Cyrus Omar Research Statement

I apply mathematical principles to design **user interfaces for programming languages** that **augment human intelligence** by:

- **helping people express their high-level intent** more quickly, correctly, and directly using the high-level mathematical concepts and notations, both textual and graphical, that arise most naturally in each problem domain of interest; and by
- **providing live feedback** so that people can develop a "useful degree of comprehension in situations that previously were too complex" [Engelbart, *Augmenting Human Intellect: A Conceptual Framework*, 1962].

My research is broadly applicable but I am especially interested in fundamentally improving the programming experience for **computational and data scientists** (motivated by my own past experiences as a neurobiologist), **web application programmers** (for whom **usable security** is critical), **students and educators** (for whom timely feedback and familiar notation is particularly helpful), and for **people with limited mobility and other disabilities** (who cannot effectively use many existing programming tools). I am also interested in designing better **interactive proof assistants** and in applying them in mathematics (including in my own research) and in verifying program correctness and security properties.

1 Hazel: Live Programming with Typed Holes POPL 2017 · POPL 2019

Live programming environments—for example, scientific lab notebook environments like Jupyter, spreadsheet environments, live audiovisual programming environments, and interactive proof assistants—differ from conventional batch programming environments in that they aim to provide various forms of assistance and feedback to the programmer *continously* throughout the editing process. The fundamental problem that has limited live programming environments from fully achieving continuity is the **gap problem**: programming language definitions traditionally assign meaning only to complete programs, meaning those that are syntactically fully-formed and free of type and binding errors. Consequently, editor services that require a syntactic, static or dynamic understanding of the program—everything from syntax highlighting to intelligent code completion, refactoring, debugging, program analysis, and data visualization services—**exhibit gaps in service whenever the programmer produces incomplete program text**. In some cases, these gaps are brief, such as when the programmer is in the midst of writing out a short expression, but in other cases, these gaps can persist for many minutes, hours, or even days, such as when filling in the branches of a large case analysis, after making a change to a datatype definition that causes errors to propagate throughout a large program, or simply when puzzling over what to do next. It is in situations like these where assistance and feedback might be most useful.

I lead the Hazel project, which is a live functional programming environment that **comprehensively solves the gap problem**: Hazel can **synthesize types for incomplete programs**, **run incomplete programs to produce incomplete results** and **perform various semantic transformations on incomplete programs**. In fact, Hazel makes these operations available for *every possible* incomplete program that the programmer might produce using the semantic transformations that it exposes as edit actions. Hazel represents incomplete programs as typed expressions with **holes**, with empty holes standing for missing expressions or types, and non-empty holes serving as membranes around type and binding errors. Hazel does not abort when it encounters a hole (unlike other systems that have some support for typed holes, like Haskell). Instead, evaluation continues around holes, producing a result containing **hole closures** that capture the run-time environment around each hole instance. Various editor services can use this information to provide feedback and assistance to the programmer as they fill the remaining holes in the program. When a hole is filled, evaluation can incrementally resume. The result is a fully continuous (i.e. gap-free) live programming experience.

Our approach has been to start from type-theoretic first principles with every aspect of Hazel's design. In particular, Hazel is rooted in a typed lambda calculus called Hazelnut Live. We detail its static semantics and, uniquely, its edit action semantics in a paper at **POPL 2017**. We then detail its dynamic semantics and the semantics of the fill-and-resume operation, together with a proof-of-concept UI that presents live hole closure information to the programmer, in a paper to appear at **POPL 2019**. Much of the machinery for working with type-level holes coincides with that of **gradual type theory** [Siek et al., SNAPL 2015]—we add the ability to continue evaluation around a run-time cast failure, treating it like a hole. Much of the machinery for working with hole closures is related to machinery found in **contextual modal type theory** [Nanevski et al., TOCL 2008], which exhibits a correspondence with intuitionistic contextual modal logic. These connections to well-established systems, together with the fact that we have mechanized the metatheory of Hazelnut Live using the Agda proof assistant (producing the **first mechanized specification for a live program editor**), give us confidence in the technical merits of our approach.

Our work on Hazel has been well-received both within and outside the academic community. Recently, I led the preparation of a grant proposal together with co-PIs Ravi Chugh (my post-doc advisor at UChicago) and Matthew Hammer (at CU Boulder). This proposal, which encompasses the ongoing and future work described in Sec. 1.1, Sec. 1.2 and Sec. 3 below, was selected this year for three years of funding by the **National Science Foundation**. Portions of this proposal were based on our Hazel "vision paper", which I presented at **SNAPL 2017**. More recently, I presented our vision at **Strange Loop 2018**, a highly selective multi-disciplinary conference that draws a large (2000+) predominantly non-academic audience. This led to substantial excitement around Hazel on social media (see my website for examples), which in turn has led to substantial interest from open source contributors and from undergraduate and graduate research students (see my teaching statement). Based on recent conversations, our research contributions are already having an impact on the design of a growing collection of other languages that are experimenting with typed holes, including Haskell, OCaml, Agda, Scala, and Elm. Members of our group at Chicago are integrating typed holes into Sketch-n-Sketch, a live functional graphics programming environment. Hazel is also directly influencing the design of the editor component of Dark (`darklang.com`), which is a cloud application development platform being designed by an early-stage startup (started by the founders of CircleCI). I have also been approached by several other companies interested in our approach.

1.1 Scaling Up — PSP 2014 🏆 · GPCE 2016 · Ongoing and Future Work

We are now scaling up the language of Hazel to encompass a full-scale functional programming language in the ML tradition—we are initially tracking toward parity with the popular Elm web programming language (`elm-lang.org`) because its design allows us to conserve all of the theoretical guarantees we have established so far. After that, we will address the theoretical challenges that arise when including features from **dependently typed languages** (where evaluation is in the service of equational reasoning and the type and expression levels are collapsed) and from **imperative languages** (where effects complicate the fill-and-resume operation and require exercising more caution when performing continuous evaluation).

We are also working to enrich Hazel's language of semantic edit actions to offer a more text-like "hybrid" structure editing experience that does not succumb to the usability problems of early structure editors, which offered only local tree transformations. We are following the contemporary example of the mbeddr project, which has produced a full-scale hybrid structure editor for a C-like language and shown empirically that programmers are able to use it productively [Berger et al., FSE 2016]. Hazel also offers a readable, conventional textual syntax for use in situations where the available semantic edit actions remain too rigid. We plan to introduce **text edit holes** that allow local text edits while maintaining continuity in the rest of the program.

Ultimately, we would like the UI advances pioneered in Hazel to be available to researchers and domain experts exploring many different points in the language design space. There are two ways to achieve this goal. First, we could automatically generate a dialect of Hazel from a lightly annotated language definition—this approach was explored with the Synthesizer Generator [Reps and Teitelbaum, SDE 1984] and more recently with, for example, Citrus [Ko and Myers, UIST 2005]. Recent work on the Gradualizer suggests a type-theoretic foundation for this approach [Cimini and Siek, POPL 2016]. Another approach would be to allow library providers to modularly install new **semantic fragments**. In a paper at **GPCE 2016**, we introduce such a mechanism, implement it as a statically typed language called `typy` that is embedded into Python as a library, and argue that its design is compositionally well-behaved and expressive, with examples of fragments that express the static and dynamic semantics of (1) both functional and object-oriented primitives; (2) typed foreign interfaces to Python and OpenCL; and (3) a system of regular string types useful for verifying that strings have been securely sanitized (which we introduced in a workshop paper at **PSP 2014** where it was awarded the **Best Paper Award**). I plan to continue to advance both of these approaches.

1.2 Programmable Edit Actions — Future Work

I also plan to develop mechanisms that **allow library providers to modularly install more sophisticated semantic edit actions** into Hazel while maintaining the essential continuity invariant. This will allow us to implement many common editor services as libraries, rather than building them into Hazel—examples include refactoring services, program synthesis services and program repair services (which can take advantage of hole closure information). Moreover, this opens up the possibility of library-specific edit actions, e.g. transformations that take advantage of algebraic properties of data structures or that capture common high-level idioms, which serves the goal of helping client programmers express their high-level intent more quickly, correctly, and directly.

1.3 Intelligently Suggesting Semantic Edit Actions — Future Work

Another major research direction I plan to pursue is the development of an **intelligent programmer's assistant** that **suggests edit actions that the programmer most likely intends**, taking into account both the rich semantic information available continuously in Hazel as well as statistics gathered from existing code bases and from edit action histories, from which we can learn the *idioms* that programmers prefer. This approach, where **the function of the AI is to generate idiomatic semantic edits to a program**, could help to address the important problem of **explainable AI**. I have some experience with integrating statistical machine learning into user interfaces—see Sec. 4.1 below; I also did some preliminary work in this direction as a graduate machine learning course project—and I am excited to **collaborate with researchers in AI and machine learning** in the future.

1.4 Semantic Foundations for Collaborative Editing — Future Work

Hazel currently supports only a single programmer, but modern programming is collaborative—programmers ubiquitously use version control and pair program using collaborative editors. Rather than relying on structural diffs to support these services, which can obscure the semantics of the change that a programmer has made, I would like to define a language of semantic edit actions that forms a convergent replicated data structure (CRDT), meaning that the edit action histories from different replicas of a program, being edited by different people, can be interleaved. To support this, we will need several new mechanisms including **conflict holes** that serve to maintain continuity even after there have been conflicting edits.

2 Reasonably Programmable Literal Notation — ECOOP 2014 🏆 · ICFP 2018

The surface syntax of a programming language is its primary user interface. However, programming languages typically define literal notation for only a few common data structures, like lists, so user-defined data structures must use general-purpose notation. In response, programmers often resort to "stringly-typed programming", using string encodings for data structures for which a more well-behaved composite encoding is considered too notationally costly. We performed an empirical analysis of strings in open source code, finding that a substantial fraction of them were being used in this way. String encodings can become vectors for injection attacks, which are the **leading class of security vulnerabilities** in web applications.

In response, I developed mechanisms that allow library providers to **install new literal notation** for user-defined data structures. Mechanisms like these must be introduced carefully to avoid opening the door to syntactic conflict and client confusion. The mechanisms I developed eliminate conflict and ensure that client programmers can still **reason abstractly about types and binding**. For example, consider a web programming library that defines a type, `Element`, that encodes the structure of HTML elements. Using the mechanism I have developed, the library provider can equip this type with literal notation based directly on

the HTML standard extended with support for spliced expressions, so that HTML values can be constructed programmatically. Client programmers can use this notation, within the outer delimiters ‘(and)‘, wherever a value of type `Element` is expected:

```
val body : Element = ‘(<p>Hello, {{join ’ ’ [first, last]}}</p>)‘
```

Splicing is semantic, so there is no vector for string injection attacks. Our mechanism also ensures that client programmers need not examine the underlying expansion, nor the details of the parser, to reason about the type and binding structure of the program. Instead, it suffices to know about simple annotations on the notation definition (not shown) and the locations of spliced expressions, which are tracked and can be communicated using color in the programming environment as shown above. The difficulty in designing a mechanism like this is that we must treat spliced expressions hygienically, but previous hygiene mechanisms for macro systems are unable to do so. We solved this problem by developing a two-phase expansion process.

This research was published at **ECOOP 2014**, where it received a **Distinguished Paper Award** and substantial coverage in the popular programming press (see my website). A subsequent paper, which was published at **ICFP 2018** and is based in part on mechanisms I introduced in my **doctoral dissertation** in 2017, added support for several advanced language features from ML-family languages, including pattern matching, parameterized types, type reconstruction and ML-style modules. It also further strenghthened the hygiene mechanism and **formally stated and proved all of the abstract reasoning principles**. At Chicago, I worked with a talented undergraduate, Charles Chamberlain, to implement this mechanism into Reason, which is Facebook’s increasingly popular new syntactic front-end for OCaml. Our implementation, called Relit, was released in November 2018 and it is has also received substantial interest from practitioners (see my website).

3 Palettes: Type-Specific UIs for Code Completion — ICSE 2012 · Ongoing Work

For some data types, a purely textual notation is not ideal—for example, colors, shapes, UI widgets, matrices, and audio filters are all examples where an **interactive graphical representation** may sometimes be more natural. I have developed a mechanism that allows library providers to install **type-specific graphical user interfaces**, called **palettes**, that help the programmer generate code directly within the program editor. Clients are offered access to relevant palettes via the code completion menu based on the type of expression being entered. In a paper published at **ICSE 2012**, we introduced this mechanism and described our human-centered design process. We began by gathering both quantitative and qualitative feedback on initial mockups from **nearly 500 professional software developers** using an online survey. This resulted in a ranked collection of design criteria that guided our final design. We also solicited a large number of use cases for palettes. We then implemented our refined design as an extension to Eclipse for Java called **Graphite**. Finally, we validated our design with a small controlled pilot study.

We are now integrating an improved palette mechanism into Hazel. In particular, we are **giving palettes access to hole closure information**, based on the work in our **POPL 2019** paper discussed above, so that they can show clients the dynamic implications of their UI choices. Moreover, Hazel’s palettes are not ephemeral—they persist in the program as expressions. Finally, palettes in Hazel can themselves contain typed holes, i.e. they are compositional. The reasoning principles established in our **ICFP 2018** paper, discussed above, are also relevant to this design (palettes are, in some sense, interactive graphical notation).

4 Applications

I am excited about evaluating and refining our designs by applying them in various problem domains, selected primarily based on the interests of my students. The domains that most interest me personally relate to my prior experiences as a neurobiologist.

4.1 Programming UIs for People with Limited Mobility — IJHCI 2011 · Future Work

I studied both neurobiology and computer science as an undergraduate, in large part because I was interested in designing brain-computer interfaces (BCIs) in order to help people with severely limited mobility by allowing them to program various devices, including wheelchairs, speech synthesizers, and, because fun and creative expression is so important, model airplanes and figure drawing applications. The problem is that EEG-based BCIs fundamentally provide extremely limited bandwidth so controlling standard user interfaces is infeasible. I teamed up with Prof. Todd Coleman and we decided to approach the problem using the tools of information theory. By modeling the BCI as a noisy, asymmetric channel (an odd class of channel that had been studied only sporadically by information theorists), we were able to design an optimal control scheme and to incorporate statistical models of user intent—e.g. natural language models and segmented movement models. We then demonstrated that human subjects could successfully control the resulting user interface using an EEG headset with very little training. This research culminated in a journal publication (**IJHCI 2011**). I continue to be interested in developing user interfaces for programming languages that are useful to people with disabilities. I also remain interested in statistical models of user intent (see Sec. 1.3).

4.2 Computational and Data Science — Journal of Neuroscience 2012 · Future Work

I was very curious about the source of the signals that our EEG system was picking up, so I decided to enter graduate school at CMU in the Neural Computation PhD program. There, I built a mathematical model that explained how a common excitatory-inhibitory circuit responds to external stimulation, based on data that had been gathered from the rodent whisker barrel cortex. We described that data together with my mathematical model in a publication in the Journal of Neuroscience (**J. Neurosci. 2012**). I then spent the summer at Los Alamos National Lab (as a requirement of the **DOE CSGF Fellowship** that I had received). There, I worked to build more complex computational models of the rodent visual cortex. However, I found myself frustrated with the computational tools available to me as a researcher in this complex problem domain. Whereas my dialogue with my colleagues was mediated by elegant mathematical concepts and notations, including whiteboard diagrams, my dialogue with the computer required using the limited language and notation of the machine and feedback was available only sporadically. I decided that in order to fully understand human intelligence, we would need tools that better augment human intelligence.