Squashing Bugs and Empowering Programmers with User-Centered Programming Language Design

Programming languages can provide strong safety guarantees, enabling software engineers to create reliable software. For example, Rust's type system ensures that memory is not accessed after it is freed. Unfortunately, safe languages can also be difficult to use, limiting their impact on real-world software safety. The practical benefit of a language depends on programmers' abilities to use the language *effectively*: not only without inserting critical bugs but also while creating software at reasonable cost. Although programming languages are *formal systems* that enable sound reasoning about software, they are also *user interfaces* that programmers use to specify behavior. I develop languages that take *both* aspects into account, creating languages and assessing them both theoretically and empirically.

In my research, I simultaneously (1) develop novel methods for designing and evaluating programming languages that take users' needs into account in a formal, structured way; and (2) design and evaluate concrete programming languages to show that particular design choices measurably benefit programmers. In doing so, I am both *developing the science of user-centered programming language design* and *enabling programmers to be more effective in writing safe software*. I make it easier to use languages with strong safety properties and I add strong safety properties to languages while maintaining usability. I use *qualitative* formative methods to ground designs in users' needs, and I iterate to make languages that are as effective as possible for users. Then, I conduct *summative evaluations* to assess both quantitatively and qualitatively how the designs affect users.

In the following sections, I show how I have applied the user-centered programming language design techniques I am developing to create languages that benefit programmers and the users of their programs.

Glacier: transitive class immutability for Java (ICSE 2016, 2017)

An object is *immutable* when its state cannot be changed. By restricting mutation, as recommended by security experts, programmers can avoid modifying data structures at inappropriate times and avoid adding inappropriate dependencies. In Java, a `final` field cannot be reassigned outside the constructor. Unfortunately, `final` only provides weak guarantees. A `final` field *can* reference a mutable object, so `final` cannot be used to ensure thread-safety.

I was able to show that a *stronger* language-based immutability restriction could be helpful to programmers and employed empirical methods to provide useful insight and evidence. First, I examined the space of language features that pertain to restricting state change [ICSE16]. I identified eight distinct dimensions of change restriction and hundreds of possible language designs. Which approach should a given language take? I interviewed software engineers and analyzed the data to identify *transitive class immutability* as an approach that would be beneficial in many situations. *Immutability* means that *no* reference can be used to mutate the data in an object. In *class* immutability, every instance of a particular class is immutable. *Transitive* restrictions apply to the deep structure of an object, rather than only to its immediate fields. Software engineers conjectured that this combination would provide useful properties, such as thread safety. They commonly encountered bugs when state that was supposed to be immutable was changed by mistake, and their languages did not provide tools that were appropriate for providing the guarantees they needed.

Next, I designed Glacier [ICSE17], which adds the `@Immutable` annotation to Java. I implemented Glacier as a compiler extension, which enforces that any classes annotated `@Immutable` are transitively immutable. I designed Glacier to do all checks at compile time to avoid a negative impact on performance and the runtime environment. Then, I conducted a user study that showed (1) Java programmers can obtain immutability guarantees more effectively with Glacier than with `final`; (2) Java programmers commonly unsafely mutate state of objects that are designed to be immutable. In my study of 20 Java programmers, none of the ten `final` participants could express transitive immutability because it was too easy to miss a place where an edit was required. In contrast, nine of the ten Glacier participants were able to enforce immutability. I also asked each participant to do two programming tasks in immutable classes. Of 18 tasks that `final` participants said they had finished, 11 tasks included accidental mutation of immutable state. Glacier participants finished 14 tasks within the time limit; of course, Glacier ensured correctness, so no Glacier participants mutated immutable state in their final solutions. In summary, Glacier was usable by almost all participants, and it detected serious bugs that `final` users frequently inserted. Finally, to assess the real-world applicability of Glacier, I applied Glacier to a real-world spreadsheet implementation. In doing so, I found two previously-unknown bugs in the spreadsheet.

Obsidian: a safer blockchain programming language (TOPLAS 2020, OOPSLA 2020, TOCHI 2021)

Blockchain systems support computation on shared state among participants that have not established trust, particularly for financial applications. Unfortunately, blockchain programs (*contracts*) have repeatedly been found to be buggy, and public reporting indicated that hackers had stolen over $80M by 2017. Even today, contracts are still vulnerable: as one example, hackers stole $4.4M from one contract in July 2021. Could a safer language rule out common causes of these vulnerabilities?

I observed two common patterns while studying existing applications and bugs. First, the set of operations that a program supports often depends on the state of the program. For example, a `Bond` object might accept interest payments only *after* the bond has already been sold. Second, programs frequently manipulate *assets,* such as digital currencies, which have intrinsic value and should not be accidentally lost. In prior studies[1], programmers had difficulty using stateful APIs and frequently introduced bugs that lost assets. Several of the public exploits also pertained to improper manipulation of stateful contracts.

To improve safety, I designed a language that uses *typestate* and *assets*. Typestate lifts dynamic state into static types, enabling the compiler to reason about the state of the program; *assets* represent resources that must be used or stored but not lost. Assets are represented with *linear types*, which enable formal reasoning about resources. This design posed an interesting usability challenge since it has properties that are very different from traditional type systems. For example, the type of a reference can *change* as a result of operations that occur. *Affine* types (which are like linear types, but need not be consumed) are used in other languages, such as Rust, offering the potential for broader applicability of the results.

To make this unconventional language usable, I adapted HCI techniques to develop PLIERS: the Programming Language Iterative Evaluation and Refinement System [TOCHI21]. For example, I back-ported design decisions to languages for which I could find existing programmers so that I could test particular design decisions rapidly with minimal training. In user studies, I adapted the *Wizard of Oz* prototyping technique, simulating a compiler by manually generating error messages. This allowed me to base the language on both my ideas *and* the expectations of target users. Concurrently, I developed Silica, a core calculus for Obsidian, to ensure that Obsidian rested on solid formal foundations. I proved soundness theorems to show that the language has the needed safety properties [TOPLAS20].

I based an initial design of Obsidian on prior typestate-based research languages. Users found the system confusing: they were accustomed to reasoning dynamically about properties (such as ownership, which restricts which references can exist to a given value) that Obsidian included in the static type system. Using the approaches above, I changed the language to make programming easier for users while still obtaining the same safety guarantees. For example, I combined ownership and typestate into one syntactic construct to simplify reasoning. I also designed a mechanism that allows programs to dynamically check

[1] K. Delmolino, M. Arnett, A. Kosba, A. Miller, E. Shi. (2016) Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. In International Conference on Financial Cryptography and Data Security.

the state of shared objects while still obtaining the relevant guarantees. This approach enabled relaxing the static checks, improving usability because static checks are necessarily conservative. I also created a tutorial-based teaching tool so that I could teach users the language in a consistent way.

In a controlled experiment [OOPSLA20], I found that after a 90-minute training session, Obsidian users were able to complete significantly more tasks than Solidity users. In addition, the Solidity participants frequently inserted asset-related bugs, which the Obsidian compiler detects.

I led case studies to evaluate to what extent Obsidian was useful for writing real-world programs. For example, external partners at IBM used Obsidian to implement an application for tracking shipments, and a student implemented an insurance application based on requirements from the World Bank.

Bronze: a garbage collector for Rust that improves programmer productivity (in submission)

Rust is a new programming language that aims to bring type- and memory-safety to general-purpose programming contexts. Rust's strict ownership and lifetime discipline ensure that no reference can be used after its referent has been freed. Ownership improves performance over garbage collection (GC), an alternative approach to safe memory management. However, ownership means that each value has only one reference that permits mutation. This makes it difficult to write programs that require unrestricted aliasing, such as those implementing general object-pointer graphs.

What if programmers used ownership for performance-critical components but used GC for other components? Prior work[2] showed that GC could be applied judiciously to large portions of typical codebases with minimal performance cost. This would enable people to learn Rust gradually, deferring the trickiest parts until they were needed to improve performance. However, prior Rust GC libraries required users to either copy references explicitly or specify roots (which are used to determine which objects are reachable during collection) manually, sacrificing usability relative to a free-aliasing approach. To assess the potential of GC to improve Rust usability, I designed and prototyped Bronze [Bronze], a new GC for Rust. I modified the Rust compiler to emit LLVM stack maps so that Bronze can find roots for collection automatically, rather than requiring manual rooting. I evaluated Bronze in a controlled experiment [Bronze] with 333 participants in a programming languages course. In one task, participants who used Bronze spent only *one-third* as much time as those who used regular Rust. At the same time, they were also significantly more likely to complete the task successfully ($p \approx .006$). The experiment showed that aliasing restrictions can have significant usability costs, but those costs can be mitigated with GC.

Ongoing and future work

The Bronze study focused on using GC in a complex aliasing scenario. Could GC also ease the learning curve of Rust in simpler situations — and could it scaffold learning rather than only deferring the learning cost? I plan to evaluate whether GC could be a viable solution to improving Rust's learnability at earlier stages of learning. I am also interested in supplementing GC-only languages (such as Java) with ownership, bringing the performance benefits of ownership to more usable languages.

I am also conducting an observational study of Rust programmers to understand how we might build tools to make Rust easier to learn and use. For example, a compiler gives detailed error messages but does not attempt to model the user's understanding of the system. Akin to a cognitive tutor, what if the compiler modeled the user's understanding of Rust and provided appropriate high-level guidance?

[2] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney. Region-based memory management in Cyclone. PLDI '02.

Next, APIs can be thought of as small programming languages; they can be used incorrectly, resulting in undefined behavior or run-time errors [VLHCC15]. REST APIs, commonly used on the Web, are particularly interesting. Although there are numerous opinions regarding how best to write them, the scientific data are scant. In a collaboration I am leading with faculty and students at UMD, Tufts, and Syracuse, we are interviewing users and designers of REST APIs to develop hypotheses: do particular design approaches promote or inhibit bugs in clients? We hope to develop and evaluate tools and guidance to help people create safer, easier-to-use REST APIs in an empirically-justified way.

I want to also bring the benefits of safer languages to programmers without formal training in computing. Spreadsheet references can be unstable under structural operations, such as permuting rows and columns. Based on my eight years implementing and maintaining a commercial spreadsheet application, I saw that this instability leads to bugs in practice. I am developing and evaluating a new formula language, which will be safe under these structural operations. Then, to enable computations that are challenging in the limited context of a table, I am embedding the spreadsheet in a computational notebook. I hope to show that the resulting system is safer and more productive for users than traditional spreadsheets.

A new collaboration pertains to information flow control (IFC). IFC languages guarantee that secret data cannot leak out to unauthorized users, but they are notoriously hard to use. IFC is particularly challenging when information flows through shared, mutable objects. However, languages that have alias restrictions, such as Rust, already restrict how information can flow through mutable objects. We are working to develop an IFC system that uses ownership to provide strong security guarantees in a more usable way.

Finally, in each language design context in which I work, I seek to develop methodology for designing and evaluating languages that benefit people. My recent paper [HATRA21] argues for using theories from cognitive science as well as collaborating with cognitive scientists to enable designers to predict the effects of their designs on people. It focuses on the potential benefits of leveraging cognitive load theory, but other theories are worth exploring as well. For example, *attention* pertains to people's ability to process particular information and ignore other information and may have implications on programmers' abilities to write software that meets multiple requirements simultaneously. By developing and validating theories, I hope to reduce the need for expensive empirical studies when designers make choices.

Conclusion

I develop and apply user-centered approaches and theory to create programming languages that significantly improve programmers' abilities to write safe programs effectively. Then, I use user studies to evaluate the languages' benefits for users. With Glacier, I showed that although some programmers are likely to incorrectly mutate immutable structures in traditional languages, language-based immutability support can prevent these bugs without hampering programmers' ability to complete tasks. In Obsidian, I showed how a user-centered design approach led to a safe, novel language that programmers could learn to use after only a short training period. In Bronze, I showed empirical evidence that garbage collection can shorten task completion times — in this case, by a factor of three.

My work is (and must be) interdisciplinary, and has been published in software engineering, human-computer interaction, and programming languages venues. I have also taken a leadership role in organizing researchers around this area. By co-founding and co-organizing the Human Aspects of Types and Reasoning Assistants (HATRA) workshop at ACM SIGPLAN's SPLASH conference, I seek to promote and foster work on usability of safer languages in the research community.

My research program is about empowering programmers of all kinds to be more effective in achieving their goals. By both creating languages and developing methodology, I aim to impact both developers and language designers. I am excited to continue collaborating with other researchers, designers, and software engineers to help developers of all kinds create safer software.

[VLHCC15] **Michael Coblenz**, Robert Seacord, Brad Myers, Joshua Sunshine, and Jonathan Aldrich. A Course-Based Usability Analysis of Cilk Plus and OpenMP. VL/HCC 2015. https://doi.org/10.1109/vlhcc.2015.7357223

[ICSE16] **Michael Coblenz**, Joshua Sunshine, Jonathan Aldrich, Brad A. Myers, Samuel Weber, and Forrest Shull. Exploring Language Support for Immutability. ICSE 2016. https://doi.org/10.1145/2884781.2884798

[ICSE17] **Michael Coblenz**, Whitney Nelson, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Glacier: Transitive Class Immutability for Java, ICSE 2017. https://doi.org/10.1109/ICSE.2017.52

[OOPSLA20] **Michael Coblenz**, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can Advanced Type Systems be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian. OOPSLA 2020. 28 pages. https://doi.org/10.1145/3428200

[TOPLAS20] **Michael Coblenz**, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and Assets for Safer Blockchain Programming. ACM TOPLAS (December 2020), 82 pages. https://doi.org/10.1145/3417516

[TOCHI21] **Michael Coblenz**, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. ACM TOCHI Oct. 2021. https://doi.org/10.1145/3452379

[HATRA21] **Michael Coblenz**. Toward a Theory of Programming Language and Reasoning Assistant Design: Minimizing Cognitive Load. HATRA workshop at SPLASH 2021. https://arxiv.org/abs/2110.03806

[Bronze] **Michael Coblenz**, Michelle Mazurek, and Michael Hicks. Does the Bronze Garbage Collector Make Rust Easier to Use? A Controlled Experiment. Preprint: https://arxiv.org/abs/2110.01098. Under review.