production haskell

succeeding in industry with haskell

matt parsons

Production Haskell

Succeeding in Industry with Haskell

Matt Parsons

This book is for sale at http://leanpub.com/production-haskell

This version was published on 2023-02-01

Contents

CHANGELOG

I'm working hard on the book! This section lists the things I have changed with each release.

If you have any questions or comments, feel free to reach out to my email address `parsonsmatt@gmail.com` or comment on the Leanpub Discussion Forums[1].

Thanks so much for checking the book out!

v12-2022-11-26-RC1

- Edited and proofread the book
- Added a technique for removing a constructor from a sum type with error messages to Growing Pains

v11-2022-11-07

- Incorporated feedback from Jade Lovelace
- Work on `Logger` chapter
- Dropped the Mocking chapter. I don't approve of the practice generally and would prefer folks look at other techniques first, and other resources if they really want to mock.

v10-2022-10-01

- First draft on Family Values chapter

[1] https://community.leanpub.com/c/production-hask

v9-2022-09-04

Things with Manning didn't work out, so here's an update!

- Exceptions
  - Added information on asynchronous exceptions
  - Added information on `annotated-exception`
- Trouble with Typed Errors
  - Link to Plucky and resources for that experimental technique
- MPTCs vs TyFams
  - Providing some information on bidirectional dependencies
- Profiling
  - Removed chapter stub. This is well covered in other materials.
- Growing Pains
  - Filled out the `ErrorCall` example

v8-2021-10-29

- Add Family Values chapter
- The book has been picked up by Manning Publications. Please email me with a copy of your receipt and I will ensure that you get access to the finished book through Manning.

v7-2021-09-03

- Incorporate bunch of minor edits posted on the Leanpub forms. Thanks arpl and jakalx!
- Remove the stub for the database library comparison chapter. Going into full depth on that would require a lot of time and produce an almost-immediately dated article, so a blog post is a more suitable format for that content. I'd rather spend the effort improving and documenting `persistent` and `esqueleto`.
- Begin the Logging chapter

Page count: 440

v6-2021-02-01

- Work on the Databases chapter
- Split the library eval from Databases into it’s own chapter
- Add clarifications to ‘Haskell Teams’

page count: 426

v5-2021-01-01

- Add to Growing Pains chapter

page count: 393

v4-2020-12-01

- Finish the Project Preludes chapter
- Add another intermediate datatype example to the Testing chapter.
- Compare and contrast popular testing libraries
- Explain why single constructor exceptions are best
- Elaborate on EDSL’s ability to customize resource usage
- Talk about exception hierarchies
- Outlined the Database chapter
- Outlined the Logging chapter

Page count: 373

v3-2020-11-01

- Add to Project Prelude chapter
- Add HasCallStack section to Exceptions
- Complete Reinventing section in Exceptions

- Remove the phrase “cargo cult”
- Elaborate on the Constraint Trick in EDSL Design
- Elaborate on why PolyKinds are scary

354 pages

v2-2020-10-14

- Add Prairie chapter
- Start Project Prelude chapter
- Add Changelog to book frontmatter
- Add the weightlifting logging section in EDSL Design
- Add Growing Pains chapter

318 pages

v1-2020-10-07

Initial published version

Introduction

An Opinionated Tour Guide

So you've learned Haskell. You've taught your friends about monads, you've worked through some beginner textbooks, and maybe you've played around with some open source projects. Now that you've had a taste, you want more: you want to write an application in Haskell for fun! Maybe you want to use Haskell at work!

You sit down at your computer, and you're stuck.

> How does anyone actually get anything done with this language?

This is a common thing to wonder.

Haskell has always enjoyed a wide variety of high quality learning material for advanced parts of the language, if you're not afraid of academic papers. Many people have created fantastic resources for beginners in the last five years. However, the language does not have many resources for using it in production. The Haskell ecosystems can be difficult to navigate. There are many resources of varying quality with ambiguous goals and values. Identifying the right advice is nearly as challenging as finding it in the first place.

Haskell is a hugely diverse landscape. There are many regional groups: United Kingdom, Scandinavia, mainland Europe, Russia, the USA, Japan, China, and India all have thriving Haskell ecosystems that have interesting dialects and differences in custom and practice.

People come to Haskell with many backgrounds. Some people learned Haskell well into their careers, and had a long career writing Java, Scala, or C# beforehand. Some people came to Haskell from dynamically typed languages, like LISP or Ruby. Some people started learning Haskell early on in their programming career, and use it as the basis of comparison. Some people primarily use Haskell in academic research, while others primarily use Haskell in industrial applications. Some people are hobbyists and just like to write Haskell for fun!

This book is intended for people that want to write Haskell in industry. The trade-offs and constraints that industrial programmers face are different from academic or hobbyist programmers. This book will cover not only technical aspects of the Haskell language, but also social and engineering concerns that aren't "really" about Haskell.

Part of this book will be objective. I will teach you how to use some interesting techniques and ideas to make developing with Haskell more productive. We'll learn about Template Haskell, type-level programming, and other fun topics.

However, for the most part, this book is inherently subjective. Because Haskell serves so many ecosystems, it is imperative to discern what ecosystem a something is intended for. More than just giving out prescriptions - "This library is production ready! This is a toy!" - I hope to show my thought process and allow you to make your own judgment calls.

Ultimately, this is a book about the social reality of software engineering in a niche language.

After reading this book, you should feel comfortable:

- Writing large software projects in Haskell
- Evaluating competing libraries and techniques
- Productively reading material from a variety of Haskell users

About the Author

I'm Matt Parsons.

I started learning programming in January 2014 with Computer Science 101 at the University of Georgia. At the time, I was working for the IT department, installing Windows and troubleshooting printers. My manager disliked me and made it clear that he'd throw me under the bus at every opportunity. I was desperate for a new career, and I had a bunch of college credits from a failed attempt at a biochemistry degree. Computer science seemed like the best option to get out of that job.

CS101 taught me about basic Java programming. None of the local startups or programmers used or liked Java, so I asked what I should learn to get a job fast. JavaScript and Ruby were the top choices. I learned JavaScript that summer with the excellent book Eloquent JavaScript[2], which had chapters on functional programming and object oriented programming. I found the chapter on functional programming more intuitive, so I made a mental note to learn the most functional language I could find. A few months later, I started learning Haskell and Ruby on Rails.

I quit my IT job in December 2014, so I could be a full time student. By mid-January, I had a Rails internship with a local startup - so much for full time study.

My brain picked up Haskell quickly. I had barely started learning imperative and object-oriented programming, so the difficult novelty of learning new jargon and concepts was expected. The Ruby language was remarkably receptive to implementing Haskell ideas, though the community wasn't as excited. The concepts I learned in Haskell helped me write easily tested and reliable code in Ruby.

In August 2015, I started a Haskell internship, where I got to build web applications and fast parsers. I was allowed to use Haskell in my Artificial Intelligence coursework. In my last semester of college, I used Haskell in my undergraduate thesis to study the connection between category theory, modal logic, and distributed systems.

I am fortunate to have had these opportunities, as they set me up for success to work with Haskell. My first job out of college was converting PHP applications to greenfield Haskell, and I've been working full-time with Haskell ever since. I've worked in a variety of contexts: a startup that wasn't 100% sold on Haskell, a larger company that was sold on Haskell but wrestling with social and technical difficulties of a huge code base and development team, and a startup that was sold on Haskell and working on growing. I also contribute to many open source projects, and I'm familiar with most of the ecosystems. All told, I have worked with millions of lines of Haskell code.

[2] https://eloquent-javascript.net

I’ve seen Haskell fail. I’ve seen it succeed. I’d like to help you succeed with Haskell.

Principles

This section documents guiding principles for the book. I've found these core ideas to be important for managing successful Haskell projects.

- Complexity
- Novelty
- Cohesion
- Empathy

Complexity

Managing complexity is the most important and difficult task with Haskell projects.

This is so important that I am going to make this the first principle, and I'll even say it twice:

Managing complexity is the most important and difficult task with Haskell projects.

Just like you have "technical debt," you have a "complexity budget." You spend your complexity budget by using fancy technologies, and you spend your novelty budget by picking up new or interesting or different technologies. You can increase your budget by hiring expert engineers and consultants. Unlike technical debt, these budgets have a real and direct impact on your actual financial budget.

Complexity is a Fat Tail

It's easy to decry the evils of complexity, when you're just talking about complexity. But we don't pick up complexity on it's own. Codebases adopt small features, neat tricks, and safety features slowly. Over time, these accrete into highly complex systems that are difficult to understand. This happens even when each additional bit of complexity seems to pull it's own weight!

How does this happen?

A unit of code does not stand alone. It must relate to the code that uses it, as well as the code it calls. Unless carefully hidden, complexity introduced by a unit of code must be dealt with by all code that uses it. We must consider the relationships between the code units, as well as the units themselves. This is why two bits of complexity don't simply add together - they multiply! Unfortunately, the benefits of the complexity don't multiply - they're usually only additive.

A system that is difficult to understand is difficult to work with. Eventually, a system can become so difficult to understand that it becomes a black box, essentially impossible to work with. In this case, a ground up rewrite is often the most palatable option for the project. This often kills the project, if not the company. We must avoid this.

Complexity paints us into a corner. Safety features especially limit our options and reduce the flexibility of the system. After all, the entire point of "program safety" is to forbid invalid programs. When requirements change and the notion of an "invalid program" also changes, the safety features can become a hindrance. Complexity imposes a risk on every change to the codebase.

Predicting the cost or time required to modify to a complex system is difficult. The variance of these predictions grows with the complexity of the system. Tasks that seem simple might become extremely difficult, and it will be equally troublesome to provide estimates on the remaining time left to complete a task.

In measurement, we consider accuracy and precision to be separate concepts. A precise measurement or prediction is highly consistent - for a given Truth, it will report a similar Measurement consistently. An accurate measurement or prediction is close to the actual Truth. We can imagine predictions that are precise, but not accurate, as well as accurate, but not precise.

Complex systems make both the precision and accuracy of predictions worse. Precision is the more serious problem. Businesses rely on forecasting and regularity to make plans. If prediction becomes imprecise, then this makes the business more difficult to maintain.

A highly complex system, then, is more likely to fail catastrophically than a simple system. This is true even if the system is better in every other way! Imagine two cars - one gets 100 miles per gallon, can drive at 200mph, and corners like a dream. The other is much worse: only 40 miles per gallon and a top speed of 60mph. Naturally, there is a catch: the first car will break down relatively often and randomly, and it may take up to a week to repair it. The second car isn't perfect, but it reliably breaks down once a year, and it consistently takes a day to fix.

If you need a car to get to work, and can only have one car, then you want the second car. Sure, the first car can go faster and costs less, but the essential quality you need in a commuter is reliability.

Mitigating Complexity

We return to a common theme in this book: the variety of ecosystems. Can you imagine a group who would prefer the first car? Hobbyists! And professional race drivers, who can have spare cars! And engineers who study advanced automotive technology!

Haskell primarily serves academia as a research language for functional programming. Industrial use is a secondary concern. Many Haskellers are also hobbyists, primarily using it for fun. These are all valid uses of Haskell, but academic and hobbyist practitioners usually believe that their techniques are suitable for industry. Unfortunately, they often don't work as well as they might hope.

When asking about two cars in Haskell, you'll often hear people recommend the Fast Car. Try to learn more about the people in question. Have they actually driven the Fast Car? As a commuter? Are they responsible for fixing it when it breaks down?

People will recommend fancy and fantastic and wonderful solutions to your problems. Take them with a grain of salt. There are few codebases in Haskell where any technique has been exhaustively examined. Few of those examinations make it into the folklore.

The best way you can guarantee the success of your Haskell project is by managing the complexity and preferring the simplest possible solution.

Why is this hard?

Haskell selects for a certain kind of person.

Hobbyist and industrial programmers follow one path. If you don't enjoy novelty and difficulty, you will have a difficult time learning such a novel and complex language in the first place. Most Haskell developers learn Haskell on their own time, pursuing personal projects or intellectual growth. Haskell's learning materials, while greatly improved in recent time, are still difficult enough that only determined people with a great tolerance for novelty and frustration make it through.

Academic programmers tend to follow another path. Many of them learn Haskell in university classes, with a professor, teaching assistants, and other classmates to offer support. They pursue their research and studies to push the limits of programming languages and computer science. Much academic work is a proof-of-concept, rather than a hardened industrial implementation. The resulting work is often rather complex and fragile.

The Haskell programming language is also partly responsible for this. Strong types and functional programming can provide safeguards. Programmers often feel much more confident working with these safeguards. This confidence allows the developers to reach for greater and more complex solutions.

As a result, much of the ecosystem and community tend to be less averse to complexity and novelty. Hobbyists and academics are also driven by a different set of incentives than industrial programmers. Complexity and novelty accrete quickly in Haskell projects, unless aggressively controlled for.

Novelty

Novelty is the second danger in a Haskell project. It's nearly as dangerous as complexity, and indeed, the trouble with complexity is often the novelty that comes with it.

Unlike a complexity budget, which can be increased by spending money on expertise, your novelty budget is harder to increase. New techniques are usually difficult to hire for. They're difficult to learn and document.

If you have selected Haskell for your application language, then you have already spent much of your complexity and novelty budgets. You will probably need to write or maintain foundational libraries for your domain - so you'll need to be employing library grade engineers (or comfortable with contracting this out). You will need to develop an understanding of GHC - both the compiler and the runtime system. Expertise (and consulting) on these topics is more difficult to find than tuning the JVM or the CLR. Much of this understanding can be punted past the prototype stage - Haskell's library situation is Good Enough for many domains, and GHC's performance is Good Enough out-of-the-box that you can prototype and be fine.

Since Haskell is a big complexity/novelty budget item, it is important to stick with low-cost choices for the rest of the stack. Don't try out the fancy new graph database for your app - stick with Postgres. Especially don't try any fancy in-Haskell databases! Sticking with industry standards and common technology opens up a wider and more diverse field of engineers for hire.

Every requirement you place on your job ad for a developer increases the difficulty and cost of hiring. Haskell is a rare skill. Years of experience using Haskell in production with fancy libraries and techniques is rarer still. Haskell's productivity advantages are real, but these only apply while writing, reading, and understanding code. Documentation, requirements, and QA take just as much time as in other languages.

Cohesion

Two engineers are fighting about their personal preferences, again. You sigh and evaluate the arguments. Both solutions are fine. Sure, they have trade-offs, but so does everything.

Haskell engineers are unusually opinionated, even for software engineers. Haskell itself is strongly opinionated - purely functional programming is the only paradigm that the language directly supports. Software developers who want to learn Functional Programming and aren't too opinionated about it typically learn with JavaScript or a less extreme functional language like OCaml, F#, or Scala. If you successfully learn Haskell, then you are probably pretty opinionated about how to do it!

The diversity in Haskell's ecosystem gives rise to many different practices and conventions. The Haskell compiler GHC itself has many different formatting styles and concepts, and many of these are specific to that project. I have noticed differences in the style that correspond strongly with cultural centers - the United States east and west coasts differ, as do the styles of the Netherlands vs Scotland vs Sweden.

Vanilla Haskell is flexible enough. GHC Haskell, the de facto standard implementation, permits a massive variety of semantic and syntactic variations through language extensions. `MultiWayIf`, `LamdbaCase`, and `BlockArguments` provide syntactic changes to the language. The extensions `MultiParamTypeClasses` + `FunctionalDependencies` can be used to do type-level programming in a way that is mostly equivalent to `TypeFamilies`, and which to use is often a matter of personal preference. Many problems are just as easy to solve with either `TemplateHaskell` or `Generic` deriving, but the real trade-offs are often ignored for personal preferences.

Meanwhile, the multiple ecosystems all contribute competing ideas for how to do anything. There are often many competing libraries for basic utilities, each offering a slightly different approach. People develop strong opinions about these utilities, often disproportionate to the actual trade-offs involved. I am certainly guilty of this!

A lack of cohesion can harm the productivity of a project. Successful projects should devote some effort towards maintaining cohesion. Promoting cohesion is a special case of avoiding novelty - you pick one way to do things, and then resist the urge to introduce further novelty with another way to solve the problem.

Cohesive Style

Haskell's syntax is flexible to the extreme. Significant white space allows for beautifully elegant code, as well as difficult parsing rules. Vertical alignment becomes an art form, and the structure of text can suggest the structure of the underlying computation. Code is no longer merely read, but laid out like a poem. Unfortunately, this beauty can often interfere with the maintenance and understanding of code.

Projects should adopt a style guide, and they should use automated tooling to help conformance. There are many tools that can help with this, but the variety of Haskell syntax makes it difficult to settle on a complete solution. Exploring the trade-offs of any given coding style is out of scope for this chapter, but a consistent one is important for productivity.

Cohesive Effects

Haskellers have put a tremendous amount of thought and effort into the concept of 'effects'. Every other language builds their effect system into the language itself, and it's usually just imperative programming with unlimited mutation, exceptions, and some implicit global context. In Haskell, we have a single 'default' effect system - the `IO` type. Writing directly in `IO` makes us feel bad, because it's less convenient than many imperative programming languages, so we invent augmentations that feel good. All of these augmentations have trade-offs.

If you use an exotic effect system in your application, you should use it consistently. You should be prepared to train and teach new hires on how to use it, how to debug it, and how to modify it when necessary. If you use a standard effect system, then you should resist attempts to include novel effect systems.

Cohesive Libraries

There are over a dozen logging libraries on Hackage. Non-logging libraries (such as database libraries or web libraries) often rely on a single logging library, rather than abstracting that responsibility to the application. As a result, it's easy to collect several logging libraries in your application. You will want to standardize on a single logging library, and then write adapters for the other libraries as-needed.

This situation plays out with other domains in Haskell. There are many possible situations, and some underlying libraries force you to deal with multiples. The path of least resistance just uses whatever the underlying library does. You should resist this, and instead focus on keeping to a single solution.

Cohesive Teams

If you hire two developers that have conflicting opinions, and neither are willing to back down, then you will experience strife in your project. Haskellers are particularly ornery about this, in my experience. It is therefore important to broadcast your team's standards in job ads. While interviewing new hires, you should be checking to see how opinionated they are, and whether they share your opinions.

Fortunately, none of the Strong Opinions that a Haskeller might have are along racial, gender, sexuality, or religious lines. Focusing on developing strong team cohesion is in alignment with hiring a diverse group of people.

Empathy

The final principle of this book is empathy.

Software developers are somewhat famous for getting involved in big ego contests. Consider the Python vs Ruby flame wars, or how everyone hates JavaScript. Talking down about PHP is commonplace and accepted, and I know I am certainly guilty of it. When we are limited to our own perspective, understanding why other people make different choices can be challenging.

Reality is more complex. PHP offers a short learning curve to make productive websites in an important niche. Ruby solves real problems that real programmers have. Python solves other real problems that real programmers have. JavaScript evolved well beyond it's original niche, and JavaScript developers have been working hard to solve their problems nicely.

In order to communicate effectively, we must first understand our audience. In order to listen effectively, we must first understand the speaker. This is a two-way street, and it takes real effort on all sides for good communication to occur.

When we are reading or evaluating software, let's first try to understand where it came from and what problems it solves. Then let's understand the constraints that led to the choices that happened, and not form unnecessarily broad negative evaluations. Haskell is used by a particularly broad group of people among several ecosystems, but also has a relatively small total number of people working on things at any one time. It's easy to misunderstand and cause harm, so we have to put focused effort to avoid doing this.

Empathy: For Yourself

Haskell is hard to learn. There aren't many resources on using Haskell successfully in industry. I have made plenty of mistakes, and you will too. It's important for me to have empathy for myself. I know that I am doing my best to produce, teach, and help, even when I make mistakes. When I make those mistakes - even mistakes that cause harm - I try to recognize the harm I have caused. I forgive myself, and then I learn what I can to avoid making those mistakes in the future, without obsessive judgment.

You, too, will make mistakes and cause harm while exploring this new world. It's okay. To err is human! And to focus on our errors causes more suffering and blocks healing. Forgive yourself for your difficulties. Understand those difficulties. Learn from them and overcome them!

Empathy: For your past self

Your past self was excited about a new technique and couldn't wait to use it. They felt so clever and satisfied with the solution! Let's remember their happiness, and forgive them the mess they have left us. Keep in mind the feeling of frustration and fear, and forgive yourself for encountering them. These feelings are normal, and a sign of growth and care.

Perhaps they missed something about the problem domain that seems utterly obvious to you now. They were doing the best they could with what they had at the time. After all, it took humanity nearly 9,000 years to invent calculus, which we can reliably teach to teenage children. Your frustration and disbelief is the fuel you need to help grow and be more empathetic to your future self.

Empathy: For your future self

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be

> when you write it, how will you ever debug it?
>
> - Brian Kernighan, The Elements of Programming Style, 2nd edition, chapter 2

Your future self is tired, bored, and doesn't have the full context of this line of code in mind. Write code that works for them. Write documentation that seems obvious and boring. Imagine that you've forgotten everything you know, and you need to relearn it - what would you write down?

Software is difficult, and you can't always put 100% of your brain and energy into everything. Write something that is easier to understand than you think is necessary, even if you think it has some fatal flaw.

Empathy: For your teammates

Healthy self-empathy is a prerequisite for healthy empathy for other people.

As difficult as empathizing with yourself is, empathizing with others is more difficult. You know your internal state and feeling. You possibly even remember your past states. And you may even be able to predict (with varying reliability) how you will react to something.

All of these intuitions are much weaker for other people. We must apply our practice of understanding and forgiveness to our teammates. They're working with us, and they're trying their best.

Empathy: For your audience

I apologize in advance for any harm this book might cause. I hope that the audience of my book might use it for success and happiness in their career and business projects. I also recognize that my advice will - inevitably - be miscommunicated, or misapplied, and result in harm and suffering.

Likewise, when you are writing code, you will need to consider your audience. You will be part of your audience, so the lessons you've learned about Past-You and Future-You will be helpful. If you're writing code for an application, then consider all the people that might read it. You'll want to consider the needs of beginners, newcomers, experienced developers, and experts.

Doing this fully is impossible, so you will need to consider the trade-offs carefully. If you anticipate that your audience is mostly new to Haskell, then write simply and clearly. If you require advanced power and fancy techniques, make a note of it, and write examples and documentation to demonstrate what is going on. There is nothing wrong with a warning sign or note to indicate something may be difficult!

Empathy: For the Business Folks

This one is especially hard. They'll never read your code. And they often change the requirements willy-nilly without a care for the abstractions you've developed. But - after all - we are here to write code to enable the business to be profitable.

We can have empathy for their needs by leaving our code open to their whims. A project should be able to evolve and change without needing to be reborn.

If the business fails, then we're all out of a job. If enough Haskell projects fail, then we won't have enough Haskell jobs for everyone that wants one. And, most concerning, if *too many* Haskell projects fail, then Haskell won't be a viable choice in industry.

I believe that all Haskellers in industry have a responsibility to their community to help their projects succeed. This book is a result of that belief.

References

- You Need a Novelty Budget[3]
- You have a complexity budget[4]

[3] https://www.shimweasel.com/2018/08/25/novelty-budgets
[4] https://medium.com/@girifox/you-have-a-complexity-budget-spend-it-wisely-74ba9dfc7512

I Building Haskell Teams

1. Selling Haskell

You want to use Haskell for work. Your boss is skeptical - isn't Haskell some obscure, fancy, academic programming language? Doesn't it have horrible build tooling and crappy IDEs? Isn't it super hard to learn and use?

Haskell skeptics have many mean things to say about Haskell. If your boss is a Haskell skeptic, you're probably not going to get to use Haskell in your current job. If your boss is more receptive to trying Haskell, and you've been given the task of evaluating Haskell's suitability for a task, then you will be *selling* Haskell. In order to effectively sell Haskell, we must drop our programmer mindset and adopt the business mindset.

The savvy business person is going to do what has the best profit, and will weigh long term benefits against short term costs. Haskell truly can improve the bottom line for a business, and if you are selling Haskell, you need to know how to argue for it.

1.1 Assessing Receptiveness

Is your company a Ruby shop? Do your coworkers hate static types, love monkeypatching, and don't mind occasional production crashes from `nil` errors? If so, you probably won't have a good time selling them on Haskell. In order to be sold on Haskell, it's good if the team shares the same values that the Haskell language embodies.

On the other hand, a shop like this has the *most* to gain from Haskell. Is there a piece of core infrastructure that is slow and buggy which significantly drags on profit? If so, you may be able to rewrite that bit of infrastructure and provide massive benefit to the company. My predecessors at a previous job convinced management to use Haskell for a rewrite instead of another attempt in PHP, and I was hired to do this. The Haskell version of the service required 1/10th the cloud resources to run and removed a bottleneck that allowed us to charge our larger customers more money.

If your company already employs developers that are familiar with statically typed functional languages, like Scala or F#, then you will have an easier time selling them on Haskell. Presumably they already value Haskell's strengths, which is why they went with a functional language. However, there may not be enough to gain from writing a service in Haskell - after all, isn't Scala most of the way there? The developers may feel that the additional friction of adding another language to the stack would bring minimal benefit since it's so close. In this case, you will need to sell them on other benefits that Haskell has.

1.2 Software Productivity

We want to claim that Haskell will increase software developer productivity. This will result in reduced developer costs and increased profit from new features. However, we need to understand developer productivity on a somewhat nuanced level in order to adequately sell this.

How do we measure developer productivity? It's not simple. Many studies exist, and all of them are bad. Regardless of your position on a certain practice (dynamic vs static types, pair programming, formal verification, waterfall, agile, etc), you will be able to find a study that supports what you think. We simply don't know how to effectively and accurately use the scientific method to measure developer productivity.

How do we claim that Haskell improves it, then? We can only use our experiences - anecdotal evidence. Likewise, our arguments can - at best - convince people to be receptive sharing our experience. The experiential nature of developer productivity means that we have to cultivate an open mind in the engineers we wish to convince, and then we must guide them to having the same experiences.

We'll learn a bit about this on the chapter "Learning and Teaching Haskell".

1.3 Statistics of Productivity

Management cares about productivity, but they don't just care about how fast you can bang out a feature. They care about how well you can predict how long a feature will take. They care about how big the difference in productivity among team members will be. Variance matters to management.

Statistics gives us tools for thinking about aggregations of data. The average, or mean, is calculated by summing up all the entries and dividing by the count. It's the most common statistical measure, but it can also be terribly misleading. The average income in the United States is $72,000, and you might hear that and think that most people make around that number. In fact, most people make below that amount.

A different measure, the median, is more appropriate. The median is the midpoint of the distribution, which means that half of all values are above the median and half of all values are below the median. The median household income is $61,000. The mean is much higher than the median, which means that there are a small number of people that make a huge amount of money. In order to know whether a mean or median is more appropriate for your purpose, you need to know the distribution of your data.

Your software team might have impressive average developer productivity. This might be because you have a bunch of above-average developers. It can also be because you have one extremely productive developer and a bunch of below-average developers. A team that is heavily lopsided is a *risk* for the company, because the loss of a single developer might have drastic consequences. Management will prefer a high median developer productivity over average for this reason.

However, what management *really* wants is low variance between developers. Variance is the average of the squared difference from the average. The difference is squared so that negative and positive differences are accounted for equally. A high variance team will have some developers significantly above and below the median. This is risky, because the exact assignment of developers can dramatically change how quickly and effectively software is developed. A low variance team will have most of the developers relatively close to each other in skill. This reduces risk, because a single developer can take a vacation and not significantly alter the team's median skill.

Larger companies tend to optimize for reducing variance in individual productivity. They can afford to hire tons of software engineers, and they want their staff to be replaceable and interchangeable. This isn't solely to dehumanize and devalue the employees - it is easier to take a vacation or maternity leave if you are not the only person who is capable of doing your job. However, it does usually result in reduced productivity for the highest performers.

It's a valid choice to design for low variance. Indeed, it makes a lot of sense for businesses. Elm and Go are two new programming languages that emphasize simplicity and being easy to learn. They sacrifice abstraction and expressiveness to reduce the variance of developing in the language. This means that an Elm expert won't be that much more productive than an Elm beginner. Elm or Go programmers can learn the language and be as productive as they'll ever be relatively quickly. Management loves this, because it's fast to onboard new developers, you don't have to hire the best-and-brightest, and you can reliably get stuff done.

Haskell is not a low variance language. Haskell is an extremely high variance language. It takes a relatively long time to become minimally proficient in, and the sky is the limit when it comes to skill. Haskell is actively used in academia to push the limits of software engineering and discover new techniques. Experts from industry and academia alike are working hard to add new features to Haskell. The difference in productivity between someone who has been using it for years and someone who has studied it for six months is huge.

Keeping variance in mind is crucial. If you work with Haskell, you will already be betting on the high end of the variance curve. If you select advanced or fancy Haskell libraries and features, then you will be increasing the variance. The more difficult your codebase is to jump into, the less your coworkers will enjoy it, and the more skeptical they will be of Haskell at all. For this reason, it's important to prefer the simplest Haskell stuff you can get away with.

1.4 Know Your Competition

Competition exists in any sales and marketing problem. Your competition in selling Haskell will be fierce - several other programming languages will also have compelling advantages. Haskell must overcome the other language's benefits with the technical gains to be made.

In some domains, like compiler design or web programming, Haskell has sufficient libraries and community that you can be productive quickly. The language is well situated to provide a compelling advantage. Other domains aren't so lucky, and the library situation will be much better in another language than yours.

If your Haskell project fails for whatever reason, the project will be rewritten in some other language. You probably won't get a chance to "try again." Businesses are usually unwilling to place bets outside of their core competency, and the programming language choice is probably not that core competency. So you'll want to be careful to situate Haskell as a safe, winning choice, with significant advantages against the competition.

2. Learning and Teaching Haskell

If you want your Haskell project to be successful, you will need to mentor and teach new Haskellers. It's possible to skate by and only hire experienced engineers for a while, but eventually, you'll want to take on juniors. In addition to training and growing the Haskell community, you'll be gaining vital new perspectives and experiences that will help make your codebase more resilient.

2.1 The Philology of Haskell

Learning Haskell is similar to learning any other language. The big difference is that most experiences of "learning a programming language" are professional engineers learning yet another language, often in a similar language family. If you know Java, and then go on to learn C#, the experience will be smooth - you can practically learn C# by noting how it differs from Java. Going from Java to Ruby is a bigger leap, but they're both imperative programming languages with a lot of built-in support for object oriented programming.

Let's learn a little bit about programming language history. This is useful to understand, because it highlights how different Haskell really is from any other language.

In the beginning, there was machine language - assembly. This was error prone and difficult to write, so Grace Hopper invented the first compiler, allowing programmers to write higher level languages. The 1950s and 1960s gave us many foundational programming languages: ALGOL (1958) and FORTRAN (1957) were early imperative programming languages. LISP (1958) was designed to study AI and is often recognized as the first functional programming language. Simula (1962) was the first object oriented programming language, directly inspired by ALGOL.

Smalltalk (1972) sought to reimagine Object Oriented programming from the ground up. The C programming language (1972) was invented to help with the UNIX operating system. Meanwhile, Standard ML (1973) introduced modern functional programming as we know it today. Prolog (1972) and SQL (1974) were also invented in this time period. For the most part, these languages define the language families that are in common use today.

C++ took lessons from Simula and Smalltalk to augment C with object oriented programming behavior. Java added a garbage collector to C++. Ruby, JavaScript, Python, PHP, Perl, etc are all in this language family - imperative programming languages with some flavor of object oriented support. In fact, almost all common languages today are in this family!

Meanwhile, Standard ML continued to evolve, and programming language theorists wanted to study functional programming in more detail. The programming language Miranda (1985) was a step in this direction - it features lazy evaluation and a strong type system. The Haskell Committee was formed to create a language to unify research in lazy functional programming. Finally, the first version of the Haskell programming language was released in 1990.

Haskell was used primarily as a vessel for researching functional programming technologies. Lots of people got their PhDs by extending Haskell or GHC with some new feature or technique. Haskell was not a practical choice for industrial applications until the mid-2000s. The book “Real World Haskell” by Don Stewart, Bryan O’Sullivan, and John Goerzen demonstrated that it was finally possible to use Haskell to solve industrial grade problems. The GHC runtime was fast and had excellent threading support.

As of this writing, it is 2022. Haskell is world-class in a number of areas. Haskell - beyond anything else - is radically different from other programming languages. The language did not evolve with industry. Academia was responsible for the research, design, and evolution. Almost 30 years of parallel evolution took place to differentiate Haskell from Java.

2.2 Programming Is Hard To Learn

If you are a seasoned engineer with ten years of experience under your belt, you’ve probably picked up a bunch of languages. Maybe you learned

Go, Rust, or Swift recently, and you didn't find it difficult. Then you try to learn Haskell and suddenly you face a difficulty you haven't felt in a long time. That difficulty is the challenge of a new paradigm.

Most professional programmers start off learning imperative programming, and then pick up object oriented programming later. Most of their code is solidly imperative, with some OOP trappings. There's nothing wrong with this - this style of code genuinely does work and solves real business problems, regardless of how well complex programs become as they grow. Many programmers have forgotten how difficult learning imperative programming is, or to think like a machine at all.

I want to focus on the principle of Empathy for this section. Programming is hard to learn. Trivially, this means that functional programming is hard to learn.

The experience of struggling to learn something new can often bring up uncomfortable feelings. Frustration, sadness, and anger are all common reactions to this difficulty. However, we don't need to indulge in those emotions. Try to reframe the experience as a positive one: you're learning! Just as soreness after exercise is a sign that you're getting stronger, mild frustration while learning is a sign that you're expanding your perspective. Noticing this with a positive framing will help you learn more quickly and pleasantly.

2.3 Pick Learning Materials

I'm partial to Haskell Programming from First Principles[1]. Chris Allen and Julie Moronuki worked hard to ensure the book was accessible and tested the material against fresh students. I've used it personally to help many people learn Haskell. I used the book as the cornerstone of the Haskell curriculum and training program at Mercury, where we train folks to be productive Haskell developers in 2-8 weeks.

[1] https://haskellbook.com/

2.4 Write Lots of Code

While learning and teaching Haskell, enabling a fast feedback loop is important. Minimizing distractions is also important. For this reason, I recommend using minimal tools - a simple text editor with syntax highlighting is sufficient. Fancy IDEs and plugins often impede the learning process. Time spent setting up your editor situation is time that you *aren't* spending actually learning Haskell.

Students should get familiar with `ghci` to evaluate expressions and `:reload` their work to get fast feedback. The tool `ghcid`[2] can be used to automate this process by watching relevant files and reloading whenever one is written.

When we're learning Haskell, the big challenge is to develop a mental model of how GHC works. I recommend developing a "predictive model" of GHC. First, make a single change to the code. Before saving, predict what you think will happen. Then, save the file, and see what happens.

If you are surprised by the result, that's good! You are getting a chance to refine your model. Develop a hypothesis on why your prediction didn't come true. Then test that prediction.

Compiler errors tell us that something is wrong. We should then try to develop a hypothesis on what went wrong. Why does my code have this error? What did I expect it to do? How does my mental model of Haskell differ from GHC's understanding?

Programming is tacit knowledge. It isn't enough to read about it. Reading informs the analytical and verbal parts of our brain. But programming taps into much more, which is only really trained by doing. We must write code - a lot of it - and we have to make a whole bunch of mistakes along the way!

2.5 Don't Fear the GHC

Many students pick up an aversion to error messages. They feel judgment and condemnation from them - "Alas, I am not smart enough to

[2] https://hackage.haskell.org/package/ghcid

Get It Right!" In other programming languages, compiler errors are often unhelpful, despite dumping a screen's worth of output. As a result, they often skip reading error messages entirely. Training students to *actually read* the compiler errors from GHC helps learning Haskell significantly. GHC's error messages are often more helpful than other languages, even if they can be difficult to read at first.

We want to rehabilitate people's relationships with their compilers. Error messages are gifts that the compiler gives you. They're one side of a conversation you are having with a computer to achieve a common goal. Sometimes they're not helpful, especially when we haven't learned to read between the lines.

An error message does not mean that you aren't smart enough. The message is GHC's way of saying "I can't understand what you've written." GHC isn't some perfect entity capable of understanding any reasonable idea - indeed, many excellent ideas are forbidden by Haskell's type system! An error message can be seen as a question from GHC, an attempt to gain clarity, to figure out what you really meant.

2.6 Start Simple

When writing code for a beginner, I try to stay as absolutely simple as possible. "Simple" is a vague concept, but to try and be more precise, I mean something like prefering the smallest transitive closure of concepts. Or, "ideas with few dependencies."

This means writing a lot of `case` expressions and explicit lambdas. These two features are the foundational building blocks of Haskell. Students can simplify these expressions later, as a learning exercise, but we shouldn't focus on that - instead, focus on solving actual problems! Additional language structures can be introduced as time goes on as the student demonstrates comfort and capability with the basics.

As an example, let's say we're trying to rewrite `map` on lists:

```haskell
map :: (a -> b) -> [a] -> [b]
map function list = ???
```

I would recommend the student begin by introducing a `case` expression.

```
map function list =
    case ??? of
        patterns ->
            ???
```

What can we plug in for those ??? and patterns? Well, we have a list variable. Let's plug that in:

```
map function list =
    case list of
        patterns
```

What are the patterns for a list? We can view the documentation and see that there are two constructors we can pattern match on:

```
map function list =
    case list of
        [] ->
            ???
        (head : tail) ->
            ???
```

The use of a case expression has broken our problem down into two smaller problems. What can we return if we have an empty list? We have [] as a possible value. If we wanted to make a non-empty list, we'd need to get our hands on a b value, but we don't have any, so we can't plug that in.

For the non-empty list case, we have head :: a and tail :: [a]. We know we can apply function to head to get a b. When we're looking at our "toolbox", the only way we can get a [b] is by calling map function.

```haskell
map function list =
    case list of
        [] ->
            []
        (head : tail) ->
            function head : map function tail
```

We want to start with a relatively small toolbox of concepts. Functions, data types, and case expressions will serve us well for a long time. Many problems are easily solved with these basic building blocks, and developing a strong fundamental sense for their power is important for a beginning Haskell programmer.

This ties into our Novelty and Complexity principles. We want to add concepts slowly to avoid overwhelming ourselves. As we introduce concepts, we have to consider not just the concept itself, but also how that concept interacts with every other concept we know. This can easily become too much!

2.7 Solve Real Problems

This takes some time to get worked up to, but a beginner can learn how to use the `IO` type well enough to write basic utilities without understanding all of the vagaries of monads. After all, consider this example program in Java, Ruby, and finally Haskell:

Java: `java public class Greeter { public static void main(string[] args) { Scanner in = new Scanner(System.in); String name = in.nextLine(); System.out.println("Hello, " + name); } }`

Ruby:

```ruby
name = gets
puts "Hello" + name
```

Haskell:

```haskell
main = do
    name <- getLine
    putStrLn ("Hello, " ++ name)
```

The Java code contains a ton of features. They don't need to be explained. In my Java 101 courses at university, we were told to just "copy and paste" it and an explanation would come later. That worked okay for me. After all, computers are often perceived as black boxes of mysterious magical power: treating programming languages the same feels natural and normal.

A beginner can work through the common Haskell education of defining `Functor`, `Monad`, `Monoid`, etc. instances for common types while also developing basic utilities and examples.

2.8 Pair Programming

Pair programming can be a great way to show the tacit nature of programming in Haskell. The driver may also use pairing as an opportunity to show off and feed their own ego, which harms the beginner. The teacher must take great care to have Empathy for the learner.

The driver will want to slow down and explain their thought process. I find it helpful to separate verbal explanations into "Observing Reality," "Noticing Feelings," and "Discussing Strategies." This technique comes from Nonviolent Communication. I will also verbally explain my predictive model. For example, when solving a problem, I may ordinarily jump a few predictions and make a bigger change when programming solo. When pairing, I will instead state my prediction, make the modification, and talk through the result.

The student will want to pay attention and ask questions. Don't be afraid to interrupt - the purpose of the exercise is primarily to transfer knowledge and practice from the driver. However, a big part of the benefit is in causing the driver to think clearly about what they are doing! The driver should get as much out of a good question as the student.

Unfortunately, "pair programming" in this manner is a tacit exercise, just like programming itself. I can describe my strategies and techniques for a successful session, but the best way to learn is by observing and participating. Let's walk through a hypothetical example.

2.9 A Dialogue

(Things that I might think are in parentheses. I won't actually say them, because it's important to not distract the student with extraneous digressions.)

Student: Hey, do you mind if we pair on something?

Matt: Sure! I'd be happy to.

S: My task is to take our user list and figure out how many email accounts belong to each hosting service.

M: Okay, cool. So basically count up how many `@gmail.com` and `@yahoo.com` etc there are?

S: Yeah. I'm not sure how to get started! I know I can do it in SQL, but I'd rather learn how to make it work in Haskell.

M: Sure! Okay, so first I'm going to check out our database types. I want to check my assumptions on how our data is structured, since that is the source of our information. I'll navigate to the file that contains our definition. Here's the type:

```haskell
data User = User
    { userId :: UserId
    , userName :: Text
    , userEmail :: EmailAddress
    , userIsAdmin :: Bool
    }
```

S: So we want to take each `User` and inspect the `EmailAddress`. What does that type look like?

M: Great question! If I search the file for `EmailAddress`, I don't get anything. So I'll search the file for `Email`, since it's a closer match. This takes me up to the import list, where I see that we're importing a module named `Text.Email.Validate`.

S: Where does that come from?

M: I'm not sure. I don't see that module listed in our project, which means it is in a dependency. So now I'm going to switch to my browser and search `stackage.org` for `EmailAddress`. There are a few results here, and the top one is a package named `email-validate` in a module `Text.Email.Parser`. Since it shares the same `Text.Email.*` structure, and it has validation, I'm going to guess that's it.

S: Yeah, I don't think it's coming from the crypto library, and we don't use `pushbullet`, so the `pushbullet-types` one probably isn't it.

M: Good observation!

S: OK! I think I know what comes next. The module exports a function `domainPart :: EmailAddress -> ByteString`. So we can use that to get the domain for an email!

M: That's my guess too. Now that we have our primitive types understood, let's write out the signature. What's your guess for an initial signature?

S: I think I'd start here:

```haskell
solution :: Database [(ByteString, Int)]
```

The `ByteStrings` are the `domainPart`, and the `Int` is our count.

M: That sounds good. I think I would go for something different, but let's explore this first.

S: Why?

M: Well, whenever I see a `[(a, b)]`, I immediately think of a `Map a b`. But we can keep it simple and just try to solve this problem. If it becomes too annoying, then we'll look for an alternative solution.

S: Okay, that works for me! I don't really see how a `Map` works for us right now - we're not doing lookups. So the first thing I want to do is get all the users.

```
solution = do
    users <- selectAllUsers
    ???
```

Once I have the users, I want to get the email addresses.

```
solution = do
    users <- selectAllUsers
    let emails = map userEmail users
```

M: Nice use of `map` there!

S: Next, I want to get the domain parts out.

```
solution = do
    users <- selectAllUsers
    let emails = map userEmail users
    let domains = map domainPart emails
```

And, uh, I think I am stuck right here. I want to group the list by the domains. But I don't know how to do that.

(Gonna resist the urge to make the code more concise! Just because I can write that as `map (domainPart . userEmail) <$> selectAllUsers` doesn't mean that it's important to do now.)

M: Okay! We have a `[ByteString]` right now. What might our grouping look like?

S: I guess a `[[ByteString]]`?

(Well, a `[NonEmpty ByteString]` is more precise, but we can get there later.)

M: Sounds good to me. Well, we have a few avenues available - when I'm not sure about some functionality, I'll either look at the relevant modules or search Hoogle for the type signature. If I'm not sure about the relevant modules, then I'll just go straight to Hoogle. So let's search for `[ByteString] -> [[ByteString]]`.

S: None of these are relevant! `text-ldap` isn't close. `subsequences`, `inits`, `permutations`, `tails`, none of these have anything to do with grouping.

M: Hmm. Yeah. Hoogle fails us here. What if we search `group`?

S: Oh, then we get back `group :: Eq a => [a] -> [[a]]`. That's exactly what we want!

M: Let's read the docs, just to be sure. Does anything jump out as a potential problem?

S: Yeah - the example given is a bit weird.

```
>>> group "Mississippi"
["M","i","ss","i","ss","i","pp","i"]
```

I would expect it to group all the equal elements together, but `s` appears twice. I think I can work around this!

(Hmmm, where is the student going? Sorting the list?)

S: We can call `group`, and then get the size of the lists.

```
func :: [ByteString] -> [(ByteString, Int)]
func domains =
    map (\grp -> (head grp, length grp)) (group domains)
```

Okay okay okay so this is the right shape, BUT, we have to use it in a special way!

M: How do we do that?

S: Okay so suppose we're looking for `gmail.com`. We'd filter the result list for `gmail.com` and then sum the `Int`s!

```haskell
domainCount :: ByteString -> [(ByteString, Int)] -> Int
domainCount domain withCounts =
    foldr (\(_, c) acc -> c + acc) 0 $
    filter (\(name, _) -> domain == name) withCounts
```

(Resist the urge to suggest a `sum . map snd` refactor!)

M: Nice! That works for the case when we know the domain. But if we just want a summary structure, how can we modify our code to make that work?

S: Hmm. We could map through the list and for each name, calculate `domainCount`, but that is inefficient…

M: That works! But you're right, that's inefficient. I think we can definitely do better. What comes to mind as the problem?

S: Well, there are multiple groups for each domain, potentially. If there were only a single group for each domain, then this would be easy.

M: How might we accomplish that?

S: Well, we start with a `[ByteString]`. Oh! Oh. We can `sort` it, can't we? Then all the domains would be sorted, next to each other, and so the `group` function would work!

M: Yeah! Let's try it.

S: HERE WE GO

```haskell
func domains =
    map (\grp -> (head grp, length grp)) $
    group $
    sort domains
```

(must resist the urge to talk about `head` being unsafe…)

M: Well done! Now how do we do a lookup of, say, `gmail.com`?

S: `List.lookup "gmail.com" (func domains)`.

M: Ah, but there's `lookup` - doesn't that suggest a `Map` to you?

S: Eh, sure!

```haskell
Map.lookup "gmail.com"
    $ Map.fromList
    $ map (\grp (head grp, length grp))
    $ group $ sort domains
```

But that doesn't really seem any better? I guess we have a more efficient lookup, but I think we're doing extra work to construct a `Map`.

M: We are, but much of that work is unnecessary. Let's look at the `Data.Map` module documentation for constructing `Map`s. Instead of doing all the work with lists, let's try constructing a `Map` instead.

S: Huh. I'm going to start with `foldr` since that's how you deconstruct a list.

```haskell
func domains =
    foldr (\x acc -> ???) Map.empty domains
```

M: Off to a great start! Just as a refresher, what is `acc` and `x` here?

S: `acc` is a `Map` and `x` is a `ByteString`.

M: Right. But what is it a `Map` of? Remember, we had a `[(ByteString, Int)]`.

S: Oh, `Map ByteString Int`.

M: Right. So what do we want to do with the `ByteString`?

S: Insert it into the `Map`? Hm, but what should the value be?

M: We're keeping track of the count. This suggests that we may want to update the `Map` instead of inserting if we have a duplicate key match.

S: Ah! Okay. Check this out:

```haskell
func domains =
    foldr (\domain acc ->
        case Map.lookup domain acc of
            Nothing ->
                Map.insert domain 1 acc
            Just previousCount ->
                Map.insert domain (previousCount + 1) acc
    ) Map.empty domains
```

M: Well done! We can do even better, though. Let's take a look at inserting in the documentation. Does anything here seem promising?

S: Hmm. `insertWith` might do it. Let me try:

```haskell
func domains =
    foldr (\domain acc ->
        Map.insertWith
            (\newValue oldCount -> newValue + oldCount)
            domain
            1
            acc
    ) Map.empty domains
```

M: Beautiful. This is efficient and perfectly satisfies our needs. And you got to learn about `Map`s!

Teaching Haskell is about *showing* how to *do* the action, as much as *telling* how to *understand* the concepts.

2.10 References

- History of programming languages[3]

[3] https://en.wikipedia.org/wiki/History_of_programming_languages

- Generational list of programming languages[4]
- Tacit Knowledge[5]

[4] https://en.wikipedia.org/wiki/Generational_list_of_programming_languages

[5] https://commoncog.com/blog/tacit-knowledge-is-a-real-thing/

3. Hiring Haskellers

3.1 The Double-edged Sword

Haskell is a double-edged sword with hiring. This is a consistent experience of every hiring manager I have talked to with Haskell, as well as my own experiences in looking at resumes and interviewing candidates. An open Haskell role will get a fantastic ratio of highly qualified candidates. Among them will be PhDs, experienced Haskellers, senior developers in other languages, and some excited juniors demonstrating tremendous promise. The position is possibly "beneath" the people applying, but Haskell is enough of a benefit that they're still happy.

While quality will be high, quantity will be disappointing. A Java posting may attract 1,000 applications, of which 25 are great. A Haskell posting may attract 50 applications, of which 10 are great. This is a real problem if you need to hire a large team. Haskell's productivity bonuses reduce the need for a large team, but you can only put that off for so long.

You can grow a Haskell team solely by training newcomers into the language. This requires at least one Haskell-experienced engineer with a penchant for mentorship and the restraint to keep the codebase easy to get started with. That's a tall order for the same reason that Complexity and Novelty are especially difficult problems in Haskell. If you are reading this before starting your Haskell team, then I implore you - write code that you can train a junior on without too much stress. If you already have a complex codebase, then you probably need to hire a senior.

3.2 Juniors and Seniors

This chapter will use the terms 'senior' and 'junior'. These terms carry a bit of controversy, with some amount of judgment, and I'd like to define them before moving forward.

A senior developer has had the time and opportunity to make more mistakes and learn from them. Senior developers tend to be experienced, jaded, and hopefully wise. Senior developers know the lay of the land, and can generally either navigate tough situations or avoid them altogether.

A junior developer is bright, curious, and hasn't made enough mistakes yet. Juniors are excited, learn quickly, and bring vital new perspectives into a project. They are not liabilities to quickly harden into seniors - their new energy is essential for experimentation and challenging the possibly stale knowledge of your senior team. The joyful chaos of a talented junior can teach you more about your systems than you might believe possible.

A person can have ten years of experience with Java and be a junior to Haskell. A senior Haskell engineer might be a junior in C#. A junior can be considerably smarter than a senior. A person with two years of experience may be more senior than a person with eight. The relevant characteristic - to me - is the sum of mistakes made and lessons learned.

A large team and project benefits from having both seniors and juniors. The Haskell community as a whole benefits from having junior roles - how else are we going to get experienced Haskell developers that can seed new companies and start compelling projects? If I was not offered the internship, I would not be a professional Haskell developer today. You would not be reading this book. We must pay it forward to grow the community and help cement the success of this wonderful language.

Unfortunately, most Haskell projects that I have experienced are almost entirely staffed with senior developers. There is a vicious cycle at play:

1. Alice, a brilliant Haskeller, gets to start a project. She is uniquely well suited to it - she has a ton of domain experience and knows Haskell inside and out.
2. Alice uses advanced features and libraries to develop the project. Alice leverages all the safety and productivity features to deliver the project on time, under budget, and without defects. The project is a resounding success.
3. The project accretes new features and responsibilities. While Alice is able to write code fast enough to cover this growth, the other aspects of a project begin to demand another developer. Haskell doesn't make writing documentation any faster.

4. Alice compiles a list of requirements for a new developer. To be productive, the engineer must understand advanced Haskell tricks. There isn't time to train a junior engineer to be productive.

This creates greater and greater demand for senior Haskellers. If you are a senior Haskeller, you may think this is just fine. More job opportunities and more pay competition!

This is unsustainable. The business always has the option of canning Haskell, hiring a bunch of Go/Java/C# developers and destroying the Haskell project. Not only has a Haskell project been destroyed, but another business person has real life experience with Haskell failing.

3.3 Hiring Seniors

You will probably need to hire senior Haskell engineers. Searching for raw Haskell ability is tempting, but this is not as necessary as you might think. The original Haskell developer(s) can handle all of the difficult bits that the new hire does not understand. Instead, we'll want to look for the Four Principles of this book:

1. Complexity: favors simple solutions
2. Novelty: favors traditional solutions
3. Cohesion: won't get into style arguments
4. Empathy: can show compassion for others

Hiring is a two way street, so let's first look at ways you can improve the probability of a successful hiring process. Furthermore, we're not just concerned with the hiring event - we're also concerned with retention.

Remote Friendly

If your company is in San Francisco, New York City, Glasgow, or a handful of other Haskell hubs, then you can probably hire locally. Otherwise, you will need to expand your search to full-remote candidates. Haskell developers are widely distributed across the globe, and you will dramatically increase the quality and quantity of Haskell developers if you don't require them to move to your city.

The first few hires are a great time to develop your remote-friendly workflows. These workflows work fantastically for many companies and open source communities. Beyond opening up your hiring pool for Haskell developers, remote work will increase productivity by promoting asynchronous work practices.

This isn't a book on how to successfully manage a remote team, so you'll need to look elsewhere for that. I'm merely a meager Haskell developer, and I can only tell you what makes hiring dramatically easier.

Don't Skimp

There's a misconception that developers are willing to accept lower pay to use Haskell. This is not generally true. Experienced Haskell engineers are rare and valuable, and you get what you pay for. My salary and benefits as a Haskell engineer have usually been competitive with the market for my role and experience.

Along with any misconception, there's a kernel of truth in a specific context. Some companies pay their engineers exceptionally well. Google, Facebook, Netflix, Indeed, etc are capable of offering total compensation packages in excess of $500k per year. I have not heard of a single Haskell developer making that much, though I have heard of at least one in the $300k range.

So you might be able to hire an ex-Googler who is used to making $400k, and "only" pay her $250k to use Haskell. But you should not expect to hire a senior engineer and pay under $100k - the discount doesn't work like that.

You might be able to hire an experienced Scala or F# engineer that has never used Haskell in production at a reduced rate. While this might work out OK, Haskell and Scala/F# are sufficiently different that the experience doesn't carry over as much as you might expect. Production Haskell has enough quirks and peculiarities that mere fluency with functional programming idioms won't carry you far.

Most Haskell shops pay their senior engineers a competitive rate. And most Haskell shops that only hire seniors don't require that much production experience to get hired. If you hire an experienced Scala developer to do Haskell at a deep discount, they'll take that experience and get a better paying Haskell job quickly.

This paints a picture of the Haskell salary landscape as bimodal. Experienced Haskellers can make competitive salaries, if not competitive with FAANG[1]. Less experienced Haskellers can accept a pay cut to learn Haskell on the job, but they'll soon level up and get into that second bucket. Remember that Haskell is a high variance language - you aren't betting on averages or medians, you are betting on beating the curve. Statistically, you need to be paying *better* than market average in order to select from the top end of the curve.

If you do hire a junior Haskeller (that is otherwise a senior engineer), be prepared to give them a substantial raise a year in, or prepare for turnover.

Don't Bait and Switch

Experienced Haskell engineers know this one all too well. There's a job ad posted which lists Haskell as a desired skill. Or maybe there's a Haskell Job posted, but you also need to know Java, PHP, Ruby, and Go. Unfortunately, Haskell is only a tiny minority of what the developer will be expected to do, despite the job being sold as "A Haskell Job."

Don't do this. You are going to frustrate developers who make it through, and you won't retain Haskell talent for long if you make them write another language for a majority of the time. You can't hijack their passion for Haskell and have them write PHP at the same level. As above, the developer will get to list production Haskell on their resume and skip to another company to work on Haskell the majority of the time.

This isn't to say you can't have a polyglot tech stack. There's nothing wrong with having microservices in Go, Ruby, and also Haskell. Indeed, requiring a Haskeller occasionally write another language is a good way to select for Pragmatic Haskellers instead of purists. This must be

[1] Facebook, Amazon, Apple, Netflix, Google. A common initialism for some of the major players in the tech industry hiring, with some of the highest compensation. While uncommon, Facebook and Google do have a few Haskellers in employ.

communicated up front and honestly, though. If you anticipate a person writing 10% Haskell, then don't bill it as a Haskell job. It's a Go job with a tiny Haskell responsibility.

Impurity Testing

Non-Haskell responsibilities may be wise to have in the job description.

There is a type of Haskell developer that I have seen. They only want to work with Haskell. 100% Haskell. No JavaScript, no Ruby, no Go, no Java, no bash, no PHP, no sysadmin responsibilities, nothing! Just Haskell.

These developers are often great at Haskell, and hiring them is tempting. You probably should not. While Haskell is a fantastic choice for many applications, anyone that requires a 100% Haskell experience will inevitably choose Haskell where another choice is more appropriate. You do not want this on your team. Worse, they may bring unwanted complexity and novelty into the project.

Remember, the purpose of an industrial software project is to promote the needs of the business. Haskell legitimately does this. If I didn't believe this (based on my experiences), then I wouldn't be writing a book on how to do it successfully. But Haskell isn't a sufficient cause for success. Knowing when to use Haskell and when to defer to another tool is crucial for any well-rounded Haskell engineer.

Embrace Diversity

Haskellers are weird. They're not going to look or act like typical programmers, because they're not! If you filter too hard on extraneous "culture fit" qualities, then you'll be passing up a ton of good engineers. This goes doubly so for under-represented minorities.

This isn't to say that Haskell developers are *better* or *worse* than others, just that they're different. Lean into and embrace the differences.

3.4 Hiring Juniors

A junior to Haskell is someone that hasn't made enough mistakes yet. This can be someone who just started learning to program last year and somehow picked Haskell, or it could be a grizzled Scala developer with 10 years of professional experience who just started learning Haskell. Picking out a good candidate for a Junior role is slightly different than in other programming languages.

Haskell has a much smaller population than other programming languages. The population is fractured among many communities. The total amount of support available to juniors is lower than in other languages. This means that you'll need to cover the slack.

Investing in a culture of training and mentorship is a great way to achieve this. Directly mentoring your juniors will promote Cohesion in the project. Senior developers will get valuable practice teaching and gain insights from the junior.

Haskell's selection effects mean that you will likely need less mentoring than you might expect. Haskell is sufficiently Weird and Different that most people excited about it are already self-starters and great at doing independent research. The work of the mentor is less "teach" and more "guide."

All of the above advice for hiring seniors applies to juniors, as well. You should definitely try to hire a junior Haskeller early on to a project's life cycle, for a few important reasons. A senior Haskell engineer will be dramatically more productive than in another language, but the overall workload of an engineer is only partially technical. Junior engineers are remarkably well suited to many of the tasks in software engineering that aren't directly related to technical competency and experience. This work also acts as excellent training for the junior.

Supporting Tasks

An experienced Haskeller can deploy features faster than in any other language, with less time spent towards fixing bugs. Unfortunately, documentation doesn't take any less time to write. If you hire for the capacity to deliver features, then you will not have the person-hours to write documentation or provide other forms of support to the project.

Junior developers may not have the same capacity to write code as seniors, but they are comparatively less disadvantaged at writing documentation and supporting the project in other ways. Writing this documentation and supporting the codebase will give them excellent experience with the code, which will help advance their knowledge and skill.

This is not to say that seniors shouldn't write documentation. They absolutely should! But a senior's documentation may miss context that is not obvious to someone deeply embedded into the project. The documentation a junior writes will often be more comprehensive and assume less about the reader, unless the senior is a particularly good technical writer.

The process of reviewing this documentation is an excellent opportunity for a senior to provide extra information and clarification to the junior.

Clarifying Concepts

Senior Haskellers will naturally grow the complexity of a project unless they apply consistent effort to avoid doing so. However, a Senior Haskeller is poorly situated to make judgment calls about precisely how much complexity is being introduced. After all, they know and understand it, or have already done the work to figure it out. The surrounding context and assumptions are in the background.

The act of explaining choices and concepts to a Junior is a forcing function for identifying the complexity involved. If the idea is too complicated for the Junior on the team to understand without significant guidance, then it is too complicated!

This is not to say that juniors are incapable of understanding complex ideas, or that a junior should have veto power on any concept in a codebase. Juniors provide a powerful new perspective that can inform decisions. Respecting this perspective is crucial, but bowing to it entirely is unnecessary. Supporting a Junior in learning more complex ideas and gaining hands-on experience with them is part of the process.

Institutional Knowledge

Suppose your main Haskell developer wins the lottery and quits. You'll need to replace that person. A junior may even rise to the occasion and completely replace the newly departed developer. We shouldn't expect or pressure them to do this.

However, the junior will be in an excellent position to retain that institutional knowledge. They can help to interview new candidates and provide insight on what the codebase needs. The nature of a junior developer provides them an excellent insight on what will help to grow the codebase and keep it maintainable over time. A senior engineer that is unable to teach or explain to a junior engineer is not going to be a great hire.

Juniors are in a better position to make these transfers because they have fewer internalized assumptions than seniors. This gives them a better perspective when evaluating and transferring knowledge to new hires.

4. Evaluating Consultancies

You may need to hire consultants to help work on your project. Haskell consultancies can be an excellent source of deep expertise. Due to the niche nature of the language, there are not many Haskell consultancies, and they are all brilliant. I don't expect this to remain true forever, so I'll share how I evaluate consultancies to identify where they might be best applied.

4.1 Identifying the Target

Few consultancies send potential business to their competition. As a result, a consultancy will happily accept your business, even if you might be better served by another company. Just as the Haskell language has many communities, these consultancies are usually better suited for some communities than others.

All consultancies will say that they specialize in Industrial Haskell. However, their approaches differ, and some are more or less suited to different application domains in this niche.

Haskell consultancies advertise via open source portfolios and blog posts. These portfolios form a body of evidence of work, and can be analyzed to determine proper fit. Many of the techniques for evaluating consultancies require evaluating libraries, and I don't cover that until section 5 ("Interfacing the Real"). However, we can get a general idea for the target market without too much of a deep dive.

First, we'll look up the website, blog, and other marketing materials. Consultancies generally market towards their niche, and if they're not speaking to your needs, they're probably not a great fit. Consultancies get material for blog post from independent research and lessons learned in consulting work, so this is a good way to see how they handle issues that come up. Additionally, you'll get a feel for their communication style (at least, as is presented to the world).

Next, we want to evaluate the libraries that the consultancy supports. This gives us important information about how they write code and the approaches they support. The easiest way to do this is by looking for the main source code repository organization for the consultancy (often GitHub, but GitLab and BitBucket are also possibilities). We'll also want to look at the GitHub accounts of employees, if we can find them. Consultancies usually hire engineers because of their open source contributions - if you hire the engineer that supports X library, you can sell consulting to users of that library.

Finally, we'll want to try and find experience reports of companies that have used these consultancies. Employees that worked on projects that were assisted or completed by consultancies are another valuable resource here. This information will be harder to acquire. Companies rarely want to publish details like this, so you'll more likely acquire them through community involvement. Individuals won't publish negative experience reports, as consultancies tend to have an outsized effect on public opinion in small communities like Haskell.

Let's investigate a few of the larger consultancies.

4.2 Well-Typed

Well-Typed bills themselves as "The Haskell Consultants." With community heavyweights like Duncan Coutts, Andres Löh, and Edsko de Vries, they certainly have claim to the title. The company has been active since 2008. This is a solid pedigree for success, and they have a track record to back it up.

Almost everyone in the staff listing has a degree in computer science, and more have a doctoral degree than a mere bachelor's degree. The academic background at Well-Typed is well-represented.

The GitHub at `https://www.github.com/well-typed` lists a number of repositories that will be useful to investigate. I will select a few here:

- `optics`, an alternative `lens` library that offers much improved error messages
- `generics-sop`, an alternative to `GHC.Generics`

- `ixset-typed`, a strongly typed indexed set data structure
- `cborg`, a binary serialization library

Furthermore, we see a number of contributions to the `cabal` repository from Well-Typed employees, along with other core Haskell infrastructure. The major maintainers for the Servant web library are employed by Well-Typed. The `acid-state` database is also maintained by Well-Typed employees.

I have worked directly with Well-Typed while employed by IOHK. The strong theoretical knowledge was critical for developing much of the highly theoretical aspects of the codebase. The equally strong technical knowledge for Haskell development was excellent for developing the Wallet part of the codebase.

Well-Typed delivers extremely strong on theoretical concerns and Haskell expertise. However, this reveals an operational blind-spot: relying on Haskell where other tools may be more suitable. The use of `acid-state` at IOHK was the source of numerous problems, documented in the Databases chapter of this book. Additionally, the extremely high skill of Well-Typed employees is reflected in the complexity and difficulty of the solutions delivered.

I would not hesitate to hire Well-Typed to help on a project for industrial use, especially if the project requires novel theoretical insight. I would be cautious to ensure that the resulting solution can be understood easily by the core team of engineers. Well-Typed's trainings are excellent for promoting an intermediate or advanced Haskeller to the next level.

4.3 FP Complete

FP Complete used to bill themselves as primarily Haskell consultants, but they have pivoted in recent years to devops and blockchain as well. Michael Snoyman is the director of engineering, and other than that, their website does not list any engineers. Their blog contains posts on many topics, including Rust, devops, containers, and Haskell.

Michael Snoyman and Aaron Contorer, the two driving members of FP Complete, do not have an extensive background in academia or computer science theory. Michael's degree is in actuarial sciences, while Aaron specialized in emerging technologies with Microsoft. The approach that the company takes is informed primarily by industrial needs. This helps to explain their more diverse focus - Haskell plays a prominent role, but devops, Rust, and other technologies are important to their business strategy and marketing.

The GitHub at `https://www.github.com/fpco` offers a few more clues. There are several members of the organization listed: Niklas Hambüchen, Sibi Prabakaran, Chris Done stand out as Haskell contributors. The FPCo GitHub has many repositories we can inspect:

- `safe-exceptions`, a library to assist in safe and predictable exception handling
- `stackage-server`, the code that hosts the Stackage package set
- `weigh`, a library for measuring memory allocations of Haskell functions
- `resourcet`, a library for safe and prompt resource allocation

Other relevant libraries include the Yesod web framework, the Persistent database library, and the `stack` build tool.

I have not worked with FP Complete directly, but I have extensive experience with Yesod, Persistent, and have collaborated directly with Michael Snoyman. I used these libraries to quickly and effectively deliver working and maintainable software at my first job, and the focus on real industrial concerns led to my success there. The libraries tend to be easy to work with, accept newcomer contributions regularly, and aren't terribly strict about coding standards. This is a double edged sword - many libraries throw exceptions more often than programmers might prefer instead of signaling with typed error channels. Template Haskell is often used to reduce boilerplate and provide type-safety, a choice that is pragmatic but unfashionable among more 'pure' functional personalities.

I would not hesitate to hire FP Complete for industrial use, especially if the project does not have novel theoretical requirements. FP Complete's training is proven to help get Juniors proficient with Haskell, and they have the ability to train on advanced and gritty GHC behaviors as well.

Exceptions

FP Complete wrote the definitive article on safe exception handling in Haskell[1]. It may come as no surprise that their libraries tend to throw runtime exceptions more than you might expect, or want. Some libraries in the Haskell ecosystem use the `ExceptT` monad transformer to signal exceptions. FP Complete believes this to be an anti-pattern, and wrote an article[2] saying as much. Instead, you can expect that `IO` functions in FP Complete libraries throw run-time exceptions.

`TemplateHaskell`

FP Complete's libraries tend to use `TemplateHaskell` extensively for functionality. Yesod uses a `QuasiQuoter` to define routes for the web app. Shakespeare uses a `QuasiQuoter` to interpolate values. `monad-logger` uses `TemplateHaskell` logging functions to interject the log line location. `persistent` uses a `QuasiQuoter` to define types for interacting with the database.

`TemplateHaskell` and `QuasiQuoters` are often maligned among Haskellers. They do carry some downsides. Any use of `TemplateHaskell` in a module will require GHC to fire up a code interpreter - this slows down compilation with a constant hit of a few hundred milliseconds on my laptop. However, generating the code is usually quite fast. If the resulting generated code is extremely large, then compiling that will be slow.

`QuasiQuoters` define a separate language that is parsed into a Haskell expression. A separate language has some upsides: you can define exactly what you want and need without having to worry about Haskell's restrictions. Unfortunately, you need to invent your own syntax and parser. You need to document these things and keep those documents up-to-date. The code that is generated by the `QuasiQuoter` is often not amenable to inspection - you cannot "Jump To Definition" on a type that is generated by `TemplateHaskell`, nor can you view the code easily.

[1] https://www.fpcomplete.com/haskell/tutorial/exceptions/

[2] https://www.schoolofhaskell.com/user/commercial/content/exceptions-best-practices

5. Invert Your Mocks!

Mocking comes up a lot in discussions of testing effectful code in Haskell. One of the advantages for `mtl` type classes or `Eff` freer monads is that you can swap implementations and run the same program on different underlying interpretations. This is cool! However, it's an extremely heavy weight technique, with a ton of complexity.

In the previous chapter, I recommended developing with the `ReaderT` pattern - something like this:

```haskell
newtype App a = App { unApp :: ReaderT AppCtx IO a }
```

Now, how would I go about testing this sort of function?

```haskell
doWork :: App ()
doWork = do
    query <- runHTTP getUserQuery
    users <- runDB (usersSatisfying query)
    for_ users $ \user -> do
        thing <- getSomething user
        let result = compute thing
        runRedis (writeKey (userRedisKey user) result)
```

If we have our `mtl` or `Eff` or OOP mocking hats on, we might think:

> I know! We need to mock our HTTP, database, and Redis effects. Then we can control the environment using mock implementations, and verify that the results are sound!

Mocking is awful. It complicates every aspect of our codebase, and it doesn't even make for reliable tests. I'll cover techniques on mocking in a later part of the book, but it would be significantly nicer if we never had to do it. Who knows - maybe you never will! But first, we're going to need to figure out ways to test our code without relying on mocking.

Let's step back and apply some more elementary techniques to this problem.

5.1 Decomposing Effects

The first thing we need to do is recognize that *effects* and *values* are separate, and try to keep them as separate as possible. The separation of effects and values is a fundamental principle of purely functional programming. Generally speaking, functions that look like doWork are not functional (in the "functional programming" sense). Let's look at the type signature for a few clues.

```haskell
doWork :: App ()
```

Our first warning is that this function has no arguments. That means that any input to this function must come from the App environment. These inputs are *effects*.

Likewise, this function returns () - the unit type, signifying nothing. There is no meaningful value here. If this function *does* anything at all, it must be a side-effect.

So, let's look again at what the function does. We'll need to decompose the function before we can test it.

```haskell
doWork :: App ()
doWork = do
    query <- runHTTP getUserQuery
    users <- runDB (usersSatisfying query)
    for_ users $ \user -> do
        thing <- getSomething user
        let result = compute thing
        runRedis (writeKey (userRedisKey user) result)
```

We get a bunch of stuff - inputs - that are acquired as the result of an *effect*. To test this directly, we need to somehow intercept the effect and provide some other value. This is unpleasant to do in Haskell.

Instead, let's split this into two functions. The first will be responsible for performing the input effects. The second will accept the *results* of those input effects as a pure function parameter.

```haskell
doWork :: App ()
doWork = do
    query <- runHTTP getUserQuery
    users <- runDB (usersSatisfying query)
    doWorkHelper users

doWorkHelper :: [User] -> App ()
doWorkHelper users =
    for_ users $ \user -> do
        thing <- getSomething user
        let result = compute thing
        runRedis (writeKey (userRedisKey user) result)
```

Now, to test `doWorkHelper`, we don't need to mock out the effects that get the `[User]` out. We can provide whatever `[User]` we want in our tests without having to orchestrate a fake HTTP service and database.

Now, the only remaining effects in `doWorkHelper` are `getSomething` and `runRedis`. But I'm not satisfied. We can get rid of the `getSomething` by factoring another helper out. We'll follow the same pattern: call the input effect, collect the values, and provide them as inputs to a new function.

```haskell
doWorkHelper :: [User] -> App ()
doWorkHelper users = do
    things'users <- for users $ \user -> do
        thing <- getSomething user
        pure (thing, user)
    lookMaNoInputs thing'users

lookMaNoInputs :: [(Thing, User)] -> App ()
lookMaNoInputs things'users =
    for_ things'users $ \(thing, user) -> do
        let result = compute thing
        runRedis (writeKey (userRedisKey user) result)
```

We've now extracted all of the "input effects." The function `lookMaNoInputs` (as it suggests) only performs *output* effects. If we want to test this, we can provide any `[(Thing, User)]` we want.

However, we're still stuck with our output effects. If we want to test this, we'd need to verify that the `App` environment (or real world) actually changed in the way we expect. Fortunately, we have a trick up our sleeve for this. Let's inspect our output effect:

```haskell
runRedis (writeKey (userRedisKey user) result)
```

It expects two things:

1. The user's Redis key
2. The computed result from the `thing`.

We can prepare the Redis key and computed result fairly easily:

```haskell
businessLogic :: (Thing, User) -> (RedisKey, Result)
businessLogic (thing, user) = (userRedisKey user, compute thing)

lookMaNoInputs :: [(Thing, User)] -> App ()
lookMaNoInputs users = do
    for_ (map businessLogic users) $ \(key, result) -> do
        runRedis (writeKey key result)
```

Neat! We've isolated the core business logic out and now we can write nice unit tests on that business logic. The tuple is a bit irrelevant - the userRedisKey function and compute thing call are totally independent. We can write tests on compute and userRedisKey independently. The *composition* of these two functions should *also* be fine, even without testing businessLogic itself. All of the business logic has been excised from the effectful code, and we've reduced the amount of code we need to test.

Now, you may still want to write integration tests for the various effectful functions. Verifying that *these* operate correctly is an important thing to do. However, you won't want to test them over-and-over again. You want to test your business logic independently of your effectful logic.

5.2 Streaming Decomposition

Streaming libraries like Pipes and Conduit are a great way to handle large data sets and interleave effects. They're *also* a great way to decompose functions and provide "inverted mocking" facilities to your programs. You may have noticed that our refactor in the previous section involved going from a single iteration over the data to multiple iterations. At first, we grabbed the [User], and for each User, we made a request and wrote to Redis. But the final version iterates over the [User] and pairs it with the request. Then we iterate over the result again and write to Redis at once.

We can use conduit to avoid the extra pass, all while keeping our code nicely factored and testable.

Most conduits look like this:

```haskell
import Data.Conduit (runConduit, (.|))
import qualified Data.Conduit.List as CL

streamSomeStuff :: IO ()
streamSomeStuff = do
    runConduit
         $ conduitThatGetsStuff
        .| conduitThatProcessesStuff
        .| conduitThatConsumesStuff
```

The pipe operator (.|) can be thought of as a Unix pipe - "take the streamed outputs from the first Conduit and plug them in as inputs to the second Conduit." The first part of a Conduit is the "producer" or "source." This can be from a database action, an HTTP request, or from a file handle. You can also produce from a plain list of values.

Let's look at conduitThatGetsStuff - it produces the values for us.

```haskell
-- Explicit
type ConduitT input output monad returnValue

-- Abbreviated
type ConduitT i o m r

conduitThatGetsStuff
    :: ConduitT () ByteString IO ()
--              ^  ^          ^  ^
--              |  |          |  return
--              |  |          monad
--              |  output
--              input
```

conduitThatGetsStuff accepts () as the input. This is a signal that it is mostly used to *produce* things, particularly in the monad type. So conduitThatGetsStuff may perform IO effects to produce ByteString chunks. When the conduit is finished running, it returns () - or, nothing important.

The next part of the conduit is `conduitThatProcessesStuff`. This function is right here:

```haskell
conduitThatProcessesStuff :: ConduitT ByteString RealThing IO ()
conduitThatProcessesStuff =
    CL.map parseFromByteString
    .| CL.mapM (either throwIO pure)
    .| CL.map convertSomeThing
    .| CL.filter someFilterCondition
```

This `ConduitT` accepts `ByteString` as input, emits `RealThing` as output, and operates in `IO`. We start by parsing values into an `Either`. The second part of the pipeline throws an exception if the previous step returned `Left`, or passes the `Right` along to the next part of the pipeline. `CL.map` does a conversion, and then `CL.filter` only passes along `RealThing`s that satisfy a condition.

Finally, we need to actually *do* something with the `RealThing`.

```haskell
conduitThatConsumesStuff :: Consumer RealThing IO ()
conduitThatConsumesStuff =
    passThrough print
    .| passThrough makeHttpPost
    .| CL.mapM_ saveToDatabase
  where
    passThrough :: (a -> IO ()) -> Conduit a IO a
    passThrough action = CL.mapM $ \a -> do
        action a
        pure a
```

This `print`s each item before yielding it to `makeHttpPost`, which finally yields to `saveToDatabase`.

We have a bunch of small, decomposed things. Our `conduitThatProcessesStuff` doesn't care where it gets the `ByteString`s that it parses – you can hook it up to *any* `ConduitT i ByteString IO r`. Databases, HTTP calls, file IO, or even just `CL.sourceList [example1, example2, example3]`.

Likewise, the conduitThatConsumesStuff doesn't care where the RealThings come from. You can use CL.sourceList to provide a bunch of fake input.

We're not usually working directly with Conduits here, either – most of the functions are provided to CL.mapM_ or CL.filter or CL.map. That allows us to write functions that are simple a -> m b or a -> Bool or a -> b, and these are really easy to test.

doWork: conduit-style

Above, we had doWork, and we decomposed it into several small functions. While we can be confident it processes the input list efficiently, we're not guaranteed that it will work in a constant amount of memory. The original implementation made a single pass over the user list. The second one does three, conceptually: the first for_ to grab the secondary inputs, the call to map businessLogic and the final for_ to perform the output effect. If there were more passes and we wanted to guarantee prompt effects, we can use a Conduit.

So let's rewrite doWork as a ConduitT. First, we'll want a producer that yields our User records downstream.

```
sourceUsers :: ConduitT () User App ()
sourceUsers = do
    users <- lift $ do
        query <- runHttp getUserQuery
        runDB (usersSatisfying query)
    sourceList yieldMany users
```

Now, we'll define a conduit that gets the thing for a user and passes it along.

```
-- Alternatively, using the `Conduit.List` API:
getThing :: ConduitT User (User, Thing) App ()
getThing =
    CL.mapM $ \user -> do
        thing <- getSomething user
        pure (user, thing)
```

Another conduit computes the result.

```
computeResult :: Monad m => ConduitT (User, Thing) (User, Result) m ()
computeResult =
    mapC $ \(user, thing) -> (user, compute thing)
```

The final step in the pipeline is to consume the result.

```
consumeResult :: ConduitT (User, Result) Void App ()
consumeResult = do
    CL.mapM_ $ \(user, result) ->
        runRedis $ writeKey (userRedisKey user) result
```

The assembled solution is here:

```
doWork :: App ()
doWork = runConduit
     $ sourceUsers
    .| getThing
    .| computeResult
    .| consumeResult
```

This has the same efficiency as the original implementation, and also processes things in the same order. However, we've been able to extract the effects and separate them. The `computeResult :: ConduitT _ __` is *pure* , and can be tested without running any `IO`.

Even supposing that `computeResult` *were* in plain `IO`, that's easier to test than a potentially complex App type.

5.3 Plain ol' abstraction

Always keep in mind the lightest and most general techniques in functional programming:

1. Make it a function
2. Abstract a parameter

These will get you far.

Let's revisit the doWork business up top:

```
doWork :: App ()
doWork = do
    query <- runHTTP getUserQuery
    users <- runDB (usersSatisfying query)
    for_ users $ \user -> do
        thing <- getSomething user
        let result = compute thing
        runRedis (writeKey (userRedisKey user) result)
```

We can make this *abstract* by taking concrete terms and making them function parameters. The literal definition of lambda abstraction!

```
doWorkAbstract
    :: Monad m
    => m Query -- ^ The HTTP getUserQuery
    -> (Query -> m [User]) -- ^ The database action
    -> (User -> m Thing) -- ^ The getSomething function
    -> (RedisKey -> Result -> m ()) -- ^ finally, the redis action
    -> m ()
doWorkAbstract getUserQuery getUsers getSomething redisAction = do
    query <- getUserQuery
    users <- getUsers query
    for_ users $ \user -> do
        thing <- getSomething user
        let result = compute thing
        redisAction (userRedisKey user) result
```

There are some interesting things to note about this abstract definition:

1. It's parameterized over *any* monad. `Identity`, `State`, `IO`, whatever. You choose!
2. We have a pure specification of the effect logic. This can't *do* anything. It just describes what to do, when given the right tools.
3. This is basically dependency injection on steroids.

Given the above abstract definition, we can easily recover the concrete `doWork` by providing the necessary functions:

```
doWork :: App ()
doWork =
    doWorkAbstract
        (runHTTP getUserQuery)
        (\query -> runDB (usersSatisfying query))
        (\user -> getSomething user)
        (\key result -> runRedis (writeKey key result))
```

We can also easily get a testing variant that logs the actions taken:

```
doWorkScribe :: Writer [String] ()
doWorkScribe =
    doWorkAbstract getQ getUsers getSomething redis
  where
    getQ = do
        tell ["getting users query"]
        pure AnyUserQuery
    getUsers _ = do
        tell ["getting users"]
        pure [exampleUser1, exampleUser2]
    getSomething u = do
        tell ["getting something for " <> show u]
        pure (fakeSomethingFor u)
    redis k v = do
        tell ["wrote k: " <> show k]
        tell ["wrote v: " <> show v]
```

All without having to fuss about with monad transformers, type classes, or anything else that's terribly complicated.

5.4 Decompose!!!

Ultimately, this is all about decomposition of programs into their smallest, most easily testable parts. You then unit or property test these tiny parts to ensure they work together. If all the parts work independently, then they should work together when composed.

Your effects should ideally not be anywhere near your business logic. Pure functions from a to b are ridiculously easy to test, especially if you can express properties.

If your business logic really needs to perform effects, then try the simplest possible techniques first: functions and abstractions. I believe that writing and testing functions that take pure values is simpler and easier. These are agnostic to *where* the data comes from, and don't need to be mocked at all. This transformation is typically easier than introducing mtl classes, monad transformers, Eff, or similar techniques.

5.5 What if I need to?

Sometimes, you really just can't avoid testing effectful code. A common pattern I've noticed is that people want to make things abstract at a level that is far too low. You want to make the abstraction as *weak* as possible, to make it *easy* to mock.

Consider the common case of wanting to mock out the database. This is reasonable: database calls are extremely slow! Implementing a mock database, however, is an extremely difficult task – you essentially have to implement a database. Where the behavior of the database differs from your mock, then you'll have test/prod mismatch that will blow up at some point.

Instead, go a level up - create a new indirection layer that can be satisfied by either the database or a simple to implement mock. You can do this with a type class, or just by abstracting the relevant functions concretely. Abstracting the relevant functions is the easiest and simplest technique, but it's not unreasonable to also write:

```haskell
data UserQuery
    = AllUsers
    | UserById UserId
    | UserByEmail Email

class Monad m => GetUsers m where
    runUserQuery :: UserQuery -> m [User]
```

This is *vastly* more tenable interface to implement than a SQL database! Let's write our instances, one for the `persistent` [1] library and another for a mock that uses QuickCheck's `Gen` type:

```haskell
instance MonadIO m => GetUsers (SqlPersistT m) where
    runUserQuery = selectList . convertToQuery

instance GetUsers Gen where
    runUserQuery query =
        case query of
            AllUsers ->
                arbitrary
            UserById userId ->
                take 1 . fmap (setUserId userId) <$> arbitrary
            UserByEmail userEmail ->
                take 1 . fmap (setUserEmail userEmail) <$> arbitrary
```

Alternatively, you can just pass functions around manually instead of using the type class mechanism to pass them for you.

[1] https://hackage.haskell.org/package/persistent

Oh, wait, no! That `GetUsers Gen` instance has a bug! Can you guess what it is?

In the `UserById` and `UserByEmail` case, we're not ever testing the "empty list" case – what if that user does not exist?

A fixed variant looks like this:

```haskell
instance GetUsers Gen where
    runUserQuery query =
        case query of
            AllUsers ->
                arbitrary
            UserById userId -> do
                oneOrZero <- choose (0, 1)
                users <- map (setUserId userId) <$> arbitrary
                pure $ take oneOrZero users
            UserByEmail userEmail -> do
                oneOrZero <- choose (0, 1)
                users <- map (setUserEmail userEmail) <$> arbitrary
                pure $ take oneOrZero users
```

I made a mistake writing a super simple generator. Just think about how many mistakes I might have made if I were trying to model something more complex!

6. The Trouble with Typed Errors

Us Haskell developers don't like runtime errors. They're awful and nasty! You have to debug them, and they're not represented in the types. Instead, we like to use `Either` (or something isomorphic) to represent stuff that might fail:

```haskell
data Either l r = Left l | Right r
```

`Either` has a `Monad` instance, so you can short-circuit an `Either l r` computation with an `l` value, or bind it to a function on the `r` value. The names of the type and constructors are not arbitrary. We have two type variables: `Either left right`. The `left` type variable is in the `Left` constructor, and the `right` type variable is in the `Right` constructor.

So, we take our unsafe, runtime failure functions:

```haskell
head   :: [a] -> a
lookup :: k -> Map k v -> v
parse  :: String -> Integer
```

and we use informative error types to represent their possible failures:

```haskell
data HeadError = ListWasEmpty

head :: [a] -> Either HeadError a

data LookupError = KeyWasNotPresent

lookup :: k -> Map k v -> Either LookupError v

data ParseError
    = UnexpectedChar Char String
    | RanOutOfInput

parse :: String -> Either ParseError Integer
```

Except, we don't really use types like HeadError or LookupError. There's only one way that head or lookup could fail. So we just use Maybe instead. Maybe a is just like using Either () a - there's only one possible Left () value, and there's only one possible Nothing value. (If you're unconvinced, write newtype Maybe a = Maybe (Either () a), derive all the relevant instances, and try and detect a difference between this Maybe and the stock one).

But, Maybe isn't great - we've lost information! Suppose we have some computation:

```
foo :: String -> Maybe Integer
foo str = do
    c <- head str
    r <- lookup str strMap
    eitherToMaybe (parse (c : r))
```

Now, we try it on some input, and it gives us Nothing back. Which step failed? We actually can't know that! All we can know is that *something* failed.

So, let's try using Either to get more information on what failed. Can we just write this?

```
foo :: String -> Either ??? Integer
foo str = do
    c <- head str
    r <- lookup str strMap
    parse (c : r)
```

Unfortunately, this gives us a type error. We can see why by looking at the type of >>=:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

The type variable `m` must be an instance of `Monad`, and the type `m` must be *exactly the same* for the value on the left and the function on the right. `Either LookupError` and `Either ParseError` are not the same type, and so this does not type check.

Instead, we need some way of accumulating these possible errors. We'll introduce a utility function `mapLeft` that helps us:

```
mapLeft :: (l -> l') -> Either l r -> Either l' r
mapLeft f (Left l) = Left (f l)
mapLeft _ r = r
```

Now, we can combine these error types:

```
foo :: String
    -> Either
        (Either HeadError (Either LookupError ParseError))
        Integer
foo str = do
    c <- mapLeft Left (head str)
    r <- mapLeft (Right . Left) (lookup str strMap)
    mapLeft (Right . Right) (parse (c : r))
```

There! Now we can know exactly how and why the computation failed. Unfortunately, that type is a bit of a monster. It's verbose and all the `mapLeft` boilerplate is annoying.

At this point, most application developers will create a "application error" type, and they'll just shove everything that can go wrong into it.

```haskell
data AllErrorsEver
     = AllParseError ParseError
     | AllLookupError LookupError
     | AllHeadError HeadError
     | AllWhateverError WhateverError
     | FileNotFound FileNotFoundError
     | etc...
```

Now, this slightly cleans up the code:

```haskell
foo :: String -> Either AllErrorsEver Integer
foo str = do
    c <- mapLeft AllHeadError (head str)
    r <- mapLeft AllLookupError (lookup str strMap)
    mapLeft AllParseError (parse (c : r))
```

However, there's a pretty major problem with this code. `foo` is claiming that it can "throw" all kinds of errors - it's being honest about parse errors, lookup errors, and head errors, but it's also claiming that it will throw if files aren't found, "whatever" happens, and `etc`. There's no way that a call to `foo` will result in `FileNotFound`, because `foo` can't even do `IO`! It's absurd. The type is too large! The previous chapter discusses keeping your types small and how wonderful it can be for getting rid of bugs.

Suppose we want to handle `foo`'s error. We call the function, and then write a case expression like good Haskellers:

```haskell
case foo "hello world" of
    Right val ->
        pure val
    Left err ->
        case err of
            AllParseError parseError ->
                handleParseError parseError
            AllLookupError lookupErr ->
                handleLookupError
            AllHeadError headErr ->
                handleHeadError
            _ ->
                error "impossible?!?!?!"
```

Unfortunately, this code is brittle to refactoring! We've claimed to handle all errors, but we're really not handling many of them. We currently "know" that these are the only errors that can happen, but there's no compiler guarantee that this is the case. Someone might later modify foo to throw another error, and this case expression will break. Any case expression that evaluates any result from foo will need to be updated.

The error type is too big, and so we introduce the possibility of mishandling it. There's another problem. Let's suppose we know how to handle a case or two of the error, but we must pass the rest of the error cases upstream:

```
bar :: String -> Either AllErrorsEver Integer
bar str =
    case foo str of
        Right val ->
            Right val
        Left err ->
            case err of
                AllParseError pe ->
                    Right (handleParseError pe)
                _ ->
                    Left err
```

We *know* that AllParseError has been handled by bar, because - just look at it! However, the compiler has no idea. Whenever we inspect the error content of bar, we must either a) "handle" an error case that has already been handled, perhaps dubiously, or b) ignore the error, and desperately hope that no underlying code ever ends up throwing the error.

Are we done with the problems on this approach? No! There's no guarantee that I throw the *right* error!

```
head :: [a] -> Either AllErrorsEver a
head (x:xs) = Right x
head [] = Left (AllLookupError KeyWasNotPresent)
```

This code type checks, but it's *wrong*, because `LookupError` is only supposed to be thrown by `lookup`! It's obvious in this case, but in larger functions and codebases, it won't be so clear.

6.1 Monolithic error types are bad

So, having a monolithic error type has a ton of problems. I'm going to make a claim here:

> All error types should have a single constructor

That is, errors should not be sum types. The name of the type and name of the constructor should be the same. The exception should carry *actual values* that would be useful in writing a unit test or debugging the problem. Carrying around a `String` message is a no-no.

Almost all programs can fail in multiple potential ways. How can we represent this if we only use a single constructor per type?

Let's maybe see if we can make `Either` any nicer to use. We'll define a few helpers that will reduce the typing necessary:

```
type (+) = Either
infixr + 5

l :: l -> Either l r
l = Left

r :: r -> Either l r
r = Right
```

Now, let's refactor that uglier `Either` code with these new helpers:

```
foo :: String
    -> Either
        (HeadError + LookupError + ParseError)
        Integer
foo str = do
    c <- mapLeft l (head str)
    r <- mapLeft (r . l) (lookup str strMap)
    mapLeft (r . r) (parse (c : r))
```

Well, the syntax is nicer. We can `case` over the nested Either in the error branch to eliminate single error cases. It's easier to ensure we don't claim to throw errors we don't - after all, GHC will correctly infer the type of `foo`, and if GHC infers a type variable for any `+`, then we can assume that we're not using that error slot, and can delete it.

Unfortunately, there's still the `mapLeft` boilerplate. And expressions which you'd *really* want to be equal, aren't –

```
x :: Either (HeadError + LookupError) Int
y :: Either (LookupError + HeadError) Int
```

The values `x` and `y` are *isomorphic*, but we can't use them in a `do` block because they're not exactly equal. If we add errors, then we must revise *all* `mapLeft` code, as well as all `case` expressions that inspect the errors. Fortunately, these are entirely compiler-guided refactors, so the chance of messing them up is small. However, they contribute significant boilerplate, noise, and busywork to our program.

6.2 Boilerplate be gone!

Well, turns out, we *can* get rid of the order dependence and boilerplate with type classes! The first approach is to use "classy prisms" from the `lens` package. Let's translate our types from concrete values to prismatic ones:

```haskell
-- Concrete:
head :: [a] -> Either HeadError a

-- Prismatic:
head :: AsHeadError err => [a] -> Either err a


-- Concrete:
lookup :: k -> Map k v -> Either LookupError v

-- Prismatic:
lookup
    :: (AsLookupError err)
    => k -> Map k v -> Either err v
```

Now, type class constraints don't care about order - `(Foo a, Bar a) => a` and `(Bar a, Foo a) => a` are exactly the same thing as far as GHC is concerned. The `AsXXX` type classes will *automatically* provide the `mapLeft` stuff for us, so now our `foo` function looks a great bit cleaner:

```haskell
foo :: (AsHeadError err, AsLookupError err, AsParseError err)
    => String -> Either err Integer
foo str = do
    c <- head str
    r <- lookup str strMap
    parse (c : r)
```

This appears to be a significant improvement over what we've had before! And, most of the boilerplate with the `AsXXX` classes is taken care of via Template Haskell:

```haskell
makeClassyPrisms ''ParseError
-- this line generates the following:

class AsParseError a where
    _ParseError :: Prism' a ParseError
    _UnexpectedChar :: Prism' a (Char, String)
    _RanOutOfInput :: Prism' a ()

-- etc...
instance AsParseError ParseError where
```

However, we do have to write our own boilerplate when we eventually want to concretely handle these types. We may end up writing a huge AppError that all of these errors get injected into.

There's one major, fatal flaw with this approach. While it composes nicely, it doesn't decompose at all! There's no way to catch a single case and ensure that it's handled. The machinery that prisms give us don't allow us to separate out a single constraint, so we can't pattern match on a single error.

Once again, our types become ever larger, with all of the problems that entails.

6.3 Type Classes To The Rescue!

What we *really* want is:

- Order independence
- No boilerplate
- Easy composition
- Easy decomposition

In PureScript or OCaml, you can use open variant types to do this flawlessly. Haskell doesn't have open variants, and the attempts to mock them end up quite clumsy to use in practice.

Fortunately, we can use type classes and constraints to do something similar. Above, we had a bunch of problems with the "nested `Either`" pattern - `Either (Either (Either A B) C) D`. This allows us to grow and shrink the exception type, which allows us to handle cases and introduce new ones. But the usability is quite bad.

The reason is that `Either _ _` represents a *binary tree* of types. We don't want a binary tree - we want a `Set`. But a `Set` of types is best modeled as a type class constraint. So we need a way to say that `A`, `B`, `C`, and `D` are all 'in' the type.

While I'd love to include this topic fully in the book, I feel it would be dishonest. I haven't used the technique in production, and cannot fully recommend it. If you're interested in reading about more experimental stuff, then I would recommend the "Plucking Constraints"[1] blog post, as well as the `plucky`[2] proof-of-concept library, and the super experimental `prio`[3] repository, which uses the `plucky` technique with `IO`-based exceptions.

6.4 The virtue of untyped errors

We've seen that typed errors have a number of problems. It's difficult to remove error cases. The boilerplate is intense. The bookkeeping is rarely ergonomic or friendly.

Typed errors have *lots* of problems and require a *lot* of work. Meanwhile, untyped errors have *lots* of problems but require *little* work. For this reason, I think it's best to stick with untyped exceptions, until something more robust comes along.

```haskell
throwIO :: (Exception e) => e -> IO a
```

[1] https://www.parsonsmatt.org/2020/01/03/plucking_constraints.html
[2] https://hackage.haskell.org/package/plucky
[3] https://github.com/parsonsmatt/prio

7. Template Haskell Is Not Scary

7.1 A Beginner Tutorial

This tutorial is aimed at people who are beginner-intermediate Haskellers looking to learn the basics of Template Haskell.

I learned about the power and utility of metaprogramming in Ruby. Ruby metaprogramming is done by constructing source code with string concatenation and having the interpreter run it. There are also some methods that can be used to define methods, constants, variables, etc.

In my Squirrell[1] Ruby library designed to make encapsulating SQL queries a bit easier, I have a few bits of metaprogramming to allow for some conveniences when defining classes. The idea is that you can define a query class like this:

```ruby
class PermissionExample
  include Squirrell

  requires :user_id
  permits :post_id

  def raw_sql
    <<SQL
SELECT *
FROM users
  INNER JOIN posts ON users.id = posts.user_id
WHERE users.id = #{user_id} #{has_post?}
SQL
  end

```

[1] https://github.com/parsonsmatt/squirrell/

```
  def has_post?
    post_id ? "AND posts.id = #{post_id}" : ""
  end
end
```

and by specifying `requires` with the symbols you want to require, it will define an instance variable and an attribute reader for you, and raise errors if you don't pass the required parameter. Accomplishing that was pretty easy. Calling `requires` does some bookkeeping with required parameters and then calls this method with the arguments passed:

```
def define_readers(args)
  args.each do |arg|
    attr_reader arg
  end
end
```

Which you can kinda read like a macro: take the arguments, and call `attr_reader` with each. The magic happens later, where I overrode the `initialize` method:

```
def initialize(args = {})
  return self if args.empty?

  Squirrell.requires[self.class].each do |k|

    unless args.keys.include? k
      fail MissingParameterError, "Missing required parameter: #{k}"
    end

    instance_variable_set "@#{k}", args.delete(k)
  end

  fail UnusedParameter, "Unspecified parameters: #{args}" if args.any?
end
```

We loop over the arguments provided to new, and if any required ones are missing, error. Otherwise, we set the instance variable associated with the argument, and remove it from the hash.

Another approach involves taking a string, and evaluating it in the context of whatever class you're in:

```
def lolwat(your_method, your_string)
  class_eval "def #{your_method}; puts #{your_string}; end"
end
```

This line of code defines a method with your choice of name and string to print in the context of whatever class is running.

7.2 wait this isn't haskell what am i doing here

Metaprogramming in Ruby is mostly based on a textual approach to code. You use Ruby to generate a string of Ruby code, and then you have Ruby evaluate the code.

If you're coming from this sort of background (as I was), then Template Haskell will strike you as different and weird. You'll think "Oh, I know, I'll just use QuasiQuoterss and it'll all work just right." Nope. You have to think differently about metaprogramming in Template Haskell. You're not going to be putting strings together that happen to make valid code. This is Haskell, we're going to have some compile time checking!

7.3 Constructing an AST

In Ruby, we built a string, which the Ruby interpreter then parsed, turned into an abstract syntax tree, and interpreted. In Haskell, we'll skip the string step. We'll build the Abstract Syntax Tree (AST) directly using standard data constructors. GHC will verify that we're doing everything OK in the construction of the syntax tree, and then it'll print the syntax tree into our source code before compiling the whole thing. So we get two levels of compile time checking - that we built a correct template, and that we used the template correctly.

One of the nastiest things about textual metaprogramming is the lack of guarantee that your syntax is right. Debugging syntax errors in generated code can be difficult. Verifying the correctness of our code is easier when programming directly into an AST. The quasiquoters are a convenience built around AST programming, but I'm of the opinion that you should learn the AST stuff first and then dive into the quoters when you have a good idea of how they work.

Alright, so let's get into our first example. We've written a function `bigBadMathProblem :: Int -> Double` that takes a lot of time at runtime, and we want to write a lookup table for the most common values. Since we want to ensure that runtime speed is super fast, and we don't mind waiting on the compiler, we'll do this with Template Haskell. We'll pass in a list of common numbers, run the function on each to precompute them, and then finally punt to the function if we didn't cache the number.

Since we want to do something like the `makeLenses` function to generate a bunch of declarations for us, we'll first look at the type of that in the `lens` library. Jumping to the lens docs[2], we can see that the type of `makesLenses` is `Name -> DecsQ`. Jumping to the Template Haskell docs[3], `DecsQ` is a type synonym for `Q [Dec]`. `Q` appears to be a monad for Template Haskell, and a `Dec`[4] is the data type for a declaration. The constructor for making a function declaration is `FunD`. We can get started with this!

We'll start by defining our function. It'll take a list of commonly used values, apply the function to each, and store the result. Finally, we'll need a clause that passes the value to the math function in the event we don't have it cached.

```
precompute :: [Int] -> DecsQ
precompute xs = do
    -- .......
    return [FunD name clauses]
```

Since `Q` is a monad, and `DecsQ` is a type synonym for it, we know we can start off with `do`. And we know we're going to be returning a function

[2] https://hackage.haskell.org/package/lens-4.13/docs/Control-Lens-TH.html

[3] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html

[4] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html#t:Dec

definition, which, according to the Dec documentation, has a field for the name of the function and the list of clauses. Now it's up to us to generate the name and clauses. Names are easy, so we'll do that first.

We can get a name from a string using mkName. This converts a string into an unqualified name. We're going to choose lookupTable as the name of our lookup table, so we can use that directly.

```haskell
precompute xs = do
    let name = mkName "lookupTable"
    -- ...
```

Now, we need to apply each variable in xs to the function named bigBadMathProblem. This will go in the [Clause] field, so let's look at what makes up a Clause. According to the documentation[5], a clause is a data constructor with three fields: a list of Pat patterns, a Body, and a list of Dec declarations. The body corresponds to the actual function definition, the Pat patterns correspond to the patterns we're matching input arguments on, and the Dec declarations are what we might find in a where clause.

Let's identify our patterns first. We're trying to match on the Ints directly. Our desired output is going to look something like:

```haskell
lookupTable 0 = 123.546
lookupTable 12 = 151626.4234
lookupTable 42 = 0.0
-- ...
lookupTable x = bigBadMathProblem x
```

So we need a way to get those Ints in our xs variable into a Pat pattern. We need some function Int -> Pat... Let's check out the documentation[6] for Pat and see how it works. The first pattern is LitP, which takes an argument of type Lit. A Lit is a sum type that has a constructor for the primitive Haskell types. There's one for IntegerL, which we can use.

So, we can get from Int -> Pat with the following function:

[5] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html#t:Clause

[6] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html#t:Pat

```haskell
intToPat :: Int -> Pat
intToPat = LitP . IntegerL . toInteger
```

Which we can map over the initial list to get our `[Pat]`!

```haskell
precompute xs = do
    let name = mkName "lookupTable"
        patterns = map intToPat xs
    -- ...
    return [FunD name clauses]
```

Our `lookupTable` function is only going to take a single argument, so we'll want to `map` our integer `Pat`s into `Clause`, going from our `[Pat] -> [Clause]`. That will get use the `clauses` variable that we need. From above, a clause is defined like:

```haskell
data Clause = Clause [Pat] Body [Dec]
```

So, our `[Pat]` is simple - we only have one literal value we're matching on. `Body` is defined to be either a `GuardedB` which uses pattern guards, or a `NormalB` which doesn't. We could define our function in terms of a single clause with a `GuardedB` body, but that sounds like more work, so we'll use a `NormalB` body. The `NormalB` constructor takes an argument of type `Exp`. So let's dig in to the `Exp` documentation![7]

There's a lot here. Looking above, we really want to have a single thing - a literal! The precomputed value. There's a `LitE` constructor which takes a `Lit` type. The `Lit` type has a constructor for `DoublePrimL` which takes a `Rational`, so we'll have to do a bit of conversion.

```haskell
precomputeInteger :: Int -> Exp
precomputeInteger =
    LitE . DoublePrimL . toRational . bigBadMathProblem
```

We can get the `Body`s for the `Clause`s by mapping this function over the list of arguments. The declarations will be blank, so we're ready to create our `clauses`!

[7] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html#t:Exp

```
precompute xs = do
    let name = mkName "lookupTable"
        patterns = map intToPat xs
        fnBodies = map precomputeInteger xs
        precomputedClauses =
            zipWith
                (\body pat -> Clause [pat] (NormalB body) [])
                fnBodies
                patterns
    -- ......
    return [FunD name clauses]
```

There's one thing left to do here. We need to create another clause with a variable x that we delegate to the function. Since mkName allows the variable to be shadowed, and that might create a warning in the generated code, we'll want to use newName to create a hygienic name for the variable. We will have to get a bit more complicated with our Body expression, since we've got an application to a function going on.

```
precompute xs = do
    let name = mkName "lookupTable"
        patterns = map intToPat xs
        fnBodies = map precomputeInteger xs
        precomputedClauses =
            zipWith
                (\body pat -> Clause [pat] (NormalB body) [])
                fnBodies
                patterns
    x' <- newName "x"
    let lastClause = [Clause [VarP x'] (NormalB appBody) []]
    -- ...
        clauses = precomputedClauses ++ lastClause
    return [FunD name clauses]
```

Going back to the Exp type, we're now looking for something that captures the idea of application. The Exp type has a data constructor AppE which takes two expressions and applies the second to the first. That's precisely what we need! It also has a data constructor VarE which takes a Name argument. That's all we need. Let's do it.

```haskell
precompute xs = do
    let name = mkName "lookupTable"
        patterns = map intToPat xs
        fnBodies = map precomputeInteger xs
        precomputedClauses =
            zipWith
                (\body pat -> Clause [pat] (NormalB body) [])
                fnBodies
                patterns
    x' <- newName "x"
    let lastClause =
            [Clause [VarP x'] (NormalB appBody) []]
        appBody =
            AppE (VarE 'bigBadMathProblem) (VarE x')
        clauses =
            precomputedClauses ++ lastClause
    return [FunD name clauses]
```

To get the name for bigBadMathProblem, we used a Template Haskell quote. The ' character creates a Name out of a value, while two apostrophes creates a Name out of a type. This is what you often see with deriving: deriveJSON ''MyType.

We did it! We wrangled up some Template Haskell and wrote ourselves a lookup table. Now, we'll want to splice it into the top level of our program with the $( ) splice syntax:

```haskell
$(precompute [1..1000])
```

As it happens, GHC is smart enough to know that a top level expression with the type Q [Dec] can be spliced without the explicit splicing syntax. So we could have also written:

```haskell
module X where

import Precompute (precompute)

precompute [1..1000]
```

Creating Haskell expressions using the data constructors is really easy, if a little verbose. Let's look at a little more complicated example.

7.4 Boilerplate Be Gone!

We're excited to be using the excellent `users` library with the `persistent` backend for the web application we're working on (source code located here, if you're curious[8]). It handles all kinds of stuff for us, taking care of a bunch of boilerplate and user related code. It expects, as its first argument, a value that can be unwrapped and used to run a Persistent query. It also operates in the `IO` monad. Right now, our application is setup to use a custom monad `AppM` which is defined like:

```haskell
type AppM = ReaderT Config (EitherT ServantErr IO)
```

So, to actually use the functions in the `users` library, we have to do this bit of fun business:

```haskell
someFunc :: AppM [User]
someFunc = do
    connPool <- asks getPool
    let conn = Persistent (`runSqlPool` connPool)
    users <- liftIO (listUsers conn Nothing)
    return (map snd users)
```

That's going to get annoying quickly, so we start writing functions specific to our monad that we can call instead of doing all that lifting and wrapping.

[8] https://github.com/parsonsmatt/QuickLift/

```haskell
backend :: AppM Persistent
backend = do
    pool <- asks getPool
    return (Persistent (`runSqlPool` pool))

myListUsers :: Maybe (Int64, Int64) -> AppM [(LoginId, QLUser)]
myListUsers m = do
    b <- backend
    liftIO (listUsers b m)

myGetUserById :: LoginId -> AppM (Maybe QLUser)
myGetUserById l = do
    b <- backend
    liftIO (getUserById b l)

myUpdateUser
    :: LoginId
    -> (QLUser -> QLUser)
    -> AppM (Either UpdateUserError ())
myUpdateUser id fn = do
    b <- backend
    liftIO (updateUser b id fn)
```

ahh, totally mechanical code. just straight up boiler plate. This is exactly the sort of thing I'd have metaprogrammed in Ruby. So let's metaprogram it in Haskell!

First, we'll want to simplify the expression. Let's use `listUsers` as the example. We'll make it as simple as possible - no infix operators, no do notation, etc.

```haskell
listUsersSimple m =
    (>>=) backend (\b -> liftIO (listUsers b m))
```

Nice. To make it a little easier on seeing the AST, we can take it one step further. Let's explicitly show all function application by adding parentheses to make everything as explicit as possible.

```haskell
listUsersExplicit m =
    ((>>=) backend) (\b -> liftIO ((listUsers b) m))
```

The general formula that we're going for is:

```haskell
derivedFunction arg1 arg2 ... argn =
  ((>>=) backend)
    (\b -> liftIO ((...(((function b) arg1) arg2)...) argn))
```

We'll start by creating our `deriveReader` function, which will take as its first argument the `backend` function name.

```haskell
deriveReader :: Name -> DecsQ
deriveReader rd =
    mapM (decForFunc rd)
        [ 'destroyUserBackend
        , 'housekeepBackend
        , 'getUserIdByName
        , 'getUserById
        , 'listUsers
        , 'countUsers
        , 'createUser
        , 'updateUser
        , 'updateUserDetails
        , 'authUser
        , 'deleteUser
        ]
```

This is our first bit of special syntax. The single quote in `'destroyUserBackend` returns the `Name` for `destroyUserBackend`. Unlike `mkName "destroyUserBackend"`, however, this is a *globally qualified name*. This `Name` works even if the module that splices the code doesn't import the code that it came from. If you are referring to names that exist outside of the code you generate, you need to use this form. Otherwise, your users will need to import a bunch of modules to satisfy the requirements of your macro.

Now, what we need is a function `decForFunc`, which has the signature `Name -> Name -> Q Dec`.

In order to do this, we'll need to get some information about the function we're trying to derive. Specifically, we need to know how many arguments the source function takes. There's a whole section in the Template Haskell documentation about 'Querying the Compiler'[9] which we can put to good use.

The `reify` function returns a value of type `Info`. For type class operations, it has the data constructor `ClassOpI` with arguments `Name`, `Type`, `ParentName`, and `Fixity`. None of these have the arity of the function directly...

I think it's time to do a bit of exploratory coding in the REPL. We can fire up `GHCi` and start doing some Template Haskell with the following commands:

```
λ: :set -XTemplateHaskell
λ: import Language.Haskell.TH
```

We can also do the following command, and it'll print out all of the generated code that it makes:

```
λ: :set -ddump-splices
```

Now, let's run `reify` on something simple and see the output!

```
λ: reify 'id

<interactive>:4:1:
    No instance for (Show (Q Info)) arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
```

Hmm.. No show instance. Fortunately, there's a workaround that can print out stuff in the `Q` monad:

[9] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html#g:3

```
λ: $(stringE . show =<< reify 'id)
"VarI
  GHC.Base.id
  (ForallT
    [KindedTV a_1627463132 StarT]
    []
    (AppT
      (AppT ArrowT (VarT a_1627463132))
      (VarT a_1627463132)
    )
  )
  Nothing
  (Fixity 9 InfixL)"
```

I've formatted it a bit to make it a bit more legible. We've got the `Name`, the `Type`, a `Nothing` value that is always `Nothing`, and the fixity of the function. The `Type` seems pretty useful… Let's look at the `reify` output for one of the class methods we're trying to work with:

```
λ: $(stringE . show =<< reify 'Web.Users.Types.getUserById)
"ClassOpI
  Web.Users.Types.getUserById
  (ForallT
    [KindedTV b_1627432398 StarT]
    [AppT
      (ConT Web.Users.Types.UserStorageBackend)
      (VarT b_1627432398)
    ]
    (ForallT
      [KindedTV a_1627482920 StarT]
      [ AppT
          (ConT Data.Aeson.Types.Class.FromJSON) (VarT a_1627482920)
      , AppT (ConT Data.Aeson.Types.Class.ToJSON) (VarT a_1627482920)
      ]
      (AppT
        (AppT
          ArrowT
          (VarT b_1627432398)
        )
        (AppT
          (AppT
            ArrowT
              (AppT
                (ConT Web.Users.Types.UserId)
                (VarT b_1627432398)
              )
          )
          (AppT
            (ConT GHC.Types.IO)
            (AppT
              (ConT GHC.Base.Maybe)
              (AppT
                (ConT Web.Users.Types.User)
                (VarT a_1627482920)
              )
            )
          )
        )
      )
    )
  )
  Web.Users.Types.UserStorageBackend
  (Fixity 9 InfixL)"
```

Wow, that is a ton of text! Believe it or not, I formatted it to make it a bit more legible. We're mainly interested in the `Type` declaration, and we can get a lot of information about what data constructors are used from the documentation[10]. Just like `AppE` is how we applied an expression to an expression, `AppT` is how we apply a type to a type. `ArrowT` is the function arrow in the type signature.

Just as an exercise, we'll go through the following type signature and transform it into something a bit like the above:

[10] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html#t:Type

```
fmap
    :: (a -> b) -> f a -> f b
     ~ ((->) a b) -> (f a) -> (f b)
     ~ (->) ((->) a b) ((f a) -> (f b))
     ~ (->) ((->) a b) ((->) (f a) (f b))
```

Ok, now all of our (->)s are written in prefix form. We'll replace the arrows with ArrowT, do explicit parentheses, and put in the ApplyT constructors working from the innermost expressions out.

```
~ (ArrowT ((ArrowT a) b)) ((ArrowT (f a)) (f b))
~ (ArrowT ((ApplyT ArrowT a) b)) ((ArrowT (ApplyT f a)) (ApplyT f b))
~ (ArrowT (ApplyT (ApplyT ArrowT a) b))
    (ApplyT (ApplyT ArrowT (ApplyT f a)) (ApplyT f b))
~ ApplyT (ArrowT (ApplyT (ApplyT ArrowT a) b))
    (ApplyT (ApplyT ArrowT (ApplyT f a)) (ApplyT f b))
```

That got pretty out of hand and messy looking. But, we have a good idea now of how we can get from one representation to the other.

So, going from our type signature, it looks like we can figure out how we can get the arguments we need from the type! We'll pattern match on the type signature, and if we see something that looks like the continuation of a type signature, we'll add one to a count and go deeper. Otherwise, we'll skip out.

The function definition looks like this:

```
functionLevels :: Type -> Int
functionLevels = go 0
  where
    go :: Int -> Type -> Int
    go n (AppT (AppT ArrowT _) rest) =
        go (n+1) rest
    go n (ForallT _ _ rest) =
        go n rest
    go n _ =
        n
```

Neat! We can pattern match on these just like ordinary Haskell values. Well, they *are* ordinary Haskell values, so that makes perfect sense.

Lastly, we'll need a function that gets the type from an `Info`. Not all `Info` have types, so we'll encode that with `Maybe`.

```haskell
getType :: Info -> Maybe Type
getType info =
    case info of
        ClassOpI _ t _ _ ->
            Just t
        DataConI _ t _ _ ->
            Just t
        VarI _ t _ _ ->
            Just t
        TyVarI _ t ->
            Just t
        _ ->
            Nothing
```

Alright, we're ready to get started on that `decForFunc` function! We'll go ahead and fill in what we know we need to do:

```haskell
decForFunc :: Name -> Name -> Q Dec
decForFunc reader fn = do
    info <- reify fn
    arity <-
        case getType info of
            Nothing -> do
                reportError "Unable to get arity of name"
                return 0
            Just typ ->
                pure $ functionLevels typ
    -- ...
    return (FunD fnName [Clause varPat (NormalB final) []])
```

Arity acquired. Now, we'll want to get a list of new variable names corresponding with the function arguments. When we want to be hygienic with our variable names, we use the function `newName` which creates a totally unique variable name with the string prepended to it. We want `(arity - 1)` new names, since we'll be using the bound value from the reader function for the other one. We'll also want a name for the value we'll bind out of the lambda.

```
varNames <- replicateM (arity - 1) (newName "arg")
b <- newName "b"
```

Next up is the new function name. To keep a consistent API, we'll use the same name as the one in the actual package. This will require us to import the other package qualified to avoid a name clash.

```
let fnName = mkName . nameBase $ fn
```

`nameBase` has the type `Name -> String`, and gets the non-qualified name string for a given `Name` value. Then we `mkName` with the string, giving us a new, non-qualified name with the same value as the original function. This might be a bad idea? You probably want to provide a unique identifier. However, keeping the names consistent can be helpful for discovery.

Next up, we'll want to apply the `(>>=)` function to the `reader`. We'll then want to create a function which applies the `bound` expression to a lambda. Lambdas have an LamE[11] constructor in the `Exp` type. They take a `[Pat]` to match on, and an `Exp` that represents the lambda body.

```
bound  = AppE (VarE '(>>=)) (VarE reader)
binder = AppE bound . LamE [VarP b]
```

So `AppE bound . LamE [VarP b]` is the exact same thing as `(>>=) reader (\b -> ...)`! Cool.

Next up, we'll need to create `VarE` values for all of the variables. Then, we'll need to apply all of the values to the `VarE fn` expression. Function application binds to the left, so we'll have:

[11] https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH.html#v:LamE

```
fn       ~                   VarE fn
fn a     ~              AppE (VarE fn) (VarE a)
fn a b   ~        AppE (AppE (VarE fn) (VarE a)) (VarE b)
fn a b c ~ AppE (AppE (AppE (VarE fn) (VarE a)) (VarE b)) (VarE c)
```

This looks just like a left fold! Once we have that, we'll apply the fully applied `fn` expression to `VarE 'liftIO`, and finally bind it to the lambda.

```
varExprs    = map VarE (b : varNames)
fullExpr    = foldl AppE (VarE fn) varExprs
liftedExpr = AppE (VarE 'liftIO) fullExpr
final       = binder liftedExpr
```

This produces our `(>>=) reader (\b -> fn b arg1 arg2 ... argn)` expression.

The last thing we need to do is get our patterns. This is the list of variables we generated earlier.

```
varPat = map VarP varNames
```

And now, the whole thing:

```
deriveReader :: Name -> DecsQ
deriveReader rd =
    mapM (decForFunc rd)
        [ 'destroyUserBackend
        , 'housekeepBackend
        , 'getUserIdByName
        , 'getUserById
        , 'listUsers
        , 'countUsers
        , 'createUser
        , 'updateUser
        , 'updateUserDetails
        , 'authUser
        , 'deleteUser
        ]

decForFunc :: Name -> Name -> Q Dec
decForFunc reader fn = do
    info <- reify fn
    arity <-
        case getType info of
            Nothing -> do
                reportError "Unable to get arity of name"
                return 0
            Just typ ->
                pure $ functionLevels typ
    varNames <- replicateM (arity - 1) (newName "arg")
    b <- newName "b"
    let fnName     = mkName . nameBase $ fn
        bound      = AppE (VarE '(>>=)) (VarE reader)
        binder     = AppE bound . LamE [VarP b]
        varExprs   = map VarE (b : varNames)
        fullExpr   = foldl AppE (VarE fn) varExprs
        liftedExpr = AppE (VarE 'liftIO) fullExpr
        final      = binder liftedExpr
        varPat     = map VarP varNames
    return $ FunD fnName [Clause varPat (NormalB final) []]
```

And we've now metaprogrammed a bunch of boilerplate away!

We've looked at the docs for Template Haskell, figured out how to construct values in Haskell's AST, and worked out how to do some work at compile time, as well as automate some boilerplate. I'm excited to learn more about the magic of defining quasiquoters and more advanced Template Haskell constructs, but even a super basic "build expressions and declarations using data constructors" approach is useful.