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 rich 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, and thus the increasing interest in type systems and static reasoning should correspond to increasing ability to statically reason about multi-language programs. Unfortunately, that's not the case: with rare exceptions, all static invariants are thrown out in the process of compiling and linking together multi-language programs. This means programmers have to carefully wire together foreign function library invocations with no guarantee (either statically, or even often at runtime) that the calls will do what they expect.

This problem also has a dampening effect on research to improve programmer productivity, as researchers must either confine their attention to improving tooling within legacy languages, or acknowledge the 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.

Semantic Soundness of Interoperability One aspect of this problem that I have addressed is how to know if a given multi-language program is type safe. A desirable property of a single language, true (or nearly true) of many modern languages, is so-called type soundness: programs that are well-typed will only have particular well-defined runtime errors (e.g., divide by zero, uncaught exception), and as a corollary, if they do not error, will evaluate to values that conform to their static type. In **recently submitted work** [1], I show how to extend this to the setting where well-typed programs include calls to other languages, something regularly excluded in type soundness results.

The framework that I developed relies upon defining a semantic notion of *source-language types as sets of target terms* that capture the behavior of the source types. This target-language-centric definition naturally allows us to capture the behavior of compilers, but more importantly, since interoperation typically happens after compilation to some shared target, our model allows us to describe types of both interoperating languages in the medium where they actually interact. In this way, we can express what it would mean for a foreign call to be safe: that our values, possibly wrapped in some target-level glue code, conforms to the type that the foreign call expects, and what the foreign code returns, again possibly wrapped, conforms to what our type system expects. The framework lifts these properties into a declarative judgment used in an algorithmic typechecker, and allows us to prove a semantic type soundness theorem for both languages.

Semantics of Language Migration Safety of a given mixed-language program is important, but a common reason for mixed language programs is gradual migration between languages. Indeed, the persistence of any large enough (i.e., legacy) code base means that introducing a new language will result in a (perhaps temporary) mixed-language program. The act of migrating, that is, rewriting portions of a program from one language into another, should not change the functional behavior of the program. This is a well-studied problem in the context of gradual typing, where the two languages share syntax and operational behavior and primarily vary in their static semantics.

In ongoing work, I am generalizing this to the context of arbitrary pairs of languages—as indeed, unless one happens to start a project in one of the few gradually typed languages, this is the scenario you will likely end up in. In particular, much of gradual typing research relies upon the addition or removal of type annotation being what migration is, something that doesn't hold in this more general setting where the underlying languages change as well. To address this, I have developed a framework for equivalence-preserving refactoring between multi-language programs which relies upon similar source-indexed but target-inhabited models as in the above soundness work.

Types for Linking The previous two avenues of work rely upon building models that allow us to express when code from one language behaves as a type in the other, perhaps with some target-level wrapper code to adapt it. In the first, I leverage this to prove type soundness; in the second, to prove when migration preserves behavior. Sometimes, however, features in one lanugage are simply inexpressible at all in the other, at least in any reasonable way—for example, a language with substructural types like Rust linking with a language with only unrestricted types like OCaml. This is an interesting, and clearly not theoretical, situation, since one of the primary reasons for mixing languages is to gain access to new functionality. The framework that I described in a **SNAPL'17 paper** [2] allows types to be added in order to describe foreign code to be linked. These so-called "linking types" come with strong properties that allow us to maintain static reasoning while interacting with such inexpressible behavior.

Compositional Compiler Correctness Compilers are complex pieces of software, and ones for which bugs are insidious, as they do not show up anywhere in the program that you are writing. As a result, there has been increasing interest in compiler verification, and even significant results (e.g., the CompCert compiler for C, CakeML compiler for Standard ML, etc). However, initially many of these results assumed that only whole programs would be compiled, an unrealistic assumption: not only is incremental compilation critical for performance, but all programs have dependencies, standard libraries, runtimes, etc. Various attempts have been made to solve this and create so-called "compositionally correct compilers", but each proved slightly different theorems, with different assumptions, allowing different forms of linking. To bring order to this, **my ICFP'19 paper** [4] unified many existing results into a single abstract theorem that highlighted the differences and spread them out onto a spectrum, so that a reader could understand how they could use such a compiler while still gaining the benefits of its correctness theorem.

Low-level Interoperability Finally, I've explored interoperability between a high-level functional language and a low-level typed assembly in a **PLDI'17 paper** [3]. This primarily involved categorizing the different types of jumps that could occur in the assembly and tracking, in the type system, ones that corresponded to well-bracketed calls/returns vs. ones that did not return. With that discipline added, mixing the two languages became tractable, as a block of assembly could be ascribed a type that described how it would eventually return to the functional code that had invoked it.

Future Work

There are various aspects of the language interoperability problem that I am still exploring.

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 source languages 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 ever more important. 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 questions of **how 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 (or, languages with confusing yet sound semantics), can we show that the sources of unsoundness cause difficulty 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, it's important to me to **quantify 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 in general do better, 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`