My primary area of interest is optimizing runtimes for dynamic languages and building supporting tooling. For the past three years, I have been working on a fast Python interpreter and a JIT on top of CPython. While this work has been exciting and educational, I would like to spend time answering questions, solving problems, and building for a longer time horizon—hence my interest in a PhD.

From November 2018 to July 2021, I worked on an optimized green-field Python runtime called Skybison. This runtime applied a lot of the past 30 years of research in Smalltalk, Self, Java, and other dynamic language runtimes. I had three significant contributions: work on the C-API emulation layer, which allowed our runtime to load C extensions written for CPython; performance improvements in inline caching, the interpreter written in assembly, core data structures, and library code; and, finally, the experimental template JIT compiler.

Since July 2021, I have been working on a Python JIT called Cinder. I'm writing a function inliner. Unlike other languages, Python has significant support for runtime reflection baked into the language. This poses a challenge for many compiler optimizations, including inlining. We must consider deoptimization, runtime reflection, user-swappable globals and "builtins" modules, reifying lightweight stacks for sampling profilers, and more. Despite all of the hurdles, the inliner should be launching in the next month.

I have now worked on the above two large projects at Facebook. They each have radically different performance characteristics. There was a long learning curve for each, slowly and painfully building an intuition for what "hurts"—building mechanical sympathy.

Mechanical sympathy is a term coined by the racing driver Jackie Stewart which refers to the best drivers having a feel for how their cars work internally. To reframe for computing: the most effective engineers understand how the compiler, runtime, and hardware are working beneath the hood. Different groups have different ways of building such an understanding, with differing efficacy: Smalltalk programmers often have a great deal of runtime introspection available in the graphical interface; Java programmers can run HProf and Flight Recorder, or, if they are lucky, some proprietary software by Azul; some can integrate third-party profiling libraries into the runtime to get a view of where time is spent; the rest of the unfortunate world must use Callgrind, the GNU debugger, Matthew Godbolt's Compiler Explorer, and print logging.

It's a little absurd that the runtime often knows so much about what is slow but programmers must re-discover it for themselves. Runtime hooks exist for deoptimization, for cache misses, and for all manner of other slow events. The runtimes *know* the internal representations and *run* the generated code. Hardware signals exist for instruction cache misses, data cache misses, translation lookaside buffer misses, and more. So why must we manually dump and inspect the intermediate representations, generated code, and internal state when it should be nicely surfaced for us? The runtime should be screaming bloody murder about this megamorphic cache and I should get a mean red underline in my text editor. This would help not only with coming up with optimizations but with preventing mistakes, too: at least once a week I see someone manually dumping code or submitting a follow-up change titled "Actually apply the foo optimization to bar... the previous change missed a case." There are many questions to answer before and while building such software, starting from "what visualizations are useful to compiler engineers and what are useful to compiler users?" and continuing on to questions such as "given a boatload of performance data, how can we surface only the most useful annotations?" (See the appendix for more.)

I am not always a curmudgeon about tooling. In fact, quite the opposite: I love my tools. I decided earlier this year to write and teach the first offering of a course at Tufts University (CS 50ISDT - Introduction to Software Development Tooling) with my friend and twice-colleague Tom Hebb. As you read this letter, the course will have just finished. We taught eighteen students about the command line, Linux internals, and other operating systems; about version control, Git internals, and other source control systems; about build systems, Make, and other build and packaging systems; and, finally, about unit tests and the great wide world of other means to achieve software correctness (what does it even mean for software to be "correct"?). I include this note on teaching because I would like to eventually become a professor—it is my other primary motivation for applying to this PhD program. I got my first real taste for teaching in undergrad, and it has been a passion of mine ever since.

I was a teaching assistant for CS 40 (Machine Structure & Assembly-Language Programming) and a teaching fellow for both CS 15 (Data Structures) and 11 (Introduction to Computer Science). I spent, and thoroughly enjoyed, hours per week writing assignments, leading lab sections, and holding office hours. I learned to debug not student programs, but student understandings of the material. As a senior, I wrote and taught a course (EXP 57 - Tech Trends and Careers[1]) on the journey from school to "the real world" with my friend Yuki Zaninovich. In this course, we: gave an overview of the current computing landscape; taught students how to personably interact with other people and write a good resume; and provided an insider's view of the interview process and how to do well in it.

I am thrilled at the possibility of returning to an academic setting. I love academic environments: the teaching, the writing, the poking at hard problems over tea on sleepless nights. I want to answer all of the questions in the appendix and more. I would love to join your research group.

Appendix

Questions to answer about mechanical sympathy tooling:

- Can it help engineers write better code and compiler engineers write better compilers?
  - What existing "bugs" might this tooling surface in libraries and compilers?
- What information is helpful to surface from inside the runtime? For what audience? With what language?
- What visualizations are useful? Would multiple layers of IRs (a la Compiler Explorer) help? Line-based annotations in an editor? A live-updating view of runtime "hot spots"?
- How can we filter the information so the most helpful or impactful is being shown?
- How can lower level (for example, hardware) performance indicators be conceptually mapped back to higher level (for example, surface programming language) terms?
- Can we build a common data format so that this language tooling may be shared across compilers and runtimes? What challenges exist—why has this not been done before?

[1] We did not get to choose the course name. The material goes far beyond "tech trends."

  - HProf exists, but is very Java specific. Runtimes like Dart and Skybison fake the Java-specific parts.
  - Folks on the PyPy project have expressed interest in this existing.
  - How can we go the way of the Language Server Protocol and not XKCD 927?
- Can this be run in production and with a minimal performance hit?
  - If not, how can we package and ingest the data for later use?
  - If not, how can we tell when we have gathered a representative sample—that the tooling will provide "useful" information?