Research Statement

My overwhelming research interest is in solving the problem of writing correct and elegant software, and in developing tools and methods that help towards that end.

Software is rapidly eating the world. Unfortunately, as a species, humans seem to be fundamentally unsuited to writing software—at least, given the tools we currently have. While applications of software rapidly outpace most industries' ability to cope with the changes, very little of that productivity improvement seems to benefit the programmer experience. For example, most programming tasks occur in a terminal emulator: that is, an emulator of a 50 year old machine. We edit program trees by pretending they are strings of bytes. And, by and large, we use programming languages that have internalized all the pain and necessary evils of how our computer machines used to operate.

Strict, interactive tools with fast-feedback are unreasonably effective at teaching users better discipline, even in the absence of the tool. For example, users of Haskell often say they eventually gain the ability to run the typechecker in their head: a skill that can be taken to other programming languages. "Aha, this program would not typecheck in Haskell" they might think, "therefore there is probably a bug here."

I have spent over a decade of my life writing ALGOL-inspired languages, but could never shake the feeling that this couldn't be the apex of language design. Over the summer after graduating university, I taught myself Haskell, and for the first time discovered that software needn't be as awful an experience as I had previously thought. Programming languages are tools for thought, and I find myself compelled to help improve the status-quo.

Towards this goal, I have many specific interests. Effect systems, which can help enforce the Law of Demeter, ensure software is correctly modularized and make convenient-but-unmaintainable short-term fixes costly. I'm the author of Polysemy, a popular effect system in Haskell, but the formulation is too difficult to use in higher-order contexts, and completely falls apart in some desirable situations. I would like to find flexible and lightweight approaches to effect systems, and in particular, find techniques for integrating them into existing codebases.

Another of my research interests is automating tedious programming tasks. Many tasks are unambiguously stated in few words, but require thousands of keystrokes to implement. Property testing is a good example here; it's much easier to automate generating unit tests than it is to write them by hand. But the idea goes much further; why am I unable to state function f is a homomorphism with respect to X and have the computer elaborate the details? Why can't the computer write the boilerplate and ask me only about the pieces that require human insight? Interactive theorem-provers do a lot of good work in this direction, but I'd like to find ways of bringing the benefits to people who aren't yet experts.

After becoming frustrated with traditional code editing paradigms, I decided to build a tool to help. At any given step in a program, there are maybe fifty possible functions to call, variables to reference or statements to run. This is only six bits of information I should need to inform the computer of, but every keystroke asserts seven, and every language construct is on the order of 10 to 100 keystrokes. The takeaway: programming is at somewhere between two and three orders of magnitude more inefficient than it need be. My solution was to implement an editing interface that talks to the compiler, maintaining lists of all variables in scope and their types. By refining "holes" in the program, my work is capable of synthesizing code and performing many common type-directed tasks. My product, Wingman for Haskell, is now part of the Haskell Language Server and is used daily by many thousands of people.

I have also built an experimental tool for light-weight model checking. The user jots down some desirable properties of a not-yet-written software interface, and the tool performs confluence checking of those rules. Even with well-thought-out designs, usually some subset of the properties turn out to be impossible to satisfy simultaneously—something much better to learn at the beginning of a project than at the end. I would like to go further in this direction—in essence, building a composable toolkit of good software behaviors, and getting automated implementations and tests for them.

Further towards the ends of automating tedious programming tasks, I am also very interested in generic programming; that is, what are the smallest units I can write a program at and compose together to build the desired behavior? Having better tools (mental or algorithmic) for decomposing problems in this way is an area in which I see great potential for capitalizing on low-hanging fruit.

Datatype descriptions are a powerful means of doing generic programming over types, but I'd like to find a way to turn this around and factor an algorithm into generic parts, which could then be composed differently to create an infinite family of equivalent algorithms with different computational properties. I can see significant gains here in automation (general tools for automatically optimizing a brute-force function), as well as in demystifying the process of designing algorithms.

I am confident in my interest in, and successful completion of, a PhD program. My past four years attest to my ability for self-directed work as well a proven history of autodidacticism—having researched and written two books, as well as having made exceptional open-source contributions. I am curious, results-driven, and eager to learn. In addition, I believe these research directions to be tenable and important; software programming is highly paid, ubiquitous, and onerous. Even modest gains here would save huge costs, alleviate misery, and raise the waterline on the future of engineering endeavors.

Publications

Books

1. Maguire, Sandy. *Algebra-Driven Design: Elegant Software from Simple Building Blocks.* Leanpub. 2020.
2. Maguire, Sandy. *Thinking with Types: Type-Level Programming in Haskell.* Leanpub. 2018.

Conference Talks

Invited

1. Maguire, Sandy. (2022, Dec 8-9) *Just Because It Works Doesn't Mean It's Right: Finding Elegance in Quadtrees.* [Conference presentation]. Haskell eXchange 2022, London, UK.
2. Maguire, Sandy. (2021, June 18-20) *A New Kind of Programming: Tactic Metaprogramming in Haskell.* [Conference presentation]. ZuriHac, Zurich, CH. https://www.youtube.com/watch?v=BuEn1J90bKg
3. Maguire, Sandy. (2019, Oct 10-11). *Polysemy: Chasing Performance in Free Monads.* [Conference presentation]. Haskell eXchange 2019, London, UK. https://skillsmatter.com/skillscasts/14665-polysemy-chasing-performance-in-free-monads

Given

4. Maguire, Sandy. (2017, Oct 14). *Some1 Like You: Dependent Pairs in Haskell.* [Conference presentation]. LambdaConf 2017, Boulder, Colorado, USA. https://www.youtube.com/watch?v=PNkoUv74JQU
5. Maguire, Sandy. (2017, Apr 7-9). *Don't Eff It Up: Freer Monads in Action.* [Conference presentation]. BayHac 2017, San Francisco, California, USA. https://www.youtube.com/watch?v=gUPuWHAt6SA