Sandy Maguire

343.262.3363 • sandy@sandymaguire.me • https://reasonablypolymorphic.com

Research Interests

Functional programming, compiler design, free monads

Education

2010-2015 **BSE, Software Engineering, Honours, Co-operative Program**
University of Waterloo, Waterloo, ON

- 77.16 Cumulative GPA
- Graduated in Excellent Standing

Classes and Grades

2015	**Compiler Construction** - 92
2015	**Design Project 2** - 89
2015	**Introduction to Artificial Intelligence** - 84
2015	**Software Testing and Quality Assurance** - 76
2015	**Introduction to Microeconomics** - 74
2015	**Cybernetics and Society** - 66
2014	**Design Project 1** - 92
2014	**Quantum Physics 1** - 83
2014	**Computer Networks** - 78
2014	**Software Design and Architectures** - 73
2014	**Cooperative and Adaptive Algorithms** - 72
2013	**Design Project Planning** - 96
2013	**Operating Systems** - 87
2013	**The Use of English** - 86
2013	**Software Requirements Specification and Analysis** - 83
2013	**Concurrent and Parallel Programming** - 75
2013	**User Interfaces** - 74
2013	**Introduction to Database Management** - 71
2013	**Introduction to the Theory of Computing** - 70
2013	**Numerical Computation** - 68
2013	**Introduction to Feedback Control** - 65
2013	**Algorithms 1** - 58
2012	**Data Structures and Data Management** - 80
2012	**Software Engineering Principles** - 75
2012	**Introduction to Combinatorics** - 74
2012	**Waves, Electricity and Magnetism** - 72
2012	**Advanced Mathematics for Software Engineers** - 67

2012	**Engineering Economics** - 67
2011	**Foundations of Sequantial Programs** - 85
2011	**Chemistry for Engineers** - 81
2011	**Digital Circuits and Systems** - 80
2011	**Functional Programming and Data Abstraction** - 79
2011	**Calculus 2 for Engineering** - 76
2011	**Digital Computers** - 75
2011	**Statistics for Software Engineering** - 75
2011	**Physics of Electrical Engineering 2** - 73
2011	**Introduction to Philosophy** - 70
2011	**Algebra for Honours Mathematics** - 62
2011	**Logic and Computation** - 62
2010	**Linear Algebra for Engineering** - 96
2010	**Programming Principles** - 93
2010	**Physics of Electrical Engineering 1** - 86
2010	**Introduction to Methods of Software Engineering** - 85
2010	**Calculus 1 for Engineers** - 78
2010	**Linear Circuits** - 73

Professional Experience

2016-2018 **Senior Haskell Software Engineer**
Takt

- Led a team of four to reimplement a core subcomponent of the product --- increased the cadence of new feature development from months to days.
- Directed a team of three to implement a high-throughput, low-latency brokered streaming library. Resulting library became the company's core inter-service communication protocol.

2015-2016 **Engineer III (Identity and Access Management)**
Google

- Led the architectural design effort of a user-defined permission model for the cloud --- the team's only project for the subsequent quarter.
- Took ownership over an unmaintained, service-critical internal compiler; improved compile times by 96% and test coverage by 65%.

2014 **Engineering Intern (Ads Ranking)**
Facebook

- Analyzed the advertising platform's spending behaviors, and subsequently implemented algorithmic changes --- resulting in a 0.5% revenue increase.
- Parallelized the back-end graph ranker, resulting in site-wide response time gains of 0.4%.

2013 **Engineering Intern (FIFA Mobile)**
Electronic Arts Canada

- Single-handedly cleaned up 60% of the project's technical debt alongside assigned tasks.

2012 **Gameplay Programmer Intern**
Digital Extremes

- Architected and implemented the game's entire voice-over-IP stack.
- Rewrote a significant portion of the AI and PhysX engines to support arbitrary 3-space rotation.
- Designed and implemented a type-system for the game's proprietary scripting language runtime.

2011 **.NET Consultant**
Systemgroup Consulting Inc,

- Found and closed over ten highly-exploitable vulnerabilities in a multi-billion-dollar company's website.

Publications

Books

2018 **Thinking with Types: Type-Level Programming in Haskell** [link]
Sandy Maguire, self-published

- Favourably reviewed; described as being "well written and very approacable", "absolutely brilliant," and "a great read."
- Sold over 1,000 copies --- a significant portion of the estimated 10,000 members in the Haskell community.

2016 **How These Things Work** [link]
Sandy Maguire, self-published

Blog

2015- **Reasonably Polymorphic** [link]
Sandy Maguire

- Professional and research blog.

Presentations

2019 **Chasing the Performance of Free Monads**
Lambda Montreal • Montreal, QC

2018 **Existentialization: What Is It Good For?** [slides]
Denver FP • Denver, CO

2017 **Some1 Like You: Dependent Pairs in Haskell** [video]
LambdaConf • Boulder, CO

2017 **Don't Eff It Up: Freer Monads in Action** [video]
BayHac • San Francisco, CA

Open Source Contributions

Maintainer

2019- **isovector/polysemy** [link]
Higher-order, no-boilerplate, zero-cost free monad library.

2019- **isovector/suavemente** [link]
An applicative functor capable of talking directly to HTML inputs.

2018- **isovector/do-notation** [link]
Generalized do-notation for indexed monads.

2018- **isovector/latex-live-snippets** [link]
Keep code snippets up to date in LaTeX documents.

2018- **isovector/constraints-emerge** [link]
Runtime reification of Haskell's instance resolution machinery.

2018- **isovector/ecstasy** [link]
A boilerplate-free entity component system library.

2017- **isovector/th-dict-discovery** [link]
Compile-time reification of every typeclass instance in scope.

2017 **TaktInc/freer** [link]
Corporate fork of a freer monad library.

Contributor

2019 **Lysxia/first-class-families** [link]
A library for higher-order type-level programming in Haskell.

2017 **kim/opentracing** [link]
Haskell implementation of the OpenTracing instrumentation API.

2017 **google/proto-lens** [link]
API for protocol buffers using modern Haskell language and library patterns.

2017 **IxpertaSolutions/freer-effects** [link]
A free monad library in the style of "Freer Monads, More Extensible Effects."

2017 **k0001/exinst** [link]
An implementation of sigma types in Haskell.

Committer

2018 **nomeata/inspection-testing** [link]
Automated testing of Haskell's Core representations.

2018 **AshleyMoni/QuadTree** [link]
QuadTree library for Haskell, with lens support.

2018 **conal/concat** [link]
A GHC plugin to compile Haskell to constrained categories.

2018 **diagrams/diagrams-contrib** [link]
User contributed extensions to diagrams.

Honours, Awards and Scholarships

2015 **SoftEng Capstone Design Symposium (2nd place)**
Yelp

2010 **President's Scholarship**
University of Waterloo

Research Statement

A friend of mine often jokes that "Haskellers are just early adopters of correctness." Despite being tongue-in-cheek, I think he's onto something here.

My program in university was co-operative, meaning that after each school term I was expected to find a job and spend four months excelling out in the "real world." Four months is just about as long as it takes most people to start feeling comfortable in a new codebase, and as a result, I needed to learn how to learn fast.

It became clear to me that a great deal of human capital was being wasted simply trying to reverse-engineer the codebases in which my cohort and I found ourselves. The industry prioritizes writing "readable" code, characterized by being verbose and "doing nothing clever". While each step along the way is indeed comprehensible, too often this desideratum manages to lose the original *intention* behind the code. There's usually somebody who indeed understands the code, but finding that person is often nontrivial.

Some friends and I decided to solve this. As our graduation capstone project, we built an analytics tool to determine *who knows what* in a codebase. It consisted of editor plugins to record what code was being read, by whom. We cross-referenced this against diffs, and found that high scores on both measures correlated strongly with the team members considered to be "experts" in those areas.

The software would infer dependencies in the codebase by analyzing which sections were viewed simultaneously, and emit warnings if they weren't logically related. Our tool was useful not only for static analysis, but also during debugging. By following experts through their previous jumps from a problematic part of the code, we found we could hone in on the issue more quickly than other non-experts.

Our results won us second-place in a project competition hosted by Yelp.

When you're new to a codebase, it's unclear if the quirky design choices are solutions to problems you don't yet understand, or if they're just unintentionally bad. In my experience, it's usually the latter. These concerns are unsuccessfully assuaged with comments like "well we know it's ugly, but we're too afraid to change it" or "it's clearly a ticking time-bomb, but we haven't gotten a chance to get around to it."

I try to live my life by leaving everything a little nicer than I found it, and so I decided I should start trying to clean up the dirty code that nobody else wanted to touch. It's thankless work --- customers don't see it, and management doesn't care, but it makes colleagues' lives better, and that seemed worthwhile.

As I fixed more and more technical debt, I started reading between the source lines. I saw the abstractions that the original authors were grasping for, without realising it. I'd write these abstractions in the course of my tidying, and inevitably, my senior colleagues would be using them by the end of the week.

From company to company, people seemed to all be missing the same things. They'd write little domain-specific languages to solve their problems, but which would immediately balloon into ad-hoc obscurity. They'd copy and paste code between files because it wasn't clear exactly how to generalise it.

Cleaning up bad code is fun for a while, but it gets old rather quickly. It's frustrating that today's mainstream tools are simply failing people who want to write better code. These people find often themselves between doing the *right* thing and doing a *quick* thing. Every incentive structure prioritizes the latter. I don't think our tools should punish people for attempting to write good code.

I became interested in free monads at my last company, where our code was an impressively tangled ball of spaghetti. It was untestable, and so people simply didn't write any tests. A stupid bug made it into production, and it cost us a few million dollars. Several people were subsequently let go, a few of whom were good friends of mine. All of this seemed so preventable that I started looking for a solution.

As I dug in, I realized that free monads were the abstraction that *I* had been grasping for without realizing. They allowed me to express my ideas at a very high-level. They let me separate my implementation details from my business logic, and as a bonus, I could turn those details into library code. It let me write code at a faster rate than my peers, and it was easy to prove correct.

Despite my best attempts at proselytizing free monads, my communities simply didn't engage. Try as I might, people pushed back against the idea. They correctly argued that free monads didn't deliver on their promise of static analysis; they correctly argued that free monads were too slow to be practical.

So long as people had valid complaints, they simply wouldn't listen to how free monads could help. I decided to do something about it.

Following the work of Foner et al.'s work on StrictCheck, I realized a similar approach of carefully passing bottoms around could be used to plumb through binds in free monads. The result was novel, and allowed many classes of free Kleisli morphisms to be inspected statically --- an idea previously considered impossible.

The performance problem however was more taxing. While Wu and Schrijvers have shown that efficient free monads are possible, their technique requires users to write a significant amount of boilerplate --- something I knew would be a stumbling block to wide adoption. If something is too much work to do write, nobody will do it. After several months of work, I came up with a solution that could give free implementations of the boilerplate, at the expense of making the compiler work harder. Unfortunately, the compiler wasn't up to the challenge, so I had to patch it. The code review is still in process, but my local branch allows for free monads to be a true zero-cost abstraction.

I really and truly believe that useable free monads will drastically improve code "out in the wild." They'll lessen the frequency of newcomers flailing around in codebases. Perhaps most importantly, they'll cut down on the frantic four-in-the-morning broken-code alarms that are all too prevalent in the industry.

Alleviating these pain points seems like the most important thing I can be working on, and I'm determined to do whatever I must in order to make that dream a reality.