Statement on Research — Daniel Patterson

My research has been driven by a simple question:

Why can't programmers easily write multi-language software?

Everything we do as a field is eventually realized in software, and software is written in programming languages. Recently, there has been renewed interest in building new languages, especially with type systems – Typescript introduced to wrangle Javascript, Rust beginning to replace C/C++, and WebAssembly introduced as a compiler target without undefined behavior. At the same time, the real programs we write are rarely implemented in a single language, so the increasing interest in type systems and static reasoning makes the lack of ability to translate static reasoning across languages all the more troubling.

This problem has a magnifying effect in practice, as it hampers programmers ability to fully use the best tools available—and thus researchers must either confine their attention to improving productivity in legacy languages or acknowledge the huge barrier that programmers will encounter when trying to incorporate their innovative ideas into legacy systems—the maintenance of which is, after all, what most programmers do.

Past Work Thus far, I've focused on several aspects of this problem. One is to understand how we can know if a given multi-language program (as traditionally implemented with a foreign function interface) is safe. In order to do this, we must understand how the various languages are compiled to whatever intermediate or target where they actual run. To be safe, the invariants of the source must be expressed somehow in that target code – whether through actual static features of the target, or through careful insertion of glue code. The novel framework that I have developed [1] allows us to prove a semantic soundness theorem for such systems.

The aspect that I am currently exploring is how we can know the correctness of a multi-language refactoring—where we replace code from a language with that from another but intend the meaning to remain the same. This is a generalization of a well-understood problem from gradual typing, but brings interesting novel challenges, as the equivalence of program fragments from different languages and the resulting mixed types are aspects that do not come up in the simpler setting. Again, leaning on reality, we rely upon the compilation and the resulting equivalence in the intermediate or target, as that behavior is the actual behavior that programmers will have to reason with.

The final aspect of this problem that I've explored arises when code in one language cannot be expressed at all in the other—for example, a language with substructural types like Rust linking with a language with unrestricted types like OCaml. This is an interesting, but realistic, situation, since one of the reasons for mixing languages is to gain access to new functionality. The framework that I developed [2] allows novel types to be ascribed to the foreign code. These so-called "linking types" allow us to maintain static reasoning while interacting with such inexpressible behavior.

I've also explored the interaction of functional code with assembly [3] and built a general framework for compiler correctness that accounts for linking [4].

Future Work I plan on further exploring various aspects of this problem:

First, I am currently working on a project in collaboration with industrial researchers on the semantics of WebAssembly interface types. This relates to one of the recurring themes of my research: how invariants from the source language can be preserved after compilation into the medium where linking occurs. Interface types are a share-nothing description of high-level types

that can be passed across modules, where the intention is that they serve as a point of high-level commonality that various languages that compile to WebAssembly can agree upon.

I also plan on continuing to explore ideas for flexible type systems for low-level languages. As multi-language software proliferates and there is an increasing understanding that type soundness is useful for programmer productivity, the question of how to preserve invariants becomes increasingly pressing. I've explored, in early work, a flexible type annotation language called Phantom Contracts [5], that may produce interesting results in the future. And more generally, I'm interested in how to bring types to low-level languages without imposing restrictions that make them no longer behave as low-level languages.

I also consider broader questions of usability and productivity of multi-language systems. There are two (sometimes unstated) motivations for this work: one, that drives my own doctoral work, is that as legacy systems migrate, they gain new parts written in new languages, and thus multi-language systems are inevitable in large enough software. The second, that underlies the language oriented programming (LOP) paradigm, is that different languages are suited to different tasks, and that we should support this. One fascinating question is: is the latter, indeed, true? Or more specifically, is it possible to compare the productivity of a language approach to solving problems, where different domains of the problem are given appropriate domain specific languages (DSLs), and one where much coarser libraries are used instead? In particular, does the increase in productivity of the DSL offset the decrease in accessibility, since each will have different semantics, versus the single shared semantics of the library approach?

I'm also interested in how questions of semantics impact student performance, in particular in introductory classes. Much of my research has focused on how programmers can reason about multi-language software, but similar questions can be asked about single languages. In particular, if we use unsound languages, can we show that the sources of unsoundness cause confusion to our students? More generally, do students build up an accurate understanding of the semantics of the tools they are using, or does an imperfect understanding suffice to succeed?

Finally, related to the latter, I'm also very interested in quantifying the effectiveness of teaching more broadly. For example, one simple question is: what is the correlation of student knowledge before a class to their performance in it? For an introductory class, we would like a one sided correlation: i.e., it's okay if students with background do well, but not okay if they are the *only* students who do well. I'm also interested in exploring peer review [6] as a method of increasing detailed feedback even while class sizes increase.

1. Semantic Soundness for Language Interoperability. Daniel Patterson, Noble Mushtak, Andrew Wagner, Amal Ahmed. DRAFT. `https://dbp.io/pubs/2021/semanticinterop-draft.pdf`
2. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. Daniel Patterson and Amal Ahmed. SNAPL 2017.`https://dbp.io/pubs/2017/linking-types.pdf`
3. FunTAL: Reasonably Mixing a Functional Language with Assembly. Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. PLDI 2017. `https://dbp.io/pubs/2017/funtal.pdf`
4. The Next 700 Compiler Correctness Theorems (Functional Pearl). Daniel Patterson and Amal Ahmed. ICFP 2019. `https://dbp.io/pubs/2019/ccc.pdf`
5. Phantom Contracts for Better Linking. Daniel Patterson. POPL 2019 SRC. `https://dbp.io/pubs/2018/phantom-contracts-src.pdf`
6. CaptainTeach: Multi-Stage, In-Flow Peer Review for Programming Assignments. Joe Gibbs Politz, Daniel Patterson, Kathi Fisler, and Shriram Krishnamurthi. ITiCSE 2014. `https://dbp.io/pubs/2014/captainteach-iticse.pdf`