Research Statement – Cyrus Omar

My research aims to bring to life a new generation of highly *adaptable* and *intelligent* programming environments rooted in the fundamental principles of type theory.

By an "adaptable programming environment", I mean one that allows the libraries that the programmer has imported to install new syntactic, semantic and edit-time features. Adaptability has become critical as programming has pervaded society, because language and tool designers can no longer hope to satisfyingly accomodate every problem domain *a priori*. The challenge is in ensuring that these newly installed features do not interfere with one another, nor weaken the type and binding discipline of the language (because a language with a strong type and binding discipline is, I am thoroughly convinced, necessary for programming "in the large".) My research has rigorously confronted this challenge.

By an "intelligent programming environment", I mean one that combines a semantic understanding of the program being written with statistics gathered from other programs to synthesize code suggestions, and more generally *action suggestions*, for the programmer throughout the programming process. There are many important problems that await confrontation here. My interest has been in developing principled solutions to foundational problems, including (1) the problem of extracting meaning from programs that are incomplete; (2) the problem of reasoning about edit actions as formal structures; and (3) the problem of combining semantic and statistical approaches to suggestion generation.

I am particularly interested in incorporating these mechanisms into next-generation languages and tools for scientists and other professional end-users (e.g. engineers, data scientists), and for individuals with severely limited mobility and other disabilities. I have a distinctive perspective on these topics because as a former student in neuroscience, I developed computational models of neurobiological circuits in the sensorimotor system, and I designed and built rehabilitative brain-computer interfaces.

Previous and Ongoing Research

Adaptable Programming Systems Over the course of my graduate studies, I have developed several mechanisms that allow libraries to install new language and editor features, without running afoul of conflicts or causing client confusion.

My thesis research focuses on mechanisms that allow libraries to install type-specific syntactic sugar. For example, consider a web programming library that defines a type, `Element`, of HTML elements. Using the mechanism I have developed, the library provider can equip this type with standard HTML syntax. Clients of this library can use this syntax, within backticks, wherever a value of type `Element` is expected, e.g.

```
val body : Element = `<p>Hello, {[join ' ' [first, last]]}</p>`
```

Uniquely, syntactic conflicts are impossible by construction, because the outer delimiters (here, backticks) are fixed, and control over parsing the body (between backticks) is delegated in a deterministic, type-directed manner. Moreover, the mechanism comes equipped with a principle of *syntactic abstraction*, meaning that library clients can reason about types and binding without examining the expansions of newly installed syntactic forms like these. The core mechanism was formally introduced in a paper at **ECOOP 2014**, where it was awarded a **Distinguished Paper Award**. It was also covered widely by the popular programming press, and our implementation now has over 400 stars on GitHub. My dissertation formalizes variants of this mechanism that interact with advanced language features, including pattern matching, parameterized types and ML-style modules. An account of these subsequent advances was awarded **2nd place at the ICFP 2015 Student Research Competition**, and a capstone publication based on these results is in preparation for submission (to POPL in July) .

I have also developed a mechanism that gives library providers the ability to install new *semantic fragments*. These fragment definitions are delegated control over typechecking and translation (to a fixed target language) in a type-directed manner. In a paper at **GPCE 2016**, we argue that this design is compositionally well-behaved and expressive, with examples of fragments that express the static and dynamic semantics of (1) functional primitives (e.g. records and labeled sums, with support for nested pattern matching *a la* ML); (2) a variation on JavaScript's prototypal object system; (3) typed foreign interfaces to Python and OpenCL; and (4) a system of constrained string types useful for verifying that strings have been securely sanitized as intended. This last fragment, developed in collaboration with an undergraduate research student, was formalized in a paper at the **PSP 2014** workshop, where it was awarded the **Best Paper Award**. Our implementation, `typy` (now called Tidy), is embedded as a library into Python, which makes it immediately practical as a tool for scientists (Python is among the most popular languages in scientific computing today.)

Finally, I have developed a mechanism that allows library providers to install type-specific graphical user interfaces that generate code directly within the program editor. Clients are offered access to these interfaces via the standard code completion menu based on the type of expression being entered at the cursor. In a paper published at **ICSE 2012**, we introduced this system and described our human-centered design process. In particular, we began by gathering feedback on our initial mockups from over 450 professional software developers via an online survey. This resulted in a ranked collection of design criteria that guided our final design. We validated our final design with a controlled pilot study.

Intelligent Programming Systems Programming language definitions assign meaning to *complete* programs. Programmers, however, spend a substantial amount of time interacting with *incomplete* programs using tools like program editors and live programming environments (which interleave editing and evaluation.) These systems sometimes attempt to provide suggestions to the programmer. However, these suggestion systems are built largely around various *ad hoc* heuristics. I am interested in designing principled programming systems that intelligently suggest edit actions to the

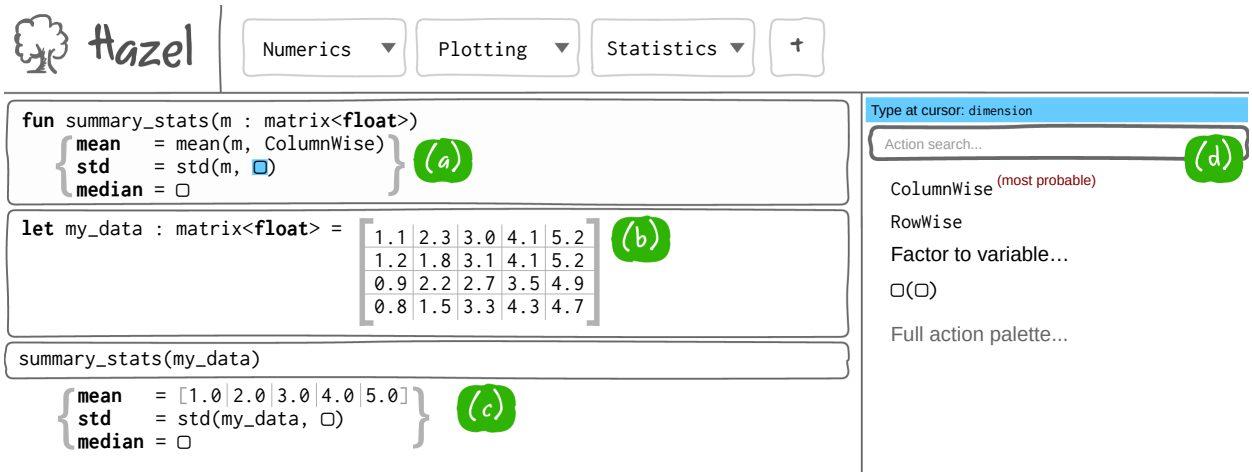

Figure 1: A mockup of Hazel.

programmer. To do so, these systems need to (1) to be able to extract meaning from incomplete programs, not just complete programs; (2) to reason about and compose structured edit actions; and (3) to have access to a sound statistical model of incomplete programs and edit actions over them, so that the suggestions can be ranked by likelihood.

I have led the development of a foundational calculus called Hazelnut that addresses the first two of these problems. This calculus, described in a paper appearing at **POPL 2017**, assigns static meaning to programs that contain *holes* and *type inconsistencies*. Interestingly and encouragingly, the machinery for type holes coincides with that developed in the study of gradual type systems. The paper then develops an *action semantics* that assigns meaning to structured edit actions and maintains a powerful invariant: that every well-defined action leaves the program statically meaningful, according to the aforementioned static semantics. We have mechanized the metatheory of Hazelnut using the Agda proof assistant. We also have a browser-based implementation of a simple structure editor based on this calculus written in OCaml using functional reactive programming techniques, and compiled using `js_of_ocaml`.

With respect to the problem of developing statistical models of edit actions, I have led two projects of note. First, during my years as a neuroscientist, I developed a principled information theoretic framework for designing *brain-computer interfaces*: systems that translate neural activity generated by paralyzed and locked-in individuals into commands (i.e. simple programs) for various devices. We used speech synthesizers, motorized wheelchairs, and radio-controlled airplanes (because everyone deserves to have some fun.) This framework, which was described in **IJHCI 2011**, relies on a statistical understanding of both the communication channel (we used an EEG headset) and the language of commands.

The second project, which began as a course project for a Machine Learning course and remains ongoing as of this writing, involves developing a principled statistical model of well-typed expressions that takes into account the form, the type and the context around the expression being evaluated for insertion. In our preliminary investigations, our simple model already keeps pace with models that use an *ad hoc* tokenized representation of programs [1].

Computational Neuroscience and Neuroinformatics I entered graduate school at CMU in the Neural Computation PhD program. There, I worked in collaboration with an experimental neurophysiologist on a computational model of a neurobiological circuit in the portion of the rat brain responsible for whisker sensations. This research was published in the **Journal of Neuroscience (2012)**. I then became interested in the problem of systematically validating scientific models like these against empirical data on an ongoing basis. Our solution, which combines statistical testing techniques and modularity techniques, was described in a short paper at **ICSE 2014** (NIER track, 18% acceptance rate.) My collaborator, Rick Gerkin, continues to push forward on this project, funded by an R01 grant from the National Institutes of Health. I contributed to the writing of this grant. My transfer to the Computer Science Department was precipitated by my experiences as a neuroscientist — I became convinced that we need better languages and tools!

Future Research Directions

Going forward, I would like to (1) continue to develop new adaptation and intelligent action suggestion mechanisms, starting from type theoretic and statistical first principles, in order to build enduring *theoretical foundations for interactive programming tools*; and (2) integrate these advances into Tidy, described above, and also into an accompanying next-generation *live lab notebook* programming environment called Hazel. A mockup of the Hazel programming environment, which is organized much like the widely adopted Jupyter (formerly IPython) lab notebook, is shown in Figure 1. Hazel will advance the state of the art in several ways.

Hazel will be a hybrid *structure editor* – incomplete programs will never be syntactically malformed. Instead, portions of the program that are not yet complete will be explicitly marked with holes, indicated by rounded squares in the cell marked (a) in Figure 1. Uniquely, Hazel, like Hazelnut, will go beyond syntactic well-formedness to maintain a stronger invariant: that every incomplete program that arises is also statically meaningful. This will require **scaling up the static semantics for incomplete programs** developed for Hazelnut (POPL 2017). I plan to do so by equipping Hazel with a fragmentary semantics, building upon my recent research in this area (GPCE 2016). In particular, I plan to develop a

variant of this mechanism within Coq, and use Coq's extraction mechanism to generate an OCaml implementation, which will be compiled by `js_of_ocaml` (as in our current implementation of Hazelnut.) This will allow us to make another fundamental advance: fragment providers will be able to develop **modular correctness proofs**.

Hazel will also allow the programmer to construct values using non-textual *projections*. For example, the cell marked (b) in Figure 1 shows a projection of a value of matrix type. Uniquely, the logic for displaying and interacting with this projection will not be built in to Hazel, but rather installed by the `Numerics` library and triggered based on the type annotation. This mechanism of **type-specific projections** will build upon the advances made in my previous work on type-specific syntax (ECOOP 2014) and type-specific code generation interfaces (ICSE 2012).

In the cell marked (c), the programmer applies `summary_stats` to the aforementioned matrix value. In a standard programming environment, the fact that `summary_stats` is incomplete would disable evaluation. For Hazel, however, I plan to investigate the problem of **evaluating incomplete programs**. Notice in cell (c) that the computation of the mean is able to proceed, and the computation of the standard deviation can proceed at least as far as is shown. This tightens the feedback loop for the client programmer, going beyond what live programming environments today are capable of. It is challenging to get the dynamic semantics of incomplete programs right, because the standard notion of type safety breaks down – evaluation can in fact "get stuck". I plan to develop a refined notion of type safety that precisely characterizes states that are appropriately stuck. I also plan to investigate the foundations of **edit and resume** functionality, which would allow the programmer to fill in holes and continue evaluation where it left off. This would be particularly useful for scientific computations involving large data sets. This line of research will build both upon the static semantics for incomplete programs we have developed (POPL 2017) and on work on contextual modal type theory (CMTT), which, by its correspondence with contextual modal logic, lays logical foundations beneath the problem of manipulating and reasoning about programs with holes [2].

The sidebar marked (d) offers feedback and action suggestions to the programmer. There are several fundamental research problems lurking behind this sidebar. The first was already discussed – we need a semantics for edit actions. Our paper at POPL 2017 lays the foundations for Hazel's primitive action semantics. We next need a system that allows library providers to modularly install new **high-level edit actions**, defined in terms of the primitive actions. For example, it should be possible to implement various type-directed program generation and bug repair techniques as libraries, and integrate them directly into the editor, using this system. Each of these are substantial research areas in their own right – I am already collaborating with Claire Le Goues at CMU, an expert on automated bug repair, and I plan to develop further collaborations in these directions in the future. Third, we need an **action suggestion semantics** that determines which suggestions are actually generated for any given edit state. This semantics must come equipped with a theorem that establishes that the suggestions offered are meaningful at the cursor (shaded in blue in Figure 1). Finally, we need a **statistical model of actions** capable of ranking the suggested actions, as shown in the sidebar in Figure 1. There are several theses worth of open problems on this front. I plan to start with a simple foundational statistical model (ongoing work, discussed above), and move forward to incorporate more sophisticated machine learning techniques, ideally in collaboration with researchers in machine learning. This will be the basis for at least one grant submission.

With Hazel, we are intentionally blurring the line between the programming language and the program editor. This opens up a number of interesting research directions in **language-editor co-design**. For example, it may be possible to recast "tricky" language mechanisms, like function overloading, type classes, implicit values and unqualified imports, as editor mechanisms. Because we will be treating programming as a structured conversation between the programmer and the programming environment, the editor can simply ask the programmer to resolve ambiguities when they arise. The programmer's choice is then stored unambiguously in the syntax tree.

Another interesting problem that I plan to explore has to do with **semantic, interactive documentation**. In particular, in Hazel, references to program structures that appear in documentation will be treated in the same way as other references and be subject to renaming and other operations. Documentation will also be capable of containing expressions of arbitrary types (e.g. of the `Image` or `Link` type.) Together with the type-specific projection mechanism mentioned above, I hope that this will allow Hazel to function not only as a structured programming environment, but also as a **structured authoring environment**.

My research philosophy is to start with the first principles of type theory and probability theory. As such, my contributions will first take the form of core calculi, mechanized proofs and simple prototypes. That said, I am a big believer in applying disciplined qualitative and empirical methods to scale up, validate and iterate on my designs. In particular, I intend to make Hazel a practical tool for data science tasks, and to conduct pilot studies on its efficacy in this domain. These pilot studies will generate data that we plan to both publish and utilize internally as we iterate on Hazel's design. The ICSE 2012 paper described above demonstrates my experience with this approach. Taken together, I believe that my research has the potential to open up exciting new research directions for semanticists, tool designers and statisticians, both individually and in interdisciplinary collaboration. My vision is that one day, scientific artifacts of all forms will be produced using Hazel, and intellectual derivatives thereof.

References

[1] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[2] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.