* This simple piece of advice, repeatedly applied over the next five years, would ultimately lead me to complete my Ph.D. on a happy note.

I immediately took this advice to heart while I was still in Boston. I *cold-emailed* (sent an unsolicited email to) an MIT computer science professor named Rob to politely request a meeting with him. In this initial email, I briefly introduced myself as a recent MIT alum and current Stanford Ph.D. student who wanted to build tools to improve the productivity of computer programmers. Since I knew that Rob was also interested in this research area, I hoped that he would respond favorably rather than marking my email as spam. Rob graciously met with me for an hour in his office, and I pitched a few project proposals to get his feedback. He seemed to like them, so I grew encouraged that my ideas were at least somewhat acceptable to a professor who worked in this research area. Unfortunately, I wouldn't be able to work with Rob since I was no longer an MIT student. At the end of our meeting, Rob suggested for me to talk to a Stanford computer science professor named Scott to see if I could sell him on my ideas.

When I returned to Stanford, I cold-emailed Scott to set up an appointment to chat. I came into the meeting prepared with notes about three specific ideas and pitched them in the following format:

1. What's the problem?
2. What's my proposed solution?
3. What compelling experiments can I run to demonstrate the effectiveness of my solution?

My friend Greg, one of Rob's Ph.D. students, taught me the importance of the third point—thinking in terms of experiments—when proposing research project ideas. Professors are motivated by having their names appear on published papers, and computer science conference papers usually need strong experiments to get accepted for publication. Thus, it's crucial to think about experiment design at project inception time.

Although none of my specific ideas won Scott over, he still wanted to work with me to develop a project related to my general interests. At the time, he was an assistant (pre-tenure) professor who had been at Stanford for only three years, so he was eager to publish more papers in his quest to earn tenure. Since I had a fellowship, Scott didn't need to fund me from his grants, so there was no real downside for him.

~

Scott specialized in a pragmatic *subfield* of computer science called HCI (Human-Computer Interaction). In contrast to many other subfields, the HCI methodology for doing research centers on the needs of real people. Here is how HCI projects are typically done:

1. Observe people to find out what their real problems are.
2. Design innovative tools to help alleviate those problems.
3. Experimentally evaluate the tools to see if they actually help people.

Since I wanted to create tools to help improve the productivity of programmers, Scott first suggested for me to observe some programmers at work in their natural habitat to discover what real problems they were facing. In particular, Scott was intrigued by how modern-day programmers write code using an assortment of programming languages and rely heavily on Web search and copying-and-pasting of code snippets. The previous few decades of research in programmer productivity tools have assumed that programmers work exclusively with

a single language in a homogeneous environment, which is a grossly outdated assumption. By observing what problems modern-day programmers face, I might be able to design new tools to suit their needs.

Now that Scott had provided a high-level goal, I set out to find some professional programmers whom I could observe at work. First, I tried to drum up some leads at Google, since I had just interned there and my former manager agreed to forward my email solicitation to his colleagues. I quickly received a few polite rejections, since nobody wanted to deal with possible intellectual property issues arising from a non-employee looking at their code. I then emailed a dozen friends at various startup companies near Stanford, figuring that they would not be subject to the same constraints as programmers at a big company. Unfortunately, they were even less accommodating since they had a greater need for secrecy due to competitive reasons. Also, they were far busier than their big-company counterparts and thus unwilling to indulge the intellectual fancy of some random graduate student.

My last-ditch attempt was to try to observe programmers at Mozilla, the nonprofit software development foundation that makes the popular Firefox web browser. Since Mozilla's software projects were all open-source, I figured that they would not be afraid of an outsider coming in to watch their programmers work. Rather than cold-emailing (I didn't even know who to email!), I decided to drive to the Mozilla headquarters and walk in the front door. I accosted the first person I saw and introduced myself. He generously gave me the names and email addresses of two Mozilla Foundation leaders who might be interested in such a research collaboration. I cold-emailed both of them later that day and was amazed that one responded favorably. Unfortunately, that was the last I heard from him; he never responded to my requests to follow up.

In retrospect, I'm not surprised that my workplace shadowing attempts failed, since I had absolutely nothing to offer these professional programmers; I would just be disrupting their workday. Fortunately, a

few years later I managed to observe a different set of (nonprofessional) programmers—fellow graduate students doing programming for scientific research—who welcomed my light intrusions and were more than happy to talk to me about their working environments. Those interviews would end up directly inspiring my dissertation work.

~

Since I had no luck in shadowing professional programmers, I decided to look within Stanford for opportunities. When I saw a flyer announcing an annual computer programming competition being held soon in my department, I immediately cold-emailed the organizer. I pitched him on the idea of having me observe students during the competition, and he happily agreed.

Although a student programming competition was not indicative of real-world programming environments, at least it was better than nothing. Joel, one of Scott's students who entered the Ph.D. program one year before me, wanted to come observe as well. Joel and I spent an entire Saturday morning watching a few students participating in the four-hour competition. It was a pretty boring sight, and we didn't end up learning much from the notes we took. However, this experience gave us the idea to plan a more controlled *laboratory study*.

At this point, Scott decided to have Joel and me team up to create a lab study together rather than working on separate projects. Since Joel and I shared similar research interests, Scott could reduce his management overhead by having us join forces. I was happy to let Joel take the lead on the lab study design, since I was concurrently working on another project with Dawson.

Joel, Scott, and I designed a lab study where we asked Stanford students to spend 2.5 hours programming a simple Web-based chat room application from scratch. They were allowed to use any resources they wanted, most notably searching the Web for existing code and tutorials. We recruited 20 students as participants, most of them

coming from the *Introduction to HCI* course that Scott was teaching at the time.

Over the next few weeks, Joel and I sat in our department's basement computer lab for 50 hours (2.5 hours x 20 students) observing study participants and recording a video feed of the lab's computer monitor. Watching the students at work was engaging at first but quickly grew tedious as we observed the same actions and mistakes over and over again. We then spent almost as much time watching replays of the recorded videos and marking occurrences of critical events. Finally, we analyzed our notes and the marked events to garner insights about what kinds of problems these students faced when working on a simple yet realistic real-world programming task.

We wrote up our lab study findings and submitted our paper to a top-tier HCI conference. Three months later, we found out that our paper was accepted with great reviews and even nominated for a *Best Paper Award*. Joel was listed as the *first author* on this paper, and I was the second author. Almost all computer science papers are coauthored by multiple collaborators, and the order in which authors are listed actually matters. The author whose name appears first is the project leader (e.g., Joel) who does more work than all subsequently listed authors and thus deserves most of the credit. All other authors are project assistants—usually younger students (e.g., me) or distant colleagues—who contributed enough to warrant their names being on the paper. Ph.D. students often list their advisor (e.g., Scott) as the last author, since the advisor helps with idea formulation, project planning, and paper writing.

Since I wasn't the first author on this paper, it didn't contribute towards my dissertation; however, this experience taught me a great deal both about how to do research and about how to write research papers. Most importantly, I felt satisfied to see this project conclude successfully with a prestigious top-tier conference publication, in stark contrast to the embarrassing Klee paper rejection from my first year.

Joel continued down this research path by turning our paper into the first contribution of his dissertation. Over the next few years, he built several tools to help programmers overcome some of the problems we identified in that initial study and published papers describing those tools. In the meantime, Scott didn't actively recruit me to become his student, so I never thought seriously about switching advisors. It seemed like my interests were too similar to Joel's, and Joel was already carving out a solid niche for himself in his subfield. Thus, I focused my efforts on my main project with Dawson, since he was still my advisor. Progress was not as smooth on that front, though.

~

Recall that at the end of my first year, I started exploring research ideas in a subfield called *empirical software measurement*—specifically, trying to measure software quality by analyzing the development history of software projects. It turned out that Dawson was also interested in this topic, so we continued working together on it throughout my second year. His main interest was in building new automated bug-finding tools (e.g., Klee), but he had a side interest in software quality in general. He was motivated by research questions such as:

- If a large software project has, say, 10 million lines of code, which portions of that code are crucial to the project, and which are not as important?
- What factors affect whether a section of code is more likely to contain bugs? For example, does code that has been recently modified by novices contain more bugs? What about code that has been modified by many people over a short span of time?
- If an automated bug-finding tool finds, say, 1,000 possible bugs, which ones are likely to be important? Programmers have neither the time nor energy to triage all 1,000 bug reports, so they must prioritize accordingly.

I investigated these kinds of questions by analyzing data sets related to the Linux kernel software project. I chose to study Linux since it was the largest and most influential open-source software project at the time, with tens of thousands of volunteer programmers contributing tens of millions of lines of code over the span of two decades. The full *revision control history* of Linux was available online, so that became my primary source of data. A project's revision control history contains a record of all modifications made to all code files throughout that project's lifetime, including when each modification was made, and more importantly, by whom. To correlate project activity with bugs, I obtained a data set from Dawson's software bug-finding company containing 2,000 bugs that one of its bug-finding tools had found in Linux. Of course, those were not the only bugs in Linux, but it was the only accessible data source with the information I needed to investigate our research questions.

My daily workflow consisted of writing computer programs to extract, clean up, reformat, and analyze the data from the Linux revision control history and those 2,000 bug reports. To help obtain insights, I taught myself the basics of quantitative data analysis, statistics, and data visualization techniques. As I worked, I kept a meticulous log of my experimental progress in a *research lab notebook*, noting which trials did and did not work. Every week or so, I would meet with Dawson to present my findings. Our meetings usually consisted of me showing him printouts of graphs or data tables that my analyses had generated, followed by him making high-level suggestions such as, "Wow, this part of the graph looks weird, why is that? Split the data up in this way and dig deeper." Years later, I learned that this working style was fairly common amongst *computational researchers* in a variety of academic fields; for my dissertation, I created tools to eliminate common inefficiencies in this pervasive type of workflow. However, back at that time, I had no such long-term visions; I just wanted to make interesting discoveries and get them published.

~

Dawson and I had a lot of trouble getting our results published. Throughout the year, we made two paper submissions that were both rejected. It would take another full year before this work finally got published as a shorter-length, second-tier conference paper, which held almost no prestige and didn't "count" as a contribution to my dissertation. But by then, I didn't care because I had already moved on to other projects.

The underlying cause of our publication troubles was that we were not "insiders" in the empirical software measurement subfield (sometimes also called *empirical software engineering*) to which our project belonged. At the time Dawson and I started working in this subfield, teams of researchers from dozens of universities and corporate research labs were already doing similar work. Dawson and I were severely outgunned by the competition, which consisted of professors and research scientists specializing in empirical software measurement who were advising armies of Ph.D. students to do the heavy number crunching. These people were hungry to publish a ton of papers, since lots of them were young professors who wanted to earn tenure. They were also experts in the statistical methodologies, framing of related work, and "marketing pitches" required to get these sorts of papers accepted. Most importantly, they frequently served on program committees and as external reviewers for the relevant conferences, so they knew exactly what it took to write publishable papers in this subfield.

Recall that each paper submission gets *peer-reviewed* by three to five volunteer experts—usually professors or research scientists—who critique its merit. If reviewers deem a paper worthy of publication, then it gets published; otherwise, authors must revise and resubmit at a later date. The purpose of peer review is to ensure that all published papers are up to a certain level of acceptable quality, as determined by the scholarly community. This vetting process is absolutely necessary, since there needs to be some arbiters of "goodness" to filter

out crackpot claims. However, the peer-review process is inherently imperfect since, despite their best efforts at being impartial, reviewers are human beings with their own subjective tastes and biases.

Since conferences usually accept less than 20 percent of paper submissions, if reviewers get a bad first impression when reading a paper, then they are likely to reject it. Dawson and I were not specialists in the empirical software measurement subfield, so we weren't able to "pitch" our paper submissions in a way that appealed to reviewers' expectations. Thus, we repeatedly got demoralized by negative reviews such as, "In the end, I find I just don't have much confidence in the results the authors present. There are two sources of doubt: I don't trust their interpretation of the measures, and they don't use very effective statistical techniques." In the cutthroat world of academic publishing, simply being passionate about a topic is nowhere near sufficient for success; one must be well-versed in the preferences of senior colleagues in a particular subfield who are serving as paper reviewers. In short, our data sets were not as good, our techniques were not as refined, and our results and presentation style were less impressive than what the veterans in this subfield expected.

In contrast, my paper submission with Scott and Joel was far more successful because Scott was an insider who had previously published and reviewed many papers in the HCI conference where we submitted our paper. Of course, being an insider didn't mean that our paper was scrutinized any less rigorously, since that would be unfair. However, Scott could leverage his experience to present our project's motivations and findings in a way that was the most palatable to those sorts of reviewers, thereby raising our paper's chances of acceptance.

~

By the end of my second year of Ph.D. (June 2008), I was growing frustrated by my lack of compelling results and overwhelmed by the flurry of papers being published in the empirical software measurement subfield. I still hadn't been able to publish my own findings and realized that I couldn't effectively compete with the veterans: Since neither Dawson nor I knew what those paper reviewers wanted to see, I sensed that trying to publish in this subfield would be a continual uphill struggle. And since publishing was a prerequisite for graduation, I had to find another project as soon as possible, or else I wouldn't be able to earn a Ph.D. As I prepared to begin my third year at Stanford, I was desperate to cling onto anything that had a reasonable chance of producing publishable results. And that's when I returned back to the project that haunted me from my first year—Klee.

Year Three: Relapse

As I began my third year of Ph.D. in the middle of 2008, I rejoined the Klee project, again as the most junior student. At that time, the only two remaining people on the Klee team were its original co-creators: Dawson and Cristi (his most senior Ph.D. student). All other Klee team members had already left the project.

I had mixed feelings about returning. On one hand, my traumatic first-year experiences with Klee made me dread both the project and also the team dynamics. On the other hand, Cristi and Dawson were both very passionate about Klee and wanted to publish additional follow-up papers. Since they were veteran insiders in the software bug-finding subfield, I felt like I had a strong chance of publishing papers together with them. The alternative would have been to continue the empirical software measurement project from my second year. Although I was much more interested in that project, I knew that it would be a continual struggle to publish in a subfield where Dawson and I were both outsiders. And since my goal was to publish and earn a Ph.D., I tucked away my ego and took the plunge into Klee again. I wrote Dawson an email announcing, "my main plan is to team up with you and cristi on klee to do something solid and hopefully make some [paper] submission in a few months. i think that leveraging klee and aligning with both of your interests and incentives will be the best way for me to both make a contribution and also to feel satisfied about making concrete forward progress every day."

~

Cristi and Dawson wanted me to experiment with a new way of running Klee called *cross-checking*, which allowed it to find inconsistencies between two different versions of similar software. For the next four months (July to October 2008), my day-to-day grind was similar to my first-year Klee assignment with Linux device drivers, except that I now paced myself a lot better to remain healthy and avoid burnout. Just like during my first year, I was doing a lot of grungy manual labor to use Klee to discover new bugs in software rather than improving Klee in any substantive way. My daily workflow consisted of setting up dozens of Klee configuration options, launching Klee to run for approximately ten hours to cross-check a set of test software programs, coming back the next morning to collect, analyze, visualize, and interpret the results, making the appropriate adjustments to Klee's options, and then firing off another ten-hour round of experiments.

Like other sophisticated software tools, Klee had dozens of adjustable configuration options. And since it was a research prototype hacked together by students, the behavior of most of those options were not clearly documented. As a result, I wasted a lot of time due to misunderstanding the subtle interactions between options as I was adjusting them. I filled my research lab notebook with curses such as: "OH SHIT, I think my mistake was in not realizing that there are certain options you're supposed to pass into Klee (e.g., -emit-all-errors) and others that you're supposed to pass into the target program to be used to set up the model environment (e.g., –sym-args), and if these are confused, then strange things happen because Klee is executing the target program with different argc and argv than you expect."

Throughout those months, Cristi and Dawson sometimes talked about submitting a Klee cross-checking paper, so I was motivated by that seemingly-concrete goal. As I was working, I wrote up an outline for a paper submission and incrementally filled it in with my notes and results. However, to my surprise, neither Cristi nor Dawson showed

much urgency in getting this paper polished and submitted. I was not yet capable of submitting a respectable paper on this topic without their expertise and assistance, since I was merely an assistant doing manual labor: The true research insights and high-level persuasive pitch still needed to come from them. In the end, we never submitted a paper, and my four months of work was again in vain, just like during my first-year Klee grind.

This project fizzled due to a combination of my own lack of technical expertise and insufficient mentorship from senior colleagues. Although Cristi was patient in advising me on cross-checking ideas and debugging Klee idiosyncrasies, his heart wasn't fully into our project. Since he was in the process of finishing up his Ph.D., his main priority at the time was applying for jobs as an assistant professor. The faculty job application process takes several grueling months of serious effort, and many applicants still end up with no offers. Each university department offers at most one or two tenure-track professor job positions per year, and over a hundred highly-qualified senior Ph.D. students, postdocs (temporary postdoctoral researchers), and research scientists fight for those coveted spots. The academic job hunt is a stressful process that consumes almost all of one's waking time and mental energy. Thus, Cristi had no incentive to spend hundreds of hours working on yet another paper submission, since even if it got accepted, the notification would come too late to matter for his job applications.

In hindsight, I can see why this project was likely to fail because of misaligned incentives, but back then, I lacked the wisdom to foresee such a failure. Recall that I decided to become a Klee assistant for Cristi and Dawson since I wanted to join an older Ph.D. student and professor who were experienced in publishing papers in their given subfield. I did so because this plan worked marvelously during the previous year when I helped Joel (and older Ph.D. student) and Scott (a professor) on their HCI project, which led to a top-tier award-nominated paper.

So what was different here? In short, neither Cristi nor Dawson were truly hungry to publish. They had already published several Klee papers together, and a cross-checking paper coauthored with me would have been a "nice-to-have" but not mandatory follow-up publication. Cristi was in his final year of Ph.D. and didn't need to publish any more papers to graduate, and Dawson already had tenure, so he wasn't in a rush to publish either. In contrast, Joel was a mid-stage Ph.D. student who was itching to publish the *first* paper of his dissertation, and Scott was an assistant professor who needed to publish prolifically to earn tenure. These two opposing experiences taught me the importance of deeply understanding the motivations and incentives of one's potential collaborators before working with them.

~

Since the cross-checking project went nowhere and Cristi was busy with his faculty job applications, I decided to take the lead on my own Klee-related project rather than continue serving as an assistant. After some discussions with Dawson, he suggested for me to try to improve a core component of Klee—its *search algorithm*. Klee finds bugs by searching through a "maze" of executable software code, so improving its search algorithm can possibly enable it to find more bugs.

For the first time, I was modifying Klee in a novel way—improving its search algorithm—rather than simply doing manual labor to run Klee to find bugs in software. One way to measure how well I was doing was to compute the percent *coverage* that Klee achieves (i.e., how much of the code maze was "covered" by Klee's searching) on a set of test software programs. Dawson's goal was simple: to get significantly better coverage than the current search algorithm reported in the latest Klee paper. On a suite of 89 test programs, Klee already achieved on average 91 percent coverage on each program (100 percent is perfect coverage). My job was to improve those coverage numbers as much as possible. Every day, I would modify Klee's search algorithm, run Klee

on the 89 test programs (which would take approximately ten hours), come back the next morning to see the coverage numbers, and then make another round of modifications to Klee's code and rerun it on the test programs.

It was now the middle of my third year, and many of my fellow students and I fell into a state of "limbo" where it became difficult to motivate ourselves to consistently come into the office every single day. We also experienced isolation and loneliness from spending day and night grinding on obscure, ultra-specialized problems that few people around us understood or even cared about. Our advisors and senior colleagues sometimes provided high-level guidance, but they rarely sat down together with us to work out all of the grimy details.

Unlike our peers with regular nine-to-five jobs, there was no immediate pressure for grad students to produce anything tangible—no short-term deadlines to meet or middle managers to please. For most students in my department, nobody would notice or care if they took one day off, so by extension, why not take two days off, a whole week off, or even a whole month off? Therefore, it's unsurprising that many Ph.D. students who drop out do so around their third year.

To fend off procrastination, I worked tirelessly to impose self-discipline and structure on my workdays. I tried to "micromanage" myself by setting small, bite-sized goals and attacking them every day, hoping that positive results would eventually come. But it was hard to keep myself motivated when I didn't see noticeable daily progress.

Discipline alone wasn't enough; I failed to achieve any favorable results after three months of tuning Klee's search algorithms. Since Klee already achieved 91 percent average coverage on our test programs, it was excruciatingly difficult for me to improve those numbers by a few percent up to an average of, say, 94 percent. Even worse, these kinds of minor improvements simply don't look impressive in a paper submission. The one-line story of our paper would be something like: "We improved Klee's search algorithm in some ways to get its average

coverage up from 91 to 94 percent." This is hardly an exciting or even interesting result in the eyes of reviewers; it's a typical example of boring incremental improvements to an existing project. Unsurprisingly, Dawson wasn't interested in attempting to submit such a lame paper.

If I had improved Klee's search algorithm in a fascinating and effective way, then Dawson might have been more excited and worked harder to try to submit a paper. But in January 2009, after three months of futile grinding, I couldn't see how my day-to-day incremental efforts would ever result in a breakthrough that met Dawson's expectations. I hate being labeled as a quitter, but I felt like this Klee search algorithm project was a dead-end, so I quit.

Looking back now, I take perverse solace in one tragic fact: After I stopped working on Klee's search algorithm, two of Dawson's other Ph.D. students worked on this exact same problem, and neither has published a single paper in the past *three years*. I don't think I could have done any better than those two students, so if I had stayed the course on this particular project, then I might have also been stuck in a three-year-long limbo.

~

Despite repeated failures with Klee, I still wanted to keep working on it because that was the only project Dawson cared about. I was starting to hate Klee, but I had already sunk thousands of hours into wrestling with its code, so I wanted something concrete to show for my efforts. It was now the middle of my third year, and I was desperate to publish a first-author paper that could form the basis for my dissertation; I felt a bit behind since a few of my classmates had already published their first dissertation-worthy paper. I naively hoped that Klee would be the "path of least resistance" to earning my Ph.D., since it was perfectly aligned with my advisor's interests.

At this time, a first-year Ph.D. student named Peter joined Dawson's lab group and was looking for a project. I talked to Dawson

about teaming up with Peter, figuring that the two of us working together might get better results than each working alone. Dawson liked the idea, so he suggested for Peter and me to reimplement *underconstrained execution* in Klee (abbreviated "Klee-UC"). Recall that Dawson and another student implemented the first version of Klee-UC during my first year. They created a rough first draft, submitted a shoddy paper hastily written in three days (a debacle I remember all too well), and then the project halted when that student dropped out of the Ph.D. program shortly thereafter. So now, two years later, it was up to Peter and me to reimplement Dawson's initial Klee-UC vision and hopefully get it working well enough to publish a paper.

I came into this new assignment with as much optimism as I could muster, trying my best to forget my past with Klee. I convinced myself that if I had any chance of publishing a Klee-related paper, it would be with this current Klee-UC project. I wholeheartedly believed that Dawson's Klee-UC idea was innovative and interesting from a research perspective, so if Peter and I could do a good enough job of implementing it and finding important software bugs, then we would have a strong paper submission. Moreover, I could reuse most of the experimental infrastructure I had set up for the Linux device drivers work from my first year, since we wanted to show how Klee-UC improves upon regular Klee in terms of finding bugs in those drivers. Finally, I fantasized about a successful Klee-UC paper being the ultimate redemption for all of those thousands of hours of manual labor I had spent on Klee. After all, it was my struggles with using Klee on Linux device drivers during my first year that directly inspired Dawson to come up with the Klee-UC idea. Thus, it would be a fitting conclusion if I were to first-author the paper that brought this idea to fruition (professors in my field usually let their students be the first author, even if the student's project was based on their ideas). I even hoped that this project would form the beginning of my dissertation and pave the way for my eventual graduation.

Over the next two months (February and March 2009), Peter and I busted our butts to build Klee-UC. We had a lot of fun programming together in the office every day; it was a welcome change from the solitary day-to-day grind that most Ph.D. students experience. However, after a while, Dawson seemed visibly disappointed with the relatively slow pace of our progress. Peter and I thought we were doing fine, but Dawson didn't seem happy with our work, so he no longer felt like aiming for an upcoming paper submission deadline.

At the time, I couldn't understand why Dawson was so impatient with us, but I can now sympathize with his feelings of frustration. He had such a crystal-clear vision for Klee-UC in his mind, and he wanted some talented and hardworking students to carry out his vision. If Dawson were still a Ph.D. student, then he would have surely been able to get Klee-UC done in a matter of weeks and then singlehandedly write up and publish a top-tier paper. His publishing track record when he was a student was beyond prolific, which is how he got a top-tier faculty job at Stanford. However, since he was now busy with professor duties such as teaching, committee work, paper reviewing, and other errands, he could not devote the thousands of hours of focused labor necessary to turn this idea into a publishable paper. Like all professors in labor-intensive research fields, Dawson needed students to execute on his visions.

I think that Dawson expected Peter and I to have gotten publishable results much faster, so to him, we either seemed incompetent or not serious enough about our jobs. As a professor at a top-tier university, it's a sad reality that all of Dawson's students are probably less competent than he was as a Ph.D. student. The explanation is simple: Only about 1 out of every 75 Ph.D. students from a top-tier university has what it takes to become a professor at a school like Stanford (or maybe 1 out of every 200 Ph.D. students from a regular university). Unsurprisingly, neither Peter nor I were of that caliber. If Dawson had worked with a younger clone of himself, then progress would have

been a lot faster!

Even though we put in a solid effort during those two months, Peter and I felt like we had really let Dawson down on a project he cared deeply about. Peter was so discouraged that he switched advisors and then later dropped out of the Ph.D. program altogether. With my teammate gone, I grew more disillusioned and decided to quit Klee for the final time.

~

Two years after Peter and I left the Klee project, Dawson finally found a new Ph.D. student who could properly implement his Klee-UC vision to fruition. In 2011, Dawson and his new student published a great paper incorporating both Klee-UC and cross-checking ideas. In the end, it took three attempts by four Ph.D. students over the course of five years before Dawson's initial Klee-UC idea turned into a published paper. Of those four students, only one "survived"—I quit the Klee project, and two others quit the Ph.D. program altogether. From an individual student's perspective, the odds of success were low.

From a professor's perspective, though, Klee-UC was a rousing success! Since Dawson had tenure, his job was never in danger. In fact, one of the purposes of tenure is to allow professors to take risks by attempting bolder project ideas. However, the dark side of this privilege is that professors will often assign students to grind on risky projects with low success rates. And the students often can't refuse, since they are funded by their advisors' grants. Thankfully, since I was funded by fellowships, it was much easier for me to quit Klee.

I don't mean to single out Dawson or Klee in particular. This mismatch of incentives between tenured professors and Ph.D. students is a common problem in *most* labor-intensive science and engineering research projects. What often happens is that a professor starts with a pile of grant money and some high-level vision (e.g., Klee-UC or cross-checking). The professor then hires several students and advises

them on implementing that vision, possibly (but not always) as part of their dissertation work. Without thousands of hours of student labor, there would be no tangible results and thus no publications.

The professor might need to go through several rounds of student failures and dropouts before one set of students eventually succeeds. Sometimes that might take two years, sometimes five years, or sometimes even ten years to achieve. Many projects last longer than individual Ph.D. student "lifetimes." But as long as the original vision is realized and published, then the project is considered a success. The professor is happy, the university department is happy, the grant funding agency is happy, and the final surviving set of students is happy. But what about the student casualties along the way? A tenured professor can survive several years' worth of failures, but a Ph.D. student's fledgling career—and psychological health—will likely be ruined by such a chain of disappointments.

~

I attended Cristi's *oral defense* in May 2009, the end of my third year. The oral defense is the final rite of passage before a student earns their Ph.D. degree: The student gives a one-hour presentation on their dissertation research and must answer critical questions from a panel of professors. Dawson, who is normally quiet and reserved, beamed with visible pride as he introduced Cristi to the audience and raved about what a pleasure it was to have worked together over the past few years to create Klee. His praise was well-deserved: Cristi did a wonderful job throughout his Ph.D., and the ideas embodied by Klee helped create a brand-new subfield (called *mixed concrete/symbolic program execution*) within the software bug-finding research world.

As I watched Cristi's oral defense presentation, it finally sank in that it would be almost impossible for me to get a substantive dissertation out of Klee, so I felt more confident in my decision to quit. Cristi's

phenomenal success made it more difficult for Dawson's younger students to publish and graduate. The groundbreaking initial Klee work had already been done; all that remained were follow-up incremental enhancements such as improving the search algorithm, Klee-UC, cross-checking, and applying Klee on new types of software such as Linux device drivers. Although these projects could certainly make for publishable papers and maybe even a dissertation, Dawson wasn't nearly as hungry to publish as our newly-arrived competitors were.

Since Klee (and a few related projects from 2005 to 2008) created a new subfield, dozens of assistant professors and young research scientists quickly jumped on the bandwagon and ferociously cranked out paper after paper describing incremental improvements to try to win tenure or job promotions. It was like an academic gold rush, prompted by the insights of Cristi, Dawson, and a few other early pioneers. Since Dawson had tenure and was already famous for creating Klee and other notable projects, he was above the fray and didn't have a desire to publish for the sake of padding his resume.

In effect, Ph.D. students working with those young researchers were more easily able to publish and graduate, while Dawson's students had a much harder time by comparison. In the three years since I quit Klee, dozens of research groups around the world have published hundreds of papers based on Klee-like ideas. Amazingly, fifteen of those papers described enhancements to Klee itself, which our lab released as open-source software to encourage further research. In the meantime, five of Dawson's Ph.D. students have seriously attempted to work on Klee; so far, *only one* has published a single paper (on Klee-UC).

The sad irony here is that since Dawson's direct competition was now serving as conference program committee members and paper reviewers, it was much harder to get his papers published despite the fact that he co-founded this subfield in the first place. Because Dawson had not been actively publishing in recent years, he no longer knew all of the rhetorical tricks, newfangled buzzwords, and marketing-related

contortions required to satisfy reviewers and get his papers accepted into top-tier conferences. Also, the more furiously his competitors published, the more strict the reviewers became about demanding for him and his students to justify the originality of their ideas in relation to the piles of related work, and the more frustrating the paper rejections became. After all, without Dawson's groundbreaking insights from the past decade, these picky reviewers would not even be working in this subfield, much less criticizing and rejecting his papers!

I calculated that the only advantage of staying with Klee was that Dawson deeply loved the project. Even if I couldn't get any papers published, he could maybe appeal to let me "pity graduate" with zero publications. But given my painful past with Klee, I couldn't stomach the possibility of grinding on it for an unknown number of additional years just for the hope of a pathetic "pity graduation."

By now, I had finished three years of my Ph.D. still without any idea of how I was going to eventually put together a dissertation. I had no concrete plan looking forward, but I knew that I wanted to get away from Klee once and for all.

Intermission

Immediately after my third year of Ph.D., I spent the summer of 2009 in Seattle, Washington as an intern at the headquarters of Microsoft Research. It was one of the most fun and productive summers of my life: My internship project led to the publication of three top-tier conference papers and, more importantly, helped establish the motivation for my dissertation work.

At present, Microsoft Research is the premier corporate institution for producing top-notch academic research. Research labs in most companies are usually focused on R&D efforts to directly improve their own future products. However, the primary mission of Microsoft Research (abbreviated "MSR") is to perform fundamental science and engineering research with the intent of publishing top-tier academic papers in computer science and a few related fields.

The best way to think of MSR is as a giant research university without any students. The full-time researchers are like professors, except that they can focus nearly all of their time on research since they don't have teaching or advising duties. But perhaps their favorite job benefit is that they don't need to apply for grant funding, which is a tedious recurring activity that saps professors' time. Since Microsoft is an immensely profitable company, it allocates hundreds of millions of dollars each year to funding academic (paper-producing) research. Microsoft is betting that some of the intellectual property created by its researchers might inspire future products, and it also wants the

best minds in computer science on staff for consultation. That's why the company gives its researchers access to all of the resources required to do their best work.

Getting a full-time researcher position at MSR is as difficult as getting a job as a professor at a prestigious university. Although MSR researchers don't technically have tenure, job security is fairly good, especially if they continually publish. Since lots of computer science research is labor-intensive, researchers often hire Ph.D. students as summer interns to help implement their ideas. It's a great deal for both parties: Researchers get students to assist with manual labor, and students get the chance to publish top-tier papers with famous researchers outside of their universities and possibly get letters of recommendation for future jobs. In the past decade, a significant fraction of the papers at top-tier computer science conferences were written by MSR researchers and their interns.

When I arrived at the MSR headquarters in the beginning of the summer, the campus was abuzz with the energy of hundreds of Ph.D. students meeting their managers and preparing to get to work. Since we were there for only three months, our managers planned well-defined projects that would likely result in a paper submission. Most of us were able to submit at least one paper from our summer work, and a fraction of those papers ended up getting published. Of course, research is inherently risky, so some interns were assigned projects that never panned out into publications. Nonetheless, almost everyone had a wonderful time—we were paid over four times our usual grad school stipends, treated to fun Microsoft-sponsored social outings, and attended lots of stimulating talks by top-notch researchers.

Perhaps the longest-lasting impact of an MSR internship is the friendships we all made. During that summer, I had the privilege of getting to know some of the brightest and most inspiring young computer science researchers of my generation. For instance, one of my three officemates was about to start her Ph.D. at MIT, and she had

already published more top-tier papers from her undergraduate research than most Ph.D. students could ever hope to publish. Another officemate was a UC Berkeley Ph.D. student who spent his nights and weekends working on a separate research project with collaborators across the country in addition to doing his internship project during workdays. These peers will likely grow into award-winning professors, research leaders, and high-tech entrepreneurs, so I am humbled to have been in their presence for a summer.

~

The story of how I arrived at MSR that summer illustrates the importance of combining concrete achievements with professional connections. Many Ph.D. students get internships (and later full-time jobs) through some sort of connection, and I was no exception.

I first applied to be an intern at MSR during my second year while I was working with Scott and Joel on their HCI programming lab study project. I applied through regular channels by submitting my resume online, and my application was quickly rejected in favor of those students with more publications and usually some inside connections.

One year later, during my third year, an MSR researcher saw that my work with Scott and Joel had been published in an HCI conference, so he emailed me to ask whether I was interested in doing an internship with him on a loosely related project. He sought me out in particular because my first undergraduate research supervisor at MIT had introduced us to one another several years earlier, so he had some recollection of who I was.

I was honored by his offer but told him that I was no longer working on HCI research; by then, I had already gone back to bug-finding work with Klee. However, I expressed a strong interest in working on empirical software measurement research at MSR, since I had spent my second year doing that sort of work with Dawson. He immediately forwarded my resume to his colleague Tom, who was a rising star in

the empirical software measurement subfield. After introducing myself via email, I sent Tom the short paper that I coauthored with Dawson from our Linux bug report measurement work. Tom liked my paper, so he decided to hire me as his summer intern. I had read several of Tom's research papers during my second year, so I was very excited about the possibility of working with him.

If I had blindly submitted my resume online like hundreds of other applicants, I would have probably not been able to attract Tom's attention. Most of my fellow interns also got their jobs through connections, although usually their advisors made a direct recommendation to a relevant MSR colleague. Interestingly, it wasn't Dawson, but rather one of my undergraduate research supervisors (from a project I did over six years earlier) who provided the much-needed connection for me. This same supervisor would later provide a crucial introduction that led to my first full-time job after graduation. From this experience, I learned about the importance of being endorsed by an influential person; simply doing good work isn't enough to get noticed in a hyper-competitive field.

~

Tom defined the high-level scope of my internship project and set a realistic yet ambitious goal of submitting a paper to a top-tier conference at the end of the summer. My project was to quantify people-related factors that affect whether software bug reports are successfully fixed, reassigned to others, or reopened after supposedly being fixed. To obtain these insights, I wrote computer programs to analyze software bug databases and employee personnel data sets within Microsoft. I was well-prepared to do this sort of data mining and analysis work, since I had spent most of my second year doing similar analyses with Dawson on Linux bug report and revision control history data.

Tom would drop by my office at 5pm each afternoon before he left work to check up on my progress. Although daily check-ups could

potentially be stressful, I actually found them immensely helpful since Tom wasn't intimidating or judgmental at all. Getting immediate daily feedback made it easy for me to stay focused and motivated. The combination of a well-defined, short-term goal and continual helpful feedback made my internship workdays much more productive than those during my previous three years of grad school. The best part was that I worked only during normal office hours (9am to 6pm). There was no possible way to take my work home with me since the data was available only within Microsoft, so I just chilled every evening and had fun without worrying about whether I ought to be working more; back at school, I constantly worried about whether I was working enough since I could potentially be working during all waking moments.

Since Tom has published and reviewed dozens of empirical software measurement papers, he was definitely an "insider" who knew what sorts of results and write-ups are well-liked by reviewers in that subfield. When it came time to submit our paper at the end of the summer, Tom was able to deftly frame our contributions in the context of related work, argue for why our results were novel and significant, and get our paper as polished as possible. Three months later, I was delighted to learn that our paper on studying causes of bug fixes was accepted at a top-tier conference where only 14 percent of all papers submitted that year were accepted.

But Tom wasn't done yet! Since he was a newly-hired researcher at MSR, he was eager to establish his reputation by publishing more follow-up papers. Over the next few years, we used the results from my summer 2009 internship to write two additional top-tier conference papers, one about bug report reassignments and another about bug report reopenings (which won a *Best Paper Award*).

~

My success in doing empirical software measurement research at MSR with Tom (resulting in three top-tier papers) was a satisfying redemption from the failures that I had experienced when working in this same subfield throughout my second year (resulting in two rejections followed by a shorter-length, second-tier paper). Since I had not grown much smarter between those two contrasting experiences, I give most of the credit for my internship project's success to two sources: Microsoft and Tom.

First, as an intern at MSR, I had access to a rich array of internal data sets about Microsoft's software bugs and personnel files. There was no way that I could have gotten access to those confidential data sets as an outsider. The richness of the Microsoft data sets enables MSR researchers such as Tom to more easily obtain groundbreaking publishable results than their competitors who don't have access to such data. In contrast, when I was working with Dawson, the Linux data sets I obtained were much smaller and of lower quality, since open-source software projects usually don't maintain records as meticulously as one of the world's largest software companies.

Second, Tom deserves lots of credit: Since he was a veteran insider in the empirical software measurement subfield, he knew how to advise me as a technical mentor and also how to craft the nuances of our paper submissions to maximize their chances of acceptance. In contrast, Dawson was an outsider who merely had a passing interest in these topics, so he had neither the motivation nor the abilities to advise projects in this subfield (even though he was world-famous in another subfield—software bug-finding).

During my second year, I lamented about how hard it was for Dawson and me to publish our work, since we had to compete with hordes of professionals who specialized in empirical software measurement. Now, it felt amazing to finally experience what it was like to be on the winning team working alongside one of those professionals.

~

Even though I had a wonderful summer "intermission" from my Ph.D. program, I still didn't have any plans for my dissertation project when I returned to Stanford in the fall. All I knew was that I didn't want to keep working on Klee, but I had no idea what I could do that was both personally motivating and, more importantly, publishable.

I contemplated trying to extend my current internship work into my dissertation. However, I ultimately concluded that it would be too hard to publish more papers once I returned to Stanford and no longer had access to Microsoft's internal data sets. In an ideal world, I would have been able to do all of my dissertation work within MSR. This option didn't seem feasible, though, since I didn't know any students who had previously done so.

As a last-ditch effort, I contacted my former internship manager at Google to ask whether I could become an intern again and access Google's internal software bug data sets to do empirical software measurement research. He seemed receptive to my idea, but I didn't follow-up with the proposal since it seemed unlikely to pan out: He wasn't an academic researcher himself, and I didn't know anybody else at Google who would be willing to support such a special arrangement. Thus, I decided not to pursue empirical software measurement for my dissertation, so the three papers that I eventually published from my MSR internship didn't help me graduate. However, this experience was still useful for improving my research and technical writing skills.

Desperate to generate another plausible dissertation idea, I spent my nights and weekends throughout the summer reading research papers and brainstorming at coffee shops. At one point, I even thought about creating a dissertation project based on Klee-like ideas but without using the Klee tool itself. This scheme would allow me to free myself from Klee while still capturing some of Dawson's interest. Unfortunately, I wasn't able to generate any substantive ideas along those lines that hadn't already been published.

And then, on July 24, 2009—halfway through my internship—inspiration suddenly struck. In the midst of writing computer programs in my MSR office to process and analyze data, I came up with the initial spark of an idea that would eventually turn into the first project of my dissertation. I frantically scribbled down pages upon pages of notes and then called my friend Robert to make sure my thoughts weren't totally ludicrous. At the time, I had no sense of whether this idea was going to be taken seriously by the wider academic community, but at least I now had a clear direction to pursue when I returned to Stanford to begin my fourth year.

Year Four: Reboot

Throughout my second year of Ph.D. and my summer 2009 internship at MSR, I performed empirical software measurement research by writing computer programs in a popular language called Python to process, analyze, and visualize data sets. After writing hundreds of these sorts of Python programs, I noticed that I kept facing similar inefficiencies over and over again. In particular, I had to tediously keep track of a multitude of data files on my computer and which programs they depended upon; I also needed to make my programs unnecessarily complex so that they could execute (run) quickly after each round of incremental changes to my analyses. After letting these observations simmer in the back of my mind during the summer, the following idea suddenly came to me on that quiet July afternoon at MSR: By altering the Python run-time environment (called an *interpreter*) in some innovative ways, I could eliminate many of these inefficiencies, thereby improving the productivity of computational researchers who use Python. I named my proposed modification to the Python interpreter "IncPy," which stands for ***Inc****remental* ***Py****thon*.

~

Like any pragmatic researcher, the first thing I did after coming up with my IncPy idea (after calming down from the initial excitement) was to frantically search the Web for research papers describing related

work. Thankfully, nobody had created exactly what I was planning to create, but there were some past research projects with a similar flavor. That was fine, though; no idea is truly original, so there will always be related projects. However, in order to eventually publish, I had to make a convincing case for how IncPy was different enough from similar projects. Within a few days, I had sketched out an initial project plan, which included arguments for why IncPy was unique, innovative, and research-worthy.

Since I still had over a month left at MSR, I took advantage of the fact that I was around lots of smart colleagues who were also performing similar kinds of data analysis programming for their research. I grabbed coffee with several colleagues and interviewed them about their data analysis work habits and inefficiencies. I then pitched them my IncPy idea and discussed possible refinements to make it both more useful and also more interesting from a research perspective. These early conversations helped boost my confidence that I was on the right track, since other people shared my frustrations with performing data analysis and my enthusiasm for the ideas that IncPy embodied.

For the rest of the summer, I spent my nights and weekends at coffee shops refining my fledgling IncPy idea, strengthening its "marketing pitch," and getting more feedback from MSR colleagues. I emailed drafts of my idea to Dawson, but I didn't actually care how enthusiastic he was about it since this was going to be my own undertaking. I wasn't asking for his permission; I was just informing him about what I planned to do once I returned to Stanford in the fall.

~

I rebooted my Ph.D. career as I began my fourth year at Stanford, severing my ties to the previous three years and starting anew. No more working on already-established research projects, no more trying to scheme up ways to align with the supposed interests of professors

and senior colleagues, and no more worrying about what kinds of research the academic establishment liked to see. I was now hell-bent on implementing my new IncPy idea, turning it into a publishable paper, and making it the first part of my dissertation.

Although I was filled with newfound excitement and passion, a part of me was scared because I was breaking away from the establishment to do something new and unproven without any professor support. Most Ph.D. students in my department work on projects that their advisors or other professors are interested in, since it's much easier to publish papers with the help of professors' enthusiasm and expertise. However, even without professor backing, my hunch was that IncPy could become a publishable idea; I had accumulated enough battle scars from the past three years of research failures to develop better intuitions for which ideas might succeed. Trusting this gut instinct became the turning point of my Ph.D. career.

On a pragmatic note, I still kept Dawson as my advisor since he was okay with me doing my own project and occasionally giving me high-level feedback on it. Since I was still self-funded by fellowships, I didn't have an obligation to continue working on Klee like the rest of Dawson's students did. My relationship with Dawson was much more hands-off throughout the rest of my Ph.D. years. We probably met less than a dozen times to talk about research; I was mostly working on my own or seeking out collaborations with others. I didn't want to formally switch advisors, because then I would need to "pay my dues" all over again with a new research group. Besides, I didn't know of any professors in my department who were excited by IncPy-like ideas, or else I might have contemplated switching.

To eventually graduate, I needed to form a thesis committee consisting of three professors who agree to read and approve my dissertation. Most students simply have their advisors choose committee members for them, since they are working on "official" advisor-sanctioned projects. Since I was going rogue and not working on Dawson's Klee

project, I didn't have that luxury. I tried hard to find professors both in my own department and in other related departments (e.g., statistics, bioinformatics) to serve on my committee. I cold-emailed a few professors and attempted to reach others via their current students. I even made a PowerPoint slide deck to pitch my dissertation proposal to potential committee members, but unfortunately, no professor was interested in meeting with me. However, instead of giving up and crawling back to a traditional project with more institutional support, I took a risk by marching forward with IncPy and hoping that the thesis committee issue would sort itself out later.

~

Taking inspiration from my HCI (Human-Computer Interaction) work with Scott and Joel during my second year, as soon as I returned to Stanford in the fall of 2009, I began interviewing colleagues who wrote Python programs to analyze data for their research. My goal was to discover what their programming-related inefficiencies were, and how IncPy could eliminate those inefficiencies. I also had some friends arrange for me to give talks on IncPy—which was only a half-baked idea at the time—at their lab group meetings. My initiative in conducting interviews and making presentations at this preliminary stage was helpful both for providing fresh ideas and also for refining IncPy's "marketing pitch." I'm immensely grateful for the friends who helped me get my project off the ground when I had nothing to show except for a few shabby PowerPoint slides.

I grew more and more encouraged as I found out that researchers in a variety of computational-based fields such as machine learning, pharmacology, bioengineering, bioinformatics, neuroscience, and ocean engineering all performed similar sorts of data analysis for their research and could benefit from a tool such as IncPy. After a few weeks of interviews and subsequent refinements to my design plans, I felt confident that I had a good enough pitch to "sell" the project convincingly in

a future paper submission. The argument I wanted to make was that lots of computational researchers in diverse fields struggle with several common inefficiencies in their daily programming workflow, and IncPy provides a new kind of fully automated solution to such inefficiencies that nobody else has previously implemented. This initial pitch would later evolve into the theme of my entire dissertation.

With an overall strategy in place, I was ready to begin the thousands of hours of hard labor—grinding—required to turn IncPy from an idea into a real working prototype tool. I had spent the end of my summer playing the "professor role" of sketching out high-level designs, giving talks, and refining conceptual ideas. Now I was ready to play the "student role" of massively grinding throughout the next year to implement IncPy.

~

By now it should be clear that having a good idea is necessary but nowhere near sufficient for producing publishable research. Junior researchers—usually Ph.D. students—must spend anywhere from hundreds to thousands of hours "sweating the details" to bring that idea to fruition. In computer science research, the main form of labor is performing computer programming to build, test, and evaluate new software-based prototype tools and techniques. The nearly ten thousand hours I had spent doing many types of programming over the past decade—in classroom, hobby, research lab, and engineering internship settings—prepared me to endure the intensely intricate programming required to implement research ideas such as IncPy.

Implementing IncPy involved some of the grimiest programming grind that I had ever done. If I didn't have all of those hours of trial-by-fire training over the past decade, then I would have never even attempted such a labor-intensive project. Other people have undoubtedly recognized the same inefficiencies that I observed in computational research workflows, but what set me apart from would-be

competitors was that these people didn't have the deep programming expertise required to create a *fully automated* solution such as IncPy. At best, they might create semi-automated solutions that would not be substantive enough to publish in a respectable conference.

Even though I wasn't in love with my previous research projects earlier in grad school and during college, the technical skills and judgment that I gained from those experiences made it possible for me to now implement my own ideas that I truly cared about. Over the next three years (2009 to 2011), I grinded non-stop on creating five new prototype tools to help computational researchers (IncPy was the first), published one or more first-author papers describing each tool, and then combined all of that work together into a Ph.D. dissertation that I was extremely proud of. Those three years—my latter half of grad school—were the most creative and productive of my life thus far, in stark contrast to my meandering first half of grad school.

I became fiercely self-driven by an enormous amount of *productive rage*. I turned steely, determined, and ultra-focused. Every time I reflected back on the inefficiencies, failures, and frustrations that I had endured during my first three years of grad school, I would grow more enraged and push myself to grind even harder; I was motivated by an obsessive urge to make up for supposedly lost time. Of course, those early years weren't actually lost; without those struggles, I wouldn't have gained the inspiration or abilities to create the five projects that comprised my dissertation.

~

However, as I was about to begin my fourth year of Ph.D. in September 2009, I had no sense of what awaited me in the future and definitely no grand plans for a five-project, 230-page dissertation. I didn't even know whether any professors would take my unproven ideas seriously enough to agree to serve on my thesis committee. All I

wanted to do was implement IncPy, try to get it published, and then figure out my next move when the time came.

I was about to start implementing (programming) IncPy when an unexpected bit of fortune struck. At the time, I had no idea that this one minor event would lead to a cascade of other good luck that would pave the way for my graduation. One day at lunch, my friend Robert told me he was planning to submit a paper about his new research project to a workshop whose deadline was in 2.5 months.

Normally, I wouldn't have thought twice about such an announcement for two reasons: First, Robert's research topic (in a subfield called *data provenance*) was unrelated to IncPy, so where he planned to submit papers had no relevance to me. Second, a published *workshop paper* usually does not "count" as a dissertation contribution in our department. Workshop papers are meant to disseminate early ideas and are not scrutinized as rigorously as conference papers. Whereas a conference paper submission has an 8 to 30 percent chance of acceptance, a workshop paper submission has a 60 to 80 percent chance. Most professors in our department don't encourage students to submit to workshops, since if the paper gets accepted (which is likely), the professor must pay around $1,500 of travel, hotel, and registration costs using their grant money for the student to attend and give a talk at the workshop. It costs about as much to send a student to a workshop as to a conference, and conference papers are much more prestigious both for the student and the professor. Thus, top-tier computer science professors strongly encourage students to publish more selective conference papers and eschew workshops altogether.

I asked Robert why his advisor encouraged him to submit his early-stage idea to a workshop rather than developing it further and submitting to a conference at a later date. He said that it was partly out of convenience, since the workshop was located in nearby San Jose. His entire research group planned to attend since it was only 20 miles from Stanford, so he also wanted to present a paper there.

Out of curiosity, I looked at the workshop's website to see what topics were of interest to the organizers. Although it was a workshop on data provenance (Robert's research area), the topics list included a bullet point entitled "efficient/incremental recomputation." I thought to myself: *Hmmm, IncPy provides efficient incremental recomputation for Python programs, so if I pitch my paper properly, then maybe it's sort of appropriate for this workshop!* Since I didn't need to travel to attend the workshop, I wasn't afraid to ask Dawson to pay for my registration fee if my paper was accepted. I emailed Dawson my plans to submit a paper to this workshop, and he wrote back with a lukewarm response. As expected, he wasn't enthusiastic about workshops in general, but he was okay with me submitting since he respected the workshop's program committee co-chair, a Harvard professor named Margo (who would later play a pivotal role in helping me to graduate).

The timing was perfect: I now had 2.5 months to super-grind on implementing a first working prototype of IncPy and then write a workshop paper submission. Since I knew that the bar for workshop paper acceptance was a lot lower than for a conference paper, I wasn't too stressed. I just needed to get a basic prototype working reasonably well and anecdotally discuss my experiences. I discovered that this strategy of finding and setting short-term deadlines for myself would work wonders in keeping me focused throughout the rest of my Ph.D. years. Without self-imposed deadlines, it becomes easy to fall into a rut and succumb to chronic procrastination.

My paper was accepted with great reviews, and I attended the workshop in February 2010 to give a 30-minute talk on it. In particular, I was honored that Margo, the program committee co-chair, personally praised my paper and mentioned how it was related to a new Python-based project that her student Elaine was starting. Since Elaine wasn't at the workshop, Margo gave me Elaine's email address and suggested for us to get in touch.

At first, I was afraid that Elaine would be my direct competition and publish a conference paper before I did, which would make it harder for me to publish a conference paper on IncPy: Since being first is highly valued in academia, once another researcher publishes a similar idea and "scoops" you, then it becomes much harder to get your own idea published. But after exchanging some reconnaissance emails and video chatting with her, I was relieved to find out that she didn't have plans to gear up for a conference paper submission. Also, her tool lacked the fully automatic capabilities that set IncPy apart from related work. Once we realized that we weren't rivals, we quickly became friends. I'm glad that Elaine and I kept in touch over the next few years, since she would be partially responsible for reuniting me with Margo towards the end of my Ph.D. journey.

~

Although giving a talk on my IncPy workshop paper was great for getting feedback and especially for meeting Margo, that paper didn't count as a "real" publication for my dissertation. I knew that I still needed to publish this work in a conference that professors in my department recognized as legitimate. The biggest difference between a workshop and a conference paper is that a conference paper must have a convincing *experimental evaluation* that shows the effectiveness of the tool or technique being described in the paper. A paper's evaluation section can come in many flavors ranging from measurements of run-time performance to a controlled laboratory study of user behavior. Since many researchers come up with similar ideas, reviewers carefully scrutinize how those ideas are implemented and experimentally evaluated when deciding which papers to accept or reject.

Even at the start of my IncPy project, I knew that it would be difficult to present a convincing evaluation because the argument I wanted to make—that IncPy can improve the productivity of computational researchers—was a subjective and fuzzy notion. After reading several

other papers that presented similar productivity improvement claims, I devised a two-part evaluation strategy:

1. **Case Studies**: Get a collection of programs written in the Python language from a variety of computational researchers and then simulate the productivity improvements they would have achieved if they had used IncPy during their research rather than regular Python.

2. **Deployment**: Get some researchers to use IncPy rather than regular Python in their daily work and have them report if and how IncPy improved their productivity.

The next relevant top-tier conference submission deadline was in seven months, so I aimed to have a paper ready by then. I spent lots of time trying to find case study subject programs and potential users for deployment. Without those, there would be no evaluation, no conference publication, no dissertation, and no graduation. (Of course, I still spent the majority of my waking hours doing grimy programming, debugging, and testing to implement IncPy.)

I learned to be part-salesman and part-beggar, persistently asking colleagues whether they had Python programs I could use for case studies or, even better, whether they would be willing to install and use IncPy on a daily basis for their research and report their impressions to me. As expected, I mostly got "No" replies, but I politely asked for recommendations of other people I could contact. I also invited myself to give several more talks at various lab group meetings to drum up interest in IncPy. After a few months of "begging," I obtained Python programs from half a dozen researchers in a diverse variety of fields, which was enough to begin my case studies. I appreciated everyone who helped me out during that time, since they had nothing to gain besides the goodwill of an unknown Ph.D. student.

~

Although case studies might have been sufficient for the evaluation section of my paper submission, what I truly craved was *deployment*—getting real researchers to use IncPy. Not only did I feel that deployment would make for a stronger evaluation, but more importantly, I genuinely believed that IncPy could improve the day-to-day productivity of computational researchers. The majority of research prototype tools are built to demonstrate a novel idea and then discarded, but I wanted IncPy to be practical and enduring. I didn't just dream up IncPy in a vacuum for the sake of publishing papers; it was inspired by real-world problems I faced when performing programming for my research, so I wanted real-world people to actually use it.

In spite of my idealism, I understood why almost no research prototype tools get successfully deployed. The underlying reason is that people don't want to try a new tool unless they can instantly see major benefits without any drawbacks; researchers simply don't have the time or resources to get their prototypes working well enough to meet these stringent requirements. In particular, my target users are happy enough using regular Python—despite its inefficiencies—that they don't want to take a risk by switching to IncPy. To convince someone to try IncPy, I would need to guarantee that it works better than regular Python in all possible scenarios. While that's true in theory, in reality IncPy is a research prototype tool being maintained by one student, so it's bound to have numerous bugs. In contrast, the official Python project is over twenty years old and being maintained by hundreds of veteran programmers, so it's much more stable and reliable. As soon as someone hits a single IncPy bug, they will immediately dismiss it as lame and switch back to using regular Python. I knew the odds were against me, but I didn't care.

After failing to find anybody at Stanford who was willing to install and use IncPy, I looked for leads at nearby universities. In March 2010, I cold-emailed a few Ph.D. students at UC Davis (a two-hour drive from Stanford) who were friends with one of my former MSR

intern colleagues. I appealed to their sense of altruism at helping out a fellow grad student in need and managed to invite myself over for a visit to advertise IncPy. A few gracious professors even agreed to meet with me, including one I had met at the data provenance workshop. Although I received more helpful feedback, I didn't manage to find any subjects for case studies or deployment.

I spent the night at UC Davis and planned to return to Stanford the next morning. Then I thought of an impulsive idea—cold-emailing Fernando, a research scientist at UC Berkeley who was passionate about the use of Python in computational research. A few months earlier, a fellow grad student who attended one of my IncPy talks emailed me a link to one of Fernando's blog posts, and I jotted down a reminder to contact Fernando sometime in the future. Now seemed like a good time: Since UC Berkeley was located between UC Davis and Stanford, I could potentially drop by his office on the drive back home. I cold-emailed Fernando and asked if he had time to chat the next morning. It was a long shot, but he agreed to meet with me. I had a wonderful initial one-hour meeting with Fernando; it felt great to see a well-established senior scientist support my IncPy idea.

The most significant outcome of our first meeting was that Fernando invited me to return to UC Berkeley the following month to give a talk on IncPy. He told me that my audience would consist of neuroscientists who were using Python to run their computational research experiments. When I gave my one-hour talk, I was a bit taken aback by three researchers continually interrupting and pestering me with detailed questions about how well IncPy worked in practice. At first, I thought those guys were being overly pedantic, but afterwards they came up to me and expressed a strong interest in trying IncPy. They lamented about how they're currently suffering from the exact kinds of inefficiencies that inspired me to create IncPy in the first place! It turned out that they weren't trying to be annoying during my talk; they just wanted to understand the details to assess whether

it was feasible to deploy IncPy on their research lab computers.

I was thrilled by the possibility of getting my first set of real users. I exchanged some emails with them and offered to drive up to UC Berkeley again to help them install and set up IncPy. My first email started with a cautiously optimistic tone:

> Thanks for expressing an interest in becoming the first real user of my IncPy memoizing Python interpreter! I'm really enthusiastic about getting IncPy to a state where you can use it in your daily workflow. I think the main issues will be mostly grunge work with installation / setup / configuration, which I am more than happy to deal with in order to give you the smoothest possible user experience.

By the time I tried installing IncPy on the Berkeley neuroscientists' computers, I had been building and testing it for seven months, so I felt reasonably confident that it would work for them. However, shortly after installation, we discovered that IncPy was not compatible with dozens of third-party Python add-ons (called *extension modules*) that these scientists were using in their daily work. In my own private testing, I tested only the basic use cases without any extension modules. I was hit with a lesson in the harshness of real-world deployment: failures come in unexpected forms, and once the user gets a bad first impression, then it's over! Like most researchers creating prototypes in an ivory tower, I could have never predicted that this unforeseen banal extension module issue would completely derail my first deployment attempt.

I wasn't about to give up, though. I spent the next few weeks redesigning and reimplementing a critical portion of the IncPy code so that it now worked perfectly with any arbitrary Python extension modules. I emailed the UC Berkeley neuroscientists again to ask for a second chance, but I got no response. I had one shot, and I blew it.

~

This disappointment spurred me to keep improving IncPy in practical ways: Along with fixing many subtle bugs, I created an IncPy project website containing a short video demo, documentation, and beginner tutorials. None of these hundreds of hours of effort made any improvements to my original research contribution, but they were necessary if I wanted to get real user anecdotes for an evaluation section of a future paper submission.

After a few months passed, three strangers from different parts of the world found IncPy via its website, downloaded a copy, and used it to speed up some of their research programming activities. Although three is a pathetically small number of users, at least it's more than zero, which is the number of users most research prototypes have. I felt satisfied that IncPy was in a polished enough state—thanks to my post-Berkeley improvements—that strangers were now able to use it without my guidance. These three people emailed me anecdotes about how they found IncPy to be mildly useful for some parts of their work. Their anecdotes weren't strong evidence of effectiveness, but they were better than nothing.

As my fourth year of Ph.D. came to an end in September 2010, I submitted my IncPy paper to a top-tier conference with an evaluation consisting of case studies and three brief deployment anecdotes. Only 14 percent of papers submitted to that same conference last year were accepted. Thus, I knew that my paper would most likely be rejected, both due to those meager odds and especially since IncPy didn't neatly fit into any traditional academic subfield. Nonetheless, it was still wise to first aim for the top tier and then resubmit to a second-tier conference if necessary, since a top-tier publication would carry more weight in my future graduation case.

I still didn't have a clearly paved path to graduation, but at least I was able to start taking charge of my own research agenda rather than tagging along on other people's projects. I was happy that within the past year, I had turned IncPy from the spark of an idea in my mind into

a semi-practical tool that benefited three strangers who downloaded it from the Internet. Even though this minor accomplishment wouldn't help my graduation case at all—professors respect real-world deployment far less than theoretical novelty—it was a somewhat satisfying end to my fourth year.

Year Five: Production

At the beginning of my fifth year (September 2010), I still had nothing to include in my (nonexistent) dissertation. By now, most of my classmates had published at least one dissertation-worthy first-author conference paper. Since I didn't have any dissertation-worthy papers yet (the IncPy paper was still under review), I was afraid that it would take me seven or eight total years to graduate.

Within the next twelve months, though, I would publish four conference papers and one workshop paper (all as the first author), thereby paving a clear path for my graduation. Without a doubt, my fifth year was my most productive of grad school. I was relentlessly focused.

~

In the middle of summer 2010, progress on IncPy was steady, and I was on track to submitting a paper by the September deadline. However, I knew that IncPy would not be substantive enough for an entire dissertation. So besides working towards the upcoming paper deadline, I also spent some time thinking about my next project idea.

I wish I could say that my solo brainstorming sessions were motivated by a true love for the pure essence of academic scholarship. But the truth was that I was driven by straight-up fear: I was afraid of not being able to graduate within a reasonable time frame, so I pressured myself to come up with new ideas that could potentially lead to pub-

lications. I was all too aware that it might take two to three years for a paper to get accepted for publication, so if I wanted to graduate by the end of my sixth year, I would need to submit several papers this year and pray that at least two get accepted. I felt rushed because my fellowship lasted only until the end of this year. After my funding expired, I would need to either find grant funding from a professor and face all of the requisite restrictions (e.g., working on Klee again), or else become a perpetual teaching assistant and delay my graduation even further. Time was running out.

On July 29, 2010, almost exactly one year after I conceived the initial idea for IncPy, I came up with a related idea, again inspired by real problems that computational researchers encounter while performing data analysis. I observed that because researchers write computer programs in an ad-hoc "sloppy" style, their programs often crash for silly reasons without producing any analysis results, thus leading to table-pounding frustration. My insight was that by altering the runtime environment (*interpreter*) of the Python programming language, I could eliminate all of those crashes and allow their sloppy programs to produce partial results rather than no results. I named this proposed modification to the Python interpreter "SlopPy," which stands for ***Slop****py* ***Py****thon*.

Although SlopPy and IncPy are very different ideas, I implemented both by altering the behavior of the Python interpreter. The nearly one thousand hours I had spent over the past year hacking (modifying) the Python interpreter for IncPy gave me confidence that I could implement SlopPy fairly easily. It took me only two months to create a basic working prototype, run some preliminary experiments, and submit a paper to a second-tier conference. I aimed for that conference both because its deadline was conveniently timed and also because I didn't think that SlopPy was a "big" enough idea to get accepted in a top-tier conference.

~

By October 2010, I had two papers under submission. At this point, I had given up all hope of getting a job as a professor, since I still had not published a single paper for my dissertation; competitive computer science faculty job candidates have all already published a few acclaimed first-author papers by this time in their Ph.D. careers. Unless a miracle struck, I would not be able to get a respectable research university job, so I aimed to do only enough work to graduate without worrying about how my resume would appear.

I received the opposite of a miracle: Both my IncPy and SlopPy paper submissions were rejected. I was disappointed but not too shocked, since I had already grown accustomed to paper rejections by this time. There were lots of legitimate criticisms of my work, so I felt that addressing them would strengthen my future resubmissions.

Most notably, I had unwisely framed the pitch for IncPy in a way that led to my paper being reviewed by researchers in a subfield that wasn't as "friendly" to my research philosophy. In theory, technical papers should be judged on their merit alone, but in reality, reviewers each have their own unique subjective tastes and philosophical biases. So I drastically rewrote my introductory pitch with the aim of getting more amicable reviewers and then resubmitted to a second-tier conference to further improve its chances of acceptance. My plan worked, and the IncPy conference paper was accepted—albeit with lukewarm reviews—on my second submission attempt in early 2011.

I later revised and resubmitted my SlopPy paper to a workshop that was being held together with the conference where I would be presenting IncPy. This strategy worked well since it was far easier to get a paper accepted in a workshop than in a conference. Also, Dawson wouldn't need to pay much extra for me to attend the workshop since I was already going to the conference to present IncPy. As expected, the SlopPy workshop paper was accepted, and although it didn't "count" as a standalone dissertation contribution, at least it was better than no publication; I hoped to incorporate this paper as a smaller chapter

in my dissertation to supplement the more substantive chapters, which would all derive from conference papers.

~

Back in October 2010, right after submitting the IncPy and SlopPy papers, I asked Dawson what it would take for me to graduate in the next few years. Predictably, he replied that I needed publications as proof of the legitimacy of my work. He did have one concrete suggestion, though: Another dissertation-worthy project would be for me to combine my interest in Python with Klee-like ideas that he loved in order to create an automated bug-finding tool for Python programs. Since I wasn't keen on returning to Klee in any form, I discarded his suggestion and continued thinking about possible extensions to IncPy and SlopPy that could make for a follow-up paper submission.

By this time, a nascent dissertation theme was beginning to form in my head: Both IncPy and SlopPy were *software tools to improve the productivity of computational researchers*. Thus, to think of my next project idea, I returned to identifying problems computational researchers faced in their work and then designing new tools to address those problems.

Specifically, I noticed that researchers edit and execute their Python programs dozens to hundreds of times per day while running computational experiments; they repeat this process for weeks or months at a time before making a significant discovery. I had a hunch that recording and comparing what changed between program executions could be useful for debugging problems and obtaining insights. To facilitate such comparisons, I planned to extend IncPy to record details about which pieces of code and data were accessed each time a Python program executes, thereby maintaining a rich history of a computational experiment. I also thought it would be cool for researchers to share these *experiment histories* with colleagues so that they can learn from what worked and didn't work during experimental trials.

My gut told me that some ideas along these lines could be innovative and publishable, but I couldn't quite form a crisp research pitch yet; my thoughts were all fuzzy and squishy. I felt stuck, so I sought another meeting with Fernando, whom I first met during my fourth year when I introduced IncPy and gave a talk at his UC Berkeley lab group meeting. Fernando fit me into his schedule, and our one-hour meeting planted the seeds for my next project idea.

~

As soon as I mentioned my ideas about extending IncPy to record Python-based experiment histories, Fernando launched into a passionate sermon about a topic that I had never heard of but was fascinated by: *reproducible research*.

One of the cornerstones of experimental science is that colleagues should be able to reproduce anyone's research findings to verify, compare against, and build upon them. In the past decade, more and more scientists in diverse fields have been writing computer programs to analyze data and make scientific discoveries. Thousands of scientific papers published each year are filled with quantitative findings backed by numbers, graphs, and tables. However, the unspoken shame in modern science is that it's nearly impossible to reproduce or verify any of those findings, since the original computer code and data sets used to produce those findings are rarely available. As a result, lots of papers containing egregious errors—due to both honest mistakes and outright fraud—have gone unchallenged, sometimes resulting in scientific claims that have led to human deaths. In recent years, reform-minded scientists such as Fernando have been trying to raise awareness for the importance of reproducible research in the computational sciences.

Why is reproducibility so hard to achieve in practice? A few ultra-competitive scientists purposely hide their computer code and data to fend off potential rivals, but the majority are willing to share code and data upon request. The main technical barrier, though, is

that simply obtaining someone's code and data isn't enough to rerun and reproduce their experiments. Everyone's code needs a highly-specific *environment* in which to run, and the environments on any two computers—even those with the same operating system—differ in subtle and incompatible ways. Thus, if you send your code and data to colleagues, they probably won't be able to rerun your experiments.

Fernando liked my IncPy experiment history recording idea because it could also record information about the software environment where an experiment originally occurred. Then researchers who use Python can send their code, data, and environment to colleagues who want to reproduce their experiments. I came out of that meeting feeling pumped that I had found a concrete application for my idea. The reproducible research motivation seemed compelling enough to form the storyline for a second independent IncPy-based paper submission and dissertation contribution.

As I was jotting down more detailed notes, a moment of extreme clarity struck: *Why limit this experiment recording to only Python programs*? With some generalizations to my idea, I could make a tool that enables the easy reproduction of computational experiments written in any programming language! Still in a mad frenzy, I sketched out the design for a new tool named "CDE," which stands for ***C****ode*, ***D****ata*, and ***E****nvironment*.

~

When I told my idea to Dawson, he responded favorably but challenged me to think even bigger: *Why limit CDE to targeting only scientists' code? Why not make it a general-purpose packaging tool for all kinds of software programs?* Those were wise words. A variety of software creators and distributors—not only scientists—have trouble getting other people to run their programs due to the same environment incompatibility problem, which is affectionately known as "dependency hell." Dependency hell is especially widespread on Linux-

based operating systems due to the multitude of semi-incompatible Linux variants that people use; programs that run on one person's Linux computer are unlikely to run on someone else's slightly different Linux computer. With some adjustments to my original idea, CDE could enable anybody to package up their Linux programs so that others can run them without worrying about these environment mismatches. I felt thrilled that CDE could potentially alleviate the decades-old problem of dependency hell on Linux.

As my usual reality check, I scoured the Web for related work, looking for both research prototypes and production-quality tools with similar functionality. To my relief, there wasn't much prior work, and CDE stood out from the sparse competition in two important ways: First, I designed CDE to be much easier to use than similar tools. As a user, you create a self-contained code, data, and environment package by simply running the program that you want to package. Thus, if you can run a set of programs on your Linux computer, then CDE enables others to rerun those same programs on their Linux computers without any environment installation or configuration. Second, the technical mechanism that CDE employs—a technique called *system call redirection*—enables it to be more reliable than related tools in a variety of complex, real-world use cases.

At this point, CDE existed only as a collection of notes and design sketches, but I sensed its potential when I realized that it was conceptually simpler, easier to use, and more reliable than all other tools in existence. A part of me was shocked and paranoid: *Why hasn't anybody else implemented this before?!? This idea seems so obvious in retrospect!* One possible reason I dreaded was that nobody had previously built something like CDE because it was impossible to get the details right to make it work effectively in practice. Maybe it was one of those ideas that looked good on paper but wasn't practically feasible. I figured that there was no better way to find out than to try implementing CDE myself.

Over three intense weeks spanning October and November 2010, I super-grinded on creating the first version of CDE. As I suspected, although the research idea behind CDE was straightforward, there were many grimy programming-related contortions required to get CDE working on real Linux programs. I lived and breathed CDE for those weeks, forgetting everything else in my life. I programmed day and night, often dreaming in my sleep about the intricate details that my code had to wrestle with. Every morning, I would wake up and jump straight to programming, feeling scared that this would finally be the day when I hit an insurmountable obstacle proving that it was, in fact, impossible to get CDE to work. But the days kept passing, and I kept getting closer to my first milestone: demonstrating how CDE allows me to transfer a sophisticated scientific program between two Linux computers and reproduce an experiment without hassle.

I felt ecstatic when, after three weeks of coffee-fueled fully-immersed grinding, I finally got CDE working on my scientific program demo. At that point, I knew that CDE had the potential to work on many kinds of real-world Linux programs if I kept testing and improving its code. I made a ten-minute video demo introducing CDE, created a project website containing the video and a downloadable copy of CDE, and then emailed the website link to some friends. Unbeknownst to me, one of my friends posted the following blurb about CDE on Slashdot, a popular online computer geek forum:

> A Stanford researcher, Philip Guo, has developed a tool called CDE to automatically package up a Linux program and all its dependencies (including system-level libraries, fonts, etc!) so that it can be run out of the box on another Linux machine without a lot of complicated work setting up libraries and program versions or dealing with dependency version hell. He's got binaries, source code, and a screencast up. Looks to be really useful for large cluster/cloud deployments as well as program sharing.

Within 24 hours, the Slashdot forum thread had hundreds of messages, and I began receiving dozens of emails from Linux enthusiasts around the world who downloaded and tried CDE, including gems such as: "i just wanted to tell u that U ROCK! i'm really impressed to see this idea working. I will promote the usage of it on my linux community hear [sic] in Tijuana, Mexico." These unfiltered, off-the-cuff compliments from actual users meant more to me than any fellow researcher praising my previous ideas or papers.

~

From a research standpoint, my mission was now accomplished: I successfully built an initial prototype of CDE and demonstrated that it worked on a realistic example use case. The common wisdom in most applied engineering fields is that research prototypes such as CDE only serve to demonstrate the feasibility of novel ideas. The job of a researcher is to create prototypes, experimentally evaluate their effectiveness, write papers, and then move on to the next idea. As a researcher, it's foolish to expect people to use your prototypes as though they were real products; if your ideas are good, then professional engineers might adapt them into their company's future products. At best, a few other research groups might use your prototypes as the basis for building their own prototypes and then write papers citing yours (e.g., over a dozen other university research groups have extended the Klee tool and written papers about their improvements). But it's almost unheard of for non-researchers to use research prototypes in their daily work. In sum, the purpose of academic research is to produce validated ideas, not polished products.

Thus, the wise course of action at the time would have been to submit a paper on CDE and then move on to generating a new idea, implementing a new prototype, submitting a new paper, and repeating until I had enough content to fill up a dissertation. I did submit and publish two conference papers on CDE (a short introductory paper

and a longer follow-up paper). But rather than moving on to a new project idea like a prudent researcher would do, I dedicated most of my fifth year to turning CDE into a production-quality piece of software.

I had an urge to make CDE useful for as many people as possible. I didn't want it to languish as yet another shoddy research prototype that barely worked well enough to publish papers. I knew my efforts to polish up CDE wouldn't be rewarded by the research community and might even delay my graduation since I could've spent that time developing new dissertation project ideas. But I didn't care. Since I was still funded by fellowships for the rest of the year, I had full freedom to spend my time as a pro bono software maintainer rather than as a traditional researcher tied to grant funding.

Back in my fourth year, I desperately wanted people to use IncPy, so that's why I felt thrilled to get three measly users. Even though almost nobody ended up using IncPy, my irrational desire to make it into a real-world tool led me to reach out to Fernando at UC Berkeley, and it was Fernando who inspired me to create CDE. Now at the beginning of my fifth year in November 2010—within a few days of having my video demo appear on the popular Slashdot website—CDE already had dozens of users and the potential for a lot more. Judging from early email feedback, I realized that I had created something that people wanted to use in a variety of settings I hadn't originally predicted. In short, CDE struck a chord with all sorts of Linux users who were tired of suffering from dependency hell.

~

I spent the majority of my fifth year fixing hundreds of bugs to make CDE work on a dizzying array of complex Linux programs; polishing up the documentation, user manual, and FAQ to make it easier to use; exchanging emails and even a few phone calls with users from around the world; and giving numerous talks and sending "marketing" emails to attract new users.

At present (summer 2012), CDE has been downloaded and used by over 10,000 people. I've received hundreds of emails from users with feedback, new feature requests, bug reports, and cool anecdotes. Although this isn't a large number of users for a commercial software product, it's extremely large for a free and open-source research tool being maintained by a single grad student.

Here are some kinds of people who have emailed me thank-you notes and anecdotes about how they used CDE to eliminate Linux dependency hell in their daily work:

- Research scientists at NASA (Ames and JPL research centers)
- Scientists in fields such as plant biology and medical informatics
- Scientists deploying computational experiments to the European Grid distributed computing infrastructure
- Engineers prototyping experimental code at software companies
- Open-source software creators and distributors
- Linux computer system administrators
- Linux hobbyists running software on incompatible variants of Linux-based operating systems
- Computer security analysts at a large anti-virus company
- A volunteer programmer on the *One Laptop per Child* nonprofit technology project
- Computer science Ph.D. students distributing their research prototype software
- University programmers developing software such as those for medical visualization and protein crystallography
- University teaching assistants packaging up their programming-based class assignments

Those few months were by far the most enjoyable period of my Ph.D. years, even though I knew that none of my software maintenance activities would contribute towards my dissertation. However, after the initial success of CDE, I no longer cared if my graduation was delayed by a year or more due to lack of additional publications; I got so much satisfaction from knowing that a piece of software I had invented could improve many people's computing experiences.

~

CDE also enabled me to achieve one of my long-time nerd dreams: to give a Tech Talk at Google that was broadcast online on YouTube. Since the beginning of grad school, I loved watching Google Tech Talks online on a wide range of academic subjects. I dreamed of the day when I could give such a talk, but I didn't get my hopes up since it seemed like Google employees invited only famous professors and engineers—not unknown grad students—to give these talks.

One day while scouring the Web for projects related to CDE, I serendipitously noticed that my former summer 2007 Google internship manager had recently published a paper in a reproducible research workshop. I emailed him to advertise CDE as a tool for facilitating reproducible research and to ask whether his colleagues might be interested in trying it. To my pleasant surprise, he responded with a talk invitation ending in a winking smiley face: "I took a look at your tool. Looks interesting enough! Would you be interested in giving a Tech Talk about it here at Google? I would certainly help organizing and advertising. You never know how many attendees you get, could be 100, could be none ;-)"

I spent more time preparing for my Google Tech Talk than for any previous talk, since I knew that it would be recorded. My talk went quite well, and afterwards a Google engineering manager (whom I had never met) pulled me aside to ask more detailed follow-up questions. It turned out that he was interested in alleviating this Linux dependency

hell problem within the company, so that's why he loved my talk. He offered me an internship where I could spend the upcoming summer working on CDE at Google.

I was flattered by his offer and took some time to deliberate. Professors in my department usually discourage late-stage Ph.D. students from doing internships, since they want students to focus on finishing their dissertations. Also, at the time of my offer, I hadn't yet published any first-author papers for my dissertation (several papers were under review), so I was afraid that leaving Stanford for the summer might give Dawson the impression that I wasn't serious about trying to publish and graduate. However, my gut intuition was that this was a unique and well-timed opportunity that I couldn't turn down: I would be paid a great salary to spend my summer continuing to work on my own open-source software project. In contrast, almost all interns—including myself back in 2007—were expected to work on internal company projects assigned by their managers. I talked to Dawson about my conflicting feelings, and he was quite supportive, so I accepted the internship offer.

I spent a super-chill summer of 2011 at Google dedicating almost all of my workdays to improving CDE, getting new users, and finding creative uses within Google. For part of the summer, I worked closely with another Google engineer who found CDE useful for his own work, which was a great impetus for me to fix additional bugs and to improve the documentation. By this point, I was no longer developing new CDE-related research ideas: I was just sweating the details to continue making it work better. I finally stopped working full-time on CDE after my summer internship ended and my sixth year of Ph.D. began.

Out of the five projects that comprised my dissertation, CDE was my favorite since it was a simple, elegant idea that turned into a practical tool with over 10,000 users. It was by far the least sophisticated from a research standpoint, but it was the most satisfying to work on due to its real-world relevance.

~

Back at the beginning of my fifth year—long before the IncPy, SlopPy, and CDE papers had been published—I hatched a backup plan in case my own projects failed. I cold-emailed Jeff, a new assistant professor in my department who shared some of my research interests, to ask whether he was open to collaborating on a project that might contribute to my dissertation. The two key "selling points" I mentioned were that I had my own funding and that I wanted to aim for a top-tier paper submission deadline for a conference that he liked. In exchange, he needed to serve on my thesis committee.

As expected, Jeff took me up on my offer. It was a great deal for him since our motivations were well-aligned: I was a senior student who needed to publish to graduate, and he was an assistant professor who needed to publish to earn tenure. Even better, he didn't need to fund me from his grants. And best of all, I was open to working on whatever project he wanted, since my primary solo projects already gave me the independence that I craved.

I was hedging my bets with this plan: If my IncPy, SlopPy, and CDE projects couldn't get published, then at least I would still have a "legitimate" professor-sanctioned project with Jeff, who was now one of my thesis committee members. Jeff and I decided that the best strategy was for me to build upon an interactive data reformatting tool called Wrangler that one of his other students created last year.

Towards the end of my fifth year, I took a break from CDE and spent 2.5 months creating some new extensions to Wrangler. My enhanced version was called "ProWrangler," which stands for ***Pro****active* ***Wrangler***. After implementing the ProWrangler prototype and evaluating its efficacy with controlled user testing on fellow students, I wrote up a paper submission to a top-tier HCI conference with the help of Jeff and the other creators of the original Wrangler tool.

In the midst of my summer 2011 Google internship, I received the happy news that our ProWrangler paper had been accepted with great

reviews. By far the biggest contributor to our success was Jeff's amazing job at writing both our paper's introduction and the interpretation of our evaluation results. Our user testing had failed to show the productivity improvement effects that we originally hoped to see, so I was afraid that our paper would be rejected for sure. But miraculously, Jeff's technical writing and argument framing skills turned that near-defeat into a surprise victory. The reviewers loved how we honestly acknowledged the failures of our evaluation and extracted valuable insights from them. Without a doubt, our paper would have never been accepted if not for Jeff's rhetorical expertise. He had a lot of practice, though. Back when he was a Ph.D. student, Jeff published 19 papers mostly in top-tier conferences, which is five to ten times more than typical computer science Ph.D. students. That's the sort of intensity required to get a faculty job at a top-tier university like Stanford.

~

Throughout my fifth year, I had to carefully split my time between developing new ideas, implementing prototypes, and submitting, revising, and resubmitting papers for four projects—IncPy, SlopPy, CDE, and ProWrangler—whose relevant conference submission deadlines were spread throughout the year. Even though I spent a lot of time nurturing CDE, I had to switch to focusing on other projects whenever deadlines arose. By summer 2011, all four projects were successfully published, usually after several rounds of paper revisions. I felt relieved that my intricate planning had paid off and that a full dissertation now seemed almost within reach.

Year Six: Endgame

At the end of my fifth year—right before I went to Google for the summer—I met with Dawson and asked him what it would take for me to graduate within the next year. At the time, IncPy and CDE were published as second-tier conference papers, SlopPy was a workshop paper, and the ProWrangler paper submission was under review. Dawson expressed concerns that my publication record still wasn't sufficient to graduate and that I needed one more substantive contribution to round out my dissertation. His expectations seemed reasonable, so my plan was to return to Stanford in the fall and spend a few months working on new research that could complete my dissertation. My fear, though, was that I was already exhausted from my past year of super-grinding and had no new project ideas brewing. So I went into scheming mode once again, thinking of ways to get that final as-yet-unknown piece of work that would enable me to graduate.

As part of my strategy, I also wanted to find a third (and final) thesis committee member who could strongly vouch for my graduation case. Most Ph.D. students in my department don't need to do so much planning because they work on advisor-sanctioned projects. They don't stress about who their other two thesis committee members are since their advisor vouches for them and the other members usually agree. However, my situation was unique since I hadn't been working on projects that Dawson was passionate about, so I couldn't count on him to wholeheartedly endorse my work. Having Jeff on my

committee helped since he was personally invested in our ProWrangler project and could vouch for its legitimacy. But I still needed one more committee member to support my graduation case.

I first emailed Tom, my former MSR manager, and pitched him on the idea of me spending a few months in the fall of 2011 interning at MSR and doing a new project that could contribute towards my dissertation. I wanted him to be on my thesis committee so that I could also include the three papers I published with him from my summer 2009 internship work in my dissertation. Unfortunately, he didn't seem enthusiastic about the idea, so I didn't push further.

I then raised the possibility of extending SlopPy from a workshop paper into a full-fledged conference paper so that it could "count" as a more substantive dissertation contribution. Back in my fifth year, I talked with Martin, an MIT professor whose influential paper directly inspired SlopPy, about working together to extend SlopPy into a conference paper. He was interested in collaborating, but the timing didn't work out since I was busy with CDE and ProWrangler during the latter part of that year. But now, I figured that if I could spend a few months in the beginning of my sixth year working with Martin and have him serve on my thesis committee, then that could be my ticket to graduation. Dawson liked this plan, since (unsurprisingly) he had thoughts about how to combine Klee-like ideas with SlopPy. I planned to email Martin in midsummer to propose this collaboration, but by then, another better opportunity had come along so I no longer pursued this one.

As I began my final summer internship at Google, I looked forward to spending three carefree months polishing up CDE, but I felt a bit anxious because graduation wasn't yet guaranteed upon my return to campus. I needed one more burst of inspiration, and it ended up coming from an unexpected source.

~

In summer 2011, I finally made up my mind to "retire" from academia after graduation: I didn't know what I was going to do for a career, but I wasn't planning to apply for tenure-track university faculty jobs in the upcoming year.

I made this decision for two main reasons: First, I sensed that my current publication record wasn't impressive enough to earn me a respectable tenure-track faculty job. My hunch was confirmed months later when I saw that, sadly, fellow students with slightly superior publication records still didn't get any job offers. Of course, I could always try to work as a postdoc (temporary postdoctoral researcher) for a few years, publish more papers, and then reapply to faculty jobs.

But the second and more important reason makes doing a postdoc meaningless for me: The kinds of research topics I'm deeply passionate about aren't very amenable to winning grant funding, because they aren't well-accepted by today's academic establishment. Without grants, it's impossible to pay for students. And without motivated students to grind through the tough manual labor, it's impossible to get respectable publications. And without a significant number of publications each year, it's impossible to get tenure. Even if I do earn tenure, I would still need new grants to pay for new students to help me implement my ideas; the funding cycle never ends. Given my research interests, I wasn't emotionally prepared to fight the uphill battles required to get my proposals taken seriously by grant funding agencies. I had a hard enough time convincing peer-reviewers to accept my papers; grant reviewers will likely be even less sympathetic since they are the gatekeepers to millions of dollars and would rather hand the money to colleagues who are doing more mainstream types of computer science research.

I had been considering leaving academia for quite a few years, but I now felt that I had justified reasons for doing so: I understood enough about how the "academic game" worked in computer science to know that I didn't want to keep playing it. I summarized my

feelings in an email to a friend who had recently started her job as an assistant professor: "I discovered over the past 5 years that I love being a spectator of research, but the burden of being a continual producer of new research is just too great for me."

Since my mother is a wildly successful professor and my father also deeply respects academia, it was hard for me to tell them my decision. I didn't think they truly understood my rationale; I was afraid that they felt I was giving up and selling myself short when in reality, becoming a professor hadn't been a real goal of mine for years. One of the claimed benefits of academia is the allure of creative freedom, but my decision to leave academia actually freed up my mind to become even more creative in pursuing my true professional passions, both during my final year of grad school and in searching for a new career.

~

The immediate impact of my decision to quit academia was that I didn't need to worry about "networking" at the three academic conferences I attended that summer, where I gave talks on IncPy, SlopPy, and CDE. Academic conferences are filled with senior Ph.D. students, postdocs, and pre-tenure professors schmoozing like crazy in attempts to impress their senior colleagues. For these junior researchers, professional networking at conferences is a serious full-time job, since their budding careers and academic reputations depend upon excelling at it. But since I was getting out of this academic game, I didn't care at all and enjoyed myself without being nervous or calculating.

During a break between sessions at one of the conferences, I spotted Margo sitting by herself working on her laptop. Recall that I had met Margo during my fourth year at the San Jose workshop where I presented my original IncPy paper. I debated whether to approach her and to reintroduce myself. A part of me was afraid that she wouldn't remember me and also that I wouldn't have anything interesting to say. But since I had no real schmoozing agenda due to my imminent

"retirement" from academia, I had nothing to lose if the conversation ended up fizzling. So I just went up and said hello. I reminded her about how we had previously met, and she seemed to remember me. I briefly told her that I was about to give a talk on my new CDE project and then needed to catch a flight back to California. We had a quick five-minute chat about CDE, and then I had to run to give my talk. After returning home that night, I emailed her a quick follow-up message with a link to the CDE project webpage in case her students were interested in playing with it for their research. This was my standard polite message when advertising CDE to professional colleagues, so I didn't really expect her to follow up.

Two weeks later, I received a surprise email from Margo saying that she had been talking about me with her student Elaine. The two of them wanted me to come work with them at Harvard for a few years as a postdoc after completing my Ph.D. The broader research themes surrounding my IncPy and CDE projects resonated with Margo's interests in creating tools to help make computational researchers more productive. I was very flattered by her offer, but the opportunity didn't make sense since I had already decided to retire from academia. It would be useless for me to do a postdoc, since the main purpose of a postdoc is to boost one's resume to improve the chances of getting a university faculty job.

And then inspiration struck again. Since I was in dire need of one more substantive project and thesis committee member before I could graduate, I made the following counterproposal to Margo: Instead of doing a postdoc after my Ph.D., I asked whether I could visit Harvard for four months in the fall of 2011 to work on a project with her. We could submit a paper to a conference in January 2012 and then include that project as the final portion of my dissertation. I also asked whether she could serve as the final member of my thesis committee. Margo liked this idea but didn't have sufficient grant funding for me, since she needed to fund her own students. I talked to Dawson,

and he was generously willing to fund me for those months using his grants even though I wasn't working on Klee (my fellowship had already expired). Margo happily agreed to this arrangement, and after my summer internship ended in September 2011, I moved to Boston, Massachusetts to begin my sixth and final year of grad school.

This final grad school adventure would not have been possible without me actively seizing opportunities that I was fortunate enough to have been given. If Robert hadn't told me about the San Jose workshop two years ago, if I hadn't submitted and presented my IncPy paper there, if Margo hadn't liked my paper and introduced me to Elaine, if I hadn't kept in touch with Elaine, if I hadn't spontaneously said hello to Margo again at last summer's conference where I presented CDE, if she didn't send me a gracious follow-up email, and if I didn't take a risk with my unusual counterproposal to her, then I would have still been back at Stanford struggling to find one last project and thesis committee member.

~

I had an amazingly fun and productive four months in Boston as a visiting researcher at Harvard. The change of scenery was refreshing: I could focus intensely on research without the usual errands of life back home. Elaine helped me find a wonderful studio apartment within a five-minute walk to my office, and I could easily buy food both on campus and in nearby Harvard Square. This ideal living arrangement enabled me to concentrate on my work without any distractions.

I spent my first month mostly socializing with old college friends since my alma mater, MIT, was located right near Harvard. I also met with Margo a few times to discuss potential research ideas. She was open to me working on my own project under her loose supervision, so I had nearly full intellectual freedom. However, I took a pragmatic approach to my brainstorming since I wanted her to be excited about my project and to strongly support its inclusion in my dissertation.

Thus, I read some of her recent papers and grant applications to get a sense of her research philosophy so that I could cater my ideas towards her tastes. By now, I understood the importance of aligning with the subjective preferences of senior collaborators (and paper reviewers), even when doing research in supposedly objective technical fields.

After batting around a few ideas, I came up with something that Margo loved: a tool that monitors researchers' computer-based activities and helps them organize and take notes on their experiments. It was an innovative twist on the traditional electronic lab notebook. Margo jokingly suggested the temporary codename "BurritoBook" to describe our proposed tool, since it seamlessly wraps many layers of activity monitoring around the user's normal workflow. Elaine later shortened the name to "Burrito," which grew on me and eventually became the official project name.

At the time, I thought that my Burrito idea arose spontaneously from combining my hunches with Margo's preferences, but after looking back at old notes, I realized that similar ideas had been brewing in my head for several years. I started thinking about Burrito-like ideas as early as my second year of grad school when I wanted to monitor how people performed programming, and more concretely at the beginning of my fifth year when I wanted to extend IncPy to record Python-based experiment histories. Throughout grad school, I had been keeping a research lab notebook in various ad-hoc formats to document the process of building prototypes and running experiments, so I personally felt the pain of notetaking-related inefficiencies. Finally, although I wasn't a real HCI (Human-Computer Interaction) researcher, my HCI training with Scott and Joel during my second year and with Jeff during my fifth year gave me a keen sensitivity to user needs that greatly influenced the design of Burrito.

I spent a few weeks sketching out high-level plans for Burrito and discussing preliminary design details with Margo. Many refinements to my initial idea came from observing computational researchers at work

and interviewing them about the challenges they faced in managing their multitude of experiment notes, code, and data files; most of my observation subjects were Elaine's friends who worked in various MIT and Harvard science labs. I also received useful early-stage feedback from giving a talk on my Burrito proposal at a lab group meeting led by Rob, the MIT professor I met at the beginning of my second year who encouraged me to pursue my HCI interests with Scott and Joel.

~

And then social time was over; it was time to grind. In early November 2011, I turned into a programming beast for the final time in grad school to transform my Burrito idea into a working prototype. I did 72 consecutive days of programming with only 5 days of breaks spread intermittently throughout the 2.5-month sprint. This period was the longest I had ever sustained an almost-painful level of nonstop intensity thus far. My initial CDE burst during my fifth year was only 21 days of grinding, and this burst was over three times as long. I worked straight through Thanksgiving, Christmas, and New Year's Eve, relentlessly focused on my goal of getting Burrito working well enough to submit a conference paper by the middle of January 2012.

For those few months, I morphed into an antisocial grump who shunned all distraction and became deeply immersed in my craft. All I thought about was computer code; I could barely speak in coherent English sentences except during my weekly progress meetings with Margo. Even though I appeared and acted subhuman (i.e., an unshaven disheveled mess), my emotional state was blissful. I was programming and debugging for over ten hours per day, but my mind was quite relaxed since my technical skills were well-calibrated for the challenges I faced. By now, I had accumulated enough experience in designing, implementing, and "marketing" research prototypes that I was confident in my abilities to make this project work. I received wonderful feedback and support from Margo along the way, so I sensed

that she would strongly endorse the inclusion of Burrito in my dissertation. After years of grinding on uncertain and failed projects earlier in grad school, I now felt invigorated working intensely towards a target that I knew I could feasibly achieve.

By mid-January 2012, the Burrito prototype was in fine shape, so we ran an informal evaluation, wrote up a paper, and submitted it to the conference as planned. I took a few days off to return to normal human mode, said goodbye to my Boston friends, and flew back to California for the Ph.D. endgame.

~

The popular view of how a Ph.D. dissertation arises is that a student comes up with some inspired intellectual idea in a brilliant flash of insight and then spends a few years writing a giant treatise while sipping hundreds of lattes and cappuccinos. In many science and engineering fields, this perception is totally inaccurate: The "writing" is simply combining one's published papers together into a single document and surrounding their contents with introductory and concluding chapters. All of the years of sweaty labor has already been done by the time a student sits down to "write" their dissertation document.

In my department, the most important milestone in a Ph.D. student's career is when their advisor gives the *thumbs up* to begin the dissertation writing process. This gesture signals that the student has done enough work—usually publishing two to four conference papers on one coherent theme—and deserves to graduate within a few months.

When I returned to Stanford in January 2012, my goal was to secure that vital *thumbs up* from Dawson as soon as possible. I wrote up a short document presenting evidence for why I felt I had done enough work to graduate. My argument was simple: I created five innovative software tools to improve the workflow of computational research programmers—IncPy, SlopPy, CDE, ProWrangler, and Burrito—and published 1 top-tier conference paper, 3 second-tier conference papers,

and 3 workshop papers from my body of work (the Burrito conference submission ended up being rejected, so we resubmitted and published in a workshop). As an added bonus, my two other thesis committee members, Jeff and Margo, could also vouch for my graduation case since I had done successful projects with them (ProWrangler and Burrito, respectively). I emailed the document to Dawson and nervously awaited his response. I thought my case was pretty strong, but I had no idea whether he expected me to do more work before allowing me to graduate. To my great relief, he quickly gave me the *thumbs up*, and that's when I knew that I was essentially done with grad school.

I spent the next two months combining all of my papers together into a 230-page dissertation document entitled *Software Tools to Facilitate Research Programming*. Here is the abstract (summary) from the first page of my dissertation:

> Research programming is a type of programming activity where the goal is to write computer programs to obtain insights from data. Millions of professionals in fields ranging from science, engineering, business, finance, public policy, and journalism, as well as numerous students and computer hobbyists, all perform research programming on a daily basis.
>
> My thesis is that by understanding the unique challenges faced during research programming, it becomes possible to apply techniques from dynamic program analysis, mixed-initiative recommendation systems, and OS-level tracing to make research programmers more productive.
>
> This dissertation characterizes the research programming process, describes typical challenges faced by research programmers, and presents five software tools that I have developed to address some key challenges. 1.) ProWrangler is an interactive graphical tool that helps research programmers reformat and clean data prior to analysis. 2.) IncPy is a Python interpreter that speeds up the data analysis scripting cycle and helps programmers manage code and data dependencies. 3.)

> SlopPy is a Python interpreter that automatically makes existing scripts error-tolerant, thereby also speeding up the data analysis scripting cycle. 4.) Burrito is a Linux-based system that helps programmers organize, annotate, and recall past insights about their experiments. 5.) CDE is a software packaging tool that makes it easy to deploy, archive, and share research code. Taken together, these five tools enable research programmers to iterate and potentially discover insights faster by offloading the burdens of data management and provenance to the computer.

I spent a lot of effort crafting new introductory and concluding chapters to turn my dissertation into more than merely a description of five separate tools that I had built over the past few years. Throughout the writing process, Jeff and Margo gave me great feedback on how to frame my research contributions in a more substantive intellectual light. Even though I know that few people will end up reading my dissertation—the constituent papers are far more accessible—it felt satisfying to collect all of my ideas, insights, tool descriptions, and evaluation results together into one cohesive document.

~

I scheduled my oral defense for Monday, April 23, 2012. The biggest challenge was finding a two-hour time slot where five busy professors (my three thesis committee members plus two additional oral committee members) were available. Margo was visiting California for a conference during that week, so I planned around her schedule. In my department, the format of the oral defense is that the student gives a one-hour public talk summarizing their dissertation research, and then there is a one-hour private session where the committee asks probing questions. Afterwards, the committee votes to either pass or fail the student. In reality, almost nobody fails their defense unless they act totally moronic: The committee will usually

have read through and approved a student's dissertation before they let that student defend, so there should be no surprises.

I didn't have time to present all five projects during my oral defense talk, so I chose to present three projects, one that I did with each member of my thesis committee: IncPy with Dawson, ProWrangler with Jeff, and Burrito with Margo. Most Ph.D. students publish papers with only their advisor, so it was a rare honor to get to talk about research that I did with all three of my committee members. I was also happy that many of my friends and former colleagues—including Scott, Joel, Peter, Robert, Greg, and Fernando—attended my defense.

Even though I had given dozens of academic talks throughout grad school, I was more tense than usual during my defense, perhaps because I knew almost everybody in the audience. Strangely, I feel much more at ease giving talks to rooms filled with strangers rather than familiar faces. The private session wasn't as grueling as I had anticipated, but my committee did raise some questions and suggestions that ended up improving my dissertation.

After I passed, my committee and friends all gave me polite congratulations, which was a nice but expected gesture. The compliment that I will cherish the most came from a senior professor with whom I had only spoken once. I was a bit surprised to see him at my defense since I didn't think he would be interested in the topic. After my defense, he sent me the following email praising my talk: "I just wanted to say that I really enjoyed it, partly because of the creativity of the work, partly because of the well-prepared talk, and partly because I had spent the previous year doing research programming."

~

Of the 26 Stanford Computer Science Department Ph.D. graduates in my year, I consider myself fairly mediocre from an academic perspective since most of my papers were second-tier and not well-received by

the establishment. My dissertation work awkwardly straddled several computer science subfields—Programming Languages, Human-Computer Interaction, and Operating Systems—so it wasn't taken seriously by the top people in any one particular subfield.

Despite lack of mainstream acceptance, I still thought that my Ph.D. ended successfully because I was able to carry several of my own ideas to fruition and graduate with a dissertation that I was very proud of. I took a highly entrepreneurial approach to my Ph.D.—opportunistically seeking out projects and collaborators, walking a fine line between being unconventional while conforming enough to get my papers published. I feel extremely lucky to have been able to take charge of my Ph.D. career in creative ways; I wouldn't have had nearly as much freedom without the fellowships that funded five out of my six years at Stanford.

In the end, like most Ph.D. dissertations, mine expanded the boundaries of human knowledge by a teeny microscopic amount. The five prototype tools that I built contain some interesting ideas that can be adapted by future researchers. In fact, I will be honored if future researchers cite my papers as examples of shoddy primitive hacks and argue for why their techniques are far superior. That's how research marches forward bit by bit: Each successive generation builds upon the ideas of the previous one.

However, to me, the most significant contribution of my dissertation wasn't those specific prototype tools. Rather, it was that, to the best of my knowledge, I was one of the first computer science Ph.D. students to identify a pervasive problem—the lack of software tools catered to the needs of a large and growing population of computational research programmers—and to offer some early-stage prototype solutions that others can improve upon. I believe that these ideas will become more important in the upcoming decades, but since I'm retiring from academia, I won't be around to directly promote them.

Since my dissertation topic is far from being mainstream, any junior professor or scientist who tries to build their academic career upon its ideas will struggle to gain the respect of grant funding agencies, which are the gatekeepers to launching new projects, and their senior colleagues, who are the gatekeepers to publication and tenure. I will be more than happy to assist anybody who wants to take on this noble fight, but I'm not brave enough to stake my own career on it. Instead, I plan to now pursue a completely different professional passion, which might someday be the subject of a future book :-)

~

In preparation for writing this memoir, I dug through lots of my old research notes. One day, I found the following snippet about a topic that I was interested in investigating:

> Research into software development tools for non-software engineers, but rather for scientists, engineers, and researchers who need to program for their jobs – they're not gonna care about specs., model checking, etc. – they just want pragmatic, lightweight, and conceptually-simple tools that they can pick up quickly and use all the time.

The shocking thing about this note is that I wrote it six years ago in the summer of 2006, right before I started the Ph.D. program at Stanford. It's been a long, circuitous, and unpredictable journey, but I'm incredibly grateful that I was able to turn this broad topic—one out of dozens that caught my interest over the years—into my Ph.D. dissertation. This accomplishment wouldn't have been possible without a rare combination of great luck, personal initiative, insightful nudges from generous people, and nearly ten thousand hours of grinding.

Epilogue

If you are not going to become a professor, then why even bother pursuing a Ph.D.? This frequently-asked question is important because most Ph.D. graduates aren't able to get the same jobs as their university mentors and role models—tenure-track professors. There simply aren't enough available faculty positions, so most Ph.D. students are directly training for a job that they will never get. (Imagine how disconcerting it would be if medical or law school graduates couldn't get jobs as doctors or lawyers, respectively.)

So why would anyone spend six or more years doing a Ph.D. when they aren't going to become professors? Everyone has different motivations, but one possible answer is that a Ph.D. program provides a safe environment for certain types of people to push themselves far beyond their mental limits and then emerge stronger as a result. For example, my six years of Ph.D. training have made me wiser, savvier, grittier, and more steely, focused, creative, eloquent, perceptive, and professionally effective than I was as a fresh college graduate. (Two obvious caveats: Not every Ph.D. student received these benefits—many grew jaded and burned-out from their struggles. Also, lots of people cultivate these positive traits without going through a Ph.D. program.)

Here is an imperfect analogy: *Why would anyone spend years training to excel in a sport such as the Ironman Triathlon—a grueling race consisting of a 2.4-mile swim, 112-mile bike ride, and a 26.2-mile run—when they aren't going to become professional athletes?* In short,

this experience pushes people far beyond their physical limits and enables them to emerge stronger as a result. In some ways, doing a Ph.D. is the intellectual equivalent of intense athletic training.

~

Here are twenty most memorable lessons that I've learned throughout my Ph.D. years. My purpose in sharing is not to provide unsolicited advice to students, since everyone's Ph.D. experience differs greatly; nor is it to encourage people to pursue a Ph.D., since these lessons can come from many sources. Rather, this section merely serves as a summary of what I gained from working towards my Ph.D.

1. **Results trump intentions**: Nobody questions someone's intentions if they produce good results. I didn't have so-called pure intellectual motivations during grad school: I started a Ph.D. because I wasn't satisfied with engineering jobs, pressured myself to invent my own projects out of fear of not graduating on time, and helped out on HCI projects with Scott, Joel, and Jeff to hedge my bets. But I succeeded because I produced results: five prototype tools and a dozen published papers. Throughout this process, I developed strong passions for and pride in my own work. In contrast, I know students with the most idealistic of intentions—dreamy and passionate hopes of revolutionizing their field—who produce few results and then end up disillusioned.

2. **Outputs trump inputs**: The only way to earn a Ph.D. is by successfully producing research outputs (e.g., published papers), not merely by consuming inputs from taking classes or reading other people's papers. Of course, it's absolutely necessary to consume before one can produce, but it's all too easy to over-consume. I fell into this trap at the end of my first year when I read hundreds of research papers in a vacuum—a consumption binge—without being

able to synthesize anything useful from my undirected readings. In contrast, related work literature searches for my dissertation projects were much more effective because my reading was tightly directed towards clear goals: identifying competitors and adapting good ideas into my own projects.

3. **Find relevant information**: My Ph.D. training has taught me how to effectively find the most relevant information for what I need to accomplish at each moment. Unlike traditional classroom learning, when I'm working on research, there are no textbooks, no lecture notes, and no instructors to provide definitive answers. Sometimes what I need for my work is in a research paper, sometimes it's within an ancient piece of computer code, sometimes it's on an obscure website, and sometimes it's inside the mind of someone whom I need to track down and ask for help.

4. **Create lucky opportunities**: I got incredibly lucky several times throughout grad school, culminating in getting to work with Margo at Harvard during my final year. But these fortuitous opportunities wouldn't have arisen if I didn't repeatedly put myself and my work on display—giving talks, chatting with colleagues, asking for and offering help, and expressing gratitude. The vast majority of my efforts didn't result in serendipity, but if I didn't keep trying, then I probably wouldn't have gotten lucky.

5. **Play the game**: As a Ph.D. student, I was at the bottom of the pecking order and in no position to change the "academic game." Specifically, although I dreaded getting my papers repeatedly rejected, I had no choice but to keep learning to play the publication game to the best of my abilities. However, I was happy that I played in my own unique and creative way during the second half of grad school by pursuing more unconventional projects while still conforming to the "rules" well enough to publish and graduate.

6. **Lead from below**: By understanding the motivations and personalities of older Ph.D. students, professors, and other senior colleagues, I was able to lead my own initiatives even from the bottom of the pecking order. For example, after I learned Margo's research tastes by reading her papers and grant applications, I came up with a project idea (Burrito) that we were both excited about. If I were oblivious to her interests, then it would have been much harder to generate ideas to her liking.

7. **Professors are human**: While this might sound obvious, it's all too easy to forget that professors aren't just relentless research-producing machines. They're human beings with their own tastes, biases, interests, motivations, shortcomings, and fears. Even well-respected science-minded intellectuals have subjective and irrational quirks. From a student's perspective, since professors are the gatekeepers to publication, graduation, and future jobs, it's important to empathize with them both as professionals and also as people.

8. **Be well-liked**: I was happier and more productive when working with people who liked me. Of course, it's impossible to be well-liked by all colleagues due to inevitable personality differences. In general, I strived to seek out people with whom I naturally clicked well and then took the time to nurture those relationships.

9. **Pay some dues**: It's necessary for junior lab members to pay their dues and be "good soldiers" rather than making presumptuous demands from day one. As an undergraduate and master's student at MIT, I paid my dues by working on an advisor-approved, grant-funded project for two and a half years rather than trying to create my own project; I was well-rewarded with admissions into top-ranked Ph.D. programs and two fellowships, which paid for five years of graduate school. However, once I started at Stanford, I paid my dues for a bit too long on the Klee project before quitting. It took me years to recognize when to defer to authority figures and

when to selfishly push forward my own agenda.

10. **Reject bad defaults**: Defaults aren't usually in the best interests of those on the bottom (e.g., Ph.D. students), so it's important to know when to reject them and to ask for something different. Of course, there's no nefarious conspiracy against students; the defaults are just naturally set up to benefit those in power. For example, famous tenured professors like Dawson are easily able to get multi-year grants to fund students to work on "default" projects like Klee. As long as some papers get published from time to time, then the professor and project are both viewed as successful, regardless of how many students stumbled and failed along the way. Students must judge for themselves whether their default projects are promising, and if not, figure out how to quit gracefully.

11. **Know when to quit**: Quitting Klee at the end of my third year was my most pivotal decision of grad school. If I hadn't quit Klee, then there would be no IncPy, no SlopPy, no CDE, no ProWrangler, and no Burrito; there would just be three or more years of painful incremental progress followed by a possible "pity graduation."

12. **Recover from failures**: Failure is inevitable in grad school. Nothing I did during my first three years made it into my dissertation, and many paths I wandered down in my latter three years were also dead-ends. Grad school was a safe environment to practice recovering from failures, since the stakes were low compared to failing in real jobs. In my early Ph.D. years, I would grow anxious, distraught, and paralyzed over research failures. But as I matured, I learned to channel my anger into purposeful action in what I call a *productive rage*. Every rejection, doubt, and criticism spurred me to work harder to prove the naysayers wrong. Lessons learned from earlier failures led to successes later in grad school. For example, my failure to shadow professional programmers at the beginning of my second year taught me how and who to approach for these

sorts of favors, so I later succeeded at shadowing computational researchers to motivate my dissertation work; and my failure to get lots of real users for IncPy taught me how to better design and advertise my software so that I could get 10,000 users for CDE.

13. **Ally with insiders**: I had an easy time publishing papers when allied with expert insiders such as Scott and Joel during my second year, Tom during my MSR internship, and Jeff during my fifth year. They knew all the tricks of the trade required to get papers published in their respective subfields; the five papers that I co-wrote with these insiders were all accepted on their first submission attempts. However, struggling as an outsider—with Dawson on empirical software measurement in my second year and then on my solo dissertation projects—was also enriching, albeit more frustrating due to repeated paper rejections.

14. **Give many talks**: I gave over two dozen research presentations throughout my Ph.D. years, ranging from informal talks at university lab group meetings to conference presentations in large hotel ballrooms. The informal talks I gave at the beginning of projects such as IncPy were useful for getting design ideas and feedback; those I gave prior to submitting papers were useful for discovering common criticisms that I needed to address in my papers. Also, every talk was great practice for improving my skills in public speaking and in responding to sometimes-hostile questions. Finally, talks sometimes sparked follow-up discussions that led to serendipity: For example, after watching my first talk on IncPy, a fellow grad student emailed me a link to Fernando's blog post about Python in science; that email encouraged me to reach out to Fernando, who would later inspire me to improve IncPy and then to invent CDE. Over a year later, my Google Tech Talk on CDE directly led to my super-chill summer 2011 internship.

15. **Sell, sell, sell**: I spent the majority of my grad school days heads-down grinding on implementing research ideas, but I recognized that convincingly selling my work was the key to publication, recognition, and eventual graduation. Due to the ultra-competitive nature of the paper publication game, what often makes the difference between an accept and a reject decision is how well a paper's "marketing pitch" appeals to reviewers' tastes. Thus, thousands of hours of hard grinding would go to waste if I failed to properly pitch the big-picture significance of my research to my target audience: senior academic colleagues. More generally, many people in a field have good ideas, so the better salespeople are more likely to get their ideas accepted by the establishment. As a low-status grad student, one of the most effective ways for me to "sell" my ideas and projects was to get influential people (e.g., famous professors such as Margo) excited enough to promote them on my behalf.

16. **Generously provide help**: One of my favorite characteristics of the Ph.D. experience was that I wasn't in competition with my classmates; it wasn't like if they did better, then I would do worse, or vice versa. Therefore, many of us generously helped one another, most notably by giving feedback on ideas and paper drafts before they were subject to the harsher critiques of external reviewers.

17. **Ask for help**: Over the past six years, I became good at determining when, who, and how to ask for help. Specifically, whenever I felt stuck, I sought experts who could help me get unstuck. Finding help can be as simple as asking a friend in my department, or it might require getting referrals or even cold-emailing strangers.

18. **Express true gratitude**: I learned to express gratitude for the help that others have given me throughout the years. Even though earning a Ph.D. was a mostly-solitary process, I wouldn't have made it without the generosity of dozens of colleagues. People feel good when they find out that their advice or feedback led to concrete

benefits, so I strive to acknowledge everyone's specific contributions whenever possible. Even a quick thank-you email goes a long way.

19. **Ideas beget ideas**: As I discovered at the end of my first year, it's nearly impossible to come up with substantive ideas in a vacuum. Ideas are always built upon other ideas, so it's important to find a solid starting point. For instance, the motivations for both IncPy and SlopPy came from my frustrations with programming-related inefficiencies I faced during my 2009 MSR internship. A year later, some of my ideas for extending IncPy, mixed with Fernando's insights on reproducible research and Dawson's mention of Linux dependency hell, led to the creation of CDE. Also, ideas can sometimes take years to blossom, usually after several false starts: I started pondering Burrito-like ideas during my second year and then at the end of my fourth, but it wasn't until my sixth year that I was able to solidify those fuzzy thoughts into a real project.

20. **Grind hard and smart**: This book is named *The Ph.D. Grind* because there would be no Ph.D. without ten thousand hours of unglamorous, hard-nosed grinding. This journey has taught me that creative ideas mean nothing without the extreme effort to bring them to fruition: showing up to the office, getting my butt in the seat, grinding hard to make small but consistent progress, taking breaks to reflect and refresh, then repeating day after day for over two thousand consecutive days. However, grinding smart is just as important as grinding hard. It's sad to see students blindly working themselves to death on tasks that won't get favorable results: approaching a research problem from an unwise angle, using the wrong kinds of tools, or doing useless errands. Grinding smart requires perceptiveness, intuition, and a willingness to ask for help.

~

I'll end by answering a question involving the F-word: *Was it fun?*

Some aspects of the Ph.D. experience were very fun: Coming up with new ideas was fun; sketching out software designs on the whiteboard was fun; having coffee with colleagues to chat about ideas was fun; hanging out with interesting people at conferences was fun; giving talks and inciting animated discussions was fun; receiving enthusiastic emails from CDE users around the world was fun. But I probably spent only a few hundred hours on those activities throughout the past six years, which was less than five percent of my total work time.

In contrast, I spent about ten thousand hours grinding alone in front of my computer—programming, debugging, running experiments, wrestling with software tools, finding relevant information, and writing, editing, and rewriting research papers. Anyone who has done creative work knows that the day-to-day grind is rarely fun: It requires intense focus, rigorous discipline, keen attention to detail, high pain tolerance, and an obsessive desire to produce great work.

So, *Was it fun?*

I'll answer using another F-word: *It was fun at times, but more importantly, it was fulfilling.* Fun is often frivolous, ephemeral, and easy to obtain, but true fulfillment comes only after overcoming significant and meaningful challenges. Pursuing a Ph.D. has been one of the most fulfilling experiences of my life, and I feel extremely lucky to have been given the opportunity to be creative during this time.