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

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[1], 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.

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

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

Acknowledgements

Thank you to Sandy Maguire for inspiring me to write a book in the first place!

Thank you to Chris Allen for the encouragement and example in writing Haskell material. I would not have been able to learn Haskell without the material you collected and created.

Thank you to Michael Snoyman for making my Haskell career possible. Without `stack` and the other FPComplete libraries, I wouldn't have had the luck and success to be a Haskeller today.

Thank you to Jordan Burke for giving me my first real software internship.

Thank you to Andrew Martin for hiring me as a Haskell intern.

Thank you to Ben Kovach for getting me that first Haskell job, and then inspiring me to apply to Mercury.

Thanks to the following folks who provided suggestions on the LeanPub forums: jakalx, arpl.

Thanks to Jade Lovelace, who read through an early version of the book and provided great feedback.

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[2]
- You have a complexity budget[3]

[2] https://www.shimweasel.com/2018/08/25/novelty-budgets
[3] 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:

```
1 map :: (a -> b) -> [a] -> [b]
2 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:

```
1  data User = User
2      { userId :: UserId
3      , userName :: Text
4      , userEmail :: EmailAddress
5      , userIsAdmin :: Bool
6      }
```

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:

```
1  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!

```
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 `Maps`. 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.

```
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 `Maps`!

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 docu-

mentation 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

II Application Structure

5. The Beginning

> How do I structure an application in Haskell?

This is one of the most common questions I receive. Haskell is a fundamentally different programming language than what many people are accustomed to, and people reasonably suspect that application structure may also be different. I have personally found that Haskell's easy refactoring and re-architecting qualities make application structure less important than other languages, where widespread changes are difficult to safely accomplish. Structure still matters, but possibly a bit less than other languages.

We will keep the Novelty and Complexity principles in mind here. Overblown application structure is something we see in other languages to overcome language-level difficulties that Haskell may simply not have. As a result, I am a fan of keeping things as utterly simple as possible, until I am feeling a certain amount of pain caused by that simplicity.

We'll start simple - as simple as possible. Then we will begin adding complexity. With each bit of complexity, I'll explain my perception of the trade-offs, and my experiences of the trade-offs. Each style also has a tendency to organically evolve in a few ways, some of which can cause issues.

All Haskell applications must eventually boil down to this function:

```
1  main :: IO ()
```

So we must accept the complexity of `IO` to do anything at all. Haskell applications often have the basic form:

```
main :: IO ()
main = do
    environment <- collectInformation
    result <- app environment
    communicate result

collectInformation :: IO Env
app                :: Env -> IO Result
communicate        :: Result -> IO ()
```

This is a batch application. We run the program, and it collects information from the environment that it needs. This can include environment variables, command-line arguments, standard input, reading files, making HTTP requests, etc. `app` does the meat of the work, processing the necessary environment into the final result we care about. Finally, `communicate` takes this result and makes it useful to the outside world.

Server applications aren't much different, conceptually. We have some initial application configuration that must be acquired. Then we need to have some request-specific information that we want to act on. We return the result to the client. When the entire process is done, we close the server out.

```
main :: IO ()
main = do
    appEnv <- collectAppInformation
    finalResult <- serve appEnv withClient
    communicate finalResult

withClient
    :: Env -> ClientInfo -> IO ()
withClient appEnv clientInfo = do
    x <- collectClientInfo clientInfo
    response <- makeClientResponse appEnv clientInfo x
    communciateClient appEnv clientInfo response

serve
    :: Env
```

```
16      -> (Env -> ClientInfo -> IO ())
17      -> IO Result
```

We've taken a live, responsive program design - a client/server program - and we've transformed it into a batch program that runs for each client request. We also keep the structure of:

1. Acquire Input
2. Process Input
3. Communicate Result

At some level, this is how most business applications work. Haskell excels at this structure. Working with plain `IO` functions can carry you quite far. And, since the compiler can guide large-scale refactors, changing the architecture is relatively cheap if you really need to later.

This style is simple. No language extensions are required. Exception handling and state management is straightforward, since "parameter passing" is how everything is accomplished. GHC is excellent at optimizing the `IO` type in the generated code, so the program is efficient.

This style is explicit and verbose. If a function needs access to some piece of data, then you simply pass it in as a parameter. Understanding how to call any given function is easy: supply all the necessary parameters! Knowing where things come from is also easy: the parameter list describes the complete set of requirements for a function.

"Explicit and verbose" has problems. Every added parameter to a function requires an update at every use-site of the function. This produces large diffs for potentially small changes. Additionally, some functions might take a large amount of parameters. This can produce really long lines of code which are annoying to type and read. There's no guarantee that parameters are passed in any given order, so it can become tedious to plumb parameters around if this is not consistent.

We run into a classic issue: the "signal to noise ratio." This approach has a large amount of 'volume' with changes, and it's difficult to tell whether any given change is signal or noise. Noisy diffs can cause real production problems. At one company I worked at, a change in the logging style was done. This PR touched every function that logged in order to add a

parameter for logging - it was over 3000 lines of additions and deletions. Due to this noise, a fatal bug slipped in, which crashed production. The PR was combed over by myself and another competent engineer, and neither of us found the bug.

"What is noise?" is a great question. In my opinion, "noise" is any part of the diff that is not directly related to the functionality being implemented. Let's look at what a change in the logging example above might have looked like. The following code currently uses type class methods to log, and these are defined for `App`.

```
myFunc :: Int -> Char -> App Text
myFunc i c = do
    x <- otherFunc i
    withFoo x $ \foo ->
        tellTheWorld foo c
```

Now, we're going to use an explicit logger.

```
-myFunc :: Int -> Char -> App Text
+myFunc :: Logger -> Int -> Char -> App Text
-myFunc i c = do
+myFunc logger i c = do
-    x <- otherFunc i
+    x <- otherFunc logger i
     withFoo x $ \foo ->
-        tellTheWorld foo c
+        tellTheWorld logger foo c
```

Adding a parameter to each log-requiring function tells us very little of interest - `tellTheWorld` uses the logger, as does `otherFunc`, but `withFoo` does not. However, we had to edit every single call-site of every one of these functions. Indeed, the above diff doesn't even compile - `tellTheWorld` *actually* should accept the `foo` before the `logger`.

The pain points around noisy diffs and big parameter lists are bad enough that you will try to solve them. The most obvious way is to stuff the "common" parameters into a datatype, which you pass around.

5.1 Abstraction for Mocking

You may want to make some behavior in your application abstract, so that you can easily test it. Suppose you have some business logic that depends on an external HTTP service, but you don't want to use that service in testing or for local development.

```
doWork :: String -> IO ()
doWork request = do
    response <- makeExternalCall request
    processResponse response
```

We want to mock `makeExternalCall`. To do this, we pull the function into the parameter list.

```
doWorkAbstract
    :: (String -> IO Response)
    -> String
    -> IO ()
doWorkAbstract mockExternalCall request = do
    response <- mockExternalCall request
    processResponse response

doWork :: String -> IO ()
doWork = doWorkAbstract makeExternalCall
```

This technique is easy to apply on demand, as you need it, without complicating the rest of your codebase. We'll cover this technique more in the "Invert Your Mocks!" chapter.

5.2 Forward Compatibility

I like to minimize the effort required to change the application. Haskell makes extensive changes relatively straightforward, so these changes are often *possible* and even feasible. I still prefer to avoid needing to

make the change at all. I have rarely regretted writing internal types and functions that wrap external types.

As an example, we might anticipate wanting to change away from `IO` as the main type of our application. We could write a type synonym.

```
type App = IO

doWork :: String -> App ()
```

However, this doesn't prevent us from using `IO` directly. If we ever modify the `App` type, then we'll need to modify each direct use of `IO`. We can preempt this by writing a `newtype` and deriving `MonadIO`:

```
import Control.Monad.IO.Class (MonadIO(..))

newtype App a = App { unApp :: IO a }

instance MonadIO App where
    liftIO ioAction =
        App ioAction

-- or, with GeneralizedNewtypeDeriving:
newtype App a = App { unApp :: IO a }
    deriving newtype
        (Functor, Applicative, Monad, MonadIO)
```

This requires us to write `liftIO` before every use of a plain `IO` function. This is extra boilerplate, but it protects us from needing to insert those calls whenever we change `App`. The trade-off is definitely doing a bit of work now, or possibly doing a lot of work all at once later.

Another option is to have a module of pre-lifted functions.

```
module App.LiftedFunctions where

getCurrentTime :: App UTCTime

openFile :: IOMode -> App Handle

getHttpRequest :: String -> App Response
```

This pattern makes it easy to know exactly how you depend on `IO` and external libraries. You can even skip the `MonadIO` type class. What's more, you can *forbid* people from using it with a `TypeError` instance:

```
instance (TypeError UseAppInstead) => MonadIO App where
    liftIO = undefined

type UseAppInstead = 'Text "Use App instead"
```

Writing a custom `TypeError` is out of scope for this chapter, but we'll cover it later. This facility allows you to explain to developers why things are the way they are, which is great for onboarding new developers.

A `newtype` around `IO` also makes it possible to write custom instances for type classes on your application type. This is a common way to opt-in to library features.

The general practice around writing internal wrappers for external types can help with switching libraries and growing your code. It's easy to let a library get deeply coupled into your codebase, but you should avoid that where possible.

5.3 `AppEnvironment`

Your codebase is in the previous state: `IO` functions everywhere, passing whatever parameters you need manually. You are feeling the pain points mentioned above, and you want a solution. So you make a distinction between the "common" parameters and the "specific" parameters, and package the common ones up in a new datatype. Your old code might look like this:

```
someFunction
    :: DbConn
    -> Logger
    -> HttpManager
    -> String
    -> Int
    -> IO ()
```

The `DbConn`, `Logger`, and `HttpManager` values are ubiquitous - almost every function in your codebase relies on them to do basic work. But the exact order isn't consistent, and the PR that added the `HttpManager` was way more annoying that it needed to be. So we're going to write a datatype and package it up:

```
data AppEnvironment = AppEnvironment
    { dbConn :: DbConn
    , logger :: Logger
    , httpManager :: HttpManager
    }

someFunction
    :: AppEnvironment
    -> String
    -> Int
    -> IO ()
```

Now, instead of passing the parameters separately, we pass the single "combined" parameter. This reduces the explicitness and verbosity of our code. It also adds some complexity - we now have to make a decision about *how* to pass parameters to functions! But this also reduces complexity - we now have a single place to put application concerns.

Reducing verbosity and explicitness has trade-offs. The diffs in this style of code are significantly less noisy, and reviewing changes to the codebase is easier. Writing, editing, and experimenting with the code is easier, since each change is more directly relevant. If we need to propagate a bit of information to many places in the code, we can put it into the `AppEnvironment` type, and we will only see diffs where that

is relevant. For new people to the codebase, it can be difficult to know where to get a certain item. Programmers will eventually learn what is stored in the `AppEnvironment` type, but this is an extra bit of information for them to learn to be productive.

By creating an `AppEnvironment` type, we give a name to the most common aspects of the application. This gives us the ability to write Haddock documentation for the environment. The liability of 'hidden' information can become an asset by giving us a place to name and document patterns and practices.

When to use the Environment

> When should I put a value in the environment?

This is an excellent and common question when you have an `AppEnvironment` type. Both the environment and the parameter list for a function are equivalent. I've developed a few clarifying questions:

1. Is the value being passed to most functions?
2. Does the value rarely change?
3. Do functions often pass the value unchanged to other functions, without using it directly?
4. Is the value created at environment startup and then used throughout?

If you answered "yes" to any of the above, then including the type in an `Environment` is probably a good idea. Putting a value in the environment hides it from the explicit parameter list and makes it much more difficult to modify. It also makes it easier to centralize documentation for the meaning of that parameter. Rather than commenting on the meaning in every location, we can centralize the commentary at the datatype definition.

Nesting Environments

I have rarely seen an application with a single environment. Most applications have multiple possible environments, and they are usually

'nested'. I might have an `ApplicationEnvironment` which contains stuff my entire app needs. But then I'll also have a `RequestEnvironment` which contains stuff specific to a single request.

We're immediately confronted with a choice: do we nest the `ApplicationEnvironment` directly into the `RequestEnvironment`, or do we pass them separately?

```
1  data RequestEnvironment = RequestEnvironment
2      { ...
3      }
4
5  someFunction
6      :: AppEnvironment
7      -> RequestEnvironment
8      -> IO ()
9
10 -- vs:
11 data RequestEnvironment = RequestEnvironment
12     { appEnvironmnet :: AppEnvironment
13     , ...
14     }
15
16 someFunction
17     :: RequestEnvironment
18     -> IO ()
```

This is similar to the above question - "When to use the Environment." There is a subtle difference in the meaning of these two approaches, however. If we do not nest the environments, then this implies that there are two separate environments with some amount of overlap. It implies the existence of functions that only exist in the `RequestEnvironment`, totally outside of our `AppEnvironment`.

Is it possible to do a `Request` without the larger `AppEnvironment`? If it is, then maybe combining the two types is premature.

If we nest the environment types, then it implies that *all* `RequestEnvironments` are specializations of the `AppEnvironment`. There is a small

amount of additional noise from this approach. We must call `appEnvironment` on the `RequestEnvironment` to call any `AppEnvironment` functions.

```
appFunction
    :: AppEnvironment
    -> IO ()

requestFunction
    :: RequestEnvironment
    -> IO ()
requestFunction requestEnv = do
    putStrLn "doing work"
    appFunction (appEnvironment requestEnv)
    putStrLn "work is done"
```

Contrast this with the two parameter approach:

```
requestFunction
    :: AppEnvironment
    -> RequestEnvironment
    -> IO ()
requestFunction appEnv requestEnv = do
    putStrLn "doing work"
    appFunction appEnv
    putStrLn "work is done"
```

Generalizing Environments

The ambitious reader will note that we can use type classes to solve the above issue with nesting environments. We can write a type class `HasAppEnvironment` and require that instead of a concrete `AppEnvironment`:

```haskell
class HasAppEnvironment a where
    getAppEnvironment :: a -> AppEnvironment

appFunction
    :: (HasAppEnvironment a)
    => a
    -> IO ()

data RequestEnvironment = RequestEnvironment
    { appEnvironment :: AppEnvironment
    }

instance HasAppEnvironment RequestEnvironment where
    getAppEnvironment = appEnvironment

requestFunction
    :: RequestEnvironment
    -> IO ()
requestFunction reqEnv = do
    putStrLn "doing work"
    appFunction reqEnv
    putStrLn "work is done"
```

I generally recommend against this approach. Type class polymorphism is a great way to introduce confusing type errors into a project.

5.4 The `ReaderT` Pattern

The next big step in complexity is to add our first monad transformer. The `ReaderT` monad transformer looks like this:

```haskell
newtype ReaderT r m a = ReaderT
    { runReaderT :: r -> m a
    }

ask :: ReaderT r m r
ask = ReaderT (\r -> pure r)
```

The type in the `r` parameter is passed implicitly, and we call `ask` to get access to it. This reduces the noise in passing the `AppEnvironment` parameter directly, which tends to make the code a bit more readable.

If you've adopted the suggestion in "Forward Compatibility," then you'll have an app type like this:

```haskell
newtype App a = App
    { unApp :: ReaderT AppEnvironment IO a
    }
    deriving (Functor, Applicative, Monad, MonadReader AppEnvironment)

runApp :: AppEnvironment -> App a -> IO a
runApp appEnv action =
    runReaderT (unApp action) appEnv
```

This approach is documented in The ReaderT Design Pattern[1].

Monad transformers increase the complexity of the codebase significantly. `ReaderT` is the easiest one to understand, but we run into a number of issues. It becomes difficult to call certain kinds of `IO` actions directly. There are two general solutions: `MonadBaseControl`[2] and `UnliftIO`[3]. `MonadBaseControl` is deeply complicated and difficult to use. `UnliftIO` is rather restrictive, and only supports transformers that can be translated to `ReaderT`.

[1] https://www.fpcomplete.com/blog/2017/06/readert-design-pattern/
[2] https://lexi-lambda.github.io/blog/2019/09/07/demystifying-monadbasecontrol/
[3] https://github.com/fpco/unliftio/tree/master/unliftio#readme

5.5 Embed, don't Stack

You may be tempted to add a ton of monad transformers in your app type. One monad transformer for every effect you might care about. Every effect represented by a monad transformer.

```
newtype App a = App
    { unApp ::
        ExceptT
            AppErr
            (StateT AppState (ReaderT AppEnv (LoggingT IO)))
            a
    }
```

This is a mistake. Each monad transformer incurs significant complexity in implementation and locks you in.

Fortunately, `ReaderT Env IO` is powerful enough that we don't need any additional transformers. Every additional transformer provides more choices and complexity, without offering significant extra power. We can mimic `StateT` by putting an `IORef` into the `Env`.

```
newtype StateRef s m a = StateRef (ReaderT (IORef s) m a)

instance (MonadIO m) => MonadState s (StateRef s m) where
    get a = StateRef $ do
        ref <- ask
        liftIO $ readIORef ref
    put a = StateRef $ do
        ref <- ask
        liftIO $ writeIORef ref a
```

We can mimic `ExceptT` by throwing exceptions in `IO`. I cover in "The Trouble With Typed Errors" why this is fine, and why `ExceptT` has many problems with correctness and safety.

We can write an instance of `MonadLogger` directly to avoid needing the `LoggingT` type.

```
instance MonadLogger App where
    monadLoggerLog = appLog
```

The general pattern that I recommend is to *embed* things into your App type.

```
runSql   :: Database a -> App a

runRedis :: Redis a -> App a
```

Interleaving effects can dramatically complicate code. Factoring your code such that effects are run independently of each other makes for a simpler time to understand and read the code. You also avoid many common issues.

For example, suppose `runSql` implies a database transaction. You may want to run `Redis` actions inside a database action.

```
writeRedis :: Key -> Value -> Redis ()

liftRedis :: Redis a -> Database a

databaseTransaction = do
    someRecord <- get key
    liftRedis $ writeRedis (id someRecord) (val someRecord)
    insert newRecord
    liftRedis $ writeRedis (id newRecord) (val someRecord)
    insert otherRecord
```

What happens if `insert otherRecord` fails - possibly due to a foreign key constraint, or a violated uniqueness constraint? Then the effect of `insert newRecord` will be rolled back with the rest of the transaction. However - we can't "undo" the `writeRedis` action. So our Redis service will act like `insert newRecord` has succeeded while our `Database` will not have it.

A type - like `Database`, `App`, or `Redis` - is like a mini-language. Many small, simple languages are easier than a single mega-language to rule

them all. Embedding these small languages into a larger one is relatively straightforward. By embedding our effects, rather than stacking them, we have a much easier time structuring our code.

6. Three Layer Haskell Cake

There are many perspectives and choices on how to design a Haskell application. Choosing a single approach is understandably difficult. Should I use plain monad transformers, `mtl`, just pass the parameters manually and use `IO` for everything, the `ReaderT` design pattern[1], free monads[2], free*r* monads, some other kind of algebraic effect system?!

Each approach has pros and cons. Instead of sticking with one technique for everything, let's instead leverage many techniques where they shine. Lately, I've been centering on an application design architecture with roughly three layers.

Trying to satisfy every need with a single technique is bound to fail. Some thoughts are awkward to encode or test in `IO`. Some techniques don't have sufficient control to provide for every use case. And finally, some techniques are just hard to use, no matter how powerful they might be!

This approach is all about figuring out the best technique to solve the problem, and embedding it in the larger context. Layer 1 is our foundation. Everything eventually gets interpreted into it. Items in Layer 2 and 3 are ignorant of this foundation - and they are then embedded in it.

This is another presentation of the "functional core, imperative shell"[3] model of programming, so if you're familiar with that, you might enjoy this take on it.

6.1 Layer 1: Imperative Programming

The first layer is a thin veneer over `IO`.

[1] https://www.fpcomplete.com/blog/2017/06/readert-design-pattern
[2] http://www.parsonsmatt.org/2017/09/22/what_does_free_buy_us.html
[3] https://www.destroyallsoftware.com/talks/boundaries

```haskell
newtype App a
    = App
    { unApp :: ReaderT YourStuff IO a
    }
    deriving newtype
        (Functor, Applicative, Monad, etc)
```

The ReaderT Design Pattern[4]. This is the foundation of the app: what everything eventually gets interpreted in. This type is the backbone of your app. This layer defines how the upper layers work, and for handling operational concerns like performance, concurrency, etc. For some components, you carry around some information or state (consider `MonadMetrics`[5] or `katip`'s[6] logging state/data).

At IOHK, we had a name for this kind of thing: a "capability". We have a big design doc on monads[7], and the doc goes into what makes something a capability or not. IOHK has since deleted this design document and decided that it wasn't good to follow.

You can write all of your code in this layer, without doing anything in the upper layers. Indeed, some simple applications may not require anything more complex.

Testing

If you need to test items in this layer, then you are writing integration tests. Integration tests can be slow and cumbersome, but they provide confidence that the pieces fit together correctly. Ideally, you won't need to write many tests for this layer. The composition of well-tested code with great unit and property tests should hopefully be correct enough that integration testing doesn't provide much value.

If you find yourself writing tests for Imperative Programming code, consider instead shifting the logic into one of the higher layers. Testing pure functions or otherwise abstracted code is easier and faster.

[4] https://www.fpcomplete.com/blog/2017/06/readert-design-pattern
[5] https://hackage.haskell.org/package/monad-metrics
[6] https://hackage.haskell.org/package/katip-0.5.2.0/docs/Katip.html
[7] https://github.com/parsonsmatt/cardano-sl/blob/10e55bde9a5c0d9d28bca25950a8811407c5fc8c/docs/monads.md

How do you shift something up to a higher layer? The chapter "Inverting your Mocks" covers a general strategy for accomplishing this, but the general routine is:

- Factor the *input effects* of code out as function parameters
- Represent the *output effects* as data returned from pure functions

If I had to give a name to this layer, I'd call it the "orchestration" layer. All of the code has been composed, and now we're arranging it for a real performance.

This layer works well when imperative programming is the most suitable paradigm to approach the problem with. After all, according to Simon Peyton Jones, Haskell is the world's finest imperative language. We might as well take advantage of that where appropriate.

6.2 Layer 2: Object Oriented Programming

This layer is primarily good for providing a limited interface to a complex, external API. We abstract that into a function parameter, effect, or capability. Limiting the interface and abstracting it into a runtime parameter allows us to swap in different tested implementations.

Here, we're mostly interested in abstracting and encapsulating external services and dependencies. The most convenient way I've found to do this are `mtl` style classes, implemented in terms of domain resources or effects. This is a trivial example:

```
1  class MonadTime m where
2      getCurrentTime :: m UTCTime
```

`MonadTime` is a class that I might use to "purify" an action that uses IO only for the current time. Doing so makes unit testing a time based function easier. However – this isn't a great use for this. The best "pure" instance of this is

```
instance MonadTime ((->) UTCTime) where
    getCurrentTime = id
```

And, if you've factored your effects out, this will already be done for you. Furthermore, it would actually be quite difficult to write a realistic `MonadTime` mock. One law we might like to have with `getCurrentTime` is that:

```
timeLessThan = do
    x <- getCurrentTime
    y <- getCurrentTime
    pure (x < y)
```

A pure implementation returning a constant time would fail this. We could have a `State` with a random generator and a `UTCTime` and add a random amount of seconds for every call, but this wouldn't really make testing any easier than just getting the actual time. Getting the current time is best kept as a Layer 1 concern - don't bother abstracting it.

The real benefit to introducing a `MonadTime` class and constraint is to avoid `MonadIO`. `MonadIO` allows you to do almost anything, and you may prefer to restrict the function to *only* getting the current time.

So, if `MonadTime` isn't a good "external service" to swap out, what is? In my opinion, a service is worth abstracting if you can limit the API to a sufficiently small set and if the service is in any way frustrating or annoying to setup in a development or test environment. Both of these components are inherently subjective. If your application depends tightly on PostgreSQL features, then you probably can't easily mock it, regardless of how frustrating it may be to setup a Postgres instance locally.

A more realistic example from a past codebase is `MonadLock`. This type class defined an interface for acquiring a `Lock` based on a `Key` in a distributed computer network. This was used to ensure that only one server was working on a task at any given point when the task delivery mechanism wasn't guaranteed to provide unique messages.

```haskell
class Monad m => MonadLock m where
    acquireLock
        :: NominalDiffTime
        -> Key
        -> m (Maybe Lock)
    renewLock
        :: NominalDiffTime
        -> Lock
        -> m (Maybe Lock)
    releaseLock
        :: Lock
        -> m ()
```

This class describes logic around implementing distributed locks. The production instance talked to a Redis instance. Setting up Redis for dev/test sounded annoying, so I implemented a testing mock that held an `IORef (Map Key Lock)`.

Another good class is a simplified DSL for working with your data. In OOP land, you'd call this your "Data Access Object." It doesn't try to contain a full SQL interpreter, it only represents a small set of queries/data that you need.

```haskell
class (Monad m) => AcquireUser m where
    getUserBy :: UserQuery -> m [User]
    getUser :: UserId -> m (Maybe User)
    getUserWithDog :: UserId -> m (Maybe (User, Dog))

class AcquireUser m => UpdateUser m where
    deleteUser :: UserId -> m ()
    insertUser :: User -> m ()
```

We can use this class to provide a mock database for testing, without having to write an entire SQL database mocking system. These classes also come in handy because you can swap out the underlying production implementations. Suppose you have a microservices system going on, and `AcquireUser` is done through an HTTP API. Suddenly, your boss is convinced that monoliths are king, and gives you One Large Server To

Rule Them All. Now your HTTP API has direct database access to the underlying data – you can make SQL requests instead of HTTP! How wonderful. This may seem contrived, but it was a big deal when we did exactly this at my first job.

You don't have to use type classes for this. Indeed, type classes have a lot of downsides for abstraction. If there are multiple reasonable behaviors for a givne type, then type classes make it difficult to select the instance you want. In this case, a data type with function fields may be more appropriate. You can translate a type class into a data type by turning the methods into function fields. Gabriella Gonzalez's blog post "Scrap your type classes"[8] goes into great detail, but an example transformation is here:

```
class Monad m => MonadLock m where
    acquireLock
        :: NominalDiffTime
        -> Key
        -> m (Maybe Lock)
    renewLock
        :: NominalDiffTime
        -> Lock
        -> m (Maybe Lock)
    releaseLock
        :: Lock
        -> m ()

data LockService = LockService
    { acquireLock :: NominalDiffTime -> Key -> IO (Maybe Lock)
    , renewLock :: NominalDiffTime -> Lock -> IO (Maybe Lock)
    , releaseLock :: Lock -> IO ()
    }
```

One especially nice thing about this transformation is that you can capture runtime parameters easily. Suppose we want to close over a `RedisConnection` in order to provide this. With the type class instance, we must provide the value through the types, typically with a `ReaderT` parameter:

[8] https://www.haskellforall.com/2012/05/scrap-your-type-classes.html

```
instance MonadIO m => MonadLock (ReaderT RedisConnection m) where
    acquireLock time key = do
        redisConn <- ask
        liftIO $ do
            runRedis ...

    {- etc... -}
```

With the record data type, we can provide that as a runtime parameter, without involving any types:

```
redisLock :: RedisConnection -> LockService
redisLock redisConnection =
    LockService
        { acquireLock = \time key -> do
            rundRedis redisConnection ...
        , {- etc... -}
        }
```

If you want to modify the behavior of the service, you can easily manipulate the runtime value. Manipulating the type class behavior is more tricky, since you need to call the service at a different type.

```
lockForTenTimesLonger :: LockService -> LockService
lockForTenTimesLonger LockService {..} =
    LockService
        { acquireLock = \time key ->
            acquireLock (time * 10) key
        , renewLock = \time lock ->
            renewLock (time * 10) lock
        , ..
        }

withLongerLocks :: App a -> App a
withLongerLocks =
    local (\appEnvironment ->
```

```
        appEnvironment
            { appEnvironmentLockService =
                lockForTenTimesLonger $
                    appEnvironmentLockService appEnvironment
            }
        )
-- vs,

newtype TenTimesLonger m a = TenTimesLonger (m a)

instance MonadLock m => MonadLock (TenTimesLonger m) where
    acquireLock time key =
        TenTimesLonger $ acquireLock (time * 10) key
    {- etc... -}
```

In order for `withLongerLocks` to work with the type class, we have to somehow "swap out" the type class instance that is getting selected for that inner action. This isn't trivial to do.

On the other hand, a type class instance is canonical - if you know the type something is called at, then you can easily determine the implementation that is being used. The data type approach makes it difficult to know exactly what code is being called without tracing the call graph.

These are higher level than `App` and delimit the effects you use; but are ultimately lower level than real business logic. You might see some `MonadIO` in this layer, but it should be avoided where possible. This layer should be expanded on an as-needed (or as-convenient) basis. As an example, implementing `MonadLock` as a class instead of directly in `AppT` was done because using Redis directly would require that every development and test environment would need a full Redis connection information. That is wasteful so we avoid it. Implementing `AcquireModel` as a class allows you to omit database calls in testing, and if you're real careful, you can isolate the database tests well.

DO NOT try to implement `MonadRedis` or `MonadDatabase` or `MonadFilesystem` here. That is a fool's errand. Instead, capture the tiny bits of your domain: `MonadLock`, `MonadModel`, or `MonadSpecificDataAcquisition`. The smaller your domain, the easier it is to write mocks and tests for it. You probably don't want to write a SQL database, so don't –

capture the queries you need as methods on the class so they can easily be mocked. Alternatively, present a tiny query DSL that *is* easy to write an interpreter for.

This layer excels at providing swappable implementations of external services. This technique is still quite heavy-weight: `mtl` classes require tons of `newtypes` and instance boilerplate. Other mocking techniques aren't much better. This layer should be as thin as possible, preferring to instead push stuff into Layer 3.

This layer is essentially "object oriented programming." We have objects - interfaces with messages we care about - and we swap in the object we want at runtime to have the behavior we care about. Code in this layer can be difficult to understand and follow. A nice property about Layer 1 code is that you can always just jump to the definition of the function and know how it is defined. But once you're in an "object", knowing the definition of the term is not sufficient. You have to trace where the value comes from and what modifications happen along the way.

6.3 Layer 3: Functional Programming

Business logic. This should be entirely pure, with no `IO` component at all. This should almost always just be pure functions and relatively simple data types. Reach for *only* as much power as you need - and you often need much less than you think at first.

All the effectful data should have been acquired beforehand, and all effectful post-processing should be handled afterwards. The chapter "Invert Your Mocks" goes into detail on ways to handle this. If you need streaming, then you can implement "pure" `conduit` or `pipes` with a type signature like this:

```
1 pureConduit
2     :: Monad m
3     => ConduitT i m o r
```

This expresses no dependency on *where* the data comes from, nor on how the output is handled. We can easily run it with mock data, or put it in

the real production pipeline. It is *abstract* of such concerns. As a result, it's lots of fun to test.

If the result of computation needs to perform an effect, then it is useful to encode that effect as a datatype. Free monads are a technique to encode computation as data, but they're complicated; you can usually get away with a much simpler datatype to express the behavior you want. Often times, a simple non-recursive sum type "command" suffices as an interface between a pure function and an effectful one. A list of commands adds a lot of flexibility without dramatically complicating things.

Before you jump to monads, consider: would a `Monoid` work to construct or read the data? If not, what about an `Applicative` interface? What about a limited recursive sum type, perhaps a GADT, that can express what I want to do?

Testing pure functions with easy data types as output is dreamlike and easy. This is the Haskell we know and love. Try to put as much of your code into the pleasant, testable, QuickCheck-able, type-verified bits as possible. If you manage to isolate your application like this, then you won't need to test your IO stuff (aside from the usual integration testing).

May all of your tests be pleasant and your software correct.

6.4 Examples

I get folks asking me for examples fairly regularly. Unfortunately, I haven't had time to write an OSS app using this technique. Fortunately, other folks have!

- Holmusk/three-layer[9]
- thomashoneyman/purescript-halogen-realworld[10]
- incoherentsoftware/defect-process[11] is a ~62kloc Haskell video game project that uses the Three Layer Cake

[9] https://github.com/Holmusk/three-layer
[10] https://github.com/thomashoneyman/purescript-halogen-realworld
[11] https://github.com/incoherentsoftware/defect-process

7. 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:

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

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

```
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.

7.1 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.

```
1 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.

```
1 doWork :: App ()
2 doWork = do
3     query <- runHTTP getUserQuery
4     users <- runDB (usersSatisfying query)
5     for_ users $ \user -> do
6         thing <- getSomething user
7         let result = compute thing
8         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.

```
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:

```
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.

7.2 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:

```
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.

```
-- 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:

```
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 `RealThings` that satisfy a condition.

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

```
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 `prints` 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 `ByteStrings` 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.

```
1  sourceUsers :: ConduitT () User App ()
2  sourceUsers = do
3      users <- lift $ do
4          query <- runHttp getUserQuery
5          runDB (usersSatisfying query)
6      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.

7.3 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
```

```
13          let result = compute thing
14          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:

```
1  doWork :: App ()
2  doWork =
3      doWorkAbstract
4          (runHTTP getUserQuery)
5          (\query -> runDB (usersSatisfying query))
6          (\user -> getSomething user)
7          (\key result -> runRedis (writeKey key result))
```

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

```
1  doWorkScribe :: Writer [String] ()
2  doWorkScribe =
3      doWorkAbstract getQ getUsers getSomething redis
4    where
5      getQ = do
6          tell ["getting users query"]
7          pure AnyUserQuery
8      getUsers _ = do
9          tell ["getting users"]
10         pure [exampleUser1, exampleUser2]
11     getSomething u = do
12         tell ["getting something for " <> show u]
```

```
13          pure (fakeSomethingFor u)
14      redis k v = do
15          tell ["wrote k: " <> show k]
16          tell ["wrote v: " <> show v]
```

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

7.4 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.

7.5 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:

```
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:

```
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:

```
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!

8. Project Preludes

In Haskell, the `Prelude` is a module that is implicitly imported. It provides a small set of default functions and types that you will probably find useful.

However, the default `Prelude` isn't perfect. The `Prelude` was mostly designed to make research papers and projects have pretty, concise syntax. Notably, the `Prelude` is not intended to be a robust and fully-featured standard library. That's not what we want as industrial developers, and many industrial projects will benefit from a custom Prelude-like module.

We can disable implicit import of the `Prelude` with the language extension `NoImplicitPrelude`. If we do that, nothing will be in scope by default. We need to import everything explicitly.

One way to improve productivity, documentation, safety, and convenience is with a custom, project-specific prelude.

8.1 Prelude Problems

The Haskell Prelude is old. Breaking changes to the Prelude break almost every Haskell program in existence. A change like the Semigroup Monoid Proposal[1] that added the `Semigroup` class as a superclass of the `Monoid` class broke almost every `Monoid` instance in Haskell. The Foldable/Traversable in Prelude Proposal[2] was so controversial that one person resigned from his position[3]. That proposal generalized the types of functions like `length :: [a] -> Int` and `foldr :: (a -> b -> b) -> b -> [a] -> b` to use the `Foldable` or `Traversable` type classes instead of the concrete type for lists.

[1] https://www.reddit.com/r/haskell/comments/39tumu/make_semigroup_a_superclass_of_monoid/

[2] https://wiki.haskell.org/Foldable_Traversable_In_Prelude

[3] https://mail.haskell.org/pipermail/ghc-devs/2015-October/010068.html

Changing anything in the Prelude is so expensive and difficult that only the most pressing needs are considered. Minor problems and annoyances are allowed to stick around forever. Let's look at some of the problems the Prelude has.

Much of this design is "set in stone" by the Haskell Report[4] that officially defines much of the language and standard library. Changing it isn't even as simple as making a proposal on the right mailing list and getting buy-in - you have to come up with a whole new standard. The Haskell Report hasn't seen an update since 2010, though there was almost a new version for 2020 (it fizzled out).

Partial Functions

A "partial function" is one which throws an exception (or loops forever) on some input. We contrast this with a "total function," which is guaranteed to return in finite time with a valid value.

The most infamous partial function is `head`[5]:

```
1  head :: [a] -> a
2  head (a : _) = a
3  head [] = error "Prelude.head: Empty List"
```

Debugging the error is extremely annoying, because that message is almost all that you get in most cases. The `HasCallStack` machinery might give you a source location and context. Furthermore, we're writing *Haskell*! Should we have runtime errors on the case where a list is empty? That's such a common occurrence!

`head` isn't the only partial function exposed by the Prelude. `last`, `tail`, `init`, `minimum`, `maximum`, `foldr1`, and `scanr1` fail on the empty list with the same bad error message. Lists have an indexing operator `(!!) :: [a] -> Int -> a` which fails with a runtime exception if the index is negative or greater than the length of the list. `read :: String -> a` fails on any invalid input.

[4] https://www.haskell.org/onlinereport/haskell2010/

[5] Fortunately, between this chapter being written initially and publication, a proposal was accepted to add a `WARNING` pragma to `head` and `tail`.

The class `Num` has a method `fromInteger :: Integer -> a`. This method must fail with a runtime exception if the `Integer` isn't a valid value of type `a`. For example, the `Natural` type represents whole numbers greater than 0. `fromInteger (-5) :: Natural` fails with a runtime error. If you can't sensibly define `(+)`, `(*)`, and `negate` for the type - then you must throw a runtime exception when they're called.

The class `Enum` is used for sequence literals, like `[0 .. 10]`. The class throws runtime errors left and right. Derived instances (and instances for all types in `base`) throw an exception if you call `succ` on the "last value", or `pred` on the first value. The function `toEnum :: Int -> a` fails if the provided number is too big. There's a warning for `fromEnum :: a -> Int` that behavior is "implementation dependent" if the type is too large to fit in an `Int`, but I think it's illegal to have more than 9,223,372,036,854,775,807 constructors (the value of `maxBound :: Int`).

```
λ> succ False
True
λ> succ True
*** Exception: Prelude.Enum.Bool.succ: bad argument
λ> toEnum 3 :: Bool
*** Exception: Prelude.Enum.Bool.toEnum: bad argument
```

These partial functions are a big drag on productivity. They all throw exceptions via `error`, which means the exception type is `ErrorCall` so you only get a `String` of information. This makes it virtually impossible to catch or work with them programmatically.

Many functions don't throw exceptions, but they don't enable safe usage, either. The function `group` returns a nested list of elements grouped by equality. You're guaranteed that the inner lists all have at least one element in them, but this isn't documented in the types. A better `group` would have the type `[a] -> [NonEmpty a]`. `length` returns an `Int`, but a more appropriate type is `Word` or `Natural` because these types are guaranteed to be non-negative.

A custom prelude allows you to redefine these functions or hide them. You become free from the shackles of history.

Limited Utility

The `Prelude` is smaller than most standard libraries. A module that uses significant features in Haskell will possibly import `Control.Monad`, `Data.Maybe`, `Data.Traversable`, `Data.Foldable`, `Data.Either`, `Control.Applicative`, etc in order to work.

The `base` library is also small. You'll almost certainly want to include the `containers`, `text`, and `bytestring` packages for common tasks. Then you'll need to import `Data.Text` for an efficient text type, `Data.ByteString` for an efficient binary data type, `Data.Map` for a key/value dictionary, etc.

This lack is felt more keenly as the project grows. It's not uncommon to have 50-100 lines of imports, much of which brings in basic functionality. As new dependencies bring in new terms, you have to know how to import them to make them useful. Starting a new module becomes a major chore - you need to figure out exactly what you need to import to get started.

Some terms are kept in odd places. `mapM` is in the Prelude along with `traverse`. `mapM_` (a variant that throws away the return type) is also in the Prelude, but `traverse_` must be imported from `Data.Foldable` (not `Data.Traversable`). The function `mapMaybe` is in `Data.Maybe` which might make you think it has the type `(a -> b) -> Maybe a -> Maybe b`, but it actually has the type `(a -> Maybe b) -> [a] -> [b]`. It's an operation on lists - why is it in `Data.Maybe`? Meanwhile, most of the functions defined in `Data.List` aren't specific to lists, but are generalized over any `Foldable`. The functions `words` and `lines` (and their inverses) are in `Data.String`, but also reexported from `Data.List`.

A custom prelude allows you to bring in whatever you want. It can import all the types you need for your project. This can dramatically reduce drag on development time, especially if you're in the habit of making many modules. Small modules are easier to understand than large ones, so it's good to make new modules to break ideas up rather than adding to a megamodule for convenience's sake.

Conflicting Names

The designers of the Haskell language are so enamored with the identity function that they gave it one of the shortest possible names: `id`. In business programming, you use the `id` function somewhat rarely, but you frequently want to talk about identifiers. Since the Prelude exports `id`, you can't use `id` without causing a warning.

The Haskell designers also felt that most people would be doing math with their programs, so `log` is exported by default as the logarithm. This means you can't write `log` as a function that logs some output. The Prelude exports the term `asin` for arcsin. If you work with Amazon, then `asin` has another meaning - Amazon Standard Identification Number.

Haskell gives us both `map`, which is specific to lists, and `fmap`, which works for any `Functor`. The "Foldable Traversable in Prelude" proposal didn't cover `Functor`.

Maybe you're mad that `head` is partial, so you want to redefine it to be safe: `head :: [a] -> Maybe a`. You can't do this without hiding `head` from the Prelude. So you instead write `headMay`, or `headSafe`, or `listToMaybe`. The easy, cheap, convenient name is given to the unsafe function. This is unjust!

`String`

The default text type in Haskell is `String`, a linked list of characters. Literally:

```
1  type String = [Char]
```

For many text processing performance requirements, this is abysmal. `String` is baked in to many functions and core parts of the Haskell Prelude, so there's no avoiding it. Even a function like `readFile` returns a `String` of the file contents.

A linked list takes up quite a lot of space. Each element in a list takes up 3 machine words in memory, plus a single word for the terminal, plus the element itself. A `Char` takes up two words in memory, so each character takes up 5 words in memory. On modern 64-bit architectures, with 4

byte words, we're up to 20 bytes per character! A `Text` value uses 2 bytes per character with 6 words constant overhead. These memory numbers are pulled from Johann Tibell's[6] blog post from 2011, and may not be accurate anymore.

Lists are also linear in many operations that are common on textual algorithms, like `length`.

The idea that text is a linked list of characters is also not a safe assumption to make. We can break `String` with this relatively innocuous code:

```
λ> let oops = "schleißen"
λ> putStrLn oops
schleißen
λ> putStrLn (map toUpper oops)
SCHLEIßEN
```

Calling `toUpper` on ß should result in SS. The `text` library gets this right:

```
λ> Text.putStrLn oopsText
schleißen
λ> Text.putStrLn (Text.toUpper oopsText)
SCHLEISSEN
```

Laziness

Haskell's IO functions are lazy by default. If we run this sequence:

```
main = do
  contents <- readFile "Foo.txt"
  writeFile "Foo.txt" "written"
  putStrLn contents
```

Then we'll get an error from GHC:

[6] https://blog.johantibell.com/2011/06/memory-footprints-of-some-common-data.html

```
*** Exception: Foo.txt: openFile: resource busy
(file is locked)
```

`readFile` doesn't actually read anything before it returns. Evaluating the return value is what forces the file to actually be read from the disk. If we want this program to work, we can add a `length` call:

```
main = do
  contents <- readFile "Foo.txt"
  print (length contents)
  writeFile "Foo.txt" "written"
  putStrLn contents
```

The program now works.

Lazy IO is a neat trick to implement streaming and constant memory use. It can cause major annoying problems in the runtime of an application, and should generally be avoided in favor of explicit streaming libraries like `conduit`, `pipes`, or `streaming`.

Some functions in the `Prelude` are lazy in a way that is rarely what you want. The function `foldl` is almost never what you want - it has to traverse the entire list in order to produce a result, but it can't yield anything lazily. The end result is that you build up a huge thunk in memory. `sum` and `product` are similarly lazy, which needlessly wastes memory.

8.2 Custom Benefits

Custom preludes can solve all of the problems that the standard Prelude offers. The cost is Novelty and potentially Complexity, depending on how much you want to change. Every difference from the usual Haskell Prelude will cause some amount of friction for developers as they onboard to the project. There's also a momentary cost when switching contexts if two projects use dramatically different preludes.

Documentation

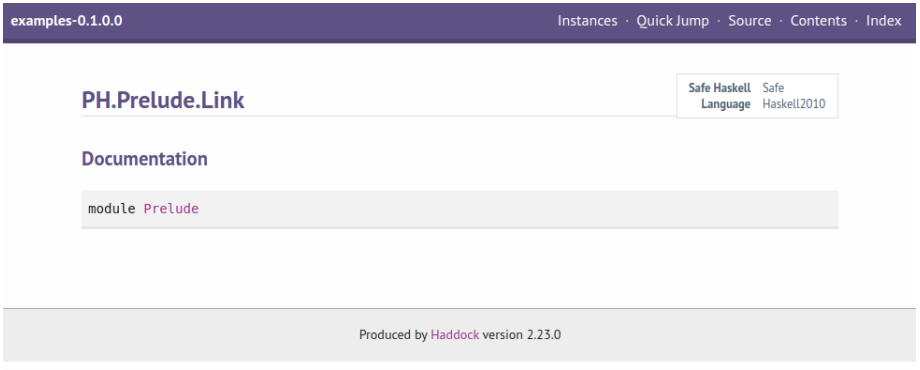
Possibly the greatest benefit to a custom prelude is that it can serve as a documentation point for your project. Most projects are a little bit weird. There's also some amount of onboarding you need to do in order for new hires to get their bearings. That onboarding must be repeated every time a person needs to re-orient to the project after some time away, too.

A custom Prelude allows you to attach documentation comments to your Prelude. These comments will show up in the Haddock rendered output for your project. If you make a strange choice or have an uncommon idiom, that's a perfect place to document it.

If you re-export a module unmodified, then the Haddocks will only show a link to the module in your documentation.

```
1  module PH.Prelude.Link
2      ( module Prelude
3      ) where
4
5  import Prelude
```

This produces the following page with `stack haddock`:



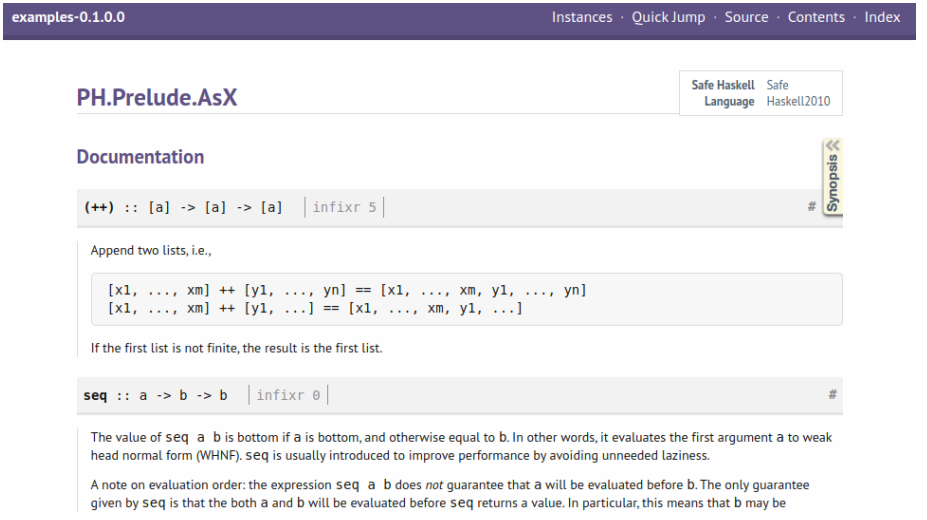
`PH.Prelude.Link` module documentation

Linking directly to the module in question is nice but fragile. If we hide anything from that module, or if we re-export two modules with the same

alias, then the rendered documentation dumps the entire contents of the import directly into the page. The following code yields the following documentation page:

```
module PH.Prelude.AsX
    ( module X
    ) where

import Prelude as X
import Data.Map as X (Map)
```

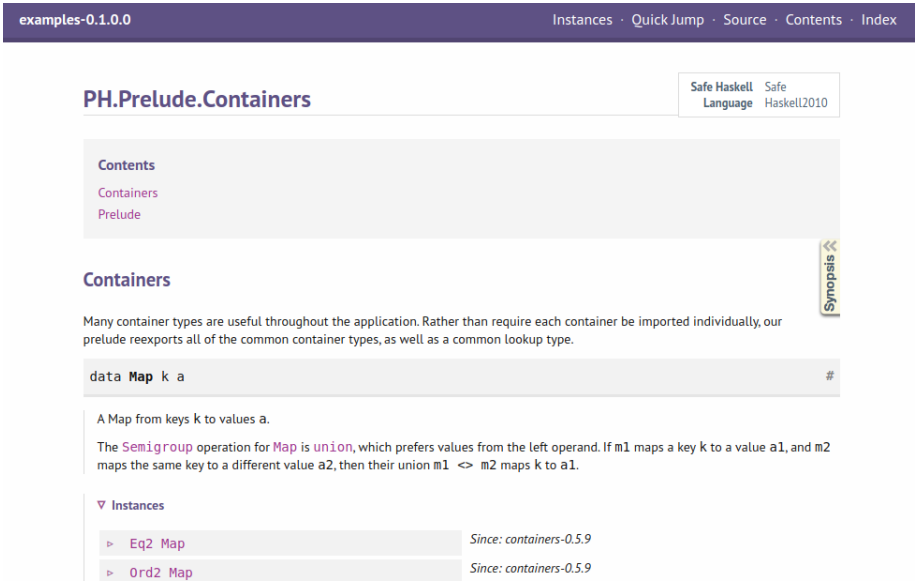


`PH.Prelude.AsX` module documentation

This makes the Haddocks messy and difficult to browse. Instead of using a catch-all re-export alias like `X`, group related imports into a similar alias and provide Haddock section documentation for them. We can use the section comment syntax `-- *` in the export list to provide a table-of-contents in the generated documentation, which also gives us the ability to link to the underlying module. Let's look at a more legible example:

```haskell
module PH.Prelude.Containers
    ( -- * Containers
      --
      -- | Many container types are useful throughout the
      -- application. Rather than require each container be
      -- imported individually, our prelude reexports all of
      -- the common container types, as well as a common
      -- lookup type.
      module Containers
      -- * "Prelude"
      --
      -- | We hide the import of 'id' since it overlaps with
      -- our domain.
    , module Prelude
    ) where

import Prelude hiding (id)
import Data.Map as Containers (Map)
import Data.Set as Containers (Set)
```



`PH.Prelude.Containers` documentation

The link to the `Prelude` module along with an explanation on why it is expanded helps alleviate the complexity and novelty cost of the custom prelude.

Domain Specificity

A custom prelude can be tailored to your application domain. While the standard Prelude may have name conflicts with common domain terms, your custom prelude can export whatever you want.

A Point of Control

Putting a type or function in your custom prelude gives you a point of control over how the type is imported. This can make upgrading versions of dependences easier.

Suppose that `Data.Text` decided to drop the `Data` module prefix in the imports. If you import `Text` through `Data.Text`, you're going to need to modify it in every single file that uses it. This is tedious and annoying.

However, if the import is provided through your prelude, then you only need to change it in one place.

Suppose that a library you use deprecates and/or removes a function you're using. You don't want to migrate away from the deprecated function now, but you *also* need a bugfix in the new version! With a custom prelude, you can provide a vendored implementation. Suppose that `Data.Text` decided to remove the `titleCase` function, encouraging users to adopt the new `Data.Text.Casing` library instead. You could either re-export `Data.Text.Casing.titleCase`, or you could copy the old implementation of `titleCase` and define it in your prelude.

This point is actually a bit more general - any time you depend on something defined outside your application, it can be a good idea to wrap it in something you do control. This can be a significant amount of work, but it will save you a lot of time and trouble if your dependency changes. For example, wrapping a database interaction library's core functionality in your own modules makes it easier to switch database libraries without needing to touch every file in your codebase.

8.3 Off-The-Shelf Preludes

If you've decided to build a custom project prelude, you may want to choose one of the many Prelude replacement libraries available as a basis. You may like one of these preludes enough to use it without modification. However, I'd still recommend wrapping the module in one you control, and importing your custom prelude instead. Having a single "entry point" to control what (almost) every module in your program sees can be a powerful tool for keeping your code resilient and robust.

I've evaluated a few of the more common and more expansive preludes available. They are listed in ascending order of difference from the basic `Prelude`.

base-prelude

`base-prelude`[7] has no dependencies other than `base`. It re-exports almost everything from the common modules in `base`, making a few

[7] https://hackage.haskell.org/package/base-prelude

upgrades along the way. The function composition operator is replaced with the version in `Control.Category`, which abstracts over what you're composing.

This package is a great choice if you don't want to make any changes to the existing Prelude, but you're also tired of importing `Data.Foldable`, `Data.IORef`, `Control.Concurrent`, and `System.IO` in many modules.

protolude

`protolude`[8] is quite opinionated. The README says:

> A sensible starting Prelude for building custom Preludes.
>
> **Design points:**
>
> - Banishes String.
> - Banishes partial functions.
> - Compiler warning on bottoms.
> - Polymorphic string IO functions.
> - Polymorphic show.
> - Automatic string conversions.
> - Types for common data structures in scope.
> - Types for all common string types (Text/ByteString) in scope.
> - Banishes impure exception throwing outside of IO.
> - StateT/ReaderT/ExceptT transformers in scope by default.
> - Foldable / Traversable functions in scope by default.
> - Unsafe functions are prefixed with "unsafe" in separate module.
> - Compiler agnostic, GHC internal modules are abstracted out into Base.
> - sum and product are strict by default.
> - Includes Semiring for GHC >= 7.6.
> - Includes Bifunctor for GHC >= 7.6.
> - Includes Semigroup for GHC >= 7.6.

This prelude attempts to solve all of the problems brought on by the stock Prelude. `id` is renamed to `identity`. The inefficient function `nub` (which removes duplicates in a list) is removed, replaced with the much more efficient `ordNub`. `undefined` and `error` are not exposed, replaced with a function `panic`. `head` is defined quite a bit differently:

[8] https://hackage.haskell.org/package/protolude

```
head :: (Foldable f) => f a -> Maybe a
```

There's one usability issue: excess polymorphism. Let's consider the `show` and `putStrLn` functions:

```
show :: (Show a, ConvertText String b) => a -> b
putStrLn :: (Print a, MonadIO m) => a -> m ()
```

These functions are totally polymorphic. The relatively innocuous function `putStrLn "Hello, World"` will now cause an error with `OverloadedStrings`, as GHC doesn't know what type the string literal should be. Likewise, writing the function `print = putStrLn . show` will have an ambiguity error, because the input of `putStrLn` is polymorphic, and the output of `show` is polymorphic, and you have to pick what the intermediate type is.

This is a common problem with code that is highly polymorphic. Ambiguity errors require type annotations in frustrating spots. Occasionally, a polymorphic function is used at the "wrong" type - consider `length :: Foldable f => f a -> Int`. `length [1,2,3]` returns `3`, as you might expect. However, suppose we have `length (foo [1,2,3])`, where `foo :: [Int] -> [Int]`. And then someone changes `foo` to return a `Maybe [Int]`. `length (foo [1,2,3])` will now always return either `0` or `1` - not the length of the underlying list.

For this reason, it's a good idea to liberally use `TypeApplications` when writing code against a heavily polymorphic library. `length @[] (foo [1,2,3])` would cause a type error when `foo` changes to be `Maybe`, which would prevent a bug from sneaking in. `putStrLn @Text` fixes the ambiguity error, and is defined in `protolude` as `putText`.

classy-prelude

`protolude` uses quite a few type classes, but `classy-prelude`[9] takes it to the next level. The biggest difference here is the use of the `mono-traversable`[10] library for much functionality, including `foldMap`, `null`,

[9] https://hackage.haskell.org/package/classy-prelude-1.5.0/

[10] https://hackage.haskell.org/package/mono-traversable-1.0.9.0/docs/Data-MonoTraversable-Unprefixed.html

sum, etc. The mono-traversable package defines a monomorphic hierarchy similar to Functor/Foldable/Traversable, which allows you to have instances for types like Text, ByteString, etc. For some functions - like foldMap - the MonoFoldable and Foldable variants are the exact same, so this library exports only these.

classy-prelude attaches WARNING pragmas to the debugTrace family of functions. This makes it easy to know when you've forgotten to remove them from code, while still making them easily accessible from the prelude.

A big advantage of the classy-prelude is that it was partially designed for the Yesod[11] web framework by FP Complete. As such, many snags with concurrency[12], runtime exceptions[13], and file IO[14] are already handled for you. As with protolude, the common container types are re-exported.

relude

relude[15] is a custom prelude from the Haskell consultancy Kowainik[16], who produce many fine tools and libraries. One thing you'll immediately notice is the abundance of documentation and examples. This is the main thing that differentiates it from protolude. Otherwise, relude offers another point in the design space of a Prelude that minimally changes base.

rio

rio[17] is not merely a Prelude replacement - it's an alternative standard library for Haskell. It re-exports the common types like Map, Set, Text, ByteString, etc that you are most likely to use, as well as functions for operating on them. The big differentiating factor is that rio is opinionated on how to structure applications and includes many IO utilities to solve common gotchas in the base library.

[11] https://www.yesodweb.com/
[12] https://www.snoyman.com/blog/2016/11/haskells-missing-concurrency-basics
[13] https://www.fpcomplete.com/haskell/tutorial/exceptions/
[14] https://www.snoyman.com/blog/2016/12/beware-of-readfile
[15] https://hackage.haskell.org/package/relude
[16] https://kowainik.github.io/
[17] https://hackage.haskell.org/package/rio

`rio` advocates the `ReaderT` Design Pattern, and encodes it with the main newtype:

```
newtype RIO r a = RIO { unRIO :: ReaderT r IO a }
```

You're encouraged to design your app such that the environment is kept polymorphic as much as possible. Logging provides an excellent example. Instead of `LoggingT` or a `MonadLogger m` constraint, we have a constraint on the `r` type parameter:

```
-- simplified
logInfo
  :: (MonadIO m, MonadReader r m, HasLogFunc env)
  => Utf8Builder -> m ()
logInfo message = do
  env <- ask
  let logFunc = view logFuncL env
  liftIO $ runLogFunc logFunc message

class HasLogFunc env where
  logFuncL :: Lens' env LogFunc

newtype LogFunc = LogFunc (Utf8BUilder -> IO ())

runLogFunc :: LogFunc -> Utf8Builder -> IO ()
runLogFunc (LogFunc f) message = f message
```

We can instantiate this to `RIO LogFunc`. Or, for any type `Env` that contains a `LogFunc`, we can write an `instance HasLogFunv Env` that shows how to get and modify the logging capability. By writing abstractly against the capabilities, it is relatively straightforward to factor out code such that it only depends on what it needs in the context. This makes it easy to run code in multiple contexts.

generic-lens

The boilerplate for defining fresh `Has$(ThingInEnv)` classes can be frustrating. With `generic-lens`[18], you can use the `typed`[19] lens to dig things out automatically.

```
typed :: HasType a s => Lens' s a
```

With this in mind, we could rewrite our `logInfo` to look like this:

```
logInfo
  :: (MonadIO m, MonadReader r m, HasType LogFunc r)
  => Utf8Builder -> m ()
logInfo message = do
  env <- ask
  let logFunc = view (typed @LogFunc) env
  liftIO $ runLogFunc logFunc message
```

Since this is `Generic`, it is defined for all records which contain exactly one `LogFunc` in any of the fields. It is also defined on tuples, where any element contains a `LogFunc`. This makes for a flexible and relatively low boilerplate method of writing in this style.

While `view` has the simplified type `view :: Lens' s a -> s -> a`, it's actually more general:

```
view :: (MonadReader s m) => Lens' s a -> m a
```

There is an instance of `MonadReader r ((->) r)`, which is why this works for the simple function case. The implementation of `logInfo` can be made more concise:

[18] https://www.stackage.org/lts-16.19/package/generic-lens-2.0.0.0

[19] https://www.stackage.org/haddock/lts-16.19/generic-lens-2.0.0.0/Data-Generics-Product-Typed.html

```
logInfo message = do
    logFunc <- view (typed @LogFunc)
    liftIO $ runLogFunc logFunc message
```

The Trouble with MonadReader

rio is a champion of the generalized ReaderT design pattern. Unfortunately, I've generally found the approach to be somewhat disappointing.

The first problem is that ReaderT will steal any MonadReader instance for itself. This becomes a problem when you want to use ReaderT for a temporary task. Consider this foo function, using a ReaderT ExtraStuff. If the Extra datatype does not contain the LogFunc that the generic logInfo we defined above requires, then it will fail to work.

```
foo :: ReaderT ExtraStuff (RIO AppEnv) Int
foo = do
    extraStuff <- ask
    logInfo "Oh no"
```

We can see what happens by looking at the instance of MonadReader for ReaderT:

```
instance (Monad m) => Monad (ReaderT r m)
```

When GHC is type checking foo, it compares the expected type with the general type:

```
-- Unify:
ReaderT ExtraStuff (RIO AppEnv) Int
ReaderT r          m            a
r ~ ExtraStuff
m ~ RIO AppEnv
a ~ Int
```

Then, GHC needs to prove some things. First, it needs to prove that r has an instance of Has LogFunc.

```
-- Want:
Has LogFunc r =>
```

Then GHC will notice that `ExtraStuff` *does not* have a `LogFunc`. GHC already decided to commit to the `MonadReader r (ReaderT r m)` instance. The compiler will not "back track" to attempt the `MonadReader r (RIO r)` instance. You can fix this temporarily by calling `lift $ logInfo msg`, but that's unappealing. The entire point of these techniques is to avoid manual lifting.

Instead, I prefer to define classes that are polymorphic in the monad, not the reader environment.

```
class MonadLog m where
    logInfo :: LogMsg -> m ()
```

This can be used in many more contexts, and permits many more implementations. The `MondaLog m` style does require more boilerplate, though, so you may want to instead ban those `ReaderT` uses. Consider this instead:

```
newtype HasExtraStuffT m a
    = HasExtraStuffT
    { unHasExtraStuff :: ReaderT ExtraStuff m a
    }
```

This has *other* boilerplate concerns, of course. Trade-offs abound.

foundation

`foundation`[20] is a radical departure from the usual Haskell ecosystem. All other alternative preludes on this list reuse types that are ubiquitous to the ecosystem: `Text` and `ByteString`. `foundation` breaks from this by defining its own type for `String` using a packed array of UTF8 codepoints. (The `Text` library has switched to this representation since

[20] https://hackage.haskell.org/package/foundation

version 2, as well). It includes several novel approaches to solving problems - a custom and efficient `Parser` type, a custom `StopWatch`, and an improved hierarchy of numeric classes. A lot of code here is copied from other libraries - a copy[21] of `conduit`[22] is included.

`foundation` is in a class of replacement preludes that seek to reimagine how Haskell works to better suit certain domains. Every implementation choice in `foundation` seems to have two motivating factors: memory efficiency and developer convenience. This makes `foundation` a decent choice if you're building memory sensitive applications and also need to move quickly. However, investing in a prelude that is so different from the ordinary Haskell ecosystem is going to impose significant costs on your project.

Ultimately, there's a sweet spot when deciding on a custom prelude. Each step you take towards perfection brings you farther from the familiar. If you can't document your journey, you may find yourself lost and alone.

8.4 Nesting Preludes

If it is useful to have a custom prelude for your entire project, then it might also be useful to have custom preludes for a small part of the project. You may even go so far as to have a prelude for every level in your module hierarchy!

As an example, let's say we have our program namespaced under `PH`, and we have `PH.Prelude` that defines most of what we need. The namespace `PH.DB` contains database access code, so we might have a prelude for it:

```
1  module PH.DB.Prelude
2    ( module PH.DB.Prelude
3    , module PH.Prelude
4    , module Database.Persist.Sql
5    -- ^ from the `persistent` package
6    )
```

[21] https://hackage.haskell.org/package/foundation-0.0.25/docs/Foundation-Conduit.html
[22] https://hackage.haskell.org/package/conduit

Now, under `PH.DB`, we've got another namespace: `Query`, for `esqueleto` queries! `esqueleto` and `persistent` have a few name conflicts - both export an operator `==.` for comparing two SQL terms. We don't want to import `PH.DB.Prelude` because we'd need to hide the `==.` references from it in favor of the one from `esqueleto`. So, instead, we'd have:

```
1  module PH.DB.Query.Prelude
2    ( module PH.DB.Prelude
3    , module Database.Esqueleto
4    ) where
5
6  import PH.DB.Prelude hiding ((==.))
7  import Database.Esqueleto
```

By providing many levels of preludes, you can more easily and concisely control what is in scope at each point. Each additional `.Prelude` module becomes a great place to attach documentation in every namespace, too!

8.5 Downsides

A custom prelude isn't a silver bullet.

Custom preludes impose a Novelty cost on a project. Some preludes are strictly additions to the normal `Prelude`. These don't carry a large cost. The biggest thing to learn is when you *don't* need to write an import.

However, some custom preludes have massive changes. `foundation` is a batteries-included complete reworking of Haskell's basic types. It's totally different from `Prelude`, replacing `bytestring`, `array`, `text`, `time`, and many other core libraries and types. The benefits are supposedly great - but deviating from the norm in a tiny community means that you'll have a significant ramp-up cost, even with experienced Haskellers that are merely unfamiliar with this standard library.

8.6 Using a Custom Prelude

So you've decided that you want to try out a custom prelude. There are a few ways to accomplish this task. You can use the `NoImplicitPrelude` language extension and import it manually in every project. You can implicitly import the custom prelude using a new feature of Cabal called `mixins`, or you can take advantage of a loophole in GHC by defining your own `Prelude` module.

I recommend explicitly importing your custom Prelude and using `NoImplicitPrelude` in your package's default extensions. This option works everywhere and has minimal magic. If you are using nested preludes, then you don't need a separate `hiding` step if there are any conflicts.

9. Optimizing GHC Compile Times

You're a Haskell programmer, which means you complain about compilation times.

We typically spend *a lot* of time waiting for GHC to compile code. To some extent, this is unavoidable - GHC does a tremendous amount of work for us, and we only ever ask it to do more. At some point, we shouldn't be terribly surprised that "doing more work" ends up meaning "taking more time." However, there are some things we can do to allow GHC to avoid doing more work than necessary. For the most part, these are going to be code organization decisions.

In my experience, the following things are true, and should guide organization:

- Superlinear: GHC takes more time to compile larger modules than smaller modules.
- Constant costs: GHC takes a certain amount of start-up time to compile a module
- Parallelism: GHC can compile modules in parallel (and build tools can typically compile packages in parallel)
- Caching: GHC can cache modules
- Type class deriving: GHC's type class derivation is slow
- `TemplateHaskell`: can cause excess recompilation

So let's talk about some aspects of project organization and how they can affect compile times.

9.1 The `Project.Types` Megamodule

You just start on a new project, and you get directed to the God module - `Project.Types`. It's about 4,000 lines long. "All the types are defined in here, it's great!" However, this is going to cause big problems for your compilation time:

- A super large module is going to take way longer to compile
- Any change to any type requires touching this module, and recompiling everything in it
- Any change to this module requires recompiling any module that depends on it, which is usually everything

We pretty much can't take advantage of caching, because GHC doesn't cache any finer than the module-level. We can't take advantage of parallelism, as GHC's parallelism machinery only seems to work at module granularity. Furthermore, we're tripping this constantly, which is causing GHC to recompile a lot of modules that probably don't need to be recompiled.

Resolution

Factor concepts out of your `Project.Types` module. This will require manually untangling the dependency graph, which isn't fun. You may also find it to be a good excuse to learn `.hs-boot` files for breaking mutual recursion.

There's a small constant cost to compile a module, so you probably shouldn't define a module for every single type To optimize compilation times, group related types into modules. My intuition feels pretty happy with "50-200" lines as a guideline, but I haven't studied this in depth. On the other hand, a single module per type aids discovery, since you can do a fuzzy file search for the type name to find a corresponding definition site.

This process can be done incrementally. Pick a concept or type from the bottom of your dependency graph, and put it in its own module. You'll need to import that into `Project.Types` - but do not reexport it! Everywhere that complains, add another import to your new module.

As you factor more and more modules out, eventually you'll start dropping the dependency on `Project.Types`. Now, as you edit `Project.Types`, you won't have to recompile these modules, and your overall compile-times will improve dramatically. All the types that are pulled out of `Project.Types` will be cached, so recompiling `Project.Types` itself will become much faster.

Before too long, you'll be minimizing the amount of compilation you have to do, and everything will be happy.

9.2 Package Splitting

Okay so you think "I know! I'll make a bunch of packages to separate my logical concerns!" A package boundary comes with some important trade-offs for development velocity and compile-times.

A good reason to factor out a package is to open source a library. If you can't reasonably consider your package a library that other packages could depend on, then it is likely not an appropriate time to begin factoring it out.

GHCi

GHCi is pretty picky about loading specific targets, and what you load is going to determine what it will pick up on a reload. You need to ensure that each target has the same default extensions, dependencies, compiler flags, etc. because all source files will be loaded as though they were in a single project. This is a good reason to either use Cabal or `hpack` common stanzas for all of this information, or to use file-specific stuff and avoid using implicit configuration.

While `stack` has no problem loading different targets into `ghci`, `cabal` refuses to load multiple targets. You can work around this limitation by defining a "mega" component which includes everything you want to include: library, tests, executables, benchmarks, even other libraries. Jade Lovelace wrote a blog post titled "The cabal test-dev trick"[1] that details this approach.

What's a "load target"? A target is a part of a package, like a library, a specific test-suite, a specific executable, or a sub-library. In a multi-package Cabal or Stack project, load targets can come from different packages.

Another gotcha is that any relative file paths must resolve based on where you're going to invoke `{stack,cabal} ghci`. Suppose you decide you want to split your web app into two packages: `database` and `web`, where `database` has a file it loads for the model definitions, and `web` has a bunch of files it loads for HTML templating. The Template Haskell

[1] https://jade.fyi/blog/cabal-test-dev-trick/

file-loading libraries pretty much assume that your paths are relative to the directory containing the `.cabal` file. When you invoke `stack ghci` (or `cabal repl`), it puts your current working directory in the directory you launch it, and the relative directories there are probably not going to work.

Once you've created that package boundary, it becomes difficult to operate across it. The natural inclination - indeed, the reason why you might break it up - is to allow them to evolve independently. The more they evolve apart, the less easily you can load everything into GHCi.

You can certainly load things into GHCi - in the above example, `web` depends on `database`, and so you can do `stack ghci web`, and it'll compile `database` just fine as a library and load `web` into GHCi. However, you won't be able to modify a module in `database`, and hit `:reload` to perform a minimal recompilation. Instead, you'll need to kill the GHCi session and reload it from scratch. This takes a lot more time than an incremental recompilation.

A single large package has advantages, but it does increase the initial boot time of `ghci`. `ghci` needs to interpret all of the modules in your project, and it is not capable of caching the modules or interpreted code. This means that reloading `ghci` will always begin recompiling at module 1. If there are modules that take a long time to compile and are rarely changed, then these can represent good candidates for splitting into a separate package.

Module Parallelism

GHC is pretty good at compiling modules in parallel. It's also pretty good at compiling packages in parallel.

Unfortunately, it can't see across the package boundary. Suppose your package `hello` depends on module `Tiny.Little.Module` in the package `the-world`, which also contains about a thousand utility modules and Template Haskell splices and derived Generic instances for data types and type family computations and (etc.....). You'd *really* want to just start compiling `hello` as soon as `Tiny.Little.Module` is completely compiled, but you can't - GHC must compile everything else in the package before it can start on yours.

Breaking up your project into multiple packages can cause overall compile-times to go up significantly in this manner. If you do this, it should ideally be to split out a focused library that will need to change relatively rarely while you iterate on the rest of your codebase. I'd beware of breaking things up until absolutely necessary - a package boundary is a heavy tool to merely separate responsibilities.

At one company I worked for, we had a package graph that looked like this:

```
1          +-> C
2  A -> B --|
3          +-> D
```

By combining `A` and `B` into a single package, we sped up compile times for a *complete* build of the application by 10%. A clean build of the new `AB` package was 15% faster to build all told,and incremental builds were improved significantly.

Package parallelism

The good news is that it is quite easy to cache entire packages, and the common build tools are quite good at compiling packages in parallel. It's not a big deal to depend on `lens` anymore, largely because of how good sharing and caching has gotten. So certainly don't be *afraid* to split out libraries and push them to GitHub or Hackage, but if you're not willing to GitHub it, then it should probably stay in the main package.

9.3 Big Ol' Instances Module

Well, you did it. You have a bunch of packages and you don't want to merge them together. Then you defined a bunch of types in `foo`, and then defined a type class in `bar`. `bar` depends on `foo`, so you can't put the instances with the type definitions, and you're a Good Haskeller so you want to avoid orphan instances, which means you need to put all the instances in the same module.

Except - you know how you had a 4,000 line types module, which was then split-up into dozens of smaller modules? Now you have to import *all* of those, and you've got a big 5,000 line class/instance module. All the same problems apply - you've got a bottleneck in compilation, and any touch to any type causes this big module to get recompiled, which in turn causes everything that depends on the class to be recompiled.

A solution is to ensure that all your type classes are defined above the types in the module graph. This is easiest to do if you have only a single package. But you may not be able to do that easily, so here's a solution:

Hidden Orphans

The real problem is that you want to refer to the class and operations without incurring the wrath of the dependency graph. You can do this with orphan instances. Define each instance in its own module and import them into the module that defines the class. Don't expose the orphan modules - you really want to ensure that you don't run into the practical downsides of orphans while allowing recompilation and caching.

You'll start with a module like this:

```
module MyClass where

import Types.Foo
import Types.Bar
import Types.Baz

class C a

instance C Foo
instance C Bar
instance C Baz
```

A change to any `Types` module requires a recompilation of the entirety of the `MyClass` module.

You'll create an internal module for the class (and any helpers etc.), then a module for each type/instance:

```haskell
module MyClass.Class where

    class C a


module MyClass.Foo where

    import MyClass.Class
    import Types.Foo

    instance C Foo


module MyClass.Bar where

    import MyClass.Class
    import Types.Bar

    instance C Bar


module MyClass.Baz where

    import MyClass.Class
    import Types.Baz

    instance C Baz


module MyClass (module X) where

    import MyClass.Class as X
    import MyClass.Foo as X
    import MyClass.Bar as X
    import MyClass.Baz as X
```

So what happens when we touch `Types.Foo`? With the old layout, it'd

trigger a recompile of `MyClass`, which would have to start entirely over and recompile *everything*. With the new layout, it triggers a recompile of `MyClass.Foo`, which is presumably much smaller. Then, we do need to recompile `MyClass`, but because all the rest of the modules are untouched, they can be reused and cached, and compiling the entire module is much faster.

This is a bit nasty, but it can break up a module bottleneck quite nicely, and if you're careful to only use the MyClass interface, you'll be safe from the dangers of orphan instances.

9.4 `TemplateHaskell`

`TemplateHaskell` has a nasty reputation with compilation speed. This reputation pushes folks away from using it, even when it is an excellent choice. In my experience, `TemplateHaskell` is often much faster than alternatives when considering an individual complete compilation.

To test this hypothesis, I attempted to switch all of the `aeson` derivation in a company's application from `TemplateHaskell` to `Generic` based derivation. The project ended up taking twice as long to compile. `Generic` derivation has significant costs, especially for large types.

There is a tiny cost to running any `TemplateHaskell` splices at all - GHC must prepare a Haskell interpreter. In earlier versions of GHC, I was able to notice a 200-500ms penalty. With GHC 8.10 and newer, I can't detect any penalty for `TemplateHaskell` splices.

The main problem with `TemplateHaskell` for compilation time is that GHC will recompile modules more often if they use `TemplateHaskell`. Prior to GHC 9.4, if a module uses `TemplateHaskell`, then it must recompile if *any* module in the transitive dependencies were modified. This is because `TemplateHaskell` behavior can change due to new instances being in scope, or new values being available. With GHC 9.4, this is much improved, and `TemplateHaskell` recompilation is much less of a problem.

Actually running code

GHC has two phases for TH:

1. Generating Code
2. Compiling Code

Generating code typically doesn't take much time at all, though this isn't guaranteed.

Fortunately, we can easily write a timing utility, since the `Template-Haskell` generation type allows you to run arbitrary `IO` operations.

```
1  import Data.Time (getCurrentTime, diffUTCTime)
2  import Language.Haskell.TH (Q, runIO, reportWarning)
3
4  timed :: String -> Q a -> Q a
5  timed message action = do
6      begin <- runIO getCurrentTime
7      result <- action
8      end <- runIO getCurrentTime
9      let duration = end `diffUTCTime` begin
10     reportWarning $ concat [ "[", message, "]: ", show duration]
11     pure result
```

Expert benchmarkers will complain about using `getCurrentTime` since it isn't monotonic, which is a valid complaint. But we're not getting a real benchmark anyway, and we're mostly just going to see whether generation or compilation is dominating the elapsed time (hint: compilation almost always dominates).

With this, we will get a reported warning about the duration of the code generation. In this reddit comment[2], I used `timed` to determine that generation of some code was taking `0.0015s`, while compilation of the resulting code took `21.201s`. The code looks like this:

[2]https://www.reddit.com/r/haskell/comments/oi1x5v/tiny_use_of_template_haskell_causing_huge_memory/h4tr7n8/

```
module Main where

import TuplesTH

$(timed "tuples" $ generateTupleBoilerplate 62)

main :: IO ()
main = do
    print $ _3 (1,2,42,"hello",'z')
```

The output looks like this:

```
Building executable 'th-perf-exe' for th-perf-0.1.0.0..
[1 of 2] Compiling Main

/home/th-perf/app/Main.hs:11:2: warning: [tuples]: 0.001553454s
   |
11 | $(timed "tuples" $ generateTupleBoilerplate 62)
   |  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[2 of 2] Compiling Paths_th_perf
 21,569,689,896 bytes allocated in the heap
  6,231,564,888 bytes copied during GC
    594,126,600 bytes maximum residency (17 sample(s))
      3,578,104 bytes maximum slop
           1641 MiB total memory in use (0 MB lost due to frag...)

                                     Tot time (elapsed)  Avg pause  Max pause
 Gen  0  1097 colls,  0 par    4.919s   4.921s     0.0045s    0.1072s
 Gen  1    17 colls,  0 par    4.466s   4.467s     0.2628s    1.0215s

 TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

 SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

 INIT    time    0.001s  (  0.001s elapsed)
 MUT     time   11.813s  ( 12.135s elapsed)
 GC      time    9.385s  (  9.388s elapsed)
```

```
26  EXIT    time    0.001s  (  0.007s elapsed)
27  Total   time   21.201s  ( 21.530s elapsed)
28
29  Alloc rate    1,825,890,582 bytes per MUT second
30
31  Productivity  55.7% of total user, 56.4% of total elapsed
```

This sort of timing is usually only useful to determine whether you need to benchmark and optimize the *generation* phase or the *compilation* phase. Optimizing *generation* is a relatively standard Haskell performance optimization process, which I won't cover here. If your code is mostly pure functions (or, with GHC 9, the new `Quote`[3] type class), then it's straightforward to do. Many `Q` features are not supported in `IO`, and it's difficult to accurately benchmark them.

Fortunately, you *can* profile them using the `-fexternal-interpreter` facility. Ben Gamari's "Profiling Template Haskell Splices"[4] is an excellent resource for this.

Optimizing Compilation

In the above example, GHC spends a tiny amount of time generating code, and then spends a *huge* amount of time compiling it. What's going on?

Above, I write that GHC compiles modules superlinearly in the size of the module. That means that large modules take longer to compile than the same amount of code split up over several modules. `TemplateHaskell` has no way of creating *modules*, or even altering the imports/exports of a given module. If you're generating a large amount of code with `TemplateHaskell`, there's no way to split it into separate modules.

We have two means of reducing generated code: spreading the use over multiple modules, and optimizing how we generate the code.

[3] https://www.stackage.org/haddock/nightly-2021-07-11/template-haskell-2.17.0.0/Language-Haskell-TH.html#t:Quote

[4] https://well-typed.com/blog/2020/05/profiling-template-haskell/

Fewer Calls to TH

In Splitting Persistent Models[5], I wrote how to speed up compile-times by isolating the `persistent` model definitions into separate modules. This results in many smaller modules, which GHC can compile much faster - in part because the modules are able to be parallelized, and in part because they are smaller, and don't hit the superlinearity.

You can do this with any other thing, too. A large module that has a ton of data types and a `TemplateHaskell` declaration for each type will quickly become a problem in compilation. Separating it out into multiple modules, each exporting a small subset of those types, will allow GHC to avoid recompiling them as often and share the cache more effectively.

Smaller Code

It's relatively easy to generate a massive amount of Haskell code. After all, the entire *point* of `TemplateHaskell` is to make GHC generate code for us that we don't want to write ourselves.

In order to see how much code we're generating in a module, it's useful to enable the `-ddump-splices` option. We can do this with a `GHC_OPTIONS` pragma above the module header:

```
1 {-# language TemplateHaskell #-}
2 {-# OPTIONS_GHC -ddump-splices #-}
3
4 module Lib where
5
6 import Language.Haskell.TH.Syntax (liftTyped)
7
8 asdf :: Int
9 asdf = $$(liftTyped 3)
```

With this option, GHC will print the splice and the corresponding output while compiling the module.

[5] https://www.parsonsmatt.org/2019/12/06/splitting_persistent_models.html

```
Building library for th-perf-0.1.0.0..
[2 of 3] Compiling Lib
/home/matt/Projects/th-perf/src/Lib.hs:10:10-22:
    Splicing expression
        liftTyped 3 ======> 3
```

However, if you've got a performance problem, then you've probably got more output here than you have any idea what to do with. In the reddit thread[6], we ended up generating enough code that I couldn't scroll back to the top! So, we'll want to dump the resulting splices to a file. We can use the `-ddump-to-file`, and GHC will store the splices for a module in a file named `$(module-name).dump-$(phase)`. If you're building with `stack`, then the files will be located in the `.stack-work` file. We can get the resulting size of the file using `wc` and a bit of a glob. In that investigation, this is the command and output:

```
$ wc -l .stack-work/**/*.dump-splices
15897 .stack-work/.../Main.dump-splices
```

That's 15,897 lines of code! You can open that file up and see what it generates. In that example, there wasn't much to optimize.

Beware Splicing and Lifting

At the work codebase, we had a `TemplateHaskell` function that ended up taking several minutes to compile. It iterated through all of our database models and generated a function that would stream each row from the database and verify that we could successfully parse everything out of the database. This is nice to check that our `PersistField` definitions worked, or that our JSONB columns could all still be parsed.

I investigated the slow compile-time by dumping splices, and managed to find that it was splicing in the entire `EntityDef`[7] type, multiple times,

[6] https://www.reddit.com/r/haskell/comments/oi1x5v/tiny_use_of_template_haskell_causing_huge_memory/h4tr7n8/

[7] https://www.stackage.org/haddock/lts-18.2/persistent-2.13.1.1/Database-Persist-EntityDef-Internal.html#t:EntityDef

for each table. This is a relatively large record, with a bunch of fields, and each `FieldDef` *also* is relatively large, with a bunch of fields!

The resulting code size was enormous. Why was it doing this? I looked into it and discovered this innocuous bit of code:

```
do
    -- ...
    tableName <- [| getEntityHaskellName entityDef |]
    dbName <- [| getEntityDBName entityDef |]
    -- ...
    pure $ mkFun tableName dbName
```

You might expect that `tableName` would be an expression containing *only* the Haskell name of the entity. However, it's *actually* the *entire expression* in the `QuasiQuote`! Haskell allows you to implicitly lift things, sometimes, depending on scope and context etc. The `lift` in question refers to the `Lift` type class[8], not the `MonadTrans` variant. This ends up being translated to:

```
tableName <- [| $(lift getEntityHaskellName) $(lift entityDef) |]
```

Lifting a function like this is relatively easy - you just splice a reference to the function. So the resulting expression for the *function name* is something like:

```
lift getEntityHaskellName
===>
    VarE 'getEntityHaskellName
```

In order to `lift` the `EntityDef` into the expression, we need to take the *complete run-time value* and transform it into valid Haskell code, which we then splice in directly. In this case, that looks something like this:

[8] https://www.stackage.org/haddock/lts-18.2/template-haskell-2.16.0.0/Language-Haskell-TH-Syntax.html#t:Lift

```
lift entityDef
===>
    EntityDef
        { entityHaskell =
            EntityNameHS (Data.Text.pack "SomeTable")
        , entityDB =
            EntityNameDB (Data.Text.pack "some_table")
        , entityId =
            EntityIdField (
                FieldDef
                    { fieldHaskell =
                        FieldNameHS (Data.Text.pack "id")
                    , fieldDB =
                        FieldNameDB (Data.Text.pack "id")
                    , fieldType =
                        -- ....
                    , fieldSqlType =
                        -- ...
                    , -- etc...
                    }
        , entityFields =
            [ FieldDef { ... }
            , FieldDef { ... }
            , FieldDef { ... }
            , ...
            ]
        }
```

The combined expression splices this in:

```
VarE 'getEntityHaskellName
`AppE`
    (ConE 'EntityDef
    `AppE`
        (ConE 'EntityNameHS
        `AppE`
            (VarE 'pack `AppE` LitE (StringL "SomeTable"))
        )
    `AppE`
        (ConE 'EntityNameDB ...)
    )
```

Which is no good - we're obviously *only* grabbing a single field from the record. Fortunately, we can fix that real easy:

```
tableName <- lift $ getEntityHaskellName entityDef
dbName <- lift $ getEntityDBName entityDef
```

This performs the access before we generate the code, resulting in significantly smaller code generation.

Recompilation Avoidance

GHC is usually pretty clever about determining if it can avoid recompiling a module. However, `TemplateHaskell` defeats this, and GHC doesn't even *try* to see if it can avoid recompiling - it just recompiles. (Note: this was considerably improved in GHC 9.4)

We can't *fix* this, but we can work around it. Try to isolate your `TemplateHaskell` use to only a few modules, and keep them as small as possible.

For example, suppose you have a ~500 line module that contains a bunch of data types, `deriveJSON` calls for those types, business logic, and handler API functions. If *any* dependency of that module changes, you need to recompile the whole module due to the `TH` recompilation rule. This needlessly recompiles everything - the datatypes, functions, JSON derivation, etc.

If you pull the datatypes and `TemplateHaskell` into a separate module, then *that* module needs to be recompiled every time. However, GHC is smart enough to avoid recompiling the dependent module. Suppose you split the 500 line module into two files, one of which is 20 lines of `data` and `TemplateHaskell`, and the other is 480 lines of functions, code, etc. GHC will always recompile the 20 line module quickly, and intelligently avoid recompiling the 480 lines when it doesn't need to.

There's another advantage to splitting modules like this. If your `.Types` module doesn't have any business logic, just type and instance declarations, then you likely depend on fewer imports. This means that the `.Types` module will be recompiled significantly less often, which cascades and means that the business logic that depends on that `.Types` *also* does not need to be recompiled so often.

Recompilation Cascade

Recompilation Cascade is the name I've given to a problem where a tiny change triggers a `[TH]` rebuild of a module, and, since that module got rebuilt, every dependent module using `TH` gets rebuilt. If you use `TemplateHaskell` pervasively, then you may end up having `[TH]` rebuilds for your entire codebase! This can wreck incremental compile times.

Try to avoid this by separating out your `TemplateHaskell` into isolated modules, if at all possible.

If you use the `typed QQ literals` trick, then you can isolate those literals into a `Constants` module, and use those constants directly. Instead of:

```
1  module X where
2
3  sendEmailToFoo = sendEmail [email|foobar@gmail.com|] "hello world"
```

Consider using this instead:

```
module Email.Constants where

    foobar_at_gmail = [email|foobar@gmail.com|]

module X where

    import Email.Constants

    sendEmailToFoo = sendEmail foobar_at_gmail "hello world"
```

With the latter form, `X` does not use `TemplateHaskell`, and therefore can skip recompilation if any dependencies change.

9.5 Some random parting thoughts

- Don't do more work than you need to. Derived type class instances are work that GHC must redo every time the module is compiled.
- Keep the module graph broad and shallow.
- The following command speeds up compilation significantly, especially after exposing all those parallelism opportunities: `stack build --fast --file-watch --ghc-options "-j4 +RTS -A128m -n2m -qg -RTS"` These flags give GHC 4 threads to work with (more didn't help on my 8 core computer), and `-A128m` gives it more memory before it does garbage collection. `-qg` turns off the parallel garbage collector, which is almost always a performance improvement. Thanks to /u/dukerutledge[9] for pointing out `-n2m`, which I don't understand but helped!
- Try to keep things `ghci` friendly as much as possible. `:reload` is the fastest way to test stuff out usually, and REPL-friendly code is test-friendly too!

[9] https://www.reddit.com/r/haskell/comments/e2l1yj/keeping_compilation_fast/f8wt34p/

III Domain Modeling

10. Type Safety Back and Forth

Types are a powerful construct for improving program safety. Haskell has a few notable ways of handling potential failure, the most famous being the venerable `Maybe` type:

```
data Maybe a
    = Nothing
    | Just a
```

We can use `Maybe` as the result of a function to indicate:

> Hey, friend! This function might fail. You'll need to handle the `Nothing` case.

This allows us to write functions like a safe division function:

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide i 0 = Nothing
safeDivide i j = Just (i `div` j)
```

I like to think of this as pushing the responsibility for failure forward. I'm telling the caller of the code that they can provide whatever `Int`s they want, but that some condition might cause them to fail. And the caller of the code has to handle that failure later on.

This is one-size-fits-all technique is the easiest to show and tell. If your function can fail, just slap `Maybe` or `Either` on the result type and you've got safety. I can write a 35 line blog post to show off the technique, and if I were feeling frisky, I could use it as an introduction to `Functor`, `Monad`, and all that jazz.

Instead, I'd like to share another technique. Rather than push the responsibility for failure forward, let's explore pushing it back. This technique

is a little harder to show, because it depends on the individual cases you might use.

If pushing responsibility forward means accepting whatever parameters and having the caller of the code handle possibility of failure, then *pushing it back* is going to mean we accept stricter parameters that we can't fail with. Let's consider `safeDivide`, but with a more lax type signature:

```
1  safeDivide :: String -> String -> Maybe Int
2  safeDivide iStr jStr = do
3      i <- readMay iStr
4      j <- readMay jStr
5      guard (j /= 0)
6      pure (i `div` j)
```

This function takes two strings, and then tries to parse `Int`s out of them. Then, if the `j` parameter isn't `0`, we return the result of division. This function is *safe*, but we have a much larger space of calls to `safeDivide` that fail and return `Nothing`. We've accepted more parameters, but we've pushed a lot of responsibility forward for handling possible failure.

Let's push the failure back.

```
1  safeDivide :: Int -> NonZero Int -> Int
2  safeDivide i (NonZero j) = i `div` j
```

We've required that users provide us a `NonZero Int` rather than any old `Int`. We've pushed back against the callers of our function:

> No! You must provide a `NonZero Int`. I refuse to work with just any `Int`, because then I might fail, and that's annoying.

So speaks our valiant little function, standing up for itself!

Let's implement `NonZero`. We'll take advantage of Haskell's `PatternSynonyms` language extension to allow people to pattern match on a "constructor" without exposing a way to unsafely construct values.

```haskell
{-# LANGUAGE PatternSynonyms #-}

module NonZero
    ( NonZero()
    , pattern NonZero
    , unNonZero
    , nonZero
    ) where

newtype NonZero a = UnsafeNonZero a

pattern NonZero a <- UnsafeNonZero a

unNonZero :: NonZero a -> a
unNonZero (UnsafeNonZero a) = a

nonZero :: (Num a, Eq a) => a -> Maybe (NonZero a)
nonZero i
    | i == 0 = Nothing
    | otherwise = Just (UnsafeNonZero i)
```

This module allows us to push the responsibility for type safety backwards onto callers.

As another example, consider `head`. Here's the unsafe, convenient variety:

```haskell
head :: [a] -> a
head (x:xs) = x
head []     = error "oh no"
```

This code is making a promise that it can't keep. Given the empty list, it will fail at runtime.

Let's push the responsibility for safety forward:

```
headMay :: [a] -> Maybe a
headMay (x:xs) = Just x
headMay []     = Nothing
```

Now, we won't fail at runtime. We've required the caller to handle a `Nothing` case.

Let's try pushing it back now:

```
headOr :: a -> [a] -> a
headOr def (x:xs) = x
headOr def []     = def
```

Now, we're requiring that the *caller* of the function handle possible failure before they ever call this. There's no way to get it wrong. Alternatively, we can use a type for nonempty lists!

```
data NonEmpty a = a :| [a]

safeHead :: NonEmpty a -> a
safeHead (x :| xs) = x
```

This one works just as well. We're requiring that the calling code handle failure ahead of time.

A more complicated example of this technique is the `justified-containers`[1] library. The library uses the type system to prove that a given key exists in the underlying `Map`. From that point on, lookups using those keys are *total*: they are guaranteed to return a value, and they don't return a `Maybe`.

This works even if you `map` over the `Map` with a function, transforming values. You can also use it to ensure that two maps share related information. It's a powerful feature, beyond just having type safety.

[1] https://hackage.haskell.org/package/justified-containers-0.1.2.0/docs/Data-Map-Justified-Tutorial.html

10.1 The Ripple Effect

When some piece of code hands us responsibility, we have two choices:

1. Handle that responsibility.
2. Pass it to someone else!

In my experience, developers will tend to push responsibility in the same direction that the code they call does. So if some function returns a `Maybe`, the developer is going to be inclined to also return a `Maybe` value. If some function requires a `NonEmpty Int`, then the developer is going to be inclined to *also* require a `NonEmpty Int` be passed in.

This played out in my work codebase. We have a type representing an `Order` with many `Items` in it. Originally, the type looked something like this:

```
1  data Order = Order  { items :: [Item] }
```

The `Items` contained nearly all of the interesting information in the order, so almost everything that we did with an `Order` would need to return a `Maybe` value to handle the empty list case. This was a lot of work, and a lot of `Maybe` values!

The type is *too permissive*. As it happens, an `Order` may not exist without at least one `Item`. So we can make the type *more restrictive* and have more fun!

We redefined the type to be:

```
1  data Order = Order { items :: NonEmpty Item }
```

All of the `Maybes` relating to the empty list were purged, and all of the code was pure and free. The failure case (an empty list of orders) was moved to two sites:

1. Decoding JSON

2. Decoding database rows

Decoding JSON happens at the API side of things, when various services `POST` updates to us. Now, we can respond with a `400` error and tell API clients that they've provided invalid data! This prevents our data from going bad.

Decoding database rows is even easier. We use an `INNER JOIN` when retrieving `Orders` and `Items`, which guarantees that each `Order` will have at least one `Item` in the result set. Foreign keys ensure that each `Item`'s `Order` is actually present in the database. This does leave the possibility that an `Order` might be orphaned in the database, but it's mostly safe.

When we push our type safety back, we're encouraged to continue pushing it back. Eventually, we push it all the way back – to the edges of our system! This simplifies all of the code and logic inside of the system. We're taking advantage of types to make our code simpler, safer, and easier to understand.

10.2 Ask Only What You Need

In many senses, designing our code with type safety in mind is about being as strict as possible about your possible inputs. Haskell makes this easier than many other languages, but there's nothing stopping you from writing a function that can take literally any binary value, do whatever effects you want, and return whatever binary value:

```
1 foobar :: ByteString -> IO ByteString
```

A `ByteString` is a totally unrestricted data type. It can contain any sequence of bytes. Because it can express any value, we have little guarantees on what it actually contains, and we are limited in how we can safely handle this.

By restricting our past, we gain freedom in the future.

These ideas are expanded upon in Alexis King's blog post Parse, Don't Validate[2]. It remains my favorite explanation on the topic. Gary Bernhardt's talk Boundaries[3] explores this as well. Data at the boundary of your program is unstructured, vast, infinitely complex and unknown. Whenever we learn something about the data, we can choose to encode that in the types (or not). The more we know about it, the less room we have for careless mistakes.

If the set of possible choices is great, then the probability of error is high. Narrowing down the possible error cases allow us to program more mechanically. This saves our brainpower for problems that require it.

[2]https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/
[3]https://www.destroyallsoftware.com/talks/boundaries

11. Keep Your Types Small...

... and your bugs smaller.

In my previous chapter "Type Safety Back and Forth", I discussed two different techniques for bringing type safety to programs that may fail. On the one hand, you can push the responsibility forward. This technique uses types like `Either` and `Maybe` to report a problem with the inputs to the function. Here are two example type signatures:

```
1  safeDivide
2      :: Int
3      -> Int
4      -> Maybe Int
5
6  lookup
7      :: Ord k
8      => k
9      -> Map k a
10     -> Maybe a
```

If the second parameter to `safeDivide` is `0`, then we return `Nothing`. Likewise, if the given `k` is not present in the `Map`, then we return `Nothing`.

On the other hand, you can push it back. Here are those functions, but with the safety pushed back:

```
safeDivide
    :: Int
    -> NonZero Int
    -> Int

lookupJustified
    :: Key ph k
    -> Map ph k a
    -> a
```

With `safeDivide`, we require the user pass in a `NonZero Int` - a type that asserts that the underlying value is not `0`. With `lookupJustified`, the `ph` type guarantees that the `Key` is present in the `Map`, so we can pull the resulting value out without requiring a `Maybe`. (Check out the tutorial[1] for `justified-containers`, it is pretty awesome)

11.1 Expansion and Restriction

"Type Safety Back and Forth" uses the metaphor of "pushing" the responsibility in one of two directions:

- forwards: the caller of the function is responsible for handling the possible error output
- backwards: the caller of the function is required to providing correct inputs

However, this metaphor is a bit squishy. We can make it more precise by talking about the "cardinality" of a type - how many values it can contain. The type `Bool` can contain two values `True` and `False`, so we say it has a cardinality of 2. The type `Word8` can express the numbers from 0 to 255, so we say it has a cardinality of 256.

The type `Maybe a` has a cardinality of `1 + a`. We get a "free" value `Nothing :: Maybe a`. For every value of type `a`, we can wrap it in `Just`.

[1] https://hackage.haskell.org/package/justified-containers-0.3.0.0/docs/Data-Map-Justified-Tutorial.html

The type `Either e a` has a cardinality of `e + a`. We can wrap all the values of type `e` in `Left`, and then we can wrap all the values of type `a` in `Right`.

The first technique - pushing forward - is "expanding the result type." When we wrap our results in `Maybe`, `Either`, and similar types, we're saying that we can't handle all possible inputs, and so we must have extra outputs to safely deal with this.

Let's consider the second technique. Specifically, here's `NonZero` and `NonEmpty`, two common ways to implement it:

```
1  newtype NonZero a
2      = UnsafeNonZero
3      { unNonZero :: a
4      }
5
6  nonZero :: (Num a, Eq a) => a -> Maybe (NonZero a)
7  nonZero 0 = Nothing
8  nonZero i = Just (UnsafeNonZero i)
9
10 data NonEmpty a = a :| [a]
11
12 nonEmpty :: [a] -> Maybe (NonEmpty a)
13 nonEmpty []     = Nothing
14 nonEmpty (x:xs) = x :| xs
```

What is the cardinality of these types?

`NonZero a` represents "the type of values `a` such that the value is not equal to `0`." `NonEmpty a` represents "the type of lists of `a` that are not empty." In both of these cases, we start with some larger type and remove some potential values. So the type `NonZero a` has the cardinality `a - 1`, and the type `NonEmpty a` has the cardinality `[a] - 1`.

Interestingly enough, `[a]` has an infinite cardinality, so `[a] - 1` seems somewhat strange - it is also infinite! Math tells us that these are even the *same* infinity. So it's not only mere cardinality that helps - it is the specific value(s) that we have removed that makes this type safer for certain operations.

These are custom examples of refinement types[2]. Another closely related idea is quotient types[3]. The basic idea here is to *restrict* the size of our inputs. Slightly more formally,

- Forwards: expand the range
- Backwards: restrict the domain

11.2 Constraints Liberate

Runar Bjarnason has a wonderful talk titled Constraints Liberate, Liberties Constrain[4]. The big idea of the talk, as I see it, is this:

> When we restrict what we can do, it's easier to understand what we can do.

I feel there is a deep connection between this idea and Rich Hickey's talk Simple Made Easy[5]. In both cases, we are focusing on simplicity - on cutting away the inessential and striving for more elegant ways to express our problems.

Pushing the safety forward - expanding the range - does not make things simpler. It provides us with more power, more options, and more possibilities. Pushing the safety backwards - restricting the domain - does make things simpler. We can use this technique to take away the power to get it wrong, the options that aren't right, and the possibilities we don't want.

Indeed, if we manage to restrict our types sufficiently, there may be only one implementation possible! The classic example is the `identity` function:

```
1 identity :: a -> a
2 identity a = a
```

[2] https://ucsd-progsys.github.io/liquidhaskell-tutorial/
[3] https://www.hedonisticlearning.com/posts/quotient-types-for-programmers.html
[4] https://www.youtube.com/watch?v=GqmsQeSzMdw
[5] https://www.youtube.com/watch?v=SxdOUGdseq4

This is the only implementation of this function that satisfies the type signature (ignoring `undefined`, of course). In fact, for any function with a sufficiently precise type signature, there is a way to automatically derive the function! Joachim Breitner's `justDoIt`[6] is a fascinating utility that can solve these implementations for you.

With sufficiently fancy types, the computer can write even more code for you. The programming language Idris can write well-defined functions like `zipWith` and `transpose` for length-indexed lists nearly automatically![7]

11.3 Restrict the Range

I see this pattern and I am compelled to fill it in:

	Restrict	**Expand**
Range		:(
Domain	:D	

I've talked about restricting the domain and expanding the range. Expanding the domain seems silly to do - we accept more possible values than we know what to do with. This is clearly not going to make it easier or simpler to implement our programs. However, there are many functions in Haskell's standard library that have a domain that is too large. Consider:

```
1 take :: Int -> [a] -> [a]
```

`Int`, as a domain, is both too large and too small. It allows us to provide negative numbers: what does it even mean to take `-3` elements from a list? As `Int` is a finite type, and `[a]` is infinite, we are restricted to only using this function with sufficiently small `Int`s. A closer fit would be `take :: Natural -> [a] -> [a]`. `Natural` allows any non-negative whole number, and perfectly expresses the reasonable domain. Expanding the domain isn't desirable, as we might expect.

[6] https://www.joachim-breitner.de/blog/735-The_magic_%E2%80%9CJust_do_it%E2%80%9D_type_class

[7] https://youtu.be/X36ye-1x_HQ?t=1140

The `base` package has functions with a *range* that is too large, as well. Let's consider:

```
1 length :: [a] -> Int
```

This has many of the same problems as `take` - a list with too many elements will overflow the `Int`, and we won't get the right answer. Additionally, we have a guarantee that we *forget* - a `length` for any container must be positive! We can more correctly express this type by *restricting* the output type:

```
1 length :: [a] -> Natural
```

A `Natural` carries the proof that it is an unbounded, positive number. This preserves information and allows us to push safety further throughout our code.

11.4 A perfect fit

The more precisely our types describe our program, the fewer ways we have to go wrong. Ideally, we can provide a correct output for every input, and we use a type that tightly describes the properties of possible outputs.

12. 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:

```
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:

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

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

```
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.

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

Now, this slightly cleans up the code:

```
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:

```
case foo "hello world" of
    Right val ->
        pure val
    Left err ->
        case err of
            AllParseError parseError ->
                handleParseError parseError
            AllLookupError lookupErr ->
                handleLookupError
```

```
10              AllHeadError headErr ->
11                  handleHeadError
12              _ ->
13                  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:

```
1  bar :: String -> Either AllErrorsEver Integer
2  bar str =
3      case foo str of
4          Right val ->
5              Right val
6          Left err ->
7              case err of
8                  AllParseError pe ->
9                      Right (handleParseError pe)
10                 _ ->
11                     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.

12.1 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.

12.2 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.

12.3 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.

12.4 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.

```
1 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

13. Exceptions

In the previous chapter, we talked about why modeling errors is difficult. GHC supports runtime exceptions. These exceptions have many of the same features that you might find in a more mainstream programming language, like Java. Exceptions are propagated through an untyped channel. They use a form of subtyping, popular in object oriented programming languages. They have runtime-type information - used to perform type-casts!

Haskell's exception system is one of the trickier parts of the language to learn. I don't believe that is a coincidence. One of the most mainstream-seeming features of the language is one of the more difficult to get right!

In this chapter, we're going to explore some of Haskell's dynamic typing facilities, too.

Let's get down to the dirty business of runtime exceptions in Haskell.

13.1 Exceptions In Five Minutes

You don't need to understand exceptions deeply in order to use them productively. Let's consider this code sample:

```
1  import Control.Monad (when)
2  import Control.Exception
3      (throwIO, catch, Exception)
4
5  data MyException = MyException
6      deriving Show
7
8  instance Exception MyException
9
10 problem :: String -> IO Int
```

```
problem foo = do
    when (foo == "uh oh") $ do
        throwIO MyException
    pure (length foo)

handler :: IO Int
handler =
    problem "uh oh"
        `catch`
            \MyException -> do
                putStrLn "handled!!!"
                pure 0
```

We can derive `Exception` if there's a `Show` instance for the datatype. Once we have that `Exception` instance, we can then use `throwIO` to throw values of this type in the runtime system. And we can use `catch`, `handle`, and `try` to deal with exceptions when they happen.

If you want to catch every possible runtime exception, use the `SomeException` type:

```
noExceptionsEver
    :: IO a
    -> IO (Either SomeException a)
noExceptionsEver action =
    try action
```

You need to specify the type you're catching somehow. It can be inferred sometimes, but a lot of the time, you'll need to provide a type annotation. You'll probably need `ScopedTypeVariables` language extension to do this easily.

The following code samples are all equivalent.

```haskell
ex0 = do
    eres <- try (pure ())
    case eres of
        Left (err :: IOException) ->
            print 10
        Right () ->
            print 20

ex1 = do
    pure () >> print 20
        `catch` \(err :: IOException) ->
            print 10

ex2 = do
    handle (\(err :: IOException) -> print 10) $ do
        pure ()
        print 20
```

So, that's the easy stuff.

13.2 Best Practices

The phrase "best practices" is riddled with assumptions, so let's pretend I titled the section "stuff I like." That's more honest, isn't it? I've been bitten by exceptions many times. They can be a nasty handful. They can also be rather pleasant and informative. I've figured out a bag of tricks that help make exceptions more useful.

Single Constructor

An exception type should have a single constructor. The single constructor should share the name of the type.

In "The Trouble With Typed Errors", I claim that is the right thing to do, and I demonstrate why multiple cases on an error type means you can't handle or catch those individual constructors - it's a package deal. But

requiring exception type and constructor names to match is a further restriction. The reason is due to the type-based mechanism by which you catch exceptions.

Consider `IOException`. It has a single constructor: `IOError`. If you want to catch it, you can write a type signature on the lambda, or you can pattern match on the constructor:

```
handle (\(err :: IOException) -> ...)
handle (\ err@IOError{..}     -> ...)
```

But it's not obvious that these two lambdas handle the same exception types. Worse, suppose we have an exception `data Err = X | Y`. I can write a handler function that will fail at runtime!

```
handle (\X -> ...)
```

This will fail if you throw a `Y` with a pattern match exception - and this destroys the original `Y` exception information! But you won't get even a warning without explicitly turning on `-Wincomplete-uni-patterns`, which is not enabled by the `-Wall` flag. Simply defining two separate error types will allow this exception handling behavior to work exactly as you might want it to.

If an exception has a single constructor with the same name as the type, it becomes obvious what type is being caught. The default Haskell exception handler calls `show` on the exception to render it to the user. If `Show` is derived for the exception type, then the user knows exactly what exception to catch - because the constructor name shares the type name. If my program dies with this output:

```
***Exception:
    UniqueKeyViolation
        (UserKey 1234)
        "the key 1234 is already present in the users table"
```

and the underlying library follows this advice, you can write:

```
try (insert user) :: IO (Either UniqueKeyViolation User)
```

This convention makes it easier to write code that handles exceptions safely.

Unit Tests

An exception should carry enough information to write a test with. Ideally, the end-user should know exactly what they did wrong, so they can fix the error on their side. It should *not* use `Text` or `String` to convey that information - prefer structured types and data whenever possible. It is always easy to destroy information by calling `show` on a value. It is much more difficult to recover that information.

A bad error message is:

```
Network.bind: socket busy
```

Which socket? What is being done? Why? The exception doesn't include this information.

A nicer exception would tell you exactly what went wrong - what values caused a problem, what errors were encountered, and why it caused an issue. This information should be conveyed - as much as possible - with Haskell values that can be manipulated. Instead of `String` messages, use constructors and well typed fields.

```
-- bad
data SqlError = SqlError Text

-- good
data UniqueKeyViolation
    = UniqueKeyViolation DbValue TableName ConstraintName
```

The more information you can include, the easier your users will find it to fix the problems in their code. More information also aids handling the exception when the exception is eventually caught.

If you write a library, and the underlying library throws exceptions, you should wrap those exceptions with additional information, if you are able to. For example, a low-level HTTP library might throw `data Err404`. A higher level library for a specific web service might wrap the `Err404` type in a more informative type that says exactly what went wrong.

Use a Hierarchy

Exception hierarchies are covered in the next section of this chapter. You should use them. Exception hierarchies allow you to catch multiple exceptions all at once and handle them uniformly. While this is bad practice when catching `SomeException`, it works great for a small domain of errors that a library or application component may throw.

You may think: "Why not just a sum type of the possible error cases?" This does give users the ability to catch "all exceptions thrown by my library." However, you run afoul of all the problems with "too large" types.

Libraries that define an exception hierarchy around the exceptions they define make it relatively easy to know what exceptions a library might throw. Putting these all in a module named `.Exceptions` makes it even easier! If you're also wrapping exceptions from libraries you call, then you ensure that your library users won't be surprised by exceptions that bubble up from your code. A little bit of effort here saves a lot of time downstream.

Decorating Exceptions with Information

Sometimes you want to annotate exceptions with information, but it's clumsy to do so for every exception type. Suppose you wanted all of your exceptions to include the `UserId` that was signed in to the service that triggered the exception. Now, you need to include the `UserId` on a field for every exception type:

```
-data MyException = MyException Int
+data MyException = MyException UserId Int

-data FooException = FooException Char
+data FooException = FooException UserId Char

{- etc... -}
```

What's worse is that you need to pass that `UserId` to every function that might throw an exception. But this doesn't even solve the problem - third party code could throw an exception that bypasses this! So you put in a bunch of work, and it's all for nothing.

I wrote `annotated-exception`[1] to solve this problem. This library allows you to attach arbitrary data to thrown exceptions. You use a function `checkpoint` to attach an `Annotation`, and that annotation is included on any exception that escapes the `checkpoint`.

With this technique, we don't have to pass extraneous information around. We don't have to modify our core exception types. We also get coverage for exceptions we never defined. This function is one of many possible "`local`-like" functions which provide additional information in a lexically scoped way. For example, we can build this into an authorization function:

```
withLoggedInUser :: Email -> Password -> (Entity User -> App a) -> App a
withLoggedInUser email password action = do
    entityUser <- login email password
    checkpoint (Annotation (entityKey entUser)) $ do
        action entUser
```

With this code, any exception that escapes the `action entUser` gets annotated with the `UserId` that was logged in. This makes debugging the problem easier.

You will likely want to extend this `local`-like with extra functionality. For example, this provides information on exceptions that *escape* the scope, but it does nothing else. We may want to append this information to each

[1] https://hackage.haskell.org/package/annotated-exception

log entry in the scope. Or, we may want to append the information to each bug report from the scope. These lexically scoped `local`-like functions are extremely powerful in their simplicity.

13.3 Hierarchies

Exception hierarchies give Haskell a flavor of subtyping polymorphism. The documentation for the `Exception` type class[2] covers this pretty well, but I'll recap briefly.

All exceptions "inherit" from `SomeException`. When you define a default instance of the `Exception` class, your exceptions can still be caught under a `SomeException` type. You can define a "subtype" wrapper that looks like this:

```
data MyException where
    MyException :: Exception e => e -> MyException

-- alternatively, without GADT syntax,
data MyException
    = forall e. Exception e => MyException e
```

You'll want to provide a means of casting to and from this wrapper type. These functions all look like this:

```
import Data.Typeable (cast)
import Control.Exception (toException, fromException)

fromMyException :: Exception e => SomeException -> Maybe e
fromMyException someExn = do
    MyException e <- fromException someExn
    cast e

toMyException :: Exception e => e -> SomeException
toMyException e = toException (MyException e)
```

[2] https://hackage.haskell.org/package/base-4.14.0.0/docs/Control-Exception.html#t:Exception

Now, armed with these functions and types, we can write an exception that inherits this.

```
data X = X

instance Exception X where
    toException = toMyException
    fromException = fromMyException
```

With this, we now have three possible ways to catch an `X` error:

1. `try :: IO a -> IO (Either X a)`
2. `try :: IO a -> IO (Either MyException a)`
3. `try :: IO a -> IO (Either SomeException a)`

But catching `MyException` means that you'll catch any *other* exceptions that are defined to inherit from this class.

All of the boilerplate for defining exception hierarchies is pretty annoying, so I created the `exception-via`[3] library to make it more convenient. Let's define another one.

```
data Y = Y
    deriving Exception via Y <!!! MyException
```

This technique uses the `DerivingVia` language extension, which allows for convenient and powerful deriving of type class instances.

13.4 Reinventing

Haskell's exceptions are mostly implemented as library code. The efficient implementation for throwing and catching in `IO` is baked into the GHC runtime. We can use `ExceptT` for a slower version, but it'll work fine to demonstrate how the exception system works.

[3] https://hackage.haskell.org/package/exception-via

Typeable

Exceptions use run-time type information to determine what exceptions to catch. Usually, Haskell types are completely erased - type information does not exist at run-time. We have the primitive function `unsafeCoerce :: a -> b` which can cast types, but it's terribly unsafe. What we want is a function `cast :: a -> Maybe b` that could allow us to make these type-casts safely.

The `Typeable` type class gives us this run-time type information. This class is automatically defined for every type - you don't need to specify it in any `deriving` clauses. You are completely forbidden from writing your own instances of it.

The key method for `Typeable` is the function `typeRep`:

```
1  typeRep :: (Typeable a) => proxy a -> TypeRep
```

Somewhat confusingly, `TypeRep` is an alias for `SomeTypeRep`, which wraps the *real* `TypeRep (a :: k)` - a kind indexed singleton datatype that represents types in Haskell and supports fast equality, serialization, and deserialization. While the details are fascinating[4], we don't need to worry about them for our purposes. If two `TypeReps` for a type are equal, then they are the same type. This means we can write our `safeCast` function.

```
1  {-# language ExplicitForAll #-}           -- [1]
2  {-# language ScopedTypeVariables #-}      -- [2]
3
4  module ExceptionPrime where
5
6  import Data.Proxy (Proxy(..))             -- [3]
7  import Control.Monad (guard)
8  import Unsafe.Coerce (unsafeCoerce)
9  import Data.Typeable (Typeable, typeRep)
10
11 safeCast
```

[4] https://www.seas.upenn.edu/~sweirich/papers/wadlerfest2016.pdf

```
12      :: forall b a.                          -- [4]
13      (Typeable a, Typeable b)                -- [5]
14      => a -> Maybe b
15  safeCast a = do
16      let
17          aRep = typeRep (Proxy :: Proxy a) -- [6]
18          bRep = typeRep (Proxy :: Proxy b)
19      guard (aRep == bRep)                    -- [7]
20      pure (unsafeCoerce a)                   -- [8]
```

Let's dig in to this.

1. We need `ExplicitForAll` in order to write the `forall a b.` syntax to explicitly introduce the type variables.
2. We need `ScopedTypeVariables` to put the type variables `a` and `b` in scope in the function body, so we can then use `Proxy :: Proxy a`. Without this extension, the `Proxy :: Proxy a` would refer to a totally different type variable, and the function wouldn't type check.
3. I usually don't bother writing imports explicitly, but since we're reinventing stuff, it's good to be explicit about what is new and what is defined here.
4. The order in which type variables are introduced is an important component of programming at the type-level. Without an explicit `forall`, it's assumed that type variables are introduced in the order they are used. With an explicit `forall`, we can provide a more convenient interface. By giving the result type as the first type variable, that allows end users to write `safeCast @Foo bar`. If the order were `forall a b`, then the end user would need to write `safeCast @_ @Foo bar` or `safeCast bar :: Maybe Foo`.
5. The `Typeable a` constraint means that this function requires run-time type information. Remember - constraints in Haskell translate to dictionaries that are passed implicitly. At run-time, this function accepts three arguments: the two `Typeable` dictionaries and the `a` value.
6. We call `typeRep (Proxy :: Proxy a)` to get the `aRep :: TypeRep` value. The definition of `typeRep` doesn't specify which `Proxy` you pass - anything can work. Before `Proxy` was a part of `base`, you could have passed `Nothing :: Maybe a`, and this would have worked fine.

7. The function `guard :: Alternative m => Bool -> m ()` will call `empty` if the argument is `False`. For `Maybe`, this means `Nothing`.
8. If the `guard` was `True`, then we know that the `TypeRep` for `a` and `b` are equal. This means that the type `a` and `b` are the same. Which means that it is safe to write `unsafeCoerce a :: b`.

`Refl`

This pattern is going to be somewhat common. We want to compare two types for equality, and conditionally act on the result. It'd be nice to avoid writing `unsafeCoerce` all over the place, too.

GHC Haskell gives us a type operator `(~) :: k -> k -> Constraint` translates to type equality. We can use this for assigning shorthand names:

```
1  before
2      :: ExceptT e (StateT s (ReaderT r IO)) a
3      -> ExceptT e (StateT s (ReaderT r IO)) b
4      -> ExceptT e (StateT s (ReaderT r IO)) c
5
6  after
7      :: (monad ~ ExceptT e (StateT s (ReaderT r IO)))
8      => monad a
9      -> monad b
10     -> monad c
```

We can write `safeCast` with this constraint:

```
1  safeCast :: (a ~ b) => a -> Maybe b
2  safeCast a = Just b
3    where
4      b = a
```

Actually, there's no way to call this with two different types. It's impossible for it to fail - we don't have any run-time type information to check on! We have to statically know that `a ~ b` is true. But then the `Maybe` is unnecessary - we can just write it as:

```
id :: (a ~ b) => a -> b
id a = b
  where
    b = a
```

That's not what we want. So how do we get the constraint in scope, conditionally, based on a run-time value?

The answer is a GADT.

```
data a :~: b where
    Refl :: a :~: a
```

This technique is a bit opaque. The type is named `:~:`, and it's trying to invoke familiarity with the constraint operator $\sim$ for type equality. `Refl` stands for "`Refl`exive Equality" - if `a == b` then `b == a` should hold too. Now that we know the name, how does it work? The GADT syntax is hiding the constraint that we're packing. We can rewrite it in the "standard" form to make it explicit:

```
data TypeEquality a b = (a ~ b) => TypesAreEqual
```

This is *still* doing something tricky - that `(a ~ b) => TypesAreEqual` looks an awful lot like a nullary constructor, like `Proxy`:

```
data Proxy a = Proxy
```

But we're actually carrying around the *constraint* in the data constructor. If we were to write it fully explicitly (using the `Dict` type from the `constraints`[5] package), it would look like this:

```
data TypeEqualityExplicit a b = TypesAreEqualExplicit (Dict (a ~ b))
```

When we pattern match on `TypesAreEqual`, we introduce the constraint `a ~ b` into scope. This is part of `GADTs`s powers - constraints and types can be brought into scope with a pattern match. Now we can limit our use of `unsafeCoerce` to a single location:

[5] https://hackage.haskell.org/package/constraints

```
typeEquality
    :: forall a b. (Typeable a, Typeable b)
    => Maybe (a :~: b)
typeEquality =
    if typeRep (Proxy @a) == typeRep (Proxy @b)
        then Just (unsafeCoerce Refl)
        else Nothing
```

Let's try to use this.

```
safeCast'' :: forall b a. (Typeable a, Typeable b) => a -> Maybe b
safeCast'' a =
    case typeEquality @a @b of
        Just Refl ->
            Just a
        Nothing ->
            Nothing
```

This works! It compiles and it runs. Pattern matching on the `Refl` constructor brings the constraint `a ~ b` into scope, which tells GHC that we can treat an `a` like a `b` in this case. We wrap it up in a `Just`, which allows us to handle the `Unknown` case with a `Nothing`.

Dynamic

We now have the ability to create a dynamic type. Interestingly, `Dynamic` types in Haskell works much like they do in other languages.

```
data Dynamic where
    Dynamic :: Typeable a => a -> Dynamic
```

A `Dynamic` is a value paired with it's run-time type information, with the concrete type hidden. We can cast from a `Dynamic`:

```
fromDyn :: forall a. Typeable a => Dynamic -> Maybe a
fromDyn (Dynamic (a :: b)) =      -- [1]
    case typeEquality @a @b of    -- [2]
        Just Refl -> Just a       -- [3]
        Nothing -> Nothing
```

1. When pattern matching on the `Dynamic`, we can provide a type signature to the `a` value. This introduces the `b` type variable.
2. Now we can call `typeEquality` and provide the `a` and `b` type variables as `TypeApplications` arguments.
3. If the two types are equal, then we can return what you expected. If not, we give you `Nothing` back.

This is a bit less useful than it is in languages with implicit subtyping. Note that we can't cast an `Int` to an `Integer`:

```
λ> fromDyn @Int (Dynamic (3 :: Integer))
Nothing
```

In Ruby, we'd expect a `Dynamic (3 :: Integer)` to be able to respond to any numeric messages. Haskell's dynamic types aren't quite as powerful.

Now that we have our `Dynamic` types, we're ready to have exceptions.

Either Dynamic

Alright, let's define the two critical functions for exceptions: `throw` and `catch`.

```
throw :: (Typeable e) => e -> Either Dynamic a
throw e = Left (Dynamic e)
```

This one is easy enough - we're hiding information, after all. `catch` will be a bit more tricky, because we're refining information: we're producing something of a type we may not know about! Let's look at our type signature:

```
catch
    :: forall e a. (Typeable e)
    => Either Dynamic a        -- ^ the action
    -> (e -> Either Dynamic a) -- ^ the exception handler
    -> Either Dynamic a        -- ^ the result
```

Before we go too much further, try writing this one yourself.

OK, let's get it, line by line.

```
catch action handler =
```

First up, let's `case` on the `action`. If it is a `Right` value, then we can just return `Right`. Easy.

```
    case action of
        Right a -> Right a
```

If it's a `Left Dynamic`, then we'll want to pattern match on `Dynamic` to bring the type variable into scope. Then, we can use `typeEquality` to compare the type that we're trying to catch and the type we actually have.

```
        Left (Dynamic (err :: err)) ->
            case typeEquality @err @e of
```

If the two types are equal, that means we know that `e ~ err`, and we can call the `handler :: e -> Either Dynamic a` function. If they are not equal, then we have to return the original error.

```
                Just Refl -> handler err
                Nothing -> Left (Dynamic err)
```

Let's try it out.

```
λ> throw (1 :: Int) `catch` (\(x :: Int) -> Right "Got it")
Right "Got it"
λ> throw (1 :: Int) `catch` (\(x :: Integer) -> Right "Got it")
Left <<dynamic>>
```

That's it - we can throw and catch dynamically typed exceptions. But we're missing the hierarchies of exceptions that gave us subtyping. Let's implement that.

A New Hierarchy

The `Dynamic` type allows us to select single, individual types. The hierarchy is flat - `Dynamic` contains everything equally. But we want to be able to talk about exceptions being subtypes of other exceptions. Let's say we've got an `AppException` type with `HttpException` as a subtype. We can represent that like so:

```
data AppException where
    AppException :: Typeable a => a -> AppException

data HttpException = HttpException
data DbException = DbException
```

We're going to want to be able to catch a mere `HttpException`, as well as any `AppException` at all that might get thrown.

```
throwApp :: Typeable e => e -> Either Dynamic a
throwApp e = Left (Dynamic (AppException e))

catchApp
    :: forall e a. (Typeable e)
    => Either Dynamic a
    -> (e -> Either Dynamic a)
    -> Either Dynamic a
catchApp action handler =
    action `catch` \e -> case e of
```

```
        AppException (e :: err) ->
            case typeEquality @e @err of
                Just Refl -> handler e
                Nothing -> throwApp e
```

This almost works. We can catch and throw them directly, but we have to remember to use the specific *App functions.

```
-- Good: should catch, does
λ> throwApp HttpException `catchApp` \HttpException -> Right "Got it"
Right "Got it"
-- Good: should not catch, doesn't
λ> throwApp HttpException `catchApp` \DbException -> Right "Got it"
Left <<dynamic>>
-- Good: should not catch, doesn't
λ> throwApp HttpException `catch` \DbException -> Right "Got it"
Left <<dynamic>>
-- Good: should catch, does
λ> throw HttpException `catch` \HttpException -> Right "Got it"
Right "Got it"

-- Bad: should catch, doesn't
λ> throwApp HttpException `catch` \HttpException -> Right "Got it"
Left <<dynamic>>
-- Bad: should catch, does not
λ> throw HttpException `catchApp` \HttpException -> Right "Got it"
Left <<dynamic>>
```

Furthermore, there's nothing specific about the `throwApp` that requires we throw a specific `App` exception. There's nothing saying we can't write `throwApp (3 :: Int)`. It seems like the way we convert a type into a `Dynamic` should be a property of the *type* and not the function we use to `throw`. So let's make a type class that tells us how to convert a type to a `Dynamic`, and we'll redefine `throw` in terms of it.

```
class Typeable e => Exception e where
    toException :: e -> Dynamic

instance Exception Dynamic where
    toException = id

instance Exception AppException where
    toException = Dynamic

instance Exception HttpException where
    toException = Dynamic . AppException

instance Exception DbException where
    toException = Dynamic . AppException

throw :: Exception e => e -> Either Dynamic a
throw err = Left (toException err)
```

Unfortunately, catching is now broken - it only works to catch an exception that is a direct inheritor of `Dynamic`, not anything else.

```
λ> throw HttpException `catch` \(AppException err) -> Right "Got it"
Right "Got it"
λ> throw HttpException `catch` \HttpException -> Right "Got it"
Left <<dynamic>>
```

This suggests that we need to also handle converting from `Dynamic` as part of the class.

```
class (Typeable e) => Exception e where
    toException :: e -> Dynamic
    fromException :: Dynamic -> Maybe e

instance Exception Dynamic where
    toException = id
    fromException = Just

instance Exception AppException where
    toException = Dynamic
    fromException = fromDyn

instance Exception HttpException where
    toException = Dynamic . AppException
    fromException dyn = do
        AppException e <- fromException dyn
        safeCast''  e

instance Exception DbException where
    toException = Dynamic . AppException
    fromException dyn = do
        AppException e <- fromException dyn
        safeCast''  e
```

Indeed, the pattern in `FromException` for our two subtypes can be factored out:

```
fromAppException :: Typeable e => Dynamic -> Maybe e
fromAppException dyn = do
    AppException e <- fromDyn dyn
    safeCast'' e

instance Exception HttpException where
    toException = Dynamic . AppException
    fromException = fromAppException

instance Exception DbException where
```

```
    toException = Dynamic . AppException
    fromException = fromAppException
```

Now, we'll rewrite `catch` to use `fromException` rather than comparing the types directly:

```
catch
    :: forall e a. (Exception e)
    => Either Dynamic a
    -> (e -> Either Dynamic a)
    -> Either Dynamic a
catch action handler =
    case action of
        Right a ->
            Right a
        Left err ->
            case fromException err of
                Just e -> handler e
                Nothing -> throw err
```

And it works!

```
λ> throw HttpException `catch` \HttpException -> Right "Got it"
Right "Got it"
λ> throw HttpException `catch` \(AppException err) -> Right "Got it"
Right "Got it"
```

Now you know exactly how exceptions work in Haskell - at least, as far as we're able to care, without digging too deep into the runtime system.

13.5 Asynchronous Exceptions

Asynchronous exceptions are an extremely tricky subject. Java *used* to have them, but they were removed due to complicating the language and implementation. Haskell retains them, and this decision has lost a lot of

programmers a lot of time. However, they have significant advantages for multithreaded programming, so it's impossible to say if they are good or bad.

I won't cover this topic in depth. Instead, I'm going to offer a brief overview, and then point you to some great resources.

In Five Minutes

`Control.Exception` exposes a function `throwTo`

```
throwTo
    :: (Exception e)
    => ThreadId
    -> e
    -> IO ()
```

When you throw an exception to a thread, that exception is delivered whenever the thread may be interrupted. Threads can be interrupted while *allocating memory*, so they can be interrupted at pretty much any time.

```
thread1 :: IO ()
thread1 = do
    threadId <-
        forkIO $ do
            let !result = sum $
                    map someExpensiveFunction [ 1 .. 100000 ]
            print result
    throwTo threadId (userError "oh no")
```

The forked thread attempts to calculate the relatively large number, but will be interrupted by the asynchronous exception before it can finish. Compare this with Java - while you can explicitly interrupt a thread, the receiving thread must either be polling for interruptions or doing a `Thread.sleep` or comparable blocking function. If we tried to write the

above example in Java that would safely be interruptible, we'd need to have an alternative `sum` or `map` that occasionally polled for interrupts.

You can `mask` an action, which means that your thread will only receive asynchronous exceptions while blocked. The runtime system is blocked when performing file IO, or when trying to `takeMVar`. This state is pretty similar to Java's system for handling thread interruption.

Finally, you can `uninterruptibleMask` an action, which means that asynchronous exceptions will *never* be delivered. While this seems appealing, it can break the functionality of other libraries that rely on this behavior. Consider the `async` library's `race` method: this runs two actions and provides the result of the first one to complete:

```
race :: IO a -> IO b -> IO (Either a b)
```

This function uses an asynchronous exception to kill the thread that hasn't finished. But if you perform `race` within an `uninterruptibleMask`, then the asynchronous exception won't be delivered, and you'll have to wait until both threads complete.

But suppose you do a relatively common polling technique:

```
updateSomeSharedStatePeriodically :: IO Void
updateSomeSharedStatePeriodically =
    forever $ do
        threadDelay 5000000
        putStrLn "still running . . ."


uninterruptibleMask_ $ do
    race updateSomeSharedStatePeriodically $ do
        performSomeLongRunningWork
```

Since the first action will run `forever`, this code will *never exit*, even when `performSomeLongRunningWork` completes.

You generally do not want to do `uninterruptibleMask` unless you know *exactly* what you are doing. The way this can bite you is if you are using

the `unliftio` suite or the `safe-exceptions` library. In that library, `bracket` and `onException` performs the cleanup action in an `uninterruptibleMask`. If you use `async` in those places, then you'll get the bad behavior.

Resources

- The `safe-exceptions`[6] package is a good starting point for most Haskell application development.
- The Asynchronous Exception Handling[7] blog post is a good resource for understanding what's going on.
- Parallel and Concurrent Programming in Haskell[8] by Simon Marlow covers asynchronous exceptions well

13.6 The Theory

Haskell did not always have exceptions. It turns out that it is difficult to reconcile laziness and exceptions. The paper 'A Semantics for Imprecise Exceptions'[9] introduced a sound theoretical basis on which to have exceptions while also preserving laziness, equational reasoning, and purity.

In most languages, throwing exceptions is something that happens in a statement. Trying to throw an exception in an expression is often a parse error.

```java
// Java
public class ExceptionTest {
    public statid void main() {
        int a = 1 + throw new Exception();
    }
}
```

This

[6] https://hackage.haskell.org/package/safe-exceptions
[7] https://www.fpcomplete.com/blog/2018/04/async-exception-handling-haskell/
[8] https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/
[9] https://www.microsoft.com/en-us/research/wp-content/uploads/1999/05/except.pdf

gives us the error:

```
error: illegal start of expression
        int x = 1 + throw new Exception();
                    ^
```

JavaScript fails in a similar way:

```
var a = 1 + throw new Error();

// Uncaught SyntaxError: Unexpected token 'throw'
```

Python fails here too:

```
a = 1 + raise Exception

# SyntaxError: invalid syntax
```

For these languages, it's easy to determine a semantics for what exceptions an expression throws: none. Exceptions are only thrown in statements. So the question becomes: what exception does a statement throw? And that's trivial, as a statement that throws an exception can only throw an exception.

Of the languages I'm familiar with, only Ruby allows throwing an exception as an expression:

```
a = 1 + raise(Exception.new 1)
# Exception (1)
```

Ruby evaluates left-to-right here - we're guaranteed that the first argument is evaluated, and then the second, which triggers the exception.

In all of these cases, we can understand precisely what exceptions will be thrown. Given this Java code:

```
throw new Exception();
throw new Exception();
```

You'll get an error - "unreachable statement." In Ruby, if we write:

```
a = raise(Exception.new 1) + raise (Exception.new 2)
```

then we are guaranteed to always receive the first one. Ruby evaluates arguments strictly, from left-to-right, before going into the method call.

In most programming languages, exceptions are a mechanism of control flow. If you have a value, you have *a value*.

Exceptions In Data

In Haskell, exceptions hide inside values. This is because Haskell values may be values, *or* they may be *thunks* that will eventually produce that value.

Haskell's laziness throws a bit of a wrench into things.

```
let a = error "1" + error "2"
```

The semantics of the language don't specify which error will get thrown. GHC is capable of reordering exactly which bit of code will get evaluated before going into the function +. "A Semantics for Imprecise Exceptions" gives the following example:

```
zipWith f [] [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _ = error "Unequal lists"
```

Note that this is not how `zipWith` is actually defined. The actual function truncates when one list ends.

We can have a few calls:

```
zipWith (+) [1] [] =
    error "Unequal lists"
zipWith (+) [1,2] [1] =
    2 : error "Unequal lists"
zipWith div [1,2] [1,0] =
    1 : error "divide by zero" : []
```

These results may or may not throw exceptions depending on how far you evaluate the data structure. Consider these examples. I'm going to step through the evaluation process on a line-by-line basis.

```
-- Initial expression
>>> head (zipWith (+) [1,2] [1])

-- `head` demands the first element of the list.
-- This forces an evaluation of `zipWith`
head ((1 + 1) : zipWith (+) [2] [])

-- head pattern matches on the `:` constructor and returns. It
-- ignores the tail of the list.
head (a : _) = a

head (1 + 1 : _)
1 + 1
2
```

Next, we'll look at an example where we might get a "Division by Zero" error:

```
>>> head (zipWith div [1,2] [1,0])

-- head demands the first element
head (div 1 2 : zipWith div [2] [0])

-- head discards the tail
head (div 1 2 : _)

-- head returns
div 1 2

-- print evaluates to 1
1
```

The two prior examples both use head to discard the tail of the list. Let's see what happens with an example that starts processing the list entirely.

```
-- Initial expression:
>>> zipWith div [1,2] [1,0]

-- GHCi calls `print` on the value, which prints each
-- character one by one.
print (zipWith div [1,2] [1,0])

-- inline definition of print
putStrLn (show (zipWith div [1,2] [1,0]))

-- putStrLn demands first character from `show`
-- show, in turn, demands the first element of the list
putStrLn (show (div 1 2 : zipWith div [2] [0]))

-- show produces an open bracket and then calls show on
-- the first element of the list. the rest of the list
-- gets show', which finishes off the list stuff.
putStrLn ('[' : show (div 1 2) : show' (zipWith div [2] [0]))

-- putStrLn happily outputs the [ character and demands
```

```
-- the next one
putStrLn (show (div 1 2) ++ ',' : show (zipWith div [2] [0]))
[

-- show evaluates it's argument and yields the string:
putStrLn ('1' : ',' : show (zipWith div [2] [0]))

-- putStrLn is able to print the two characters, and
-- then demands the next bit.
putStrLn (show' (zipWith div [2] [0]))
[1,

-- zipWith evaluates out to `div 2 0`
putStrLn (show' (div 2 0 : zipWith div [] []))

-- show' evaluates the argument and produces the error
[1,*** Exception: divide by zero
```

In the prior case, we get a runtime exception because evaluate a value that divides by 0. But we manage to print out a few characters before we even get to the exception. This is laziness.

But what if we don't evaluate it? Let's take the length of the list instead. `length` is defined (more-or-less) like this:

```
length [] = 0
length (_ : xs) = 1 + length xs
```

It does not evaluate the elements of the list - only the structure of the list.

```
>>> length (zipWith div [1,2] [1,0])

-- length demands first cons of list
length (div 1 2 : zipWith div [2] [0])

-- length does not evaluate the first element
length (_ : zipWith div [2] [0])

-- recurse
1 + length (zipWith div [2] [0])
1 + length (div 2 0 : zipWith div [] [])
1 + length (_ : zipWith div [] [])

1 + 1 + length (zipWith div [] [])
1 + 1 + length []
1 + 1 + 0

2
```

In a strict language, this would throw an exception before yielding a single character. In Haskell, we're capable of getting a meaningful result, even if there's an exception hiding in the list.

Since exceptions live in values and are only triggered upon evaluation, and since laziness means that expressions can be reordered, we can't know what exception might be thrown by a given expression. The same syntax might yield different exceptions depending on how the program is evaluated.

Imprecise Exceptions

The solution is to say that an expression can throw from a set of exceptions, not merely a single exception. When two expressions are combined, you take the union of the sets.

```
bad = foo (error "lol") (error "bar") + error "baz"
{- exception set: { "lol", "bar", "baz" } -}
```

This expression could throw any of these exceptions. It is indeterminate - imprecise.

For this reason, we can't write a *pure* function `tryPure` -

```
tryPure :: a -> Either SomeException a
```

Why not? It might have a different result depending on how GHC evaluates the argument. So if we try to `tryPure bad`, we might get `ErrorCall "lol"`, or `ErrorCall "bar"`. This gives us some problems: suppose we have this expression.

```
let x = tryPure bad
 in x == x
```

This should always return `True` - `x == x` should always be `True`! And, it should *also* always be safe to inline pure values in Haskell with their definitions. This is the 'referential transparency' thing that Haskellers love so much.

But if we inline `x`, it'll possibly break!

```
tryPure bad == tryPure bad
```

Depending on exactly how this evaluates, we may get a different result. This is why `try` must be in `IO` - it's not guaranteed to produce the same result for a given input every time, and non-determinism is the domain of `IO`.

13.7 HasCallStack

`HasCallStack` attempts to make it a little easier to understand how and why an error occurred. Unfortunately, it's not particularly useful. You

must mention it in the type signature of every function you want the callstack to appear in. If any function doesn't have it, the chain is broken, and the callstack is lost. So if you call any library code, you probably won't get a callstack.

What's worse is that the callstacks only apply to `error` calls! `throwIO` does not get a callstack. So you only get callstacks with exceptions that don't carry any useful information.

Above, I referred to the `annotated-exception`[10] library. This library provides a partial solution to the above problem: a `CallStack` is included on thrown exceptions. Additionally, the various `checkpoint`, `catch`, etc methods that add information to exceptions *also* add `CallStack` entries. This provides much more information than you get by default, even if you don't provide any other annotations to the exceptions in question.

A future version of GHC should have all exceptions decorated with backtrace information[11], which should result in a nice usability improvement for `CallStacks`.

[10] https://hackage.haskell.org/package/annotated-exception
[11] https://github.com/ghc-proposals/ghc-proposals/pull/330

14. EDSL Design

Haskell folks won't stop talking about DSLs. They mean "Domain Specific Languages" - a relatively small programming language that's designed for a specific purpose. This is in contrast with "general purpose programming languages," which are supposed to be good at doing just about anything. Java, C, Ruby, Python, Haskell, etc are all general purpose programming languages.

It's not a clear binary - some people say that C is a DSL for writing assembly language code. Modern SQL is a Turing complete programming language, but originally it's a DSL for relational algebra and database access. The Cucumber testing framework has a DSL for writing test expectation in semi-natural language. So there's a bit of a spectrum.

An EDSL is an "embedded" domain specific language. Embedded here means "embedded in the host language." A typical DSL is parsed from text into a data structure and then interpreted - much like ordinary programming languages. An embedded language is built in to the host language - it reuses the host language's parser, type checker, etc.

An embedded language is also known as a "library," particularly one that cares about syntax and looking pretty. An EDSL tries to make the host language look as much like another language as possible. Haskell is absolutely brilliant for designing embedded languages. There are a few tricks that allow for remarkable flexibility in syntax:

- `do` notation
- Overloaded literals
- `RebindableSyntax`
- White-space-as-function-application
- Type classes for name overloading
- Custom type error messages
- Infix functions

All languages are, in a sense, data structures. So languages are separated into two phases: constructing that data structure and evaluating it.

14.1 Tricks with do

Do notation is a powerful syntactic tool for lightweight, statement-based syntax. Any type that has an instance of `Monad` can work with it. `RebindableSyntax` allows you to define local `>>=` and `return` functions that don't correspond to `Monad` at all.

Building Data Structures

Lists

do notation can work quite nicely to build tree-like data structures. Programming languages are generally tree-like. Let's start with a list builder. Lists are a sort of tree!

```
1  -- abstract
2  newtype ListBuilder acc a
3      = ListBuilder
4      { unListBuilder :: Writer [acc] a
5      }
6      deriving
7          newtype (Functor, Applicative, Monad)
8
9  -- the API
10 runListBuilder :: ListBuilder acc a -> [acc]
11 runListBuilder (ListBuilder w) =
12     execWriter w
13
14 add :: acc -> ListBuilder acc ()
15 add a =
16     ListBuilder $ tell [a]
17
18 -- our desired syntax
19 list = runListBuilder $ do
20     add 1
21     add 2
```

```
    add 3
    add 4

-- result
list == [1,2,3,4]
```

We can often leverage preexisting monads, without having to write our own. The `Writer` interface is particularly useful for this. `State` is a better choice - it supports the same operations and doesn't have any downsides. The standard `Writer` monad carries a significant performance problem, and the "fix" is exactly equal to the `State` monad. Let's rewrite `ListBuilder` in terms of this:

```
newtype ListBuilder acc a
    = ListBuilder
    { unListBuilder :: State [ac] a
    }

runListBuilder :: ListBuilder acc a -> [acc]
runListBuilder (ListBuilder s) =
    execState s []

add :: acc -> ListBuilder acc ()
add a =
    ListBuilder $ modify (\s -> s ++ [a])
```

Trees

Let's make a tree builder now. Here's the API for our builder:

```haskell
add :: acc -> TreeBuilder acc ()

nest :: TreeBuilder acc a -> TreeBuilder acc ()

buildTree :: TreeBuilder acc a -> Forest acc

data Tree a = Leaf a | Node [Tree a]

type Forest a = [Tree a]
```

`add` adds a single element to the current state. `nest` creates a subtree and adds that to the state. This new function, `nest`, is what turns it into a tree. And syntactically, our expressions will even look like trees.

```haskell
example :: Forest Int
example = buildTree $ do
    add 1
    nest $ do
        add 2
        nest $ do
            add 10
            add 11
        add 3
    add 4
```

We can write it without `do` notation just fine.

```haskell
example =
    [ Leaf 1
    , Branch
        [ Leaf 2
        , Branch
            [ Leaf 10
            , Leaf 11
            ]
        , Leaf 3
```

```
        ]
    , Leaf 4
    ]
```

However, it's nice to be able to reuse `do` notation and all the niceties of `Monad` when you're writing your DSL.

```
example a = buildTree $ do
    add 1
    when (a == 3) $ do
        add 2
        add 3
```

Conditionally adding elements to a list is a little more annoying.

```
example a =
    [Leaf 1]
    ++
    if a == 3
        then
            [ Leaf 2
            , Leaf 3
            ]
        else
            []
```

We can also add further capabilities to our `TreeBuilder` type - including making it a monad transformer with `IO` capabilities. Adding `IO` functionality to the bare `Forest` variant is deeply inconvenient.

Exercise: Maps

Try to write a `Map` building DSL. This is what the end code should look like:

```
asdf :: Map String Int
asdf = runMapBuilder $ do
    set "a" 1
    set "b" 2
```

This is essentially equivalent to:

```
asdf0 :: Map String Int
asdf0 = Map.fromList
    [ ("a", 1)
    , ("b", 2)
    ]
```

BlockArguments

The recent `BlockArguments` language extension improves Haskell syntax by allowing certain constructs (like `do`, `case`, `if`, and lambdas) to be applied as arguments to functions directly. PureScript had this first, and it was good a great idea that Haskell stole it.

The clearest win is with the `runST :: (forall s. ST s a) -> a` function. You'd usually write it like:

```
runST $ do
    a <- newSTRef 0
    forM_ [1..10] $ \i -> do
        modifySTRef a (+i)
    readSTRef a
```

However, `$` doesn't have the right type if you define it yourself. There's a lot of weird compiler magic to special-case `$` that doesn't work for `(.)` or any other operator. `BlockArguments` lets you write the much nicer:

```
runST do
    a <- newSTRef 0
    forM [1..10] \i -> do
        modifySTRef a (+ i)
    readSTRef a
```

There's much less noise from $ operators. We were able to remove the $ after `runST` as well as `forM [1..10]`.

I don't like explaining weird special case rules. They're annoying to look up and talk about, especially in operators, which aren't easily Googleable. That `BlockArguments` allows you to avoid the ubiquitous $ in many cases is reason enough to use it globally, in my opinion.

But - my favorite bit is that it enables paren-free multiple function arguments. Let's consider our own variant of the `if` syntax. The function is easy enough to define:

```
if_ :: Bool -> a -> a -> a
if_ b t f = if b then t else f
```

But it's annoying to use. We have to wrap arguments in parentheses, which isn't at all like the actual `if` expression in Haskell.

```
if a && b
    then do
        x <- readFile "what"
        pure 10
    else do
        putStrLn "it was false"
        pure 20

if_ (a && b)
    (do
        x <- readFile "what"
        pure 10)
    (do
        putStrLn "it was false"
        pure 20)
```

I'm not a fan of parentheses, to be honest. You put a paren *here*, and now you have to put a close-paren somewhere over *there*. Where? It's not immediately knowable. It could be way over there! Then you have to specially structure the text to make it obvious which close-paren goes with which open-paren. But the structure may be wrong and misleading, which makes it harder to read the code.

Fortunately, `do` can be used for non-monadic things, as long as you don't use a bind arrow (`<-`).

```
1  if_
2      do a && b
3      do
4          x <- readFile "what"
5          pure 10
6      do
7          putStrLn "it was false"
8          pure 20
```

We can use `do` as a bullet point of sorts. This makes it especially nice for functions that act as control structures in an EDSL.

In SQL, there's no such thing as `if`. You write a `CASE` expression instead, which is a more flexible variant.

```
1  CASE WHEN cond0 THEN res0
2       WHEN cond1 THEN res1
3       ELSE res2
4  END
```

It's usually nice to have `if`, so I've implemented it in the work codebase for the `esqueleto` database library.

```
if_ :: SqlExpr (Value Bool) -> SqlExpr a -> SqlExpr a -> SqlExpr a
if_ bool whenTrue whenFalse =
  case_
    [ when_ bool then whenTrue
    ]
    (else_ whenFalse)
```

With `BlockArguments`, this cleans up the syntax nicely. You get another nice trick with `do`.

14.2 Overloaded Literals

Haskell allows you to define instance of type classes so you can use literals for your EDSL. This can be a powerful and convenient way to make using your EDSL as seamless as possible.

It does carry some problems. Type inference can become really bad in the presence of overloaded literals. This might mean that your users will be required to provide more type annotations than you might want.

Num

The only one enabled by default is overloaded numeric literals. The `Num` type class includes a method `fromInteger` that is used for integer literals. You need to write an instance of `Fractional` to use decimal points.

Consider this toy language for summing numbers:

```
data Expr = Lit Integer | Add Expr Expr
```

We can give it a `Num` instance and then use integer literals:

```haskell
instance Num Expr where
    fromInteger = Lit
    (+) = Add
    (*) = undefined
    (-) = undefined
    negate = undefined
```

Unfortunately, the `Num` class carries quite a bit of baggage. It also expects you to define several numeric operations on the type. For `Add`, we can actually provide all of these. But there are many types where these won't make sense. It's not uncommon to write a "dummy" instance here - one that promotes numeric literals but errors on anything else.

The above `Num` instance allows us to write:

```haskell
four = Add 2 (Add 1 1)
-- instead of,
four = Add (Lit 2) (Add (Lit 1) (Lit 1))
```

Actually, it also allows us to use + for the `Add` constructor.

```haskell
four :: Expr
four = 2 + 1 + 1
```

Strings

With the `OverloadedStrings` language extension, we can write instances for the `IsString` type class, which allows GHC to interpret any literal string as our data type. This is most commonly used with the `Text` and `ByteString` types.

```haskell
{-# LANGUAGE OverloadedStrings #-}

nonOverloaded :: Text
nonOverloaded = Text.pack "hello world"

overloaded :: Text
overloaded = "hello, world!"
```

Lists

The `OverloadedLists` extension works much like `OverloadedStrings`. We enable the relevant extension, import the relevant module, and write the relevant instance.

The `Map` type from `containers` has an instance of this type class.

```haskell
{-# language OverloadedLists #-}

import GHC.Exts (IsList(..))
import Data.Map (Map)
import qualified Data.Map as Map

instance Ord k => IsList (Map k v) where
    type Item (Map k v) = (k, v)

    fromList = Map.fromList
    toList = Map.toList
```

This would allow you to write:

```haskell
myMap :: Map String Int
myMap =
    [ ("hello", 3)
    , ("foo", 5)
    ]
```

The Function Trick

You can write instances of the literal type classes for functions. This allows you to use literals *as* functions!

```
instance Num (() -> Integer) where
    fromInteger i () = i

lol :: Integer
lol = 5 ()
```

That example is silly, but we can do something more powerful. Let's write a `Day` literal syntax.

```
import Data.Time (Day, fromGregorian)

-- | It's my birthday!
birthday :: Day
birthday = 1988 09 29

instance (a ~ Int, b ~ Int) => Num (a -> b -> Day) where
    fromInteger y m d = fromGregorian y m d
    (+) = undefined
    (*) = undefined
    abs = undefined
    signum = undefined
    negate = undefined
```

Okay, *what*? This warrants some deeper inspection. Let's desugar this a bit.

GHC starts with all whole numbers having the type `Integer`. GHC then inserts a call to `fromInteger` on the type.

```haskell
-- What you write
1988 09 29 :: Day

-- What GHC sees
(fromInteger 1988) (fromInteger 09) (fromInteger 29)
    :: Day
```

Now GHC has to figure out type inference. Syntactically, we're applying the first term to two arguments.

```haskell
(fromInteger 1988 :: a -> b -> Day)
    (fromInteger 09 :: a)
    (fromInteger 29 :: b)
```

GHC searches for an instance that matches `a -> b -> Day`. It finds the one we provided. Then, the `(a ~ Int, b ~ Int)` constraints become part of the constraints that GHC has to satisfy. This unifies fine, and it type checks and works.

Remember the type of `fromInteger :: Integer -> a`. Let's specialize to our new type:

```haskell
fromInteger :: Integer -> a
fromInteger :: (a ~ Int, b ~ Int) => Integer -> (a -> b -> Day)
fromInteger :: Integer -> Int -> Int -> Day
```

This technique was written up by Chris Done as the Constraint Trick for Instances[1]. I'll cover it more in an upcoming section.

We can also do this with strings.

[1] https://chrisdone.com/posts/haskell-constraint-trick/

```haskell
instance (a ~ Int) => IsString (a -> String) where
    fromString str i = concat (replicate i str)

-- >>> "asdf" 3
-- "asdfasdfasdf"
```

This is a powerful way to provide some additional data to a literal value.

14.3 Type Inference Trick

Chris Done has written about this technique and called it The constraint trick for instances[2]. Chris Allen termed the technique instance local fundeps[3]. It's a neat trick that relies on some knowledge of how GHC tracks instances and satisfies constraints.

The type operator $(\sim)$ asserts that two types are equal. The following two type signatures look like they should be identical:

```haskell
foo :: (a ~ Int) => a   -> a   -> a
bar ::              Int -> Int -> Int
```

They are, for all intents and purposes. You might then look at the following instance declarations and wonder what the difference is:

```haskell
instance             IsString (Writer [String] ()) where
      fromString str = tell [str]

instance (a ~ ()) => IsString (Writer [String] a ) where
      fromString str = tell [str]
```

The answer has to do with constraint resolution. When GHC is looking for type class instances, it doesn't care about the constraints on the instance. All it cares about is the "instance head" - the `ClassName Type` bit. Once it finds an instance, it commits to it. When GHC is doing instance lookup for the above two instances, it sees two different things:

[2] https://chrisdone.com/posts/haskell-constraint-trick/
[3] https://bitemyapp.com/blog/instance-local-fundeps/

```
instance ... => IsString (Writer [String] ()) where
instance ... => IsString (Writer [String] a ) where
```

To understand what's going on, let's step through an example that uses this code.

```
example :: [String]
example = execWriter $
    "hello" >> "world"
```

Now, let's assign types to everything.

```
execWriter ::              Writer w a -> w
(>>)       :: (Monad m) => m a -> m b -> m b
example    ::              [String]
```

`example` is the result of calling `execWriter` on some expression, so we can infer backwards that `w ~ [String]`. If we substitute `[String]` for `w`, then we can rewrite our `execWriter` to be specialized:

```
-- substitute w ~ [String]
execWriter :: Writer w a        -> w
execWriter :: Writer [String] a -> [String]
```

We can further specialize the type of `(>>)`, as well - since we use it with `Writer [String]` in the result argument, that must be the same `m` for both inputs.

```
execWriter ::              Writer [String] a -> [String]
(>>)       :: (Monad m) => m a -> m b -> m b
-- Unify `Writer [String] a ~ m b`
-- we get `m ~ Writer [String]` and `a ~ b`
(>>)       :: Writer [String] a -> Writer [String] b -> Writer [String] b
```

Everything is going smoothly so far. GHC sees the two string literals `"hello"` and `"world"`, and assigns them their most general types: `IsString s => s` and `IsString z => z`. Now it's going to try and unify these two types. `"hello"` is used in the first argument to `(>>)`, so we can write that substitution equation:

```
"hello" :: IsString s => s
(>>) :: Writer [String] a -> ...
-- This gives us a constraint: s ~ Writer [String] a
```

Unifying a plain variable with a larger term pretty much always succeeds, so we can specialize.

```
"hello" :: IsString (Writer [String] a) => Writer [String] a
```

GHC now performs instance search for `Writer [String] a`. And - it does not find anything, because the only instance GHC knows about is or `Writer [String] ()`. Unfortunately, `IsString (Writer [String] ())` doesn't match. There's no reason GHC would want to specialize `a` to `()`. So it complains that the type `a` is ambiguous, and if you maybe try specifying it, that'd help. You *can* write `("hello" :: Writer [String] ()) >> ("world" :: Writer [String] ())`, but that sucks.

Now, let's look at the second instance - `IsString (Writer [String] a)`. This matches just fine. The instance head looks exactly like what we're trying to match on! GHC is happy to commit to it. There are strings attached, though - if GHC wants to commit to `IsString (Writer [String] a)`, then it must accept the constraint `(a ~ ())`.

GHC, then, does the following rules and substitutions to figure things out. We'll work inside-out

```
-- starting points
"hello" :: IsString s => s
"world" :: IsString z => z
(>>) :: (Monad m) => m a -> m b -> m b

-- "hello" >> "world" introduces the following constraints:
IsString s => s ~ m a
IsString z => z ~ m b

-- now we can refine the type of `"hello"` and `"world"`
"hello" :: (Monad m, IsString (m a)) => m a
"world" :: (Monad m, IsString (m b)) => m b
```

```haskell

execWriter :: Writer w a -> w
-- execWriter ... introduces another constraint
m ~ Writer w

-- we have a constraint `Monad m` - is this satisfied?
-- GHC does instance lookup and finds:
instance ... => Monad (Writer w) where
-- so yes! it is. This introduces the hidden constraint:
instance (Monoid w) => Monad (Writer w) where

-- so now we got:

"hello" :: (Monoid w, IsString (Writer w a)) => Writer w a
"world" :: (Monoid w, IsString (Writer w b)) => Writer w b

-- and with example being a [String], that means
example :: [String]
w ~ [String]
-- GHC does instance lookup on `Monoid [String]`
-- it matches on `Monoid [a]`, so we can proceed

"hello" :: (IsString (Writer [String] a)) => Writer [String] a
"world" :: (IsString (Writer [String] b)) => Writer [String] b

-- GHC now does a lookup on `IsString (Writer [String] a)`
-- It finds the instance, which satisfies our existing constraint.
-- However, the superclass constraint adds two new ones:

fromString "hello" :: (a ~ ()) => Writer [String] a
fromString "world" :: (b ~ ()) => Writer [String] b

-- GHC can immediately specialize here
fromString "hello" :: Writer [String] ()
tell ["hello"] :: Writer [String] ()

example = execWriter $
```

```
50    tell ["hello"] >> tell ["world"]
```

You may think: why not just write `instance IsString (Writer [String] a)`? Try it. It won't work. `tell :: w -> Writer w ()` - there's no way to get an `a` out of here, it has to be a `()` in the return. But, if you want GHC to commit to the instance, it has to be a general type variable.

14.4 Fluent Interfaces

A "fluent interface" here is something that reads more like natural English than Haskell syntax usually does. It is not related to fluent interfaces in object oriented programming.

In Haskell, we always have verb first:

```
1  map f xs
```

With the `&` operator, we can write object first:

```
1  import Data.Function (&)
2
3  xs & map f
```

This style is the minority for most Haskell code, but the majority outside of Haskell. This style emphasizes the input value, while Haskell style emphasizes the final result.

Dummy Arguments

Suppose we want to write an EDSL for silly math expressions, without any operators.

```
ten  = 5 times 2
four = 2 plus 2
```

Usually, you can't use `times` and `plus` like this. You're probably more used to seeing:

```
ten  = 5 `times` 2
four = 2 `plus` 2
```

But, no, we don't want backticks. Here's what we'll do:

```
data Times = Times

times :: Times
times = Times

data Plus = Plus

plus :: Plus
plus = Plus

instance (a ~ Times) => Num (a -> Expr -> Expr) where
    fromInteger x Times y =
        fromInteger x * y

instance (a ~ Plus) => Num (a -> Expr -> Expr) where
    fromInteger x Plus y =
        fromInteger x + y
```

This assumes the same `data Expr` that we defined in the `Num` section. It also relies on the overloaded literal trick defined above.

However, you can define this for non-literal functions as well. Above, we saw that `esqueleto` defined `case_` syntax for providing alternatives in SQL. The function syntax in SQL and Haskell looks like this:

```
CASE
    WHEN a
        THEN r0
    WHEN b
        THEN r1
    ELSE
        r3
```

```
case_
    [ when_ a
        then_ r0
    , when_ b
        then_ r1
    ]
    (else_ r3)
```

The implementation of `when_`, `then_`, and `else_` are only there for syntactic sugar.

```
case_
    :: [(SqlExpr (Value Bool), SqlExpr a)]
    -> SqlExpr a
    -> SqlExpr a

when_
    :: SqlExpr (Value Bool)
    -> ()
    -> SqlExpr a
    -> (SqlExpr (Value Bool), SqlExpr a)
when_ bool () value = (bool, value)

then_ :: ()
then_ = ()

else_ :: SqlExpr a -> SqlExpr a
else_ a = a
```

You could just as easily write:

```
case_
    [ (a, r0)
    , (b, r1)
    ]
    r3
```

But this doesn't look like the SQL.

14.5 Case Study: Weightlifting Logging

I like to lift weights on occasion. I've never been happy with the available weightlifting logging apps, and so I've tinkered with writing my own ever since I started learning Haskell. The first approach designed a DSL. I wrote a parser that figured out what sets, weights, and repetitions that I did. The syntax looked like this:

```
Squat:
45 x 5
95 x 5
135 x 5 x 3
115 x 5, 3, 2
```

Roughly speaking, the grammar is:

```
Lift Name:
[ Weight [x Reps [, Reps ]+ [x Sets]]
]+
```

If a rep/set number is omitted, then we infer 1. So `45 x 5` is inferred to be `45 x 5 x 1`, representing "a single set of 45lbs for 5 reps." `135 x 5 x 3` means "Three sets of Five reps at 135lbs." The comma allows you to say "Repeat the weight with a different number of reps in a new set." So

`115 x 5, 3, 2` means "115lbs in three sets, with 5 reps, 3 reps, and 2 reps."

As a challenge to myself, I decided to embed the DSL as much as possible into Haskell. This is what I got:

```
history :: NonEmpty Session
history =
    [ 2019 05 07 #:
        [ bench press %:
            155 x 5 x 5
        , deadlift %:
            [ 225 x 5
            , 275 x 3
            ] <>
            315 x 1 x 3
        , db rows %:
            45 x 12 x 6
        ]
    , 2019 04 15 #:
        [ press %:
            100 x 5 x 5
        , deadlift %:
            225 x 5 x 3
        , bb curl %:
            45 x 20 x 4
        ]
```

We have a list of dated sessions, each with a list of lifts, each with a list of weight, reps, and sets. The date uses the `Day` literal syntax that we covered earlier in the chapter. The lifts use the `x` "dummy parameter" that was covered, along with some `Num` instances that give us more structured data. Let's dig in.

```haskell
data Session = Session
    { sessionLifts :: NonEmpty Lift
    , sessionDate :: Day
    }

data Lift = Lift
    { liftName :: String
    , liftSets :: NonEmpty Set
    }

data Set = Set
    { setWeight :: Weight
    , setReps :: Reps
    }

newtype Reps = Reps { unReps :: Int }
newtype Weight = Weight { weightAmount :: Double }

-- direct syntax
firstSession =
    Sesson (fromGregorian 2019 05 07) $
        (Lift "bench press"
            (NEL.replicate 5 (Set (Weight 155) (Reps 5)))
        ) :|
        [ Lift "deadlift"
            (Set (Weight 225) (Reps 5)
            :|
            ( Set (Weight 275) (Reps 3)
            : replicate 3 (Set (Weight 315) (Reps 1))
            )
        ]
```

This direct style is ugly. There's lots of noise, parentheses, brackets, and unintuitive operators. Let's make it "Better." We can immediately improve it by turning on `OverloadedLists` and using list literal syntax for `NonEmpty`. This works great, despite the obvious runtime error in `[] :: NonEmpty Int`.

```
firstSession =
    Sesson (fromGregorian 2019 05 07)
        [ Lift "bench press"
            (NEL.replicate 5 (Set (Weight 155) (Reps 5)))
        , Lift "deadlift" $
            [ Set (Weight 225) (Reps 5)
            , Set (Weight 275) (Reps 3)
            ]
            <> NEL.replicate 3 (Set (Weight 315) (Reps 1))
        ]
```

Then, we'll make a bunch of custom, intuitive operators. This way we don't need to repeat `Session` and `Lift` as much.

```
(#:) :: Day -> NonEmpty Lift -> Session
day #: sets = Session sets day

(%:) :: String -> NonEmpty Set -> Lift
(%:) = Lift

firstSession =
  fromGregorian 2019 05 07 #:
    [ "bench press" %:
        NEL.replicate 5 (Set (Weight 155) (Reps 5))
    , "deadlift" %:
        [ Set (Weight 225) (Reps 5)
        , Set (Weight 275) (Reps 3)
        ]
        <> NEL.replicate 3 (Set (Weight 315) (Reps 1))
    ]
```

This allowed us to get rid of some parentheses, too. I generally dislike parentheses and brackets. They're non-local - they have to be paired with an ending somewhere. I'd much rather write things that are as local as possible.

Alright, let's get that `Day Literal` syntax going to drop the `fromGregorian` call. And then, I think we're going to write some instances of `Num`

for a `Set`, so we can make that nicer, too. The syntax we want for a set is like `weight x reps`.

```
-- dummy argument
data X = X

x :: X
x = X

instance (a ~ Int) => Num (X -> a -> Set) where
  fromInteger i = \X r -> Set (Weight (fromInteger i)) (Reps r)

firstSession =
  2019 05 07 #:
    [ "bench press" %:
        NEL.replicate 5 (155 x 5)
    , "deadlift" %:
        [ 225 x 5
        , 275 x 3
        ]
        <> NEL.replicate 3 (315 x 1)
    ]
```

Looking much nicer. Working with the list constructor functions is annoying though. A common pattern is to have a single listing: `NEL.replicate 5 (155 x 5)` is okay, but it'd be nicer to write that as `155 x 5 x 5`. So let's write an instance of `Num` for our `NonEmpty Set`.

```
instance
    (a ~ Int, b ~ Int)
  =>
    Num (X -> a -> X -> b -> NonEmpty Set)
  where
    fromInteger i = \X r X s -> fromInteger i x r *: s

(*:) :: Set -> Int -> NonEmpty Set
(*:) w i = w :| replicate (i-1) w
```

```
10
11  firstSession =
12      2019 05 07 #:
13          [ "bench press" %:
14              155 x 5 x 5
15          , "deadlift" %:
16              [ 225 x 5
17              , 275 x 3
18              ]
19              <> 315 x 1 x 3
20          ]
```

Alright. I like this. But... can we do better?

Exercise:

Write a `NonEmptyBuilder` type that will allow you to have the following syntax:

```
1  firstSession =
2      2019 05 07 #: do
3          "bench press" %: do
4              155 x 5 x 5
5          "deadlift" %: do
6              225 x 5
7              275 x 3
8              315 x 2 x 3
```

It's common to have a lift name that is modified. For example, "bench" is a modification of "press". There's also overhead press, push press, strict press, incline press. And you'd hate to typo `perss` and then mess up your stats. So let's make these into real names.

```
bench :: String -> String
bench str = "bench " <> str

press :: String
press = "press"

deadlift :: String
deadlift = "deadlift"

firstSession = 2019 05 07 #: do
    bench press %: do
        155 x 5 x 5
    deadlift %: do
        225 x 5
        275 x 3
        315 x 2 x 3
```

And now we have a nice and lightweight EDSL for logging our weightlifting sessions.

14.6 Case Study: `rowdy`

`rowdy` is a library that I wrote for designing HTTP API structure. A common complaint with Yesod is the QuasiQuoter strategy for structuring routes. Yesod has a DSL (not embedded) for writing routes. Since it isn't embedded in Haskell, you don't get any of Haskell's niceties when developing routes. It's quick and relatively easy to learn, but it is another Thing you have to learn.

The QuasiQuoter parses the input `Text` into a data structure: a `[ResourceTree String]`. The Template Haskell code turns that datatype into the relevant definitions for the router. `ResourceTree` is a fairly standard tree datatype.

```
data ResourceTree typ
    = ResourceLeaf (Resource typ)
    | ResourceParent String CheckOverlap [Piece typ] [ResourceTree typ]
    deriving (Lift, Show, Functor)

data Resource typ = Resource
    { resourceName :: String
    , resourcePieces :: [Piece typ]
    , resourceDispatch :: Dispatch typ
    , resourceAttrs :: [String]
    , resourceCheck :: CheckOverlap
    }
```

The `ResourceLeaf` constructor contains the actual route information that's necessary. `ResourceParent` carries information that is important for nested routes and resources.

`rowdy` dispenses with the QuasiQuoter by embedding the language into Haskell. Ultimately, we're building a tree, and we've already seen how easy it is to build a `Tree` with a monadic EDSL. The datatype to describe a route is the `RouteTree`:

```
data RouteTree nest capture terminal
    = Leaf terminal
    | PathComponent capture (RouteTree nest capture terminal)
    | Nest nest [RouteTree nest capture terminal]
    deriving (Eq, Show, Functor, Foldable)
```

`RouteTree` is polymorphic so it can be used for other backends - I am planning on adding `servant` support, once I can figure out how to make it sufficiently flexible. To specialize it to Yesod, we have the following types:

```haskell
-- | An endpoint in the Yesod model.
data Endpoint
    = MkResource Verb String
    -- ^ A resource identified by a 'Verb' and a 'String' name.
    | MkSubsite String String String
    -- ^ A subsite.
    deriving (Eq, Show)

-- | The type of things that can affect a path.
data PathPiece
    = Literal String
    -- ^ Static string literals.
    | Capture Type
    -- ^ Dynamic captures.
    | Attr String
    -- ^ Route attributes. Not technically part of the path, but
    -- applies to everything below it in the tree.
    deriving (Eq, Show)

type Dsl = RouteDsl String PathPiece Endpoint
```

With just this, we can specify our routes as a datatype.

```haskell
mkYesod "App" $ routeTreeToResourceTree $
    [ Leaf $ MkResource Get "HelloR"
    , PathComponent (Capture (Type (Proxy :: Proxy UserId))) $
        Nest "users"
            [ Leaf Get "UserR"
            , Leaf Post "UserR"
            ]
    ]
```

But that's ugly! We want a cute little language to show off on our `README.md`. So, we have our `Dsl` monad, which allows us to instead write:

```
mkYesod "App" $ toYesod $ do
    get "HelloR"
    "users" // capture @UserId // do
        get "UserR"
        post "UserR"
    "posts" // capture @PostId //
        resource "PostR" [get, post, delete, put]
```

There's a lot going on here. Let's dig into it, one by one.

Terminals

```
    get "HelloR"
```

`get` is a "terminal." It's a leaf on the tree. The implementation is pretty simple:

```
get = doVerb Get

doVerb v s = terminal (MkResource v s)

terminal :: endpoint -> RouteDsl nest capture endpoint ()
terminal = tell . pure . Leaf
```

The `RouteDsl` type actually does use `Writer` under the hood! I ignored my own advice of "just use `State`, it doesn't have performance problems." Are the performance problems bad? I don't know. Routes are small enough that you may never notice. This all happens at compile-time anyway.

The // and /: operators

`//` is an operator that roughly corresponds with the `PathComponent` constructor in our tree. It mirrors the `/` that you'd see in a URL string. `/:` is similar, but it introduces an explicit "nesting" operation. Both operators can accept a full `RouteDsl` value on the right hand side, so what's the difference between these two APIs?

```
"users" // do
    get "UserIndexR"
    post "CreateUserR"

"users" /: do
    get "UserIndexR"
    post "CreateUserR"
```

When you use `//`, the `"users"` path is appended to each child route, while it is shared with `/:`.

```
[ PathComponent (Literal "users") (get "UserIndexR")
, PathComponent (Literal "users") (post "CreateUserR")
]
-- vs,
[ Nest "users"
    [ get "UserIndexR"
    , post "CreateUserR"
    ]
]
```

For a Yesod app, this doesn't matter at all. The route structure is identical. This matters more with Servant APIs[4] where the two structures, while isomorphic, have different properties in generated clients and other uses of the API type.

`PathComponent`

In Yesod, the path pieces for a route can be described with the following datatype:

[4] https://www.parsonsmatt.org/2018/03/14/servant_route_smooshing.html

```
data PathPiece
    = Literal String
    | Capture TypeRep
    | Attr String
```

While we could require users to write this directly, that's not a great API.

```
Literal "users" // Capture (typeRep @UserId) // do
    get "GetUserR"
    put "UpdateUserR"
```

We can hook in to `IsString` to get `Literal` support. And we can write a helper for `Capture` that will accept the type argument directly:

```
instance IsString PathPiece where
    fromString = Literal

capture :: forall ty. PathPiece
capture = Capture (typeRep @ty)

"users" // capture @UserId // do
    get "GetUserR"
    put "UpdateUserR"
```

`capture` is going to require the `AllowAmbiguousTypes` and `TypeApplications` extensions.

By separating out the tree-building EDSL and the actual web API route machinery, we've made it relatively straightforward to add `rowdy` support to any web library.

14.7 Case Study: `hspec`

`hspec` is a testing library that mimics the Ruby library RSpec for behavior driven development. It's common to call libraries like these EDSLs. Let's look at some sample `hspec` code:

```
main = hspec $ do
    describe "Some Thing" $ do
        it "has a test case" $ do
            True `shouldBe` True
        describe "Can Nest Things" $ do
            it "etc" $ do
                3 `shouldSatisfy` (< 5)

    describe "Some Other Thing" $ do
        it "has tests" $ do
            print 10 `shouldReturn` ()
```

hspec leverages do notation allows for tests to be grouped and nested with clean, easy syntax. Infix assertion functions read like English, sort of.

We could write a much simpler testing library. We could omit do notation and instead have nested lists. We could skip the silly infix functions and write comparisons directly. That library might look like this:

```
main = runTests
    [ group "Some Thing"
        [ test "has a test case"
            (shouldBe True True)
        , group "Can Nest Things"
            [ test "etc"
                (shouldSatisfy 3 (< 5))
            ]
        ]

    , group "Some Other Thing"
        [ test "has tests"
            (shouldReturn (print 10) ())
        ]
    ]
```

This choice is employed by `HUnit`, `tasty` and hedgehog. I prefer hspec for much the same reason that I like the `TreeBuilder` EDSL rather than

manual data construction. There are a ton of nice utilities when working with a `Monad`, and they are usually closer to what I want than the explicit data structure operations. Furthermore, it's easy to encode additional operations in the EDSL that might be awkward in the data constructor notation.

Let's consider `before` and `after`. `before` runs an `IO` action before each spec item and provides the result of that action to the spec item. `after` runs an `IO` action that receives the same value as the spec item and can do some post-test cleanup.

```
before :: IO a -> SpecWith a -> Spec
after  :: (a -> IO ()) -> SpecWith a -> SpecWith a

main :: IO ()
main = hspec $
    before (pure "Hello") $
    after (\str -> putStrLn str) $ do
    describe "has string" $ do
        it "has string" $ \str -> do
            str
                `shouldBe`
                    "Hello"
```

This test case provides `"Hello"` to each spec item, and prints `Hello` after every test. We can implement this with a `testGroup` formulation:

```
groupBefore :: String -> IO a -> [a -> Assertion] -> TestGroup
groupBefore msg make assertions =
    group msg $
        map (\k -> bracket make (const (pure ())) k) assertions
```

But the result can be a bit awkward and unwieldy, especially with nested groups.

```
groupBefore "with string" (pure "hello")
    [ \str ->
        test "with string" $
            str `shouldBe ` "hello"
    , \str ->
        group  "with string 2" $
        [ test "still has string" $
            str `shouldBe` "hello"
        ]
    ]
```

With the EDSL approach, the passing of the parameter is implied - only at the actual *use sites* of the parameter do we need to mention it. With the explicit data structure approach, we can't defer any more than the first layer.

15. Growing Pains

As your code grows and changes, you will break it. Some domains are so easy to model and understand that you'll never mess it up, but most aren't. Some changes won't cause any problems to users and clients of your code. These are wonderful and rare.

Breaking changes happen. We want to minimize how many breaking changes occur in our code, and there are a number of techniques available to reduce the incidence and severity of a breaking change. Likewise, when we do need to make a breaking change, we want to communicate this to our client's effectively so that they know how to migrate from the broken code. It's unpleasant to upgrade to a new version of a library and see that some function was removed or changed with no guidance on how to fix it.

15.1 A Taxonomy of Breaking Changes

You can break consumers of your library in many ways. The most obvious (and annoying) is to completely remove things from your library. This is always a breaking change. Fortunately, this sort of change is caught by the compiler - GHC will complain that it cannot find whatever it was that your library used to provide.

Changing the type of an exposed term is another breaking change, also caught by the compiler. Adding or removing an argument to a function - that's a breaking change! In this category, we also have adding fields to an exposed data constructor. Adding constructors to an exposed sum type is *also* a breaking change.

Even adding things to your library might break downstream consumers. Suppose your release `v0.1.2` of your library, which exposes a new `lookup` function. This will conflict with `Data.List.lookup` - and any user with an open import of your module and `Data.List` will now get ambiguous name errors! Is this a breaking change?

The PVP

Before Semantic Versioning was invented, Haskell had the Package Versioning Policy. The PVP specifies how a library author should set their version bounds to prevent their build from breaking. The biggest difference between SemVer and the PVP is that the PVP has two major version numbers: `A.B.C.D`. This allows you to express two different "kinds" of "breaking changes" - the first digit is reserved for major rewrites or API changes to the library, while the second is for more ordinary breaking changes.

The PVP says that a bug or documentation fix that doesn't alter the exports of any module, or the type or essential behavior of any value, is a patch version change. That means you can increment `D` only - `A.B.C.(D+1)`. It should always be safe to upgrade a patch version.

The PVP says that if you add a new term to a module, then you should make that a minor version bump. If a downstream consumer has any open imports, then they should specify a strict upper bound on the libraries that define those modules.

For example, if I have an open import of `Data.Map`, then, per the PVP, my version range for the `containers` package should be:

```
    containers >= 1.2.3.0 && < 1.2.4.0
```

A release of `containers-1.2.5.0` *might* break my code. This break is unlikely, but possible. Until `containers-1.2.5.0` is actually released and tested, I can't claim to support it.

If I have a qualified or explicit import list, then I can relax that version bound.

```
-- No name ambiguities
import qualified Data.Map as Map

-- New names won't enter the scope of the module if I do
-- an explicit import
import Data.Map (insert, lookup)
```

With the above forms, I can relax my upper bounds to any minor version.

```
    containers >= 1.2.3.0 && < 1.3
```

However, the version `containers-1.3.0.0` may break my code - by removing things or changing types. So, until I have tested on the new version of `containers`, I cannot claim to support it.

Hackage Revisions

Hackage is the package repository for Haskell. It supports a feature called "metadata revisions" that allow package authors (and Hackage Trustees) to edit the metadata of a package. This means that the version bounds you provide at upload/publish time can be modified.

The PVP Controversy

The strictness of upper bounds is a source of constant pain and anguish for Haskell developers. If you want to maintain Haskell libraries, then you have two choices:

1. Strict upper bounds (and spend time revising them to be more lax)
2. Lax upper bounds (and spent time revising them to be more strict)

Let's say you are the author of package `email-server`, and you depend on `email-types`. We'll see what happens as `email-types` updates. Strict upper bounds and lax upper bounds both start here, with `0.1.0.0` as the version. So we publish `email-server-0.1.0.0` as well, with these constraints:

```
email-server-strict-0.1.0.0:
    email-types >= 0.1.0.0 && < 0.1.1.0

email-server-lax-0.1.0.0:
    email-types >= 0.1.0.0
```

All is well.

email-types-0.1.1.0

Upstream added a new datatype, which triggers a minor version bump. email-server-strict-0.1.0.0 now fails to build with the new version of email-types, because email-types-0.1.1.0 violates the constraint < 0.1.1.0.

Our users may want (or need) to update to email-types-0.1.1.0 for a variety of reasons. Until we fix email-server-strict, they are blocked from using that library. If they're using Cabal or Stack projects, then library consumers can bypass our version bounds with the allow-newer flag. However, if they're not using a project (ie cabal.project file or stack.yaml file), then they're blocked.

As authors, we have a few things we can do:

1. Relax the constraint on email-server-strict-0.1.0.0 using a Hackage revision.
2. Update the email-server-strict.cabal file with the new version bound and publish a new patch version.

If we choose to relax version constraints, then it may make sense to test every released version of email-server that has the version bound to see if they are compatible. If so, we may choose to relax the version constraint for *each released version*.

email-server-lax-0.1.0.0 does not have a conflict, and fortunately builds successfully. Neither the maintainer of email-server-lax- nor any downstream consumer need to do anything to update to email-types-0.1.1.0.

email-types-0.1.2.0

This time, upstream added a new function, and it does have a name conflict with a function in email-server. Both email-server-strict and email-server-lax need to make code changes and release a new patch version.

email-server-lax now has versions on Hackage that we *know* won't build successfully - the version constraints allow for failing builds! This

is bad. The maintainer of `email-server-lax` should now go through and add a Hackage revision that adds an upper bound to each version of `email-server-lax`:

```
email-server-lax-0.1.0.1:
    - email-types >= 0.1.0.0

# Now, for all prior versions, add the upper bound
email-server-lax-A.B.C.D:
    - email-types >= 0.1.0.0 && < 0.1.2.0
```

A Missing Logic

There is a natural tension here. The operator that we use to say `<` means "Cannot build with this version or greater." But if the version has not been released yet, we can't know that. If the version is unreleased, it's more accurate to say "Cannot guarantee that it will build with this version or greater."

We have two conflated ideas:

- Known to not build with
- Not known to build with

Proponents of `strict` upper bounds want `<` to mean the first point. If they say `email-types < 0.1.2.0`, then they mean that we know that we can't build with that version. They think it is less work to occasionally create revisions which tighten upper bounds.

Proponents of `lax` upper bounds want `<` to mean the second point. If they say `email-types < 0.1.2.0`, then they mean that we don't yet know that we can build with that version. They think it is less work to occasionally create revisions which relax upper bounds.

Personally, I maintain a number of packages, and I use both strategies. I usually find that the packages with lax upper bounds are less work. Occasionally, I'll get a GitHub issue filed reporting that some version isn't compatible, and then I fix it. With strict upper bounds, I'm constantly

getting pinged on Stackage that my package isn't building anymore due to some new version. I'll clone the code, relax the bound, test it out, and it rarely happens that the build fails. So I push a release and call it done.

Ultimately, I think this is a technical problem. It has become a social problem because coordinating between the two camps is often frustrating. A technical solution - allowing people to express both "known to not build" and "not known to build" - would allow everyone to work together with minimal friction.

15.2 Avoiding Breaking Changes

The easiest way to make your library easy-to-use over the long term is to avoid making breaking changes at all. This means:

- Don't ever remove anything
- Don't ever change anything
- Only ever add things

These requirements make it quite difficult to maintain the library - you'll often want to add things to a datatype to support a new feature! Fortunately, there are a few ways to avoid these pains.

Additive Changes

Additive changes - like a new function, datatype, or type synonym - are totally fine. These all require only a minor version bump.

If you want to add a new parameter to a function, and that parameter has a sensible default, then consider adding a new function that accepts the new parameter. Delegate to the old function, and link to the new function in the documentation.

Don't expose constructors

Let's say you've got this module in your library.

```
module Foo where

import Data.Text (Text)

data Foo = Foo
    { name :: Text
    , age :: Integer
    }
```

You realize that `name` isn't right - you should be recording the `firstName` and `lastName` separately[1].

```
module Foo where

import Data.Text (Text)

data Foo = Foo
    { firstName :: Text
    , lastName :: Text
    , age :: Integer
    }

name :: Foo -> Text
name foo =
    mconcat [firstName foo, " ", lastName foo]
```

This is a breaking change, because the type of `Foo` has changed -

```
- Foo ::         Text -> Integer -> Foo
+ Foo :: Text -> Text -> Integer -> Foo
```

Anyone that has pattern matched on `Foo` with positional arguments will get a compile-time failure:

[1] Well, that's not right either. See the excellent "Falsehoods Programmers Believe About Names".

```
what (Foo name age) =
    error "Foo has three arguments but was only given two..."
```

Likewise, anyone that has done a `RecordWildCards` or record match on `name` will get an error.

```
-- error: name has type `Foo -> Text`, not `Text`
ohno Foo {..} =
    name

-- error: Foo has no field name
ohno Foo { name = fooName } =
```

If we want to avoid this problem, then we need to make the constructor abstract. We also need to hide the field labels and only provide accessor functions. Let's look at an example that does this:

```
module Foo
    ( Foo
    , mkFoo
    , name
    , age
    )
    where

import Data.Text (Text)

data Foo = Foo
    { _name :: Text
    , _age :: Integer
    }

age :: Foo -> Integer
age = _age

name :: Foo -> Text
```

```
20  name = _name
21
22  mkFoo :: Text -> Integer -> Foo
23  mkFoo = Foo
```

Now we can change the `_name` into a `_firstName` and `_lastName` combo:

```
1   module Foo
2       ( Foo
3       , mkFoo
4       , name
5       , firstName
6       , lastName
7       , age
8       )
9       where
10
11  import Data.Text (Text)
12  import qualified Data.Text as Text
13
14  data Foo = Foo
15      { _firstName :: Text
16      , _lastName :: Text
17      , _age :: Integer
18      }
19
20  age :: Foo -> Integer
21  age = _age
22
23  firstName :: Foo -> Text
24  firstName = _firstName
25
26  lastName :: Foo -> Text
27  lastName = _lastName
28
29  name :: Foo -> Text
30  name foo =
```

```
31      mconcat [firstName foo, " ", lastName foo]
32
33  mkFoo :: Text -> Integer -> Foo
34  mkFoo name age = Foo
35      { _firstName =
36          f
37      , _lastName =
38          Text.dropWhile isSpace lastNameWithSpace
39      , _age =
40          age
41      }
42    where
43      (firstName, lastNameWithSpace) =
44          Text.breakOn " " name
```

Now, we haven't broken anything! How fantastic. If we settled on this pattern for everything, then you may want to invoke a bit of `Template-Haskell` to define getters and setters for all the fields of the record.

If you're writing an application, or otherwise don't expect to have any consumers of this type, then don't bother with this - it's just extra complexity and encapsulation for no benefit. This technique is useful when you're developing a library, and you don't control all uses of the relevant type.

For a concrete example, consider the `SqlBackend` type from the `persistent` database library. This type was completely exposed to library consumers. Any time the maintainers want to add new functionality to the type, we have to release a new major version of the package. The vast majority of users of the library will not experience any breaking changes, but the version number indicates a possibility for breaking changes.

As a result, the maintainers are encouraged to bundle up functionality upgrades with other breaking changes so as to release fewer major version bumps. Ideally, `persistent` users would be able to enjoy new functionality without corresponding breaking changes. Keeping the type abstract would help with that. The `persistent` library did make the `SqlBackend` type abstract, and has been able to release new functionality as minor changes that would have previously been major breaking changes.

When you're writing an application, you control all of the code. GHC tells you about all of the breaking changes you make at every use site, so you can fix them up.

Sum Types

It's less clear that encapsulation provides any benefit with sum types. Sum types are useful precisely because you define a closed set of possibilities with defined payloads. Pattern matching on a sum type (or using an equivalent `fold`) means that you *expect* your code to break if a new case is added.

Let's consider a quick example.

```
data Animal
    = Cat String Int Int
    | Dog String Int
    | Bird String
```

We can break users of this type in two ways:

1. Adding or removing a constructor
2. Adding or removing a field to a constructor

So, right off the bat, we have an annoying problem that we don't know exactly what those values are supposed to represent. Let's fix that up with record labels:

```
data Animal
    = Cat
        { name :: String
        , age :: Int
        , lives :: Int
        }
    | Dog
        { name :: String
        , age :: Int
```

```
        }
    | Bird
        { phrase :: String
        }
```

Unfortunately, this has some major problem: all of these record field selector functions can throw errors. `name :: Animal -> String` - however, if you call `name (Bird "asdf")`, you'll get an error that the field `name` is not defined on that constructor. Likewise, `lives` is only defined on `Cat`.

For this reason, I suggest that a sum-type constructor should contain only one field.

```
data Animal
    = AnimalCat Cat
    | AnimalDog Dog
    | AnimalBird Bird

data Cat = Cat { name :: String, age, lives :: Int }
data Dog = Dog { name :: String, age :: Int }
data Bird = Bird { phrase :: String }
```

This is a bit more cumbersome, but it works out to be much safer and resilient to breaking changes. When you pattern match on `Animal` you never need to worry about adding or removing positional parameters. You can rely on `RecordWildCards` or `NamedFieldPuns` or even just accessor functions to be perfectly clear about what you mean. This converts the most annoying problem of sum-type breaking changes into a problem of record-type breaking changes.

To encapsulate a sum type, you want to be able to add/remove fields without breaking downstream users. This means we need a set of functions that can "project" the sum type into one case, as well as "inject" a smaller case into a larger one.

```haskell
projectBird :: (Bird -> r) -> Animal -> Maybe r
projectBird withBird animal =
    case animal of
        AnimalBird b -> Just (withBird b)
        _ -> Nothing

injectBird :: Bird -> Animal
injectBird = AnimalBird

-- etc, for each different constructor
```

Now the library maintainer is free to add cases without breaking any downstream code.

Unfortunately, we lose the killer feature of a sum-type with this: exhaustivity. If we want to guarantee that we've handled every case, we need a *fold* function:

```haskell
foldAnimal
    :: (Cat -> r)
    -> (Dog -> r)
    -> (Bird -> r)
    -> Animal -> r
foldAnimal onCat onDog onBird animal =
    case animal of
        AnimalDog d -> onDog d
        AnimalCat c -> onCat c
        AnimalBird b -> onBird b
```

But - if we add a case to `Animal`, we now need to add a parameter to this function, which is a breaking change. So we might as well expose the constructors and allow people to pattern match on it.

Pattern Synonyms

Haskell has an extension `PatternSynonyms`[2] that allow you to define *patterns* that refer to existing data types. This can significantly ease the

[2] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pattern_synonyms.html

migration path for changing a data constructor. For a real world test case, let's investigate the `ErrorCall`[3] type. In `base-4.8`[4], this type was defined like this:

```
-- |This is thrown when the user calls 'error'. The @String@ is the
-- argument given to 'error'.
newtype ErrorCall = ErrorCall String
    deriving (Eq, Ord, Typeable)
```

In `base-4.9`, a second `String` was added - this one giving the location by rendering the `CallStack`. Adding location information to `ErrorCall` is great, but by extending the constructor, we're forcing everyone to modify their code whenever the construct or pattern match on the `ErrorCall` type.

```
-- if it was changed to,
data ErrorCall = ErrorCall String String

-- then this breaks:
blah (ErrorCall errMessage) = putStrLn errMessage

-- and this breaks:
main =
    throw (ErrorCall "oh no")
```

Fortunately for us, GHC provided a `PatternSynonym` with the old name, and changed the name of the new constructor.

[3] https://hackage.haskell.org/package/base-4.14.0.0/docs/Control-Exception.html#t:ErrorCall
[4] https://hackage.haskell.org/package/base-4.8.0.0/docs/Control-Exception.html#t:ErrorCall

```
-- | This is thrown when the user calls 'error'. The first @String@ is the
-- argument given to 'error', second @String@ is the location.
data ErrorCall = ErrorCallWithLocation String String
    deriving ( Eq  -- ^ @since 4.7.0.0
             , Ord -- ^ @since 4.7.0.0
             )

pattern ErrorCall :: String -> ErrorCall
pattern ErrorCall err <- ErrorCallWithLocation err _ where
  ErrorCall err = ErrorCallWithLocation err ""
```

Users of `ErrorCall` are now unaffected by our change, and are still able to pattern match on the constructor, *or* use the `ErrorCall` pattern as a constructor directly. `PatternSynonyms` are fantastic for providing backwards compatibility when evolving APIs that expose constructors directly.

Removing Constructors from a sum type

The Problem

You have a sum type, and you want to delete a redundant constructor to refactor things.

```
data Foo
    = Bar Int
    | Baz Char
    | Quux Double
```

That `Quux` is double trouble. But if we simply delete it, then users will get a `Constructor not found: Quux`. This isn't super helpful. They'll have to go find where `Quux` came from, what package defined it, and then go see if there's a Changelog. If not, then they'll have to dig through the Git history to see what's going on. This isn't a fun workflow.

But, let's say you *really need end users to migrate off `Quux`*. So we're interested in giving a compile error that has more information than `Constructor not in scope`.

Here's what some calling code looks like:

```
blah :: Foo -> Int
blah x = case x of
    Bar i -> i
    Baz c -> fromEnum c
    Quux a -> 3
```

will give the output:

```
/home/matt/patsyn.hs:24:5: error:
    Not in scope: data constructor ‘Quux’
   |
24 |     Quux a -> 3
   |     ^^^^
Failed, no modules loaded.
```

Fortunately, we can make this nicer.

GHC gives us a neat trick called PatternSynonyms[5]. They create constructor-like things that we can match on and construct with, but that are a bit smarter.

Matching

Let's redefine Quux as a pattern synonym on Foo. We'll also export it as part of the datatype definition.

[5] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pattern_synonyms.html

```
{-# language PatternSynonyms, ViewPatterns #-}

module Wow (Foo (.., Quux)) where

data Foo
    = Bar Int
    | Baz Char

pattern Quux :: a -> Foo
pattern Quux i <- (const Nothing -> Just i)
```

This does something tricky: we always throw away the input with the `ViewPattern`, and we can summon whatever we want in the left hand side. This allows us to provide whatever `a` is needed to satisfy the type. This match will *never* succeed - so `Quux` behavior will never happen.

Now, we get a warning for the match:

```
[1 of 1] Compiling Main             ( /home/matt/patsyn.hs, interpreted )

/home/matt/patsyn.hs:25:5: warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In a case alternative: Quux a -> ...
   |
25 |     Quux a -> 3
   |     ^^^^^^^^^^
Ok, one module loaded.
```

But an error for constructing:

```
[1 of 1] Compiling Main             ( /home/matt/patsyn.hs, interpreted )

/home/matt/patsyn.hs:28:10: error:
    • non-bidirectional pattern synonym ‘Quux’ used in an expression
    • In the expression: Quux 3
      In an equation for ‘blargh’: blargh = Quux 3
   |
28 | blargh = Quux 3
   |          ^^^^
Failed, no modules loaded.
```

So we need to construct with it, too. We can modify the pattern synonym by providing a `where`, and specifying how to construct with it. Since we're intending to prevent folks from using it, we'll just use `undefined`.

```
pattern Quux :: a -> Foo
pattern Quux i <- (const Nothing -> Just i) where
    Quux _ = undefined
```

With this, we get just the warning about a redundant pattern match. Now it's time to step up our game by providing a message to the end user.

Warnings

GHC gives us the ability to write `{-# WARNING Quux "migrate me pls" #-}`. This can make sense if we expect that the runtime behavior of a program won't be changed by our pattern synonym.

So let's write a warning:

```
pattern Quux :: a -> Foo
pattern Quux i <- (const Nothing -> Just i) where
    Quux _ = undefined

{-# WARNING
  Quux
    "Please migrate away from Quux in some cool manner.\n\
    \See X resource for migration tips."
  #-}
```

Now, when compiling, we'll see the warnings:

```
/home/matt/patsynimp.hs:11:5: warning: [-Wdeprecations]
    In the use of data constructor ‘Quux’ (imported from PatSyn):
    "Please migrate away from Quux in some cool manner.
     See X resource for migration tips."
   |
11 |     Quux _ -> 3
   |     ^^^^

/home/matt/patsynimp.hs:11:5: warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In a case alternative: Quux _ -> ...
   |
11 |     Quux _ -> 3
   |     ^^^^^^^^^^

/home/matt/patsynimp.hs:14:10: warning: [-Wdeprecations]
    In the use of data constructor ‘Quux’ (imported from PatSyn):
    "Please migrate away from Quux in some cool manner. See X resource for migr\
ation tips."
   |
14 | blargh = Quux (3 :: Int)
   |          ^^^^
```

But this may not be good enough. We may want to give them an error, so they can't build.

TypeError

base defines a type TypeError[6], which GHC treats specially - it raises a type error. This isn't generally useful, but can be great for marking branches of a `type family` or type class `instance` as "impossible." The error message can be fantastic for guiding folks towards writing correct code.

PatternSynonyms can have two sets of constraints: the first is *required* when constructing, and the second is *provided* when matching. So let's just put an error in the first and see what happens:

```
pattern Quux
    :: (TypeError ('Text "please migrate ..."))
    => ()
    => a -> Foo
pattern Quux i <- (const Nothing -> Just i) where
    Quux _ = undefined
```

Unfortunately, GHC blows up immediately while compiling the synonym!

```
[1 of 2] Compiling PatSyn           ( PatSyn.hs, interpreted )

PatSyn.hs:20:1: error: please migrate ...
   |
20 | pattern Quux
   | ^^^^^^^^^^^...
Failed, no modules loaded.
```

We can't even `-fdefer-type-errors` this one. Are we hosed?

What about the second position? Same problem. We can't put a bare TypeError in there at all.

Fortunately, we can have a lil' bit of laziness by introducing it as a *constraint*.

[6] https://www.stackage.org/haddock/lts-19.31/base-4.15.1.0/GHC-TypeLits.html#t:TypeError

```
class DeferredError
instance (TypeError ('Text "please migrate ...")) => DeferredError

pattern Quux
    :: DeferredError
    => DeferredError
    => a -> Foo
pattern Quux i <- (const Nothing -> Just i) where
    Quux _ = undefined
```

This actually *does* give us a warning now - at the `const Nothing -> Just i` line, we have a deferred type error.

This gives us the error behavior we want!

```
/home/matt/patsynimp.hs:14:10: error:
    • please migrate ...
    • In the expression: Quux (3 :: Int)
      In an equation for ‘blargh’: blargh = Quux (3 :: Int)
   |
14 | blargh = Quux (3 :: Int)
   |          ^^^^^^^^^^^^^^
Failed, one module loaded.
```

We only get the one error - but if we delete it, we can see the other error:

```
[2 of 2] Compiling Main             ( /home/matt/patsynimp.hs, interpreted )

/home/matt/patsynimp.hs:11:5: error:
    • please migrate ...
    • In the pattern: Quux _
      In a case alternative: Quux _ -> 3
      In the expression:
        case x of
          Bar i -> i
          Baz c -> fromEnum c
          Quux _ -> 3
```

```
12      |
13  11  |      Quux _ -> 3
14      |      ^^^^^^
15  Failed, one module loaded.
```

What's fun is that we can actually provide *two* different messages. Constructing something will give both error messages, and pattern matching only uses the "required" constraint.

This should make it *much* easier for end users to migrate to new versions of your library.

Final Code and Errors

```
1   {-# language PatternSynonyms #-}
2   {-# language KindSignatures #-}
3   {-# language FlexibleContexts #-}
4   {-# language FlexibleInstances #-}
5   {-# language ViewPatterns #-}
6   {-# language MultiParamTypeClasses #-}
7   {-# language UndecidableInstances #-}
8   {-# language DataKinds #-}
9
10  {-# OPTIONS_GHC -fdefer-type-errors #-}
11
12  module PatSyn where
13
14  import Prelude
15  import GHC.Exts
16  import GHC.TypeLits
17
18  data Foo
19      = Bar Int
20      | Baz Char
21
22  class DeferredError (a :: ErrorMessage)
23  instance (TypeError a) => DeferredError a
24
```

```
pattern Quux
    :: DeferredError ('Text "please migrate (required constraint)")
    => DeferredError ('Text "please migrate (provided constraint)")
    => a -> Foo
pattern Quux i <- (const Nothing -> Just i) where
    Quux _ = undefined
```

Matching a constructor:

```
[2 of 2] Compiling Main       ( /home/matt/patsynimp.hs, interpreted )

/home/matt/patsynimp.hs:11:5: error:
    • please migrate (required constraint)
    • In the pattern: Quux _
      In a case alternative: Quux _ -> 3
      In the expression:
        case x of
          Bar i -> i
          Baz c -> fromEnum c
          Quux _ -> 3
   |
11 |     Quux _ -> 3
   |     ^^^^^^
Failed, one module loaded.
```

Using a constructor:

```
[2 of 2] Compiling Main             ( /home/matt/patsynimp.hs, interpreted )

/home/matt/patsynimp.hs:14:10: error:
    • please migrate (required constraint)
    • In the expression: Quux (3 :: Int)
      In an equation for ‘blargh’: blargh = Quux (3 :: Int)
   |
14 | blargh = Quux (3 :: Int)
   |          ^^^^^^^^^^^^^^
```

```
10
11  /home/matt/patsynimp.hs:14:10: error:
12      • please migrate (provided constraint)
13      • In the expression: Quux (3 :: Int)
14        In an equation for ‘blargh’: blargh = Quux (3 :: Int)
15    |
16  14 | blargh = Quux (3 :: Int)
17    |          ^^^^^^^^^^^^^^^
18  Failed, one module loaded.
```

15.3 Communicating To Users

Sometimes you just have to make a breaking change. GHC can tell them that something happened and went wrong, but ideally, you can provide better diagnostics and even help them migrate to the new version of the code.

The Status Quo

GHC has notoriously difficult to read error messages. Long-time Haskellers have learned how to extract value from them, but it's a chore to learn and perform. Sometimes, these error messages are fine. There are a few cases where you'll want to be careful:

Polymorphic Monad

Changing the number of arguments in a function that is polymorphic in the monad gives awful error messages.

Let's define a function `x` that's polymorphic in `MonadIO m => m UTCTime`, and see what happens when we provide an extra argument.

```
λ> import Control.Monad.Reader
λ> import Data.Time
λ> let x y = liftIO getCurrentTime
λ> :t x
x :: MonadIO m => p -> m UTCTime
λ> x 3
2020-12-23 22:37:53.869381009 UTC
λ> x 3 5

<interactive>:7:1: error:
    • No instance for (MonadIO ((->) Integer))
        arising from a use of ‘x’
    • In the expression: x 3 5
      In an equation for ‘it’: it = x 3 5
```

We get `No instance for (MonadIO ((->) Integer))`. GHC continues along trying to solve constraints until it gets here. This is bad because the error message can show up in all kinds of weird places. If you've composed multiple functions and collected all their constraints, then the error could apply at any of the problematic functions. This makes it difficult to diagnose what to fix.

If you're going to expose polymorphic functions like this, please be careful not to remove function arguments.

Warnings/Deprecations

GHC allows you to write `WARNING` and `DEPRECATED` pragmas that apply to certain things. Whenever someone uses a term that has one of these pragmas, it will render a warning message to the user. This warning message can be used to point the library user to an upgrade path.

Here's an example:

```haskell
foo :: Int -> IO ()
foo x = print (x + 1)

{-#
  DEPRECATED foo
    "Don't use foo. The function bar allows you to specify \
    \ the increment."
 #-}

bar :: Int -> Int -> IO ()
bar x y = print (x + y)
```

We've communicated to the end user that the `foo` function is bad, and they should instead use the `bar` function. The `WARNING` pragma has the same format. Deprecations should be used when you plan on removing a function or type from the library. Warnings are a bit more flexible in their intended use.

In the `esqueleto` library, we have a function `random_`, which provides a random number. However, it turns out this was a mistake! Not all database backends use the same syntax. To fix it, we added a `DEPRECATED` pragma to `random_` which pointed users to new database-specific modules such as `Database.Esqueleto.Postgresql` which contain a specific `random_` function. This allows users to upgrade with confidence and speed.

Suppose you wanted to change the behavior of a function without wrecking your users. The first thing to do is release a version with a `WARNING` pragma, specifying how the function is going to break in the *next* version. You may also want to point the user to a variant - like `myFunctionOldBehavior` - that is appropriately named and documented to smooth the upgrade process. Then, in the next *major* version, you change the behavior of `myFunction`. This process allows you to change your library and help end users perform upgrades.

Please do not remove things from your API without providing a warning or deprecation. Diagnosing a failure to build on a new version of your library can be pleasant with the right deprecation message. Diagnosing an error message "`foobar` is not in scope" is extremely unpleasant, especially if you're not even sure where `foobar` came from in the first place!

TypeError

The module `GHC.TypeLits` provides us with a special type, `TypeError`, which causes a type error with a custom message.

We can attach this to functions, type class declarations, instance declarations, or really anything where we don't want the end-user to use it, but we also don't want to delete it. Suppose we attached a `WARNING` pragma to a function in a previous version. Users of our library may still use it and simply ignore the `WARNING`. If we attach a `TypeError` to the function, then they can no longer use it at all without triggering the error. Fortunately, we are able to display a custom message to the user. This can be used to give an upgrade path or maybe diagnose what went wrong.

Let's say we're designing a variant Prelude that should be easier to learn. A common error that Haskell beginners run into is trying to add lists.

```
λ> [1,2,3] + 4

<interactive>:14:1: error:
    • Non type-variable argument in the constraint: Num [a]
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Num a, Num [a]) => [a]
```

This is an awful error message for a beginner to see. Fortunately, we can provide a better one:

```
instance
    (TypeError
        ( 'Text "You tried to treat a list as a number. "
        :$$: 'Text "Did you mean ++ for list concatenation?"
        )
    ) => Num [a]
```

Now, armed with this instance, our beginners will see this message instead

```
λ> [1,2,3] + 4

<interactive>:22:1: error:
    • You tried to treat a list as a number.
      Did you mean ++ for list concatenation?
    • In the expression: [1, 2, 3] + 4
      In an equation for ‘it’: it = [1, 2, 3] + 4
```

We can use this to provide a nice error message for extra function arguments, too.

```
type FnErrMsg =
    'Text "You probably need to remove an argument to a function."

instance (TypeError FnErrMsg) => MonadIO ((->) r)
```

Now, when our user calls a function with a polymorphic monad that can't work out, we get this message.

```
λ> x 3 5

<interactive>:29:1: error:
    • You probably need to remove an argument to a function.
    • In the expression: x 3 5
      In an equation for ‘it’: it = x 3 5
```

Contrast it with the original error message that GHC gives:

```
λ> x 3 5

<interactive>:7:1: error:
    • No instance for (MonadIO ((->) Integer))
        arising from a use of ‘x’
    • In the expression: x 3 5
      In an equation for ‘it’: it = x 3 5
```

What's even better is that this error message trips first. It happens exactly on the line that has a problem.

The `unliftio` library provides a class `MonadUnliftIO` that doesn't support certain monad transformers. This is by design - `StateT`, `WriterT`, and `ExceptT` behave oddly in the presence of `IO`, concurrency, and state mutations. Right now, there are merely missing instances. However, they could extend the library with `TypeError` instances that give helpful information as to *why* they can't have instances. Perhaps a link to the relevant GitHub issue[7], even!

For more on this, see Dmitrii Kovanikov's post "A story told by Type Errors"[8].

[7] https://github.com/fpco/unliftio/issues/68
[8] https://kodimensional.dev/type-errors

IV Interfacing the Real

It's not enough to write pure functions that transform data. We must also interact with the real world. While you can do quite a bit with compile-time code, ultimately most projects benefit from accepting arguments at run-time and sending useful output to other programs. Haskell's `IO` type makes interacting with the real world a little more cumbersome than in other languages with implicit effects. We gain a tremendous amount of power from that type, though, and I believe that it's well worth it.

16. Testing

People like to write Haskell because they don't have to write tests. I've heard this from so many people. Haskell does offer a ton of compile-time guarantees, and there are many times "if it compiles, it works" genuinely holds true.

Unfortunately, that's not always the case. Testing is sometimes necessary, and if you never write tests, then you don't learn how to test code effectively, or how to write code in a manner that is amenable to testing. We're going to learn how to do both of these things in this chapter.

16.1 Libraries and Tools

This section will be brief, as much of this material is covered in good depth in Haskell Programming from First Principles[1].

`HUnit`

`HUnit`[2] is a minimalist testing library that is heavy on the operators. There's not much here to learn - assertions are `IO ()` actions, and a test succeeds if it doesn't throw an exception. A test fails if it throws *any* exception, with `HUnit` handling the `HUnitFailure` exception to render pretty error messages.

`HUnit` is mostly used as a foundation for other test frameworks, like `hspec`, `tasty`, and `test-framework`. It can be useful to see how it works and know the assertion functions, but I wouldn't reach for it as a first approach.

[1] https://haskellbook.com/
[2] https://hackage.haskell.org/package/HUnit-1.6.1.0/docs/Test-HUnit.html

hspec

I like hspec[3] as a test framework. I learned test driven development in Ruby with the RSpec library, and hspec is a nice port that provides a number of conveniences. hspec uses "Behavior Driven Development" (BDD) idioms, so test suites end up looking like this:

```
main :: IO ()
main = hspec $ do
    describe "Some thing" $ do
        it "does the thing" $ do
            someThing "a"
                `shouldBe`
                    "the thing"
        it "has another property" $ do
            someThing "b"
                `shouldBe`
                    "other example"
```

As you can tell, the EDSL attempts to read like English language. Whether this is good or bad is an intensely personal decision.

hspec supports an autodiscovery tool called hspec-discover[4], which automatically stitches together a test suite via metaprogramming. Any module ending in Spec with a top-level term spec :: Spec' will get included in the test suite. This is a great convenience, and guarantees that you don't forget to wire up a test.

hspec has helpers for providing before and after hooks. You can use this to provide database connections or API servers as resources to individual tests. A before call will initialize a value and pass it as a value to every it defined within:

[3] https://hackage.haskell.org/package/hspec
[4] https://hackage.haskell.org/package/hspec-discover

```
before (mkDatabase :: IO Conn) $
    describe "withDatabase" $ do
        it "has a database" $ \conn -> do
            results <- runDb conn query
            results `shouldBe` ...
```

The `it` function accepts a `String` label and an `Assertion`, which is really an `IO ()` from `HUnit`. If you throw an exception, the test fails. If there's no exception, the test succeeds.

You can mark spec items as pending. Pending items are "skipped," and show a warning that some tests aren't actually passing. This can be used to write failing test cases, and fix them in a later PR. Most of the test organization functions (`describe`, `it`, etc) have a variant with an `x` prepended that mark everything under it as `pending`.

You can mark spec items as focused. If any test case is marked as focused, then focused tests are the only ones that will run. This can be nice while using `ghcid` to automatically re-run tests, ensuring that you only get output you care about while testing some part of the codebase.

`hspec` defines how to build a test suite, as well as a set of expectation functions. While `hspec`'s expectations are built on top of `HUnit`, it also integrates with other libraries like `QuickCheck`, and the library `hspec-hedgehog`[5] provides integration with the `hedgehog` property testing library.

tasty

`tasty`[6] is a relatively new competitor to `hspec`. The main differentiating feature is that it does not use a BDD EDSL to structure tests, instead relying on constructing an ordinary tree data structure with list literals. The autogeneration facilities in `tasty-th`[7] encourage you to write tests as top-level definitions, which can make it easier to run them directly in GHCi. And `tasty-discover`[8] appears to be about as powerful as `hspec-discover`.

[5] https://hackage.haskell.org/package/hspec-hedgehog
[6] https://hackage.haskell.org/package/tasty-1.4/docs/Test-Tasty.html
[7] https://hackage.haskell.org/package/tasty-th
[8] https://hackage.haskell.org/package/tasty-discover

`tasty` does not define expectations or assertions. It is only a library for building and filtering test suites. You can use `HUnit` or `hspec` expectations with `tasty` as a test-runner. There are many integrations for other libraries available, including `QuickCheck`, `SmallCheck`, `hedgehog`, and golden tests.

I've used `hspec` and `tasty` test suites, and while I prefer `hspec`, it's a toss up. I probably wouldn't bother converting an existing test suite to a different library, and they're similar enough that it is probably easier to implement a killer feature in one than swap over entirely.

doctest

`doctest`[9] allows you to write examples in comments and then verify them automatically. This approach works well with simple combinator libraries that operate on relatively small types, where writing examples in line is easy and informative. The `lens` library[10] uses them to great effect.

In practice, I have found `doctest` suites to be difficult to maintain. Since they aren't checked by GHC automatically as part of building the code, it's easy for the tests to become stale. And it's also easy for the test setup to become cumbersome.

Property Based Testing

There are many compelling property testing libraries in Haskell. The original property testing library, `QuickCheck` is popular and in wide-use. After many years of working with property tests in production, the idea has been refined. The state of the art is `hedgehog`, which differentiates with QuickCheck on several import design decisions.

This won't be a tutorial on property based testing. For that, I recommend Oskar Wickström excellent series of blog posts[11]. Instead, we'll compare and contrast how the libraries work and why you might select one over the other.

[9] https://hackage.haskell.org/package/doctest

[10] https://hackage.haskell.org/package/lens-4.19.2/docs/Control-Lens-Getter.html#v:view

[11] https://wickstrom.tech/programming/2019/03/02/property-based-testing-in-a-screencast-editor-introduction.html

QuickCheck

QuickCheck[12] is the original and definitive "property testing" library. Where most test cases have concrete examples, QuickCheck encourages you to write tests that are *abstract* of specific values and express relational properties on them. With a normal test-case approach, you might test `reverse` like:

```
describe "reverse" $ do
    it "empty list is empty list" $ do
        reverse []
            `shouldBe`
                []
    it "1,2,3 is 3,2,1" $ do
        reverse [1,2,3]
            `shouldBe`
                [3,2,1]
```

Property testing instead encourages you to say:

> For some random input `x`, what properties can I say about `reverse x`?

Well, the length should always be the same.

```
prop_length xs = length (reverse xs) === length xs
```

And reversing it twice gives you the original list back.

```
prop_id xs = reverse (reverse xs) === xs
```

QuickCheck works by generating totally random values and trying to find examples that fail the given properties. When it finds the edge-case, it reports it back to you, so you know exactly what input breaks the code.

[12] https://hackage.haskell.org/package/QuickCheck

As the original property based testing library, it has the most examples and documentation online. It's also the most widely accessible library - QuickCheck has been ported to many other programming languages, so even non-Haskellers are sometimes familiar with it.

QuickCheck has a few downsides.

The ability of QuickCheck to find a bug is dependent on how well tuned the random value generators are to the problem. QuickCheck uses type classes for generating values by default, and the default generator is probably not well tuned to your domain. It's entirely possible that the random number generation just never generates a value that trips an edge case. Since the tests are non-deterministic, it's also easily possible that you'll write test cases that fail sometimes but not others. QuickCheck allows you to provide a seed value to reproduce these test, of course, but randomly failing tests can be difficult to track down.

Once a failing value is found, *shrinking* that value becomes essential - otherwise you'll end up with a massive and difficult to understand input test case. QuickCheck defines `shrink` with a default method that does nothing, so most custom types don't shrink at all. This limits the usefulness of QuickCheck with custom types, unless you manually define shrinking functions, and there's no guarantee that your shrinking function works properly for your domain.

SmallCheck

`smallcheck`[13] is inspired by Quickcheck, but instead of checking a random set of values, it exhaustively checks with small values. Since tests are exhaustive, you don't need to worry about random number generation. And since test values start small, you don't have to worry about shrinking.

SmallCheck obviously doesn't catch errors when the required values are large. The academic paper introducing SmallCheck[14] is accessible and not terribly theoretical.

SmallCheck uses type classes for generating data, much like `QuickCheck`.

[13] https://hackage.haskell.org/package/smallcheck
[14] https://www.cs.york.ac.uk/fp/smallcheck/smallcheck.pdf

Hedgehog

hedgehog[15] is a novel approach to property based testing. Unlike `QuickCheck`, it does not use type classes for generating values. Instead, you generate values explicitly. This makes it easy to tailor generation to your domain. Sampling from integers requires that you specify the distribution of integers that you sample from.

The killer feature is that shrinking values comes for free. With `QuickCheck`, you had to manually specify how to shrink a value of a given type. But hedgehog uses the actual generation of values to determine how to shrink a value. This means that shrinking is much more likely to produce useful values.

hedgehog has wonderful output on failing test cases, too. For each line in a failing test case that generates a value, it'll show you the value that was generated. And for an assertion that fails gets pretty-printed and rendered in color. The test output is so good that you may want to use hedgehog even for unit tests!

While most expositions of property testing use pure functions, you can use properties with effects, too. My blog post Effectful Property Testing[16] goes into detail on how to use hedgehog to test database queries.

16.2 Designing Code for Testing

Fortunately, "writing code that is easy to test" and "purely functional programming" have a tremendous amount of overlap. Indeed, if you take the "Single Responsibility Principle" from object-oriented design to it's natural conclusion, you get pure functions. Many of the design patterns in OOP for enabling good testing boil down to "higher order functions," which we'll use extensively. Modeling our business domain with fine-grained and precise types makes it easy to factor out logic and inspect correctness.

[15] https://hackage.haskell.org/package/hedgehog
[16] https://www.parsonsmatt.org/2020/03/11/effectful_property_testing.html

Pure Functions

Pure functions are easy to test. As much as possible, you should be using the techniques I describe in other parts of the book to factor as much of your code into pure functions. However, even pure functions can be difficult to test if the function is too big or complicated.

In OOP, code is organized into classes with methods. Refactoring usually begins by splitting methods and classes into multiple smaller methods and classes. We'll do this in Haskell too, by splitting our functions into smaller functions.

Datatypes are the bridge between functions. Functions can easily accept multiple inputs. But multiple *outputs* requires packing those outputs in a tuple or a custom datatype. This step is slightly less convenient, and so it's practiced a bit less.

Example 1: Render Email

Let's take a function that is "too large":

```
renderEmail :: User -> Order -> UTCTime -> Text
renderEmail user order currentDate =
    mconcat
        [ lots of nasty rendering logic
        ]
```

Now, I'd like you to imagine that the function goes on for hundreds and hundreds of lines. We can write tests for this function relatively easily. After all, it's a pure function, so we just need to make assertions about the inputs and output.

```
    describe "renderEmail" $ do
        let user = mkDefaultUser
            order =  mkDefaultOrder
            currentDate = mkDate
        it "complete example" $ do
            renderEmail user order currentDate
                `shouldBe`
                    "The whole thing"
```

However, this is brittle and somewhat annoying to write. If we want to assert that the email body contains the right name, we might write:

```
        it "has the right name" $ do
            renderEmail user order currentDate
                `shouldSatisfy`
                    (Text.elem (userName user))
```

This is unsatisfying, because we have no guarantee that the name appears in the right spot. Furthermore, we may have inserted a line break on a whitespace boundary - then the test fails because the string literal `"Foo Bar"` does not match `"Foo\nBar"`. We are trying to assert facts about the end result, but those facts have been destroyed, and we are performing data archaeology. This isn't good. Fortunately, we can refactor the code to preserve that information.

> If the test is painful to write, then we should accept this as feedback that the code should be refactored.

Let's split it up. We want to identify an intermediate step in this function, and possibly write a few new types that represent the intermediate steps here. Rendering an email suggests that we have some form of template, and information to plug into variables in that template.

We'll start by creating a type for the variables.

```
data Var
    = UserName
    | OrderDate
    | OrderPrice
    | ProductName
```

What are the actual contents of the message? Rather than representing them as text, let's pull out all of the semantic common factors, and create a sum type representing them. We likely have a few:

```
type Template = [TemplateFragment]

data TemplateFragment
    = Variable Var
    | Greeting
    | Goodbye
    | SalesPitch
    | CompanyEmail
```

Finally, we need some way to turn a `Variable` into what it is actually supposed to be. I'm going to use a function representation with a `newtype` wrapper, though I could also create a record with a value for each constructor in the sum type. The function approach makes it a bit easier to ensure we keep the two types up-to-date, as I can forget to add fields to the record type.

```
newtype ReplaceVar result = ReplaceVar (Var -> result)

replaceVar :: ReplaceVar result -> Var -> result
replaceVar (ReplaceVar k) v = k v

mkReplacements :: User -> Order -> UTCTime -> ReplaceVar Text
mkReplacements user order now =
    ReplaceVar $ \var -> case var of
        UserName -> userName user
        OrderDate -> orderDate order
        ProductName -> productName (orderProduct order)
```

Now, we've got the two sides of our `renderEmail` - the preparation of a `Template`, and the rendering of a `Template` into a `Text`.

```
renderEmail :: User -> Order -> UTCTime -> Text
renderEmail u o t =
    renderText (prepareTemplate u o t) (mkReplacements u o t)

prepareTemplate :: User -> Order -> UTCTime -> Template

renderText :: Template -> ReplaceVar Text -> Text
```

Fantastic - we now have two new testing points. Now that we have narrowed our types and introduced these intermediate functions, we can write much more precise tests.

Now, let's rewrite that test that asserts the user name is present in the email.

```
    it "has the user name variable in the template" $ do
        prepareTemplate user order currentDate
            `shouldContain`
                [Variable UserName]
    it "UserName shows the user name" $ do
        replaceVar (mkReplacements user order currentDate) UserName
            `shouldBe`
                userName user
```

Much nicer! These two tests now assert that the `mkReplacements` and `prepareTemplate` functions work as intended. By isolating the complex business logic into small testable functions, we can make precise assertions about their results. Instead of writing complex and fragile tests against the entirety of `renderEmail`, we can write resilient tests that say exactly what they mean.

Example 2: TemplateHaskell

For some reason, people forget that `TemplateHaskell` is ordinary functional programming. We can use all of our normal tricks for testing

and introspecting on `TemplateHaskell` functions to help verify their correctness.

For this example, we're going to refactor the `TemplateHaskell` code in my `record-wrangler`[17] library. The `wrangle` function defines the useful part of the library, and it's 55 lines of dense Template Haskell code. There's no way to test the code aside from simply running it and then verifying that the compiler generates the right code.

Before continuing, it's best to go read the source code for the function[18] as-is.

We'll start the refactoring process at the end.

```
    let newConstructors =
            map (uncurry RecC) newConstrs
    pure
        [ DataD [] newRecName tyvars Nothing newConstructors []
        , SigD (mkName convertName) sig
        , convert
        ]
```

That first entry in the list is the data declaration for the new record type. We'll write a datatype that has the information we want, and then we'll writer a "render" function that turns it into a `TemplateHaskell` declaration.

```
data NewRecord = NewRecord
    { newRecordName         :: Name
    , newRecordTyvars       :: [TyVarBndr]
    , newRecordConstructors :: [(Name, [(Name, Bang, Type)])]
    }

newRecordToDec :: NewRecord -> Dec
newRecordToDec NewRecord{..} =
    DataD [] newRecordName newRecordTyvars Nothing
```

[17] https://hackage.haskell.org/package/record-wrangler

[18] https://hackage.haskell.org/package/record-wrangler-0.1.1.0/docs/src/RecordWrangler.html#wrangle

```
        (map (uncurry RecC) newRecordConstructors)
        []
```

This is the most direct translation. However, I'm unhappy with that nested list of tuples. Tuples are almost always more confusing than a well-named datatype. We can refactor to even better datatypes:

```
data NewRecord = NewRecord
    { newRecordName         :: Name
    , newRecordTyvars       :: [TyVarBndr]
    , newRecordConstructors :: [NewConstructor]
    }

newRecordToDec :: NewRecord -> Dec
newRecordToDec NewRecord{..} =
    DataD [] newRecordName newRecordTyvars Nothing
        (map newConstructorToCon newRecordConstructors)
        []

data NewConstructor = NewConstructor
    { newConstructorName :: Name
    , newConstructorFields :: [RecordField]
    }

newConstructorToCon :: NewConstructor -> Con
newConstructorToCon NewConstructor{..} =
    RecC newConstructorName $ map recordFieldToTH newConstructorFields

data RecordField = RecordField
    { recordFieldName :: Name
    , recordFieldBang :: Bang
    , recordFieldType :: Type
    }

recordFieldToTH :: RecordField -> (Name, Bang, Type)
recordFieldToTH (RecordField n b t) = (n, b, t)
```

Now we can use this in the original code. It won't look like much, but we're going to construct the record and provide it to the function. Since this construction will occur relatively close by, this will seem useless. It's merely a step in the right direction.

```
    let mkConstructor (n, fs) =
            NewConstructor n
            $ map (\(n, b, t) -> RecordField n b t) fs
        newRec = NewRecord
            { newRecordName = newRecName
            , newRecordTyvars = tyvars
            , newRecordConstructors =
                map mkConstructor newConstrs
            }
    pure
        [ newRecordToDec newRec
        , SigD (mkName convertName) sig
        , convert
        ]
```

We're going to continue pulling the thread by moving those definitions closer to their sources. We got about 40 lines up before we hit the first term we depend on: `newConstrs`. `newConstrs`, in turn, depends on `newFields`, which only depends on the `addFields` - this is a field on the source `WrangleOpts` type, so we can move this up to the top of the function almost!

```
    let newConstrs = map mkNewConstr recConstrs
        mkNewConstr (recName, fields) =
            ( modifyName constructorModifier recName
            , map mkNewField fields  ++ newFields
            )
        mkNewField (fieldName, bang', typ) =
            ( modifyName fieldLabelModifier fieldName
            , bang'
            , typ
            )
```

```
        newRec = NewRecord
            { newRecordName = newRecName
            , newRecordTyvars = tyvars
            , newRecordConstructors =
                map mkConstructor newConstrs
            }
```

`newConstrs` is constructed by mapping over `recConstrs` - but this points to some redundancy, since we're also calling `map mkConstructor` on `newConstrs`. This suggests we can make this code quite a bit simpler by having `mkNewConstr` and `mkNewField` return our custom datatypes and inlining them.

```
    let mkNewConstr (recName, fields) =
            NewConstructor
                { newConstructorName =
                    modifyName constructorModifier recName
                , newConstructorFields =
                    map mkNewField fields  ++ newFields
                }
        mkNewField (fieldName, bang', typ) =
            RecordField
                { recordFieldName =
                    modifyName fieldLabelModifier fieldName
                , recordFieldBang =
                    bang'
                , recordFieldType =
                    typ
                }
        newRec = NewRecord
            { newRecordName = newRecName
            , newRecordTyvars = tyvars
            , newRecordConstructors =
                map mkNewConstr recConstrs
            }
```

This change causes a few other changes. There are a few tuple pattern matches that can be replaced with more informative record accessor

functions or `RecordWildCards` matches. Overall, improvements! `recConstrs` is provided effectfully, but only because we use `fail`:

```
    recConstrs <- for constrs $ \constr -> case constr of
        RecC recName fields ->
            pure (recName, fields)
        _ ->
            fail
                $ "Expected a record constructor, but got: "
                <> show constr
```

However, this isn't the only place we depend on the old constructor information. We use it when creating the conversion function. So we can't completely inline the definition here. This suggests a further refactor: an `OldRecord` type. I'll copy the three datatypes, replace `New` with `Old`, and then we can have a nice conversion function. This will be boilerplate-y and repetitive. That's okay. This is a mechanical process. We're going to do the easiest, most obvious, possibly dumbest thing that works.

After all, this is Haskell. Refactoring is easy. Refactoring with experience of the problems you're solving is even easier!

```
    recConstrs <- for constrs $ \constr -> case constr of
        RecC recName fields ->
            pure OldConstructor
                { oldConstructorName = recName
                , oldConstructorFields =
                    map (\(n,b,t) -> OldRecordField n b t) fields
                }
        _ ->
            fail
                $ "Expected a record constructor, but got: "
                <> show constr

    let newRec = NewRecord
            { newRecordName = newRecName
            , newRecordTyvars = tyvars
            , newRecordConstructors =
```

```
                map (oldConstructorToNew wo newFields) recConstrs
            }
-- snip --
oldConstructorToNew
    :: WrangleOpts
    -> [NewRecordField]
    -> OldConstructor
    -> NewConstructor
oldConstructorToNew wo newFields OldConstructor{..} =
    NewConstructor
        { newConstructorName =
            modifyName (constructorModifier wo) oldConstructorName
        , newConstructorFields =
            map mkNewField oldConstructorFields  ++ newFields
        }
  where
    mkNewField OldRecordField{..} =
        NewRecordField
            { newRecordFieldName =
                modifyName (fieldLabelModifier wo) oldRecordFieldName
            , newRecordFieldBang =
                oldRecordFieldBang
            , newRecordFieldType =
                oldRecordFieldType
            }
```

Look at this code. It's blindingly simple and obvious. There's no way to get it wrong.

Alright, let's continue pulling that thread. We're still not using the `OldRecord` type yet, which means we're slacking! The term `constrs` is pulled out in a pattern match on the reification of the type name.

```
    TyConI theDec <- reify tyName
    (name, tyvars, constrs) <-
        case theDec of
            DataD _ name tyVarBinders _ constructors _ ->
                pure (name, tyVarBinders, constructors)
            NewtypeD _ name tyVarBinders _ constructor _ ->
                pure (name, tyVarBinders, [constructor])
            _ ->
                fail
                  $ "Expected a data or newtype declaration, "
                  <> "but the given name \""
                  <> show tyName
                  <> "\" is neither of these things."
```

Another "effectful only because of `fail`" case. The `case` expression picks apart the `DataD` and `NewtypeD` cases and makes them uniform. This is what our `OldRecord` type is for. Let's use it.

```
    let mkOldConstr constr = case constr of
            RecC recName fields ->
                pure OldConstructor
                    { oldConstructorName =
                        recName
                    , oldConstructorFields =
                        map (\(n,b,t) -> OldRecordField n b t) fields
                    }
            _ ->
                fail
                    $ "Expected a record constructor, but got: "
                    <> show constr
    oldRecord <-
        case theDec of
            DataD _ name tyVarBinders _ constructors _ -> do
                constructors' <- traverse mkOldConstr constructors
                pure OldRecord
                    { oldRecordName = name
                    , oldRecordTyvars = tyVarBinders
```

```
                    , oldRecordConstructors = constructors'
                    }
            NewtypeD _ name tyVarBinders _ constructor _ -> do
                constructor' <- mkOldConstr constructor
                pure OldRecord
                    { oldRecordName = name
                    , oldRecordTyvars = tyVarBinders
                    , oldRecordConstructors = [constructor']
                    }
            _ ->
                fail
                  $ "Expected a data or newtype declaration, "
                  , "but the given name \""
                  <> show tyName
                  <> "\" is neither of these things."
```

We've pretty much completely threaded this data dependency through now, and we're ready to write some pure functions.

```
oldRecordToNew
    :: WrangleOpts -> OldRecord -> [NewRecordField] -> NewRecord
oldRecordToNew wo OldRecord{..} newFields =
    NewRecord
        { newRecordName =
            modifyName (typeNameModifier wo) oldRecordName
        , newRecordTyvars =
            oldRecordTyvars
        , newRecordConstructors =
            map (oldConstructorToNew wo newFields)
                oldRecordConstructors
        }
```

Now *this* is a good function to test. It's pure, and all data dependencies are explicit and self-contained. It's also small!

Finally, we can factor out some of the "data extraction" functions into pure variants. We can use `MonadFail` to abstract over failure - `Q` will fail with a compile-time error, while `Either` is nice and pure.

Our final effectful function looks like this:

```
mkNewRecordDec :: WrangleOpts -> OldRecord -> DecsQ
mkNewRecordDec wo oldRecord = do
    newFields <- processNewFields wo
    pure [newRecordToDec (oldRecordToNew wo oldRecord newFields)]
```

All the logic is pure. `oldRecordToNew` "parses" our datatype into the easily inspectable "new" variant. Then `newRecordToDec` "renders" our new datatype into the target we care about.

17. Logging and Observability

Everyone logs. `println` debugging is baked into basically every programming language. It's a simple and easy way to get observability into your system.

Haskell's focus on purity makes logging a bit trickier. In nearly any other context, you're able to just call `print` and see what happens. Haskell makes it more difficult - to call `print` in Haskell, you need to either refactor everything into `IO`, or you need to use `unsafePerformIO` to discharge it! Neither of these solutions are fun.

Fortunately, these same requirements make logging in production a potential joy to use. As it happens, logging can be a surprisingly complicated topic - poorly handled file system encodings can kill a logging thread, and that might bring things down in surprising ways. Explicit logging contexts and handles tends to make code that logs *easier* to understand and use, not worse. Since logging can be incorporated into the same system that provides other resources, you can easily tie logging into the rest of your application stack in a relevant way.

Let's learn about a way to log in Haskell nicely and effectively.

17.1 On `Debug.Trace`

`println` debugging is an important and useful tool. In Haskell, if you're not in `IO`, you can use the `Debug.Trace` module, and it'll print things to the standard error handle. This works *really* well for debugging pure code, and it's not well known!

Using `Debug.Trace`

While I generally prefer to factor code into smaller bits and write tests to debug and diagnose behavior, it's often *much* easier to just stick a `trace` in there and observe what is happening.

The function `trace` has an odd signature:

```
trace :: String -> a -> a
```

The `String` message is printed when the second argument is evaluated. This is how you have to do things in a lazy expression language - we don't "have" a sequential ordering, necessary, so you attach "effects" to "data" and not to lines in a control structure.

The function `trace` is mostly useful for expressions like `trace "evaluating foo" foo`. If you want details on what `foo` is, you can use `traceShowId`:

```
traceShowId :: Show a => a -> a
traceShowId a = trace (show a) a
```

There's also `traceM`, which can be used in an `Applicative` context and can be useful for debugging `Maybe` expressions:

```
what :: Maybe Int
what = do
    traceM "ok starting"
    blah <- foo
    traceM "blah was Just"
    bar <- baz
    traceM "baz returned Just"
    argh <- noooo
    traceM "argh is just"
    pure (x argh)
```

If we evaluate `what` and get `Nothing` back, then these traces can tell us exactly which line short-circuited the computation.

When do traces appear?

Understanding exactly *when* messages will get printed can be confusing. With `trace :: String -> a -> a`, the message in the first parameter

is attached to the data. It will be printed when the data is *evaluated*, and only once.

If you are confused about this, or are investigating a weird tracing error, then this section may be useful to you. Otherwise, you can safely skip it.

This is another trick of Haskell's inherent laziness. It works just like how *exceptions* are attached to data, not control flow.

This means you might not get messages printed out when you expect it. Let's consider this code block:

```
import qualified Debug.Trace as Debug

numbers :: [Int]
numbers =
    Debug.trace "Numbers evaluated!"
    $ map Debug.traceShowId [1 .. 10]
```

So when will these `trace` statements happen? It depends on how the `numbers` list is consumed. Let's open it in GHCi and see what happens.

```
>>> length numbers
Numbers evaluated!
10
>>> length numbers
10
```

Note that the message is only printed the *first* time we use the data. This is because the message printing is tied to the *evaluation* of the data, and once GHC has evaluated the data, it doesn't *reevaluate* it. Now, here's a line that you can easily copy into GHCi.

```
numbers = Debug.trace "outer eval" $ map Debug.traceShowId [1..10]
```

The only difference is that there isn't a type signature. Now, our trace output looks subtly different.

```
>>> numbers = Debug.trace "outer eval" $ map Debug.traceShowId [1..10]
>>> length numbers
outer eval
10
>>> length numbers
outer eval
10
```

This is because the *inferred type* of `numbers` is `Num a => [a]`. This `Num a` constraint is an *implicitly passed* function parameter. This function parameter causes GHCi to re-evaluate the list every time you demand it! The monomorphism restriction deals with exactly this behavior, and it can be confusing if you don't know what's going on.

It's actually nice to have the messages print each time the data is evaluated while playing with it, so we'll stick with the `Num a => [a]` version. `length numbers` causes `"Numbers evaluated!"` to be printed. What about `map (\_ -> ()) numbers`?

```
>>> map (\_ -> ()) numbers
Numbers evaluated!
[(),(),(),(),(),(),(),(),(),()]
```

We don't get any of the numeric output. What about this?

```
>>> map (\(i, x) -> if i > 3 then x else 0) (zip [0..] numbers)
Numbers evaluated!
[0,0,0,0,5
5,6
6,7
7,8
8,9
9,10
10]
```

So, this is a bit tricky! We're getting *interleaved output*. This is because we're actually evaluating the *numbers of the list* as we are printing them

for the first time! We don't actually demand anything from the list until we start printing it - consuming it in the real world.

It's important to attach *lots* of good information to a `Debug.Trace` statement. Knowing exactly when an item will br printed can be difficult to understand, and if your `Trace` statements aren't well-labeled, you can easily confuse yourself.

Beware Laziness

Attaching a `trace` message to a value prints the message when the value is demanded. However, printing that message out *itself* demands the message, which (in turn) demands anything you use to create the message. If you're writing code that relies on laziness, then you need to ensure that you don't demand things early while tracing.

This section is inspired by a debugging story, where I was trying to fix some `TemplateHaskell` code in the `persistent` database library. The `TemplateHaskell` code created a self-referential `Map Name EntityDef`, where `EntityDef` had lookups for other `Names` in their creation code. The code would loop infinitely during compilation. I inserted several `trace` expressions to try and determine what was going on, and I ended up identifying something dodgy. But I couldn't get the problem to go away! It kept looping. And it kept looping *because the `trace` expressions were forcing data*, which broke the laziness of the knot tying approach.

17.2 Prefer `do` Notation

Or, more generally, avoid point-free code.

Logging (or debug tracing) is one of the most common requirements for code. A code style that makes logging easy will be more pleasant to work with on average.

Let's look at some combinator-heavy code that deals with logic you might find in a CRUD app.

```
getUserPurchasesR :: UserId -> Handler UserPurchases
getUserPurchasesR userId =
    UserPurchases
        <$> (maybe (throwM NotFound) pure =<< (runDB (get userId)))
        <*> (runDB (selectList [UserPurchaseUserId ==. userId] [])
            >>= filterM isValidPurchase
            )
```

If we want to log the found user, it's a bit of a nuisance.

```
getUserPurchasesR :: UserId -> Handler UserPurchases
getUserPurchasesR userId =
    UserPurchases
        <$> (maybe (throwM NotFound) logAndReturn =<< (runDB (get userId)))
        <*> (runDB (selectList [UserPurchaseUserId ==. userId] [])
            >>= filterM isValidPurchase
            )
  where
    logAndReturn user = do
        log user
        pure user
```

Likewise, if we want to log the invalid purchases, it's even more of a nuisance:

```
getUserPurchasesR :: UserId -> Handler UserPurchases
getUserPurchasesR userId =
    UserPurchases
        <$> (maybe (throwM NotFound) pure =<< (runDB (get userId)))
        <*> (runDB (selectList [UserPurchaseUserId ==. userId] [])
            >>= filterM isValidPurchaseLogged
            )
  where
    isValidPurchaseLogged purchase = do
        t <- isValidPurchase purchase
        unless t (log purchase)
        pure t
```

We have to create a whole separate `where` clause and function definition, all to `log` some value out.

With `do` notation from the start, this code looks much nicer.

```
getUserPurchasesR :: UserId -> Handler UserPurchases
getUserPurchasesR userId = do
    muser <- runDB $ get userId
    user <-
        case muser of
            Nothing ->
                throwM NotFound
            Just u ->
                pure u
    allPurchases <- runDB $
        selectList [UserPurchaseUserId ==. userId] []
    validPurchases <- filterM isValidPurchase allPurchases

    pure $ UserPurchases user validPurchases
```

The code is undoubtedly more verbose, but adding logging statements becomes trivially easy. The code is also much more resilient to other modifications.

In general, I often find the need to refactor code *away* from an operator-heavy, point-free style. I never need to refactor code *to* this style. I tend to save more time by writing the more verbose style up front.

Additionally, this style is easier to read, understand, and teach.

17.3 Logging Contexts

In Haskell, you will almost be certainly writing things in a `Monad` of some sort. The transparent inclusion of monads into your code makes it pretty easy to write some slick combinators for logging with context.

Writing log messages that renders stuff into text and stores a bunch of information in strings is a great way to debug things, but it's neither fun

nor scalable. Especially since log messages generally *can't* be modified without modifying the message itself, and sometimes you want a bunch of messages to include extra information.

Logging contexts are a natural solution to this problem, and work nicely in `MonadReader` and `ReaderT` environments. If your logging library doesn't support this, it's pretty easy to bolt on. `katip` supports it natively, and that's why it's my favorite logging library.

Let's say we've got a web app that checks the authentication in the header before dispatching on a route.

```
app :: Route -> Handler ()
app route = do
    mauthInfo <- getAuthentication
    case route of
        HomeR -> getHomeR
        ProfilesR -> do
            isLoggedIn mauthInfo
            getProfilesR
        ProfileEditR profileId -> do
            user <- isLoggedIn mauthinfo
            user `canSeeProfile` profileId
            getProfileEditR profileId
```

For the `HomeR` route, we let just anyone in. To access `ProfilesR`, you must be logged in. And to edit a profile, you must have the proper authorization for that.

However, suppose we want to log the `UserId` that is logged in, along with the relevant authentication information. We don't have that information in `getProfileEditR` - we only pass in the `profileId`. The calling code would need to re-acquire the `User` to stuff it into the logs.

Instead, we'll add the user to the context. The way this works is that you have some function like:

```
addToLogsInBlock
    :: (ToLogThing a, MonadLogging m)
    => a -> m r -> m r
```

Now, let's modify our web app's router to include the user authentication information in the relevant code.

```
app :: Route -> Handler ()
app route = do
    mauthInfo <- getAuthentication
    addToLogsInBlock mauthInfo $ do
        case route of
            HomeR -> getHomeR
            ProfilesR -> do
                isLoggedIn mauthInfo
                getProfilesR
            ProfileEditR profileId ->
                addToLogsInBlock profileId $ do
                    user <- isLoggedIn mauthinfo
                    addToLogsInBlock user $ do
                        user `canSeeProfile` profileId
                        getProfileEditR profileId
```

Now we get a treat: all log entries in the router will have the `mauthInfo` relevant log information displayed along with the regular log messages. For the `ProfileEditR` route, we'll know which `ProfileId` is being modified by the log lines, even when we're calling `isLoggedIn` - which has no idea what route it is being called for. We'll also log relevant details about the user throughout the `getProfileEditR` and `canSeeProfile` function calls.

Aside from providing a lot of extra information on every log line, you also don't need to worry about stitching together messages as much. Consider these two scenarios:

```
noContext =
    log $ mconcat
        [ "The user ", show user
        , " can't modify the profile "
        , show profileId
        ]
```

However, if we know the user and profile ID are in the context, then we can just write a much simpler message:

```
context =
    withLogContext user $ withLogContext profileId $
        log "The user cannot modify the profile"
```

By delegating to a type class for representing metadata in logs, we also get the ability to give consistent log structure for items. Instead of grepping for some combination of `user-id`, `userId`, `USER_ID`, etc, you can just Know that the `instance ToLogItem UserId` renders it as a `{"userId":2134}` - no fuss.

Implementing Log Contexts

While `katip` supports this natively and `monad-logger-prefix`[1] adds this functionality to `monad-logger` in a crude way, it's not hard to bolt this functionality on to any other logging setup.

First, you want a `Reader`-like class that is specialized to a log context:

```
class Monad m => MonadLogContext m where
    askLogContext :: m LogContext
    localLogContext :: (LogContext -> LogContext) -> m a -> m a
```

You'll write instances of this class for your various monads and monad transformers. Since it's `Reader`-like, you can copy/paste implementations for `MonadReader`, or you can use `DerivingVia` with a helper class for focusing on the `LogContext`:

[1] https://hackage.haskell.org/package/monad-logger-prefix

```haskell
class HasLogContext a where
    _logContext :: Lens a LogContext

newtype ViaReader m a = ViaReader { unViaReader :: m a }
    deriving newtype (Functor, Applicative, Monad)

instance
    (HasLogContext a, MonadReader a m)
  =>
    MonadLogContext (ViaReader m)
  where
    askLogContext =
        ViaReader (view _logContext)
    localLogContext f action = do
        ViaReader $
            local (over _logContext f) (unViaReader action)
```

This form can be easier when it's easier to write instances for modifying the value in the `Reader` environment than writing the relevant `local` function.

Now, you'll just need to modify your `log` function to ask for the log context and include it in the message.

```haskell
instance MonadLogger App where
    monadLoggerLog loc src lvl msg = do
        context <- askLogContext
        underlying log function with context
```

With the right instances, you'll even get log output for libraries that have no idea about your context.

17.4 Libraries in Brief

There are many logging libraries in Haskell. In my experience, you are generally fine reaching for `monad-logger` for libraries and `katip` for applications.

monad-logger

monad-logger[2] is a respectable libary choice. It is simple, unopinionated, robust, and fast. As a result, it is a common logging interface that libraries use, and it is useful to understand how to use it.

In general, I find the TemplateHaskell log function variants to not be worth their cost: each TemplateHaskell splice causes a module to encounter the dreaded recompilation avoidance problem The CallStack variants do the same thing without a compilation time impact.

A common mispattern with this library is incorporating the LoggingT transformer directly into your own code. LoggingT is a "minimal" transformer that should be used only when providing MonadLogger behavior to types you don't control. Instead, consider writing a manual instance of MonadLogger, allowing you more flexibility in how the messages are used and formatted.

monad-logger-prefix

I wrote a library monad-logger-prefix[3] for adding prefixes to MonadLogger messages. This can be used to add log context to a library that doesn't want to add complexity beyond the usual MonadLogger types.

While this can be useful, I'd generally recommend building your own, or leveraging katip directly. The primary utility of this library is that it can transparently add context without requiring you to restructure your application or transformer code, provided that it is written in a sufficiently polymorphic style.

katip

The katip library is the gold standard of logging libraries in Haskell. It is fast, easy to use, easy to integrate, and has multiple scribes that can log to many destinations for important messages.

The KatipContext class provides you with the ability to have contextual logging. It uses JSON and text with a sophisticated censorship and verbosity system to log items configurably and clearly.

[2] https://hackage.haskell.org/package/monad-logger
[3] https://hackage.haskell.org/package/monad-logger-prefix

`katip`'s concept of a `Scribe` allows you to have multiple outputs for each log message. Each scribe can be configured separately. One particularly nice instance of this is the `katip-rollbar`[4], which sends `Error` level logs directly to Rollbar, an exception and error reporting service.

Integrating katip and monad-logger

When you've worked with enough code, you're bound to run into functions that expect a `MonadLogger` context, but you are primarily logging through `KatipContext` or `Katip` classes. You need a way to provide a "bridge."

The easiest way to do this is to produce a logging function which delegates to the `katip` underlying log function, and then use that with `runLoggingT` to discharge the constraint.

```
askMonadLoggerFunction
    :: (MonadUnliftIO m, Katip m)
    => m (Loc -> LogSource -> LogLevel -> LogStr -> IO ())
askMonadLoggerFunction = do
    UnliftIO runInIO <- askUnliftIO
    pure $ \loc src lvl logstr ->
        runInIO $
            logF () src (levelToSeverity lvl) (convertLogStr logStr)
```

Then, with the result of this function, you can use `runLoggingT` to pluck the `MonadLogger` constraint into a concrete `LoggingT` type, and it will use the `katip` log facilities:

```
cool = do
    logFunc <- askMonadLoggerFunction
    runLoggingT myAction logFunc
```

Now `MonadLogger` will work through `katip` in the `myAction` block.

[4] https://hackage.haskell.org/package/katip-rollbar

18. Databases

I like databases. I especially like a good SQL database, such as Postgres. It's an interesting set of problems - you have data at run-time, and that's all great. Totally normal programming. Values live in variables, you define functions and records and everything is easy.

Everything screeches to a halt when you need to read and write data to some other system. Suddenly you need to worry about *time* and *space* in ways that typical Haskell development don't care about. Sure, your program needs to execute quickly in minimal RAM. But once you've written some data to disk, you now need to care about how future versions of your application will read that data. We must become concerned with how our program will change over time.

We rarely store data *for fun*. Usually we need to store data so we can shut the program off and turn it back on. Or we store data so another program can interact with it. We want to be able to analyze, inspect, and query our data - and it has to be fast!

There are a huge variety of database engines. Unless you're a specialist and you know *exactly* what you're doing, you should ignore almost all of them. A SQL database is the right choice for almost every single application. PostgreSQL should be your first choice for a server-based application and works surprisingly well for desktop apps, too. SQLite occupies the other end of the spectrum - perfect for locally hosted databases and surprisingly good as a server database.

SQL databases force you to think about your data representation more than other forms. This is Great, because it means you'll come up with better answers than if you didn't think about it. Additionally, writing about data storage with SQL means you can remove functionality you don't want, and still get valuable lessons from this chapter.

18.1 Separate Database Types

Web programmers must repeat themselves a few times with many domain concepts. They'll need to have a database representation for a domain concept. They'll need a friendly representation for their domain computation. And finally, they'll need a representation for sending the concept to the front-end, rendering as HTML or JSON.

In Ruby, or other less-typed languages, you'll have a hash map or object come out of the database. It gets pushed into a class, which is the preferred domain representation. Finally, you'll convert it back to a hashmap/object and blast it over the wire.

Haskellers will adopt this pattern by starting with a single type - the domain type. We write a program without any concern for databases or serialization. Once it works, we write a `ToJSON` instance and send it over the wire. We'll write a function that saves it to the database. Everything is fine.

Except... everything is not fine. Our initial domain logic didn't need to care about identifiers. We simply *had* a `User`. And our database table doesn't look exactly like a `User` - we also want some timestamps (`created_at`, `updated_at`) to help with SQL debugging. So we attach a few fields on our domain type to make it match our database representation.

```
1  data User = User
2      { userName :: String
3      , userAge :: Int
4      -- ^ domain fields
5      , userCreatedAt :: UTCTime
6      , userUpdatedAt :: UTCTime
7      , userId :: UUID
8      -- ^ database fields
9      }
```

Too Many Responsibilities

We have now violated a valuable precept: the "Single Responsibility Principle." This type has *three* responsibilities:

1. Represent a database table
2. Represent a domain concept
3. Represent an API response shape

This doesn't seem like such a big deal in small and easy cases. However, you'll immediately run into problems. Consider: how do you create a user? A `User` that has not yet been created can't be said to have the `userCreatedAt`, `userUpdatedAt`, or `userId` fields. We can model this with `Maybe` - now the API can receive `null` values in those fields, and the handling code can set them to `Just` the appropriate things.

You may have predicted where this problem is going. Now the database is returning a `userCreatedAt :: Maybe UTCTime` field that is *always* set to `Just`. It's a runtime guarantee. So you're tempted to rely on it - if the `User` came from the database, then `fromJust` is perfectly fine!

This all works great until you have some innocuous looking function that takes a `User` from the API and calls that other function. Boom. A runtime error. And, worse, the error carries little information.

```
λ> fromJust Nothing
*** Exception: Maybe.fromJust: Nothing
CallStack (from HasCallStack):
  error, called at Data/Maybe.hs:148:21 in base:Data.Maybe
  fromJust, called at <interactive>:3:1 in interactive:Ghci1
```

Hopefully your program handles `CallStacks` well and has good error reporting facilities. It would be better if we didn't misuse `Maybe` like this.

Painted into a Corner

When you start out, your domain and database representation often correspond neatly. You probably wrote your database serialization to handle

you domain type exactly. This exact correspondence won't survive for long. Eventually, the requirements of the database, API, and domain will drift enough that you'll be forced to make more and more compromises.

If you're not careful, these compromises can seriously harm the ease-of-use and performance of your code. Suppose that you add a Dog feature to your app - now `Users` can have a list of Dogs that they take care of. How do we make this work?

The easy way for the API to accomplish this is with a `userDogs :: [Dog]` field. However, this isn't going to play nicely with SQL databases. Sure, you can encode the `[Dog]` as a JSONB column, but now you're running denormalized data - it's about to get really annoying to write SQL and analytics code! "How many dogs exist in the system?" went from a trivially easy and fast query to a slightly tricky and slow query.

Meanwhile, the easy way for the database to accomplish this is to have a Dog table, where each Dog refers to a `UserId`. The front-end of the application first requests the `User`, and then requests all of the Dogs for that user. So we have to extend our API to perform a relatively common task - "A `User` with all of their Dogs."

This problem is bad enough when we allow a single `User` for each Dog. But what if we eventually need to support multiple `Users` for Dogs? And what if a Dog doesn't currently have a `User`?

You have three choices now:

1. Write with the API in mind, compromising the database
2. Write with the database in mind, compromising the API
3. Refactor the app to use separate types

Every time you pick 1) or 2), you'll make picking 3) harder. The compromises will stack up, making your application worse. The third option is the correct one, so you might as well just start out with it.

A panoply of types

My recommendation is to write multiple types - one for each representation. Yes, you will have some boilerplate-y conversion functions. This

is not a real problem. Eventually, those functions will become more complex. That's when you know that you've saved yourself an absolutely massive amount of trouble.

You'll have a type for the database table. You'll have another type for your domain computations. For each API response shape or View, you'll have another type.

Got a problem? Write a new `data` type that fits the problem. Get in the habit of writing new data types for new problems. You may end up writing a datatype that's *exactly* the same as another one - you can choose to keep it or re-use the other one. But as *soon* as the demands from two uses cause *any* tension in the datatype, just split it in two.

By keeping our data types numerous and specialized, we're making it easier for each datatype to evolve and accommodate new features. The complexity inherent in the code base is now localized to the conversion functions. These conversion functions, at simplest, are pure functions accepting a `Domain` type and returning a `Database` type (or vice-versa).

```
1  toDb :: Domain.User -> Database.User
```

However, as your types and database representations become more complex, you may have multiple database tables that represent a single domain type.

```
1  toDomain
2      :: Database.User
3      -> [Database.Dog]
4      -> Domain.User
```

You may *also* end up needing to perform some effects in this conversion.

```
1  toDomain
2      :: Database.User
3      -> [Database.Dog]
4      -> IO Domain.User
```

This complexity will feel a bit nasty. However, you should really rejoice - this complexity is now surfaced and localized. If you had tried to maintain a single type to handle all of these responsibilities, then that complexity would have trickled throughout the rest of the application.

Normalization

So, you decide to have three representations. Hooray!

```
data User = User
    { userName :: String
    , userAge :: Int
    }

data ApiUser = ApiUser
    { name :: String
    , age :: Int
    , dogs :: [Dog]
    }

data DbUser = DbUser
    { id :: UUID
    , name :: String
    , age :: Int
    , createdAt :: UTCTime
    , updatedAt :: UTCTime
    }
```

All three types contain a `name` and `age` field. Is this a problem? Maybe, maybe not. The entire point of this exercise is that we want to allow the different types to evolve as necessary to best solve their problems. We can reduce a bit of duplication by nesting the `User` in where the two fields are:

```
data ApiUser = ApiUser
    { user :: User
    , dogs :: [Dog]
    }

data DbUser = DbUser
    { id :: UUID
    , user :: User
    , createdAt :: UTCTime
    , updatedAt :: UTCTime
    }
```

We have made some things easier. We have saved some typing and field duplication. For the purposes of our exercise, we can imagine there are tens of fields on the `User` type. Our conversion functions are likely a lot simpler, now:

```
toApiUser :: User -> [Dog] -> ApiUser
-- old
toApiUser user dogs =
    ApiUser
        { name = userName user
        , age = userAge user
        , dogs = dogs
        }
-- new
toApiUser user dogs =
    ApiUser
        { user = user
        , dogs = dogs
        }
```

If a new field gets added to the `User` type, then it is *automatically* added to the database and API types. This may or may not be good.

We have made some things *impossible*. We must have the exact same fields (and types) of a `User` in the database, API, and domain. The API and

the database have picked up a shared thread of responsibility by relying directly on the domain type.

Additionally, we've made some things harder. If the fields are all "flat", then it's easy to use `TemplateHaskell` or `Generic` to derive JSON or database serialization formats. But if your API type has a nested representation and you want to flatten it, then you need to write additional code.

```
instance ToJSON ApiUser where
    toJSON apiUser =
        object
            [ "name" .= userName (user apiUser)
            , "age" .=  userAge (user apiUser)
            , "dogs" .= dogs apiUser
            ]
```

This frustration alone may be sufficient to simply duplicate field entries.

I titled this section "normalization" because we're in a database chapter. Normalized data is often great in a relational database, because it reduces the scope for errors and problems and duplications. When we factor out the common structure of the `User` type instead of duplicating the fields, we are *normalizing* the `ApiUser` and `DbUser` types. However, the denormalized form can work better for application code.

Avoid polymorphism

I'm dodging tomatoes. The audience is brutal. They're yelling and hollering, telling me to log off, and *I will never log off*.

Polymorphism is a beloved feature of Haskell, and it *can* often work well. However, I'm here to tell you that it probably won't make your life better when dealing with this specific problem. Let's look at the technique and dig into some of the problems.

This technique looks at the "Users and Dogs" problem and decides to add a *type variable* to represent dogs.

```
data User a = User
    { userName :: String
    , userAge :: Int
    , userDogs :: a
    }
```

Now, the type variable tells us all we need to know about what's going on.

```
type UserNoDogs     = User ()
type UserWithDogIds = User [DogId]
type UserWithDogs   = User [Dog]
```

We can even get a `Functor` instance For Free so we can use `fmap` to modify the `userDogs` field.

Unfortunately, this stops being nice as soon as we need to do this with another field. Now we need to model `createdAt` -

```
data User dogs created = User
    { userName :: String
    , userAge :: Int
    , userDogs :: dogs
    , userCreatedAt :: created
    }

type CreateUser         = User () ()
type UserFromDatabase   = User () UTCTime
type UserFromDbWithDogs = User [Dog] UTCTime
```

Now, `fmap` only works on the `userCreatedAt` field, since `fmap` only works on the *last* type variable in a datatype. Any mention of the `User` type needs to be modified to accommodate this field change. The solution just doesn't scale.

I'm going to let you in on a little secret: you can do this with tuples.

```haskell
data User = User
    { userName :: String
    , userAge :: Int
    }

type CreateUser         =  User
type UserFromDatabase   = (User, UTCTime)
type UserFromDbWithDogs = (User, UTCTime, [Dog])
```

If you have a polymorphic field in a record, you can delete it and use a tuple to pair things up. Maybe use Tagged to give a name to it. Or maybe just write three data types with good names.

Avoid HKD

Higher Kinded Data (HKD)[1] is a potential solution to this problem. It looks at the issues with polymorphism and says "What if we solved the problem with *more polymorphism*?" Generally speaking, when I have dug myself into a hole, I try to stop digging. Sometimes, that means I miss gold. Sometimes, that means I get back to the surface unscathed.

The HKD pattern varies the *representation* of the fields by storing everything in a type constructor.

```haskell
data UserF f = User
    { userName :: f String
    , userAge :: f Int
    , userDogs :: f [Dog]
    , userCreatedAt :: f UTCTime
    , userUpdatedAt :: f UTCTime
    }

newtype Identity a = Identity { runIdentity :: a }

type User = UserF Identity
type UserNullable = UserF Maybe
type UserValidated = UserF (Validated [String])
```

[1] https://reasonablypolymorphic.com/blog/higher-kinded-data/

Now, we can have an `ApiUser` be represented as a `UserF Maybe`. The API could send us a value like the one in `fromApi` below, and we can have a function `saveUser` that'll stuff it into the database and return the full value.

```
fromApi :: UserF Maybe
fromApi =
    User
        { userName = Just "Bob"
        , userAge = Just 32
        , userDogs = Nothing
        , userCreatedAt = Nothing
        , userUpdatedAt = Nothing
        }

savePartial :: UserF Maybe -> IO (Maybe (UserF Identity))
savePartial umaybe = do
    now <- getCurrentTime
    let mkuser name age  = User
            { userName =
                Identity name
            , userAge =
                Identity age
            , userDogs =
                Identity []
            , userCreatedAt =
                Identity now
            , userUpdatedAt =
                Identity now
            }
    pure $
        mkuser
        <$> userName umaybe
        <*> userAge umaybe
```

We have to handle the possibility that the API didn't send up a name or age, since the `f` type is applied to every field uniformly. We also have to

wrap everything in `Identity` which isn't fun. We *could* make the `f` type only apply to the "non-essential" fields - but then you could just factor those out into their own type, and then, why are you bothering with fancy techniques? We can also upgrade to using a type family to strip out the `Identity` type, if it's there:

```
type family HKD f a where
    HKD Identity a = a
    HKD f a = f a
```

Sticking a type family in the record field complicates several things and can lead to some strange error messages.

This technique has given us flexibility over *how* the record is presented - are all fields present? are all fields nullable? are all fields in IO? - but it has not given us flexibility in distinguishing between fields. We can't add or remove fields to the record using this technique. It only solves the simplest issues we had with multiple responsibilities, and not even well.

This technique is pretty heavy and it's not pulling its weight. Maybe we can save it by having a *type of phases* that the type family accepts. This technique is known as Trees That Grow[2]. Let's look at it.

```
data UserPhase = Domain | API | Database
data UserField = Name | Age | Dogs | Timestamp | Id

data UserF (f :: UserPhase) = User
    { userName :: Pick f Name
    , userAge :: Pick f Age
    , userDogs :: Pick f Dogs
    , userCreatedAt :: Pick f Timestamp
    , userUpdatedAt :: Pick f Timestamp
    , userId :: Pick f Id
    }

type family Pick (f :: UserPhase) (field :: UserField) where
    Pick Domain Name = String
```

[2] https://www.microsoft.com/en-us/research/uploads/prod/2016/11/trees-that-grow.pdf

```
15      Pick Domain Age = Int
16      Pick Domain Dogs = ()
17      Pick Domain Timestamp = ()
18      Pick Domain Id = ()
19
20      Pick API Name = String
21      Pick API Age = Int
22      Pick API Dogs = [Dog]
23      Pick API Timestamp = UTCTime
24      Pick API Id = UUID
25
26      Pick Database Name = String
27      Pick Database Age = Int
28      Pick Database Dogs = ()
29      Pick Database Timestamp = UTCTIme
30      Pick Database Id = UUID
```

Adding a new field to `User` requires a new entry in the `UserField` type and a new entry per phase in the `Pick` type family. Adding a new phase requires a definition of how to interpret that phase for each field. You can omit fields from a phase by "picking" the `()` type for them.

This is *a lot* of complexity, all to avoid writing a few different simple and easy data types for your domain purposes. There are applications where HKD fit great, but this is *not one of them*.

18.2 Migrations

Migrations in Haskell aren't really any different than in any other language. As a result, there's not a lot of Haskell-specific advice I can give. For my workplaces, we've used a home-grown migration system, a Haskell library, a Java library, and lastly the Ruby on Rails framework for managing migrations. The difficulty is in the SQL, not the Haskell.

18.3 Access Patterns

Once you've defined a schema and some data types for your tables, you'll almost certainly need to define *yet more* data types that represent actions upon the database. At the very least, you'll want the Create, Read, Update, Delete basics - CRUD, as it is known. As above, I'm going to recommend *a panoply of types* - the more the merrier. Data types are useful for representing a snapshot of data, and they're also useful for representing an *action* that you want to perform on the world.

You may wish to abstract these patterns into type classes. Doing this has several advantages and disadvantages. The main advantage is that you can use a single name and interface for each action. The class and instances serve as excellent documentation for your domain serialization model. The main disadvantage is that the type class interface must satisfy every use-case for every model you need. This may be an onerous requirement.

Fortunately, this is Haskell - try one out, and if you don't like it, refactor the application to not need it.

Create

The naive interface to `create` is something like this:

```
data User = User
    { userId :: UserId
    , userName :: String
    , userAge :: Int
    , userCreatedAt :: Maybe UTCTime
    }

createUser
    :: DatabaseConnection
    -> User
    -> IO UserId
createUser conn user = do
    freshId <- Database.freshId conn
```

```
    now <- getCurrentTime
    let readyToInsert :: User
        readyToInsert =
            user
                { userId = freshId
                , userCreatedAt = Just now
                }
    Database.insert conn readyToInsert
```

The function accepts a domain object `User`, fills in the necessary details, and stuffs it into the database. This is unsatisfactory - what if we wanted to specify a `UserId` before the `createUser` function, like in a database export/import? Likewise, the client of our code must pass `userCreatedAt = Nothing`, and the API serialization for a new user must provide that field.

Instead, we'll want to create a datatype that represents the information necessary to add a user to our system.

```
data CreateUser = CreateUser
    { createUserName :: String
    , createUserAge :: Int
    }

createUser
    :: DatabaseConnection
    -> CreateUser
    -> IO User
createUser conn cu = do
    freshId <- Database.freshId conn
    now <- getCurrentTime
    let readyToInsert :: User
        readyToInsert =
            User
                { userId = freshId
                , userCreatedAt = Just now
                , userName =
                    createUserName cu
```

```
                , userAge =
                    createUserAge cu
                }
    Database.insert conn readyToInsert
    pure readyToInsert
```

Now our interface precisely captures what we're doing. The implementation interprets the `CreateUser` type into an *action* that actually creates the corresponding `User`. In a real database implementation, you'll likely have a trigger setting the `created_at` time stamp, and the database may even be in charge of generating the relevant `UserId` field.

Type Classes and Trade-offs

In a type class approach, you might write these as an associated type.

```
class Create a where
    type New a

    create :: New a -> DB a
```

Then, our implementation for `User` would look like this:

```
data CreateUser = CreateUser
    { createUserName :: String
    , createUserAge :: Int
    }

instance Create User where
    type New User = CreateUser

    create cu = do
        freshId <- Database.freshId
        now <- getCurrentTime
        let readyToInsert :: User
            readyToInsert =
```

```
                User
                    { userId = freshId
                    , userCreatedAt = Just now
                    , userName =
                        createUserName cu
                    , userAge =
                        createUserAge cu
                    }
        Database.insert eadyToInsert
        pure readyToInsert
```

The associated type means that we can only have a single `New` type for any record. We are forbidden from defining *another* instance of `Create User` that uses a different kind of type to initialize it. However, we are not forbidden from reusing `CreateUser` for a *different* domain model's `New` instance. This means that GHC will be unable to infer that the return type of `create CreateUser { .. }` is a `User` - this is going to require some type annotations that you may find annoying.

To work around this, you can use an *associated data family* or an *injectivity annotation*. An associated data family uses the `data` keyword instead of `type` in the associated type declaration. An injectivity annotation resembles the syntax for functional dependencies.

```
-- with a data family
class Create a where
    data New a

    create :: New a -> DB a

instance Create User where
    data New User = CreateUser
        { createUserName :: String
        , createUserAge :: Int
        }
    create cu = ...

-- with an injectivity annotation
```

```
15 class Create a where
16     type New a = r | r -> a
17
18     create :: New a -> DB a
19
20 data CreateUser = ...
21
22 instance Create User where
23     type New User = CreateUser
24
25     create cu = ...
```

A data family requires that you define the datatype inside of the type class. An injectivity annotation requires that you define the datatype separately. In either case, GHC will reject a program that tries to create another instance of `Create X` with a type `New X = CreateUser`. This means that GHC will happily infer that `create CreateUser {..}` returns a `User`, and everyone is happy.

Of course, all of these approaches only permit a single type for creating a `User`. If you want to have multiple methods of creating a `User`, then you may need to use a multiparameter type class.

```
1 class Create rec new where
2     create :: new -> DB rec
3
4 instance Create User CreateUser where
5     create cu = ...
```

However, this approach is going to require quite a few type annotations. Without a functional dependency, GHC will be unable to infer anything about this type. You will likely end up needing to supply a type annotation and write:

```
1 create @User CreateUser{..}
```

At this point, you may prefer to use conventions or specifically named functions to disambiguate these names.

```
import qualified Database.User as User

User.create CreateUser {..}

-- or,
createUser CreateUser {..}
```

Read

Converting database tables into a domain type is going to be a straightforward task, so we'll focus on the more interesting parts in this section: querying and looking things up.

Raw SQL

With a SQL library, you can almost always write some sort of "raw SQL" function to perform arbitrary queries.

```
allUsersWhere :: Text -> DB [Domain.User]
allUsersWhere where_ = do
    results <- rawSql ("SELECT ?? FROM users " <> where_)
    pure (databaseToDomain <$> results)
```

This interface is pretty bad. There's no type-safety in the form of the query. If you typo any part of this, you'll get a runtime error. If you delete a column or rename something, you'll get a runtime error. If a user manages to sneak some nasty fake data in that `where_` parameter, then you'll get a SQL injection attack. No good!

With all that said, a raw query approach has some advantages The biggest is that you don't need to learn a database library. Database libraries can be complex and difficult to learn (though not all of them are), and a raw SQL approach does allow you to use the entire power of SQL. However, I tend to get a huge amount of benefit from a database library, so I don't recommend this.

Query Types

You can prevent a lot of the problems with the raw SQL approach by constructing an intermediate representation. The intermediate representation is a datatype that contains the set of valid queries on the `User` type is.

```
data UserQuery
    = AllUsers
    | UserById UserId
    | UserWithName (Comparison String)
    | UserWithAge (Comparison Int)
    | AndUser UserQuery UserQuery
    | OrUser UserQuery UserQuery

data Comparison a
    = Comparison
    { operation :: ComparisonOp
    , value :: a
    }

renderWhere :: UserQuery -> Text
renderWhere uq = case uq of
    AllUsers ->
        "1 = 1"
    UserById uid ->
        "user.id = " <> renderUserId uid
    UserWithName (Comparison op val) ->
        "user.name " <> renderOp op <> " " <> escape val
    UserWithAge (Comparision op val) ->
        "user.age " <> renderOp op <> " " <> show val
    AndUser q0 q1 ->
        "(" <> renderWhere q0 <>
        ") AND (" <> renderWhere q1 <> ")"
    OrUser q0 q1 ->
        "(" <> renderWhere q0 <>
        ") OR (" <> renderWhere q1 <> ")"
```

This is a relatively simple query language against a single table. We can make it more powerful and more general, until eventually you've written your own database library.

Precise querying and control of the database layer is a common need in an application. I recommend keeping your database representation somewhat close-at-hand, so you can reach directly into the database to perform whatever querying and filtering you need. However, it's a good idea to develop a tighter point of control. A simpler API is always easier to test and understand than a more complicated one. If you ever want to mock or abstract a database, then having a simplified query language for your domain will come in handy.

Following the above advice, a type class API might look like this.

```
1  import Database.Esqueleto (SqlExpr, Value)
2
3  class Query a where
4      type DatabaseRepresentation a
5
6      query
7          :: (DatabaseRepresentation a -> SqlExpr (Value Bool))
8          -> DB a
```

The signature of `query` has a callback that accepts the `DatabaseRepresentation` of our query type, and returns an `esqueleto` SQL expression indicating whether the record passes the test. Then it runs that query against the entire table and returns what satisfies it. For a single table, this ends up being pretty straightforward. Supposing we have a domain type `User` and a database table type `Entity User` (from `persistent`), then we'll have this instance.

```
instance Query User where
    type DatabaseRepresentation User =
        SqlExpr (Entity DB.User)

    query predicate =
        fmap (fmap fromDatabase) $
        select $ do
            u <- from $ Table @Database.User
            predicate u
            pure u
```

The calling code might look like this:

```
import Database.Esqueleto ((>=.), (^.), val)

adultUsers :: DB [User]
adultUsers =
    query $ \user ->
        user ^. UserAge >=. val 18
```

Now, we've evolved our type, and the `User` has the field `userDogs :: [Dog]` now. Here's how we'll expand our type:

```
instance Query User where
    type DatabaseRepresentation User =
        SqlExpr (Entity DB.User) :& SqlExpr (Entity DB.Dog)

    query predicate =
        fmap combineDogs $
        select $ do
            res <- from $
                table @User
                `leftJoin`
                table @Dog
                    `on` do
                        \(u :& d) ->
```

```
14                              u ^. UserId ==. d ?. DogUserId
15              predicate res
16              pure res
17
18  adultUsers :: DB [User]
19  adultUsers =
20      query $ \(user :& maybeDog) ->
21          user ^. UserAge >=. val 18
22
23  withDogNamed :: String -> DB [User]
24  withDogNamed dogName =
25      query $ \(user :& maybeDog) ->
26          maybeDog ?. DogName ==. just (val dogName)
```

In this snippet, the `query` type calls into the database, using a `LeftOuterJoin` to associate a `User` with the given `Dogs`. This gives us the ability to further filter the `Dogs` that are returned from the list, too. Note that `withDogNamed` will return users that *do not* have a `Dog` matching the name - only `Dogs` with that name will be present in the result list.

There are a few special cases that we'll want to consider here. The first is a lookup by a unique key. Most database engines support some notion of uniqueness. The main consideration here is that a query can return `Maybe User` instead of `[User]`. The next special case is a special case of the prior general case - looking up a `User` by the primary key `UserId`. I don't think it's necessary or wise to differentiate between these two use cases, even though virtually every database library provides a `getById` sort of function.

The Primacy of Query

`Create` comes first in the acronym CRUD, but really, it is `Read` that is the foundational component of database access patterns. You must `INSERT` data to read it, but many database engines allow you to write `INSERT SELECT` statements. These statements allow you to synthesize database records from other database records - the output of `SELECT` is the input to `INSERT`.

Likewise, `UPDATE` and `DELETE` rely on querying in order to know *which* records to act upon. Trivially, you can `UPDATE` or `DELETE` all of the records

in the database, but in my experience that means you'll be restoring a backup soon.

The API you build for querying will almost certainly be reused when you're performing `UPDATE` and `DELETE` statements. If you use them at all - immutable tables work by only performing `INSERT` and `SELECT` statements, never actually deleting or updating a record - keeping a history of all the variations that a record has had throughout the life cycle of the application. For performance and convenience, though, we'll probably want to have those mutations available. And even if you *do* use an immutable table setup, you'll probably want to have an API for performing updates and deletes anyway, since they're so common.

Update

Updates are tricky to represent optimally. So let's evolve our understanding through a few suboptimal choices and cover the ways in which they fail.

Functional

In Haskell, we can represent an update on a domain type with ease.

```
1  type Update rec = rec -> rec
```

It's a function that accepts a value of type `rec` and returns a (presumably modified) value of type `rec`. This approach is fantastically flexible, but it has some pressing problems. Notably, we can't serialize this! We can't send a function over the API, and we can't parse one out of a JSON representation. We also can't write this to SQL and expect it to work out.

Functions in Haskell are *opaque*. If we could inspect the function and see what actually changed here, then we could serialize the differences and make an `UPDATE` that only contains those differences. However, we can't easily do that. We'd need to *run* the update on an original record and then compare the original with the updated variant. At that point we need a way to diff two records. The `prairie` library that I cover in this book is capable of `diff`ing two records, but it also provides a means of serializing updates. We'll get back to that in a second.

What we *really* want is a value that's open to inspection. Then we could easily convert that value to the corresponding function.

Batch Updates

The next approach that people will gravitate towards is to simply accept an entire record.

```
type Update rec = rec
```

This has a few big advantages. It's simple and easy. If `rec` is serializable (either `ToJSON` or to a database table), then we can serialize an `Update rec` - after all, they're the same! Unfortunately, this scheme can be expensive and dangerous.

Expensive

If we perform an `UPDATE` statement in SQL and we update *the entire record*, then we'll need to generate SQL like the following:

```
update :: _ => rec -> Sql ()
update record =
    runSql $ mconcat
        [ "UPDATE " <> tableName record
        , " SET "   <> convertToSet record
        , " WHERE " <> uniqueKey record
        ]
```

In `convertToSet`, we'll take each field of the record and tell the database engine to `SET` the value equal to the new record. For most columns, this is probably fine. However, any column with an index will cause the index to be recomputed *for the entire table*. This may be slow, if the index is big. It may also cause the database engine to hold a lock on the table for much longer than is required.

Is the database engine smart enough to know that a no-op modification doesn't require a rebuild? Maybe. Maybe not. Depends on the database engine and how smart the optimizer is on that particular query.

Dangerous

If you choose to represent an `Update` on a model with the whole model, then you may permit actions that you don't want to permit. For example, consider a primitive user permission model - some users are admins, and this is marked with a field on the `User` type called `isAdmin :: Bool`. Your standard boilerplate reduction functions will happily allow a `User` to modify their `isAdmin` field! You can mitigate this by removing the form field on the front-end, but as long as your API or server are parsing an entire `User`, you'll have to be careful here to prevent a privilege escalation attack.

It's easy to write an ad-hoc rule for this at some point in your codebase once you notice the problem. It's just as easy for this problem to slip in, unnoticed. A more careful approach would have never permitted this in the first place.

Update Fields

You're probably expecting this by now. "Here comes Matt to tell me to write a whole bunch of datatypes." Yup. It's only fair that you'd predict my solution - I've spent hundreds of pages arguing for it so far!

However, I'll recognize that writing a ton of datatypes can be a bit boring and boilerplate-y, especially since most of them will be similar. For the most part, you want to give the system a ton of flexibility, and having to thread update logic through multiple fields can be annoying when you have feature work to get through. For that reason, it's pretty typical to

In the `prairie` chapter, I walk through a library design that reifies record fields into first-class serializable values. `prairie` supports two important operations:

- `diff :: (Record rec) => rec -> rec -> [UpdateField rec]`
- `patch :: (Record rec) => [UpdateField rec] -> rec -> rec`

The type `UpdateField rec` specifies how to update a single record field on a `rec` with a new value. Since a `[UpdateField rec]` is a mere list

of values, it's quite easy to work with them in a way that doesn't do any waste. Rendering the `UPDATE` statement ends up being relatively simple and not particularly wasteful.

```
renderUpdate :: _ => [UpdateField rec] -> Text
renderUpdate [] = ""
renderUpdate (x:xs) =
    mconcat
        [ "UPDATE " <> tableName @rec
        , " SET " <> Text.intercalate ", " (map mkFields (x:xs))
        , " WHERE " <> uniqueKey @rec
        ]
  where
    mkFields (SetField field value) =
        mconcat
            [ renderField field
            , " = "
            , renderValue value
            ]
```

(in real code, you'd want to use parameter substitution to avoid SQL injection vulnerabilities - this is elided in this example for brevity)

We can avoid the problem with a regular user escalating their privileges by simply filtering the list:

```
filterIsAdminChanges :: [UpdateField User] -> [UpdateField User]
filterIsAdminChanges =
    filter $ \(SetField field _) ->
        field /= UserIsAdmin
```

If we're using a type class to provide security, we can make a pretty simple helper to provide this security:

```
class (Record a) => SafeUpdate a where
    permitField :: UserAuthLevel -> Field a -> Bool

instance SafeUpdate User where
    permitField Admin field =
        True
    permitField Regular field =
        field /= UserIsAdmin
```

This approach works quite well if you have a model that has a big "bulk update" form associated with it. It even works pretty well for composite models. Suppose that we have a `UserProfile` type which has the `User` and `Profile` database tables underlying it. We can render an update on multiple tables relatively easily:

```
updateUserProfile :: [UpdateField UserProfile] -> Sql ()
updateUserProfile allFields = do
    let (userFields, profileFields) =
            chooseFields allFields
    runUpdate userFields
    runUpdate profileFields

chooseFields
    :: [UpdateField UserProfile]
    -> ([UpdateField User], [UpdateField Profile])
chooseFields = ...
```

However, this approach falls apart when we have *nested* models. Consider the `UserWithDogs` type - it is backed by the `User` table and *also* the `Dog` table, but the associated `Dog`s are loaded in the table. The relevant update `SetField UserDogs listOfDogs` isn't great. It's opaque! There are a few ways we could handle this:

- Iterate over the `listOfDogs`, pulling it from the database, diffing it, and then perform an update if necessary. If the `Dog` is not present in the database, then insert it. If there are `Dog`s in the database not present in the list, remove the association. This is a lot of

complexity, and *also* brings in all of the problems with using a `Dog` type as an update for a `Dog`.
- Just delete all `Dogs` for the `User` and batch insert the new list of dogs. This is bad because any foreign keys will be upset about this change.
- Not bother. Don't allow the user to update `Dogs` while updating a `User`.

None of these are terribly satisfying. What we'd *really* like is if, instead of `SetField`, we had some kind of nested `DiffOf` type that would permit a *deep* update, instead of the shallow update on display here. However, that ends up being confusing and complicated to write generically, and specialized variants have a bunch of trade-offs and problems. We can remove the need for this complexity by defining multiple types.

Update Actions

This pattern represents the modifications to a domain model as *actions* that you can perform on the model. The simplest format accepts a list of the `UpdateField` for a record in a batch update:

```
data UserUpdate
    = BatchUpdate [UpdateField User]
```

Fortunately for us, the problem of `Dogs` can be handled nicely with separate *action* constructors that do the relevant work.

```
data UserUpdate
    = UserUpdateBatch [UpdateField User]
    | UserUpdateAddDog DogId
    | UserUpdateRemoveDog DogId
    | UserUpdateUpdateDog DogUpdate

data DogUpdate
    = DogUpdateBatchUpdate [UpdateField Dog]
```

With a richer intermediate datatype for actions we want to perform, the problems in the above methods evaporate. It's pretty easy to convert a `UserUpdate` value into a function `User -> User` - and it's *also* easy to convert it into a SQL statement that performs all the relevant updates, cheaply and efficiently.

Delete

Once you've queried a set of rows, you might decide that you're better off without them. You run a `DELETE` statement and now everything is gone.

This pattern is actually pretty straightforward. You probably don't need any help with it. If you've got a query interface you're happy with, throwing a `DELETE` in front of it works pretty well.

Cascade

If you try to `DELETE` a record that has a foreign key relationship with another table, and that table references the row you're trying to delete, then the database engine will throw an error and fail to delete. This can be really annoying. You want to delete the record, and ideally, the dependent rows should *also* be deleted, too!

We can tell SQL databases to `CASCADE` deletes.

```
CREATE TABLE user (
    id  AUTO INCREMENT PRIMARY KEY
);

CREATE TABLE dog (
    id  AUTO INCREMENT PRIMARY KEY,
    user_id INTEGER NOT NULL ON DELETE CASCADE
;
```

The magic word here is `ON DELETE CASCADE`. Now, if we delete a `User`, then all the `Dogs` that point to this `User` will also be deleted. If the `user_id` column is nullable, then we can also use `SET NULL` to orphan the dog row. The default option is `ON DELETE RESTRICT`, which causes SQL to throw an exception and abort the transaction.

If you want this behavior, then you should rely on SQL to do it. Manually digging through the foreign key relationship graph and deleting rows in the right order is both annoying and slow.

18.4 Conclusion

Ultimately, we come down to the same patterns I want to stress throughout the book. Write lots of datatypes that are easy to interpret, and then *interpret* them into some domain. Database access patterns are no different. Attempting to do everything in textual SQL quickly falls apart. Attempting to rely on opaque functional encodings can be limiting and difficult. Simple datatypes almost always win in the long run.

V Advanced Haskell

Throughout the book, I've cautioned against using some advanced Haskell techniques. That doesn't mean they're always a bad choice. It is my experience that they often don't pull their weight, especially in teams that have to work quickly.

The best way to learn *why* I think that is to learn how to use the tools themselves! So let's explore some of the advanced Haskell techniques, figure out how they work, and try to identify their pitfalls.

19. Template Haskell Is Not Scary

19.1 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:

```
1  class PermissionExample
2    include Squirrell
3
4    requires :user_id
5    permits :post_id
6
7    def raw_sql
8      <<SQL
9  SELECT *
10 FROM users
11   INNER JOIN posts ON users.id = posts.user_id
12 WHERE users.id = #{user_id} #{has_post?}
13 SQL
14   end
15
```

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

```
16    def has_post?
17      post_id ? "AND posts.id = #{post_id}" : ""
18    end
19  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:

```
1  def define_readers(args)
2    args.each do |arg|
3      attr_reader arg
4    end
5  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:

```
1  def initialize(args = {})
2    return self if args.empty?
3
4    Squirrell.requires[self.class].each do |k|
5
6      unless args.keys.include? k
7        fail MissingParameterError, "Missing required parameter: #{k}"
8      end
9
10     instance_variable_set "@#{k}", args.delete(k)
11   end
12
13   fail UnusedParameter, "Unspecified parameters: #{args}" if args.any?
14 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.

19.2 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!

19.3 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.

```
1  precompute :: [Int] -> DecsQ
2  precompute xs = do
3      -- .......
4      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.

```
1  precompute xs = do
2      let name = mkName "lookupTable"
3      -- ...
```

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:

```
1  lookupTable 0 = 123.546
2  lookupTable 12 = 151626.4234
3  lookupTable 42 = 0.0
4  -- ...
5  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

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

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

```
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 `Pats` into `Clause`, going from our `[Pat] -> [Clause]`. That will get use the `clauses` variable that we need. From above, a clause is defined like:

```
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.

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

We can get the `Bodys` for the `Clauses` 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.

```
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:

```
$(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:

```
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.

19.4 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:

```
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:

```
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.

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

The general formula that we're going for is:

```
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.

```
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
```

```
19            (VarT b_1627432398)
20          )
21          (AppT
22            (AppT
23              ArrowT
24                (AppT
25                  (ConT Web.Users.Types.UserId)
26                  (VarT b_1627432398)
27                )
28            )
29            (AppT
30              (ConT GHC.Types.IO)
31              (AppT
32                (ConT GHC.Base.Maybe)
33                (AppT
34                  (ConT Web.Users.Types.User)
35                  (VarT a_1627482920)
36                )
37              )
38            )
39          )
40        )
41      )
42    )
43    Web.Users.Types.UserStorageBackend
44    (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`.

```
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:

```
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.

20. Basic Type Level Programming

Dependently typed programming is becoming all the rage these days. Advocates are talking about all the neat stuff you can do by putting more and more information into the type system. It's true! Type level programming gives you interesting new tools for designing software. You can guarantee safety properties, and in some cases, even gain performance optimizations through the use of these types.

I'm not going to try and sell you on these benefits – presumably you've read about something like the dependently typed neural networks[1], or about Idris's encoding of network protocols in the type system[2]. If you're not convinced on the benefits of learning type level programming, then you can skip this chapter.

For this text to work as expected, you may need to `import Data.Kind (Type)`. You may also see `Type` instead of `Type` in GHCi. `Type` is an old deprecated name for `Type`.

20.1 The Basic Types

So let's talk about some basic types. I'm going to stick with the *real* basic types here: no primitives, just stuff we can define in one line in Haskell.

```
1  data Unit = MkUnit
2
3  data Bool = False | True
```

This code block defines two new types: `Unit` and `Bool`. The `Unit` type has one *constructor*, called `MkUnit`. Since there's only one constructor for this type, and it takes no parameters, there is only one value of this type. We call it `Unit` because there's only one value.

[1] https://blog.jle.im/entry/practical-dependent-types-in-haskell-1.html
[2] http://docs.idris-lang.org/en/latest/st/examples.html

`Bool` is a type that has two *constructors*: `True` and `False`. These don't take any parameters either, so they're kind of like constants.

What does it mean to be a type? A type is a way of classifying things. Things – that's a vague word. What do I mean by 'things'?

Well, for these simple types above, we've already seen all their possible values – we can say that `True` and `False` are members of the type `Bool`. Furthermore, `1`, `'a'`, and `Unit` are *not* members of the type `Bool`.

These types are kind of boring. Let's look at another type:

```
1  data IntAndChar = MkIntAndChar Int Char
```

This introduces a type `IntAndChar` with a single *constructor* that takes two arguments: one of which is an `Int` and the other is a `Char`. Values of this type look like:

```
1  theFirstOne = MkIntAndChar 3 'a'
2  theSecond   = MkIntAndChar (-3) 'b'
```

`MkIntAndChar` looks a lot like a function. In fact, if we ask GHCi about it's type, we get this back:

```
1  λ> :t MkIntAndChar
2  MkIntAndChar :: Int -> Char -> IntAndChar
```

`MkIntAndChar` is a function accepting an `Int` and a `Char` and finally yielding a value of type `IntAndChar`.

So we can construct values, and values have types. Can we construct types? And if so, what do they have?

20.2 The Higher Kinds

Let's hold onto our intuition about functions and values. A function with the type `foo :: Int -> IntAndChar` is saying:

> Give me a value with the type `Int`, and I will give you a value with the type `IntAndChar`.

Now, let's lift that intuition into the type level. A *value* constructor accepts a *value* and yields a *value*. So a *type* constructor accepts a *type* and yields a *type*. Haskell's type variables allow us to express this.

Let's consider everyone's favorite sum type:

```
data Maybe a
    = Nothing
    | Just a
```

Here, we declare a *type* `Maybe`, with two data constructors: `Just`, which accepts a value of type `a`, and `Nothing`, which does not accept any values at all. Let's ask GHCi about the type of `Just` and `Nothing`!

```
λ> :t Just
Just :: a -> Maybe a
λ> :t Nothing
Nothing :: Maybe a
```

So `Just` has that function type – and it looks like, whatever type of value we give it, it becomes a `Maybe` of that type. `Nothing`, however, can conjure up whatever type it wants, without needing a value at all. Let's play with that a bit:

```
λ> let nothingA = Nothing :: Maybe a
λ> let nothingInt = Nothing :: Maybe Int
λ> let nothingChar = Nothing :: Maybe Char
λ> nothingInt == nothingChar
```

```
<interactive>:27:15: error:
    • Couldn't match type ‘Char’ with ‘Int’
      Expected type: Maybe Int
        Actual type: Maybe Char
    • In the second argument of ‘(==)’, namely ‘nothingChar’
      In the expression: nothingInt == nothingChar
      In an equation for ‘it’: it = nothingInt == nothingChar
λ> nothingA == nothingInt
True
λ> nothingA == nothingChar
True
```

Woah – we get a type error when trying to compare `nothingInt` with `nothingChar`. That makes sense – `(==)` only works on values that have the same type. But then, wait, why does `nothingA` not complain when compared with `nothingInt` and `nothingChar`?

The reason is that `nothingA :: Maybe a` *really* means:

> I am a value of `Maybe a`, for any and all types `a` that you might provide to me.

So I'm seeing that we're passing types to `Maybe`, in much the same way that we pass values to `MkIntAndChar`. Let's ask GHCi about the type of `Maybe`!

```
λ> :type Maybe

<interactive>:1:1: error:
    • Data constructor not in scope: Maybe
    • Perhaps you meant variable ‘maybe’ (imported from Prelude)
```

Well, it turns out that types don't have types (kind of, sort of). Types have *kinds*. We can ask GHCi about the *kind* of types with the `:kind` command:

```
λ> :kind Maybe
Maybe :: Type -> Type
```

Type is the *kind* of types which have *values*. `Maybe` has the *kind* `Type -> Type`, which means:

> Give me a type that has values, and I will give you a type that has values.

Maybe you've heard about *higher kinded polymorphism* before. Let's write a data type that demonstrates that this means:

```
data HigherKinded f a
    = Bare a
    | Wrapped (f a)
```

Haskell's kind inference is awfully kind on the fingers – since we *use* `f` applied to `a`, Haskell just knows that `f` must have the kind `Type -> Type`. If we ask GHCi about the *kind* of `HigherKinded`, we get back:

```
λ> :kind HigherKinded
HigherKinded :: (Type -> Type) -> Type -> Type
```

So `HigherKinded` is a type that accepts a *type* of *kind* `Type -> Type`, and a *type* of *kind* `Type` and returns a type of kind `Type`. In plain, verbose English, this reads as:

> Give me two types: the first of which is a function that does not have values itself, but when given a type that *does* have values, it can have values. The second being a type that has values. Finally, I will return to you a type that can have ordinary values.

One last confusion: in recent GHC, you can ask "What is the kind of Type?" The answer is a bit mysterious:

```
λ> :kind Type
Type :: Type
```

Type has the kind Type.

20.3 Dynamically Kinded Programming

Haskell is currently transitioning to using Type. Previously, you had to write * to mean the same thing. * reminds me of regular expressions – "match anything." Indeed, * matches *any* type that has values, or even types that are only inhabited by the infinite loop:

```
λ> data Void
λ> :kind Void
Void :: Type
```

We don't provide any ways to construct a void value, yet it still has kind Type.

In the same way that you can productively program at the value level with dynamic *types*, you can productively program at the *type* level with *dynamic kinds*. And Type is basically that!

Let's encode our first type level numbers. We'll start with the Peano natural numbers, where numbers are inductively defined as either `Zero` or the `Successor` of some natural number.

```
data Zero
data Succ a

type One = Succ Zero
type Two = Succ One
type Three = Succ Two
type Four = Succ (Succ (Succ (Succ Zero)))
```

But this is pretty unsatisfying. After all, there's nothing that stops us from saying `Succ Bool`, which doesn't make any sense. I'm pretty sold

on the benefits of types for clarifying thinking and preventing errors, so abandoning the safety of types when I program my types just seems silly. In order to get that safety back, we need to introduce more kinds than merely `Type`. For this, we have to level up our GHC.

20.4 Data Kinds

```
{-# LANGUAGE DataKinds #-}
```

The `DataKinds` extension allows us to *promote* data constructors into type constructors, which also promotes their *type* constructors into *kind* constructors. To promote something up a level, we prefix the name with an apostrophe, or tick: `'`.

Now, let's define our *kind safe* type level numbers:

```
data Nat = Zero | Succ Nat
```

In plain Haskell, this definition introduces a new type `Nat` with two *value* constructors, `Zero` and `Succ` (which takes a value of *type* `Nat`). The `DataKinds` extension also allows us to use the data constructors as type constructors. This means that we get two new types: a type constant `'Zero`, which has the *kind* `Nat`, and a type *constructor* `'Succ`, which accepts a *type* of *kind* `Nat`. Notably, the *type* `Nat` and the *kind* `Nat` are the same. Let's ask GHCi about our new buddies:

```
λ> :kind 'Zero
'Zero :: Nat
λ> :kind 'Succ
'Succ :: Nat -> Nat
```

You might think: that looks familiar! And it should. After all, the *types* look very much the same!

```
λ> :type Zero
Zero :: Nat
λ> :type Succ
Succ :: Nat -> Nat
```

Where it can be ambiguous, the `'` is used to disambiguate. Otherwise, Haskell can infer which you mean.

It's important to note that there are *no values* of type `'Zero`. The only *kind* that can have *types* that can have *values* is `Type`.

We've gained the ability to construct some pretty basic types and kinds. In order to actually use them, though, we need a bit more power.

20.5 GADTs

```
{-# LANGUAGE GADTs #-}
```

You may be wondering, "What does GADT stand for?" Richard Eisenberg will tell you that they're Generalized Algebraic Data Types, but that the terminology isn't helpful, so just think of them as Gadts[3].

GADTs are a tool we can use to provide extra type information by matching on constructors. They use a slightly different syntax than normal Haskell data types. Let's check out some simpler types that we'll write with this syntax:

```
data Maybe a where
    Just :: a -> Maybe a
    Nothing :: Maybe a
```

The GADT syntax lists the constructors line-by-line, and instead of providing the *fields* of the constructor, we provide the *type signature* of the constructor. This is an interesting change – I just wrote out `a -> Maybe a`. That suggests, to me, that I can make these whatever type I want.

[3] https://www.youtube.com/watch?v=6snteFntvjM

```
data IntBool a where
    Int :: Int -> IntBool Int
    Bool :: Bool -> IntBool Bool
```

This declaration creates a new type `IntBool`, which has the *kind* `Type -> Type`. It has two constructors: `Int`, which has the type `Int -> IntBool Int`, and `Bool`, which has the type `Bool -> IntBool Bool`.

Since the constructors carry information about the resulting type, we get bonus information about the type when we pattern match on the constructors! Check this signature out:

```
extractIntBool :: IntBool a -> a
extractIntBool (Int _)  = 0
extractIntBool (Bool b) = b
```

Something *really* interesting is happening here! When we match on `Int`, we know that `IntBool a` $\sim$ `IntBool Int`. That $\sim$ tilde is a symbol for *type equality*, and introduces a *constraint* that GHC needs to solve to type check the code. For this branch, we know that `a` $\sim$ `Int`, so we can return an `Int` value.

We now have enough power in our toolbox to implement everyone's favorite example of dependent types: length indexed vectors!

20.6 Vectors

Length indexed vectors allow us to put the length of a list into the type system, which allows us to *statically* forbid out-of-bounds errors. We have a way to *promote* numbers into the type level using `DataKinds`, and we have a way to provide bonus type information using `GADTs`. Let's combine these two powers for this task.

I'll split this definition up into multiple blocks, so I can walk through it easily.

```
data Vector (n :: Nat) a where
```

We're defining a *type* `Vector` with *kind* `Nat -> Type -> Type`. The first type parameter is the length index. The second type parameter is the type of values contained in the vector. Note that, in order to compile something with a *kind* signature, we need...

```
{-# LANGUAGE KindSignatures #-}
```

Thinking about types often requires us to think in a logical manner. We often need to consider things inductively when constructing them, and recursively when destructing them. What is the base case for a vector? It's the empty vector, with a length of `Zero`.

```
    VNil :: Vector 'Zero a
```

A value constructed by `VNil` can have any type `a`, but the length is always constrained to be `'Zero`.

The inductive case is adding another value to a vector. One more value means one more length.

```
    VCons :: a -> Vector n a -> Vector ('Succ n) a
```

The `VCons` constructor takes two values: one of type `a`, and another of type `Vector n a`. We don't know how long the `Vector` provided is – it can be any `n` such that `n` is a `Nat`ural number. We *do* know that the *resulting* vector is the `Succ`essor of that `n`umber, though.

So here's the fully annotated and explicit definition:

```
data Vector (n :: Nat) (a :: Type) where
    VNil :: Vector 'Zero a
    VCons :: a -> Vector n a -> Vector ('Succ n) a
```

Fortunately, Haskell can infer these things for us! Whether you use the above explicit definition or the below implicit definition is a matter of taste, aesthetics, style, and documentation.

```
data Vector n a where
    VNil :: Vector Zero a
    VCons :: a -> Vector n a -> Vector (Succ n) a
```

Let's now write a Show instance for these length indexed vectors. It's pretty painless:

```
instance Show a => Show (Vector n a) where
    show VNil         = "VNil"
    show (VCons a as) = "VCons " ++ show a ++ " (" ++ show as ++ ")"
```

That n type parameter is totally arbitrary, so we don't have to worry about it too much.

The Vector API

As a nice exercise, let's write `append :: Vector n a -> Vector m a -> Vector ??? a`. But wait, what is `???` going to be? It needs to represent the *addition* of these two natural numbers. Addition is a *function*. And we don't have type functions, right? Well, we do, but we have to upgrade our GHC again.

```
{-# LANGUAGE TypeFamilies #-}
```

For some reason, functions that operate on types are called type families. There are two ways to write a type family: open, where anyone can add new cases, and closed, where all the cases are defined at once. We'll mostly be dealing with closed type families here.

So let's figure out how to add two Natural numbers, at the type level. For starters, let's figure out how to add at at the value level first.

```
add :: Nat -> Nat -> Nat
```

This is the standard Haskell function definitions we all know and love. We can pattern match on values, write `where` clauses with helpers, etc.

We're working with an inductive definition of numbers, so we'll need to use recursion get our answer. We need a base case, and then the inductive case. So lets start basic: if we add 0 to any number, then the answer is that number.

```
1 add Zero n = n
```

The inductive case asks:

> If we add the successor of a number (`Succ n`) to another number (`m`), what is the answer?

Well, we know we want to get to `Zero`, so we want to somehow *shrink* our problem a bit. We'll have to shift that `Succ` from the left term to the right term. Then we can recurse on the addition.

```
1 add (Succ n) m = add n (Succ m)
```

If you imagine a natural number as a stack of plates, we can visualize the *addition* of two natural numbers as taking one plate off the top of the first stack, and putting it on top of the second. Eventually, we'll use all of the plates – this leaves us with `Zero` plates, and our final single stack of plates is the answer.

```
1 add :: Nat -> Nat -> Nat
2 add Zero     n = n
3 add (Succ n) m = add n (Succ m)
```

Alright, let's promote this to the type level.

20.7 Type Families

The first line of a type family definition is the signature:

```
type family Add n m where
```

This introduces a new *type function* `Add` which accepts two parameters. We can now define the individual cases. We can pattern match on type constructors, just like we can pattern match on value constructors. So we'll write the `Zero` case:

```
    Add 'Zero n = n
```

Next, we recurse on the inductive case:

```
    Add ('Succ n) m = Add n ('Succ m)
```

Ahh, except now GHC is going to give us an error.

```
    • The type family application ‘Add n ('Succ m)’
        is no smaller than the instance head
      (Use UndecidableInstances to permit this)
    • In the equations for closed type family ‘Add’
      In the type family declaration for ‘Add’
```

GHC is extremely scared of undecidability, and won't do *anything* that it can't easily figure out on it's own. `UndecidableInstances` is an extension which allows you to say:

> Look, GHC, it's okay. I know you can't figure this out. I promise this makes sense and will eventually terminate.

So now we get to add:

```
{-# LANGUAGE UndecidableInstances #-}
```

to our file. The type family definition compiles fine now. How can we test it out?

Where we used `:kind` to inspect the kind of types, we can use `:kind!` to *evaluate* these types as far as GHC can. This snippet illustrates the difference:

```
λ> :kind Add (Succ (Succ Zero)) (Succ Zero)
Add (Succ (Succ Zero)) (Succ Zero) :: Nat
λ> :kind! Add (Succ (Succ Zero)) (Succ Zero)
Add (Succ (Succ Zero)) (Succ Zero) :: Nat
= 'Succ ('Succ ('Succ 'Zero))
```

The first line just tells us that the result of Adding two Natural numbers is itself a Natural number. The second line shows the actual result of evaluating the type level function. Cool! So now we can finally finish writing append.

```
append :: Vector n a -> Vector m a -> Vector (Add n m) a
```

Let's start with some bad attempts, to see what the types buy us:

```
append VNil rest = VNil
```

This fails with a type error – cool!

```
    • Could not deduce: m ~ 'Zero
      from the context: n ~ 'Zero
        bound by a pattern with constructor:
                   VNil :: forall a. Vector 'Zero a,
                 in an equation for ‘append’
        at /home/matt/Projects/dep-types.hs:31:8-11
      ‘m’ is a rigid type variable bound by
        the type signature for:
          append :: forall (n :: Nat) a (m :: Nat).
                    Vector n a -> Vector m a -> Vector (Add n m) a
        at /home/matt/Projects/dep-types.hs:30:11
      Expected type: Vector (Add n m) a
        Actual type: Vector 'Zero a
    • In the expression: VNil
      In an equation for ‘append’: append VNil rest = VNil
    • Relevant bindings include
        rest :: Vector m a
```

```
18            (bound at /home/matt/Projects/dep-types.hs:31:13)
19          append :: Vector n a -> Vector m a -> Vector (Add n m) a
20            (bound at /home/matt/Projects/dep-types.hs:31:1)
```

The error is kinda big and scary at first. Let's dig into it a bit.

GHC is telling us that it can't infer that `m` (which is the length of the second parameter vector) is equal to `Zero`. It knows that `n` (the length of the first parameter) is `Zero` because we've pattern matched on `VNil`. So, what values *can* we return? Let's replace the definition we have thus far with `undefined`, reload in GHCi, and inspect some types:

```
λ> :t append VNil
append VNil :: Vector m a -> Vector m a
```

We need to construct a value `Vector m a`, and we have been given a value `Vector m a`. BUT – we don't know what `m` is! So we have no way to spoof this or fake it. We have to return our input. So our first case is simply:

```
append VNil xs = xs
```

Like with addition of natural numbers, we'll need to have the inductive case. Since we have the base case on our first parameter, we'll want to try shrinking our first parameter in the recursive call.

So let's try another bad implementation:

```
append (VCons a rest) xs = append rest (VCons a xs)
```

This doesn't really do what we want, which we can verify in the REPL:

```
λ> append (VCons 1 (VCons 3 VNil)) (VCons 2 VNil)
VCons 3 (VCons 1 (VCons 2 (VNil)))
```

The answer *should* be `VCons 1 (VCons 3 (VCons 2 VNil))`. However, our Vector type only encodes the *length* of the vector in the type. The sequence is not considered. Anything that isn't lifted into the type system doesn't get any correctness guarantees.

So let's fix the implementation:

```
append (VCons a rest) xs = VCons a (append rest xs)
```

And let's reload in GHCi to test it out!

```
λ> :reload
[1 of 1] Compiling DepTypes         ( /dep-types.hs, interpreted )

/home/matt/Projects/dep-types.hs:32:28: error:
    • Could not deduce: Add n1 ('Succ m) ~ 'Succ (Add n1 m)
      from the context: n ~ 'Succ n1
        bound by a pattern with constructor:
                   VCons :: forall a (n :: Nat).
                            a -> Vector n a -> Vector ('Succ n) a,
                 in an equation for ‘append’
        at /home/matt/Projects/dep-types.hs:32:9-20
      Expected type: Vector (Add n m) a
        Actual type: Vector ('Succ (Add n1 m)) a
    • In the expression: VCons a (append rest xs)
      In an equation for ‘append’:
          append (VCons a rest) xs = VCons a (append rest xs)
    • Relevant bindings include
        xs :: Vector m a (bound at /dep-types.hs:32:23)
        rest :: Vector n1 a
          (bound at /dep-types.hs:32:17)
        append :: Vector n a -> Vector m a -> Vector (Add n m) a
          (bound at /dep-types.hs:31:1)
Failed, modules loaded: none.
```

Oh

no! A type error! GHC can't figure out that `Add n (Succ m)` is the same as `Succ (Add n m)`. We can kinda see what went wrong if we lay the `Vector`, `Add` and `append` definitions next to each other:

```
data Vector n a where
    VNil :: Vector Zero a
    VCons :: a -> Vector n a -> Vector (Succ n) a

type family Add x y where
    Add 'Zero n = n
    Add ('Succ n) m = Add n ('Succ m)

append :: Vector n a -> Vector m a -> Vector (Add n m) a
append VNil xs           = xs
append (VCons a rest) xs = VCons a (append rest xs)
```

In `Vector`'s inductive case, we are building up a bunch of `Succ`s. In `Add`'s recursive case, we're tearing down the left hand side, such that the exterior is another `Add`. And in `append`s recursive case, we're building up the right hand side. Let's trace how this error happens, and supply some type annotations as well:

```
append (VCons a rest) xs =
```

Here, we know that `VCons a rest` has the type `Vector (Succ n) a`, and `xs` has the type `Vector m a`. We need to produce a result of type `Vector (Add (Succ n) m) a` in order for the type to line up right. We use `VCons a (append rest xs)`. `VCons` has a length value that is the Successor of the result of `append rest xs`, which should have the value `Add n m`, so the length there is `Succ (Add n m)`. Unfortunately, our result type needs to be `Add (Succ n) m`.

We know these values are equivalent. Unfortuantely, GHC cannot prove this, so it throws up it's hands. Two definitions, which are provably equivalent, are *structurally* different, and this causes the types and proofs to fail. This is a HUGE gotcha in type level programming – the implementation details matter, a lot, and they leak, *hard*. We can fix this by using a slightly different definition of `Add`:

```
type family Add x y where
    Add 'Zero n = n
    Add ('Succ n) m = 'Succ (Add n m)
```

This definition has a similar structure of recursion – we pull the Succs out, which allows us to match the way that VCons adds Succ on top.

This new definition compiles and works fine.

20.8 This Sucks

Agreed, which is why I'll defer the interested reader to this much better tutorial[4] on length indexed vectors in Haskell. Instead, let's look at some other more interesting and practical examples of type level programming.

20.9 Heterogeneous Lists

Heterogeneous lists are kind of like tuples, but they're defined inductively. We keep a type level list of the contents of the heterogeneous list, which let us operate safely on them.

To use ordinary Haskell lists at the type level, we need another extension:

```
{-# LANGUAGE TypeOperators #-}
```

which allows us to use operators at the type level.

Here's the data type definition:

[4] https://www.schoolofhaskell.com/user/konn/prove-your-haskell-for-great-safety/dependent-types-in-haskell

```
data HList xs where
    HNil :: HList '[]
    (:::) :: a -> HList as -> HList (a ': as)

infixr 6 :::
```

The `HNil` constructor has an empty list of values, which makes sense, because it doesn't have any values! The `:::` construction operator takes a value of type `a`, an `HList` that already has a list of types `as` that it contains, and returns an `HList` where the first element in the type level list is `a` followed by `as`.

Let's see what a value for this looks like:

```
λ> :t 'a' ::: 1 ::: "hello" ::: HNil
'a' ::: 1 ::: "hello" ::: HNil
    :: HList '[Char, Int, String]
```

So now we know that we have a `Char`, `Int`, and `String` contained in this `HList`, and their respective indexes. What if we want to `Show` that?

```
λ> 'a' ::: 1 ::: "hello" ::: HNil

<interactive>:13:1:
    No instance for (Show (HList '[Char, Int, String]))
      arising from a use of ‘print’
    In the first argument of ‘print’, namely ‘it’
    In a stmt of an interactive GHCi command: print it
```

Hmm. We'll need to write a `Show` instance for `HList`. How should we approach this? Let's try something dumb first. We'll ignore all the contents!

```
instance Show (HList xs) where
    show HNil         = "HNil"
    show (x ::: rest) = "_ ::: " ++ show rest
```

Ahah! this compiles, and it even works!

```
λ> 'a' ::: 1 ::: "hello" ::: HNil
_ ::: _ ::: _ ::: HNil
```

Unfortunately, it's not useful. Can we do better? We can!

20.10 Inductive Type Class Instances

First, we'll define the base case – showing an empty HList!

```
instance Show (HList '[]) where
    show HNil = "HNil"
```

This causes a compile error, requiring that we enable yet another language extension:

```
{-# LANGUAGE FlexibleInstances #-}
```

If you're doing this in another file than the type family above, you'll also get an error about `FlexibleContexts`. It turns out that enabling `UndecidableInstances` implies `FlexibleContexts` for some reason. So let's throw that one on too, for good measure:

```
{-# LANGUAGE FlexibleContexts #-}
```

This compiles, and we can finally `show HNil` and it works out. Now, we must recurse!

The principle of induction states that:

1. We must be able to do something for the base case.
2. If we can do something for a random case, then we can do it for a case that is one step larger.
3. By 1 and 2, you can do it for all cases.

We've covered that base case. We'll assume that we can handle the smaller cases, and demonstrate how to handle a slightly large case:

```
instance (Show (HList as), Show a)
    => Show (HList (a ': as)) where
    show (a ::: rest) =
        show a ++ " ::: " ++ show rest
```

This instance basically says:

> Given that I know how to Show an `HList` of `as`, and I know how to Show an `a`: I can Show an `HList` with an `a` and a bunch of `as`.

```
λ> 'a' ::: 1 ::: "hello" ::: HNil
'a' ::: 1 ::: "hello" ::: HNil
```

Further Exercises

Write an aeson[5] instance for `HList`. It'll be similar to the Show instance, but require a bit more stuff.

20.11 Extensible Records

There are a few variants on extensible records in Haskell. Here's a tiny implementation that requires yet more extensions:

```
{-# LANGUAGE PolyKinds        #-}
{-# LANGUAGE TypeApplications #-}

import GHC.TypeLits (KnownSymbol, symbolVal)
import Data.Proxy
```

Before we jump to the implementation: a small warning.

[5] https://hackage.haskell.org/package/aeson

An Aside; Beware PolyKinds

It had just snowed the night before. The sun was beaming, glittering on the ice covered asphalt. A coworker was extending some of our internal Servant machinery. The types were as fancy as my coffee was warm. The error messages were as informative as the white snow blanketing the landscape.

GHC was failing to match two types.

```
1  Can't match type s with s
```

The types are the same. I was literally the "It's the same picture" meme for twenty minutes while staring at it.

It turns out, when PolyKinds are enabled, GHC will fail to match types if the *kinds* don't match, too. This makes perfect sense. The two `s` types above were inferred to have different kinds: `(s :: Type)` and `(s :: k0)`. As a result, they didn't match, and the kind information didn't make it into the error message.

`PolyKinds` will give odd error messages in unexpected locations. It is extremely convenient to use, but I usually find that I prefer to write explicit `KindSignatures` unless I genuinely need kind polymorphism.

Back to your regularly scheduled programming...

This generalizes definitions for type variables, which allows for non-value type variables to have kind polymorphism. Type applications allow us to explicitly pass types as arguments using an `@` symbol.

First, we must define the type of our fields, and then our record:

```
1  newtype s >> a = Named a
2
3  data HRec xs where
4      HEmpty :: HRec '[]
5      HCons :: (s >> a) -> HRec xs -> HRec (s >> a ': xs)
```

The `s` parameter is going to be a type with the *kind* `Symbol`. `Symbol` is defined in `GHC.TypeLits`, so we need that import to do the fun stuff.

We'll construct a value using the `TypeApplications` syntax, so a record will look like:

```
λ> HCons (Named @"foo" 'a') (HCons (Named @"bar" (3 :: Int)) HEmpty)

<interactive>:10:1: error:
    • No instance for (Show (HRec '["foo" >> Char, "bar" >> Int]))
        arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

So, this type checks fine! Cool. But it does not Show, so we need to define a Show instance.

Those string record fields only exist at the type level – but we can use the `KnownSymbol` class to bring them back down to the value level using `symbolVal`.

Here's our base case:

```
instance Show (HRec '[]) where
    show _ = "HEmpty"
```

And, when we recurse, we need a tiny bit more information. Let's start with the instance head first, so we know what variables we need:

```
instance Show (HRec (s >> a ': xs)) where
```

OK, so we have a `s` type, which has the kind `Symbol`, an `a :: Type`, and `xs`. So now we pattern match on that bad boy:

```
instance Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
```

OK, so we need to `show` the `a` value. Easy. Which means we need a `Show a` constraint tacked onto our instance:

```
instance (Show a)
    => Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
        let val = show a
```

Next up, we need the key as a string. Which means we need to use `symbolVal`, which takes a `proxy s` and returns the `String` associated with the `s` provided that `s` is a `KnownSymbol`.

```
instance (Show a, KnownSymbol s)
    => Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
        let val = show a
            key = symbolVal (Proxy :: Proxy s)
```

At this point, you're probably going to get an error like `No instance for 'KnownSymbol s0'`. This is because Haskell's type variables have a limited scope by default. When you write:

```
topLevelFunction :: a -> (a -> b) -> b
topLevelFunction a = go
  where
    go :: (a -> b) -> b
    go f = f a
```

Haskell interprets each type signature as it's own *scope* for the type variables. This means that the `a` and `b` variables in the `go` helper function are different type variables, and a more precise way to write it would be:

```
topLevelFunction :: a0 -> (a0 -> b0) -> b0
topLevelFunction a = go
  where
    go :: (a1 -> b1) -> b1
    go f = f a
```

If we want for type variables to have a scope similar to other variables, we need another extension:

```
{-# LANGUAGE ScopedTypeVariables #-}
```

Finally, we need to show the rest of the stuff!

```
instance (Show a, KnownSymbol s, Show (HRec xs))
    => Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
        let val = show a
            key = symbolVal (Proxy :: Proxy s)
            more = show rest
         in "(" ++ key ++ ": " ++ val ++ ") " ++ more
```

This gives us a rather satisfying Show instance, now:

```
λ> HCons (Named @"foo" 'a') (HCons (Named @"bar" (3 :: Int)) HEmpty)
(foo: 'a') (bar: 3) HEmpty
```

Exercise:

Write an Aeson ToJSON instance for this HRec type which converts into a JSON object.

Bonus points: Write an Aeson FromJSON instance for the HRec type.

20.12 Like what you read?

If you enjoyed this chapter, you should check out the book Thinking with Types[6] by Sandy Maguire. It is an extensive manual on practical type-level programming in Haskell.

[6] http://thinkingwithtypes.com/

21. Family Values

Haskell's `TypeFamilies` language extension enables four different kinds of things:

- Closed type families
- Open type families
- Associated type families
- Data families

And, as a bonus, GHC 8.0.1 introduced `TypeFamilyDependencies`, allowing us to regain a bit of the expressiveness of a `FunctionalDependency` on a type class.

We're using the world "family" a lot. What are they? What's the difference? Why are they useful? When should you pick one over the other?

Briefly,

- Type families are a means of computing at the type level.
- Associated types and data families allow you to transmit type information to the value-level world.

Multiparameter type classes with functional dependencies are another way of doing type-level computation - for an in depth comparison, see "Trade-offs in Type Programming" in this book. For an introduction to type level programming, see the chapter "Basic Type Level Programming."

21.1 Type Families

Type families attempt to provide *functions* at the type level. Let's review some of the key details of functions - they take arguments of a certain

type, and return a value of a certain type. Type functions take arguments of a certain *kind* and return a *type* of a certain *kind*. So we're promoting all of our language up a level.

It is somewhat rare for these extensions to be used alone. The examples in this chapter will have the following extensions:

- `DataKinds` so we can operate on things like `Symbol` (type level strings) and other lifted types (ie `'True` which has the kind `Bool`)
- `KindSignatures`, which allow you to write kind signatures on type parameters.

Additionally, a lot of the output will refer to types of kind `*`, which is the kind of types that have ordinary runtime values. As an example, `Int` has kind `*`, which is written as `Int :: *`. GHC is transitioning away from using this notation, and will soon instead use `Type` for this. You'd write `Int :: Type`.

Because closed type families are the most similar to plain functions, it is useful to consider them first.

Closed Type Families

A closed type family is defined with the keywords `type family`, a type name, a list of arguments, and then `where`. After the `where`, you provide a complete definition of each case.

```
1  type family PickType a where
2      PickType Int = Char
3      PickType Char = Int
```

We can use `PickType` at the type level, so we could write a value-level function that changes the type associated with a `Proxy`:

```
1  pickType :: Proxy a -> Proxy (PickType a)
2  pickType _ = Proxy
```

Then, we can evaluate this in GHCi.

```
λ> pickType (Proxy :: Proxy Int)
Proxy
λ> Data.Typeable.typeOf $ pickType (Proxy :: Proxy Int)
Proxy * Char
```

Note that we need to use `Data.Typeable.typeOf` to show the *type* of the `Proxy`, since the `Show` instance prints out `"Proxy"`.

If we look at the *kind* of `PickType`, we'll see that it is rather incomplete.

```
λ> :kind PickType
PickType :: * -> *
```

We accept a type of kind `*` (or `Type`), and we return a type with the same kind. But we only give definitions for `Int` and `Char`, and we do so by *pattern matching* on those type constructors. What happens if we ask for a type that we didn't provide?

```
-- `Int -> Char` is defined
λ> :t pickType (Proxy :: Proxy Int)
pickType (Proxy :: Proxy Int) :: Proxy Char
-- `Char -> Int` is defined
λ> :t pickType (Proxy :: Proxy Char)
pickType (Proxy :: Proxy Char) :: Proxy Int

-- But `PickType String` is not defined!
λ> :t pickType (Proxy :: Proxy String)
pickType (Proxy :: Proxy String) :: Proxy (PickType String)
```

The result is *totally fine* as a value - it's `Proxy (PickType String)`. We can even `show` it. But we get a rather odd error if we try to do the `typeOf` trick:

```
λ> pickType (Proxy :: Proxy String)
Proxy
λ> typeOf $ pickType (Proxy :: Proxy String)

\<interactive\>:15:1: error:
    • No instance for (Typeable (PickType String))
        arising from a use of ‘typeOf’
    • In the expression: typeOf $ pickType (Proxy :: Proxy String)
      In an equation for ‘it’:
          it = typeOf $ pickType (Proxy :: Proxy String)
```

GHC is complaining about not being able to find a `Typeable` instance for the `PickType String` type. We're observing a *stuck type family*. In value-level Haskell, an `undefined` value throws a runtime exception. But in type-level Haskell, an `undefined` type simply stops evaluating.

So, for this reason, it's *usually* good for your *closed* type families to avoid accepting arguments of kind `*` or `Type`, unless they're treating them totally polymorphically. For a polymorphic example, here's a type family that computes the ultimate return type of a function type.

```
type family KnowResult a where
    KnowResult (i -> o) = KnowResult o
    KnowResult a = a
```

This type family pattern matches on a function arrow and recurses on the function arrow. Because we can kind of imagine that the kind `Type` has this definition:

```
data Type
    = Constructor String
    | Type -> Type
    | Application Type Type
```

It's generally "safe" to pattern match on the `->` or `Application` constructors, but pattern matching on specific type constructors can get things Stuck.

A better use for a closed type family would be something like computing the length of a type level list.

```
type family Length (xs :: '[k]) :: Nat where
    Length '[] = 0
    Length (_ ': rest) = Length rest + 1
```

Open Type Families

An open type family is defined much like a closed type family, without the `where`. Instead, cases are provided separately.

```
type family PickTypeOpen a

type instance PickTypeOpen Int = Char
type instance PickTypeOpen Char = Int
```

With this formulation, we can *at a later module* provide a `type instance PickType String` that would allow our contrived example to work.

Associated Type Families

An associated type family is an *open* type family that is associated with a type class. In order to do anything with a type at the value-level, you need a type class to bring things back to earth.

The `mono-traversable` library defines a `MonoFunctor` class which allows you to map over monomorphic containers like `ByteString` and `Text` with the same function as you would use on polymorphic containers, like `[a]`. With an associated type family, we'd write this class definition.

```
class MonoFunctor o where
    type Element o :: Type
    omap :: (Element o -> Element o) -> o -> o
```

Now, we can write our instances:

```haskell
instance MonoFunctor ByteString where
    type Element ByteString = Word8
    omap = Data.ByteString.map

instance MonoFunctor Text where
    type Element Text = Char
    omap = Data.Text.map

instance MonoFunctor [a] where
    type Element [a] = a
    omap = map
```

There's a natural extension: `MonoTraversable`, which also suggests a `MonoFoldable`. But `Traversable` depends *separately* on both `Foldable` and `Functor` - there are `Foldable` types which are not `Functor` (like `Data.Set`). So, which type class should we associate the type family `Element` with?

We have a few options:

1. Create a new type class that only has the `Element` associated type family
2. Create an open type family `Element`
3. `MonoFunctor` and `MonoFoldable` both define their own associated type, and `MonoTraversable` requires that they agree
4. Scrap type families altogether and use `MultiParamTypeClasses`

Indeed, this is the *primary* reason that you'd want to use an open type family - if there are multiple type classes that can sensibly use the type family, without the classes themselves being sensibly arranged in a hierarchy.

So the definition of `MonoFunctor` is *really* like this:

```haskell
type family Element mono

class MonoFunctor mono where
    omap :: (Element mono -> Element mono) -> mono -> mono

type instance Element ByteString = Word8

instance MonoFunctor ByteString where
    omap = Data.ByteString.map
```

Try and define `MonoTraversable` using the above three strategies, and then use the classes in some toy code.

- What has the best error messages?
- What is easiest to implement?
- I claim that the open type family is the best. Are there advantages to the other formulations?

21.2 Open or Closed Type Families?

The question: "Should I use an open type family or a closed type family?" has an analog to simpler language features: type classes and sum types.

If you want a **closed** set, you use a sum type. If you want an **open** set, you use a type class. So if you're familiar with the trade-offs there, the trade-offs with open/closed type families are easier to evaluate.

A closed set means you can be exhaustive - "every case is handled." If you're pattern matching on a datakind, like `type family Foo (x :: Bool)`, then you can know that handling `Foo 'True` and `Foo 'False` that you've handled all cases. You don't have to worry that some user is going to add a case and blow things up (unless they are being particularly tricky with stuckness).

An open set is a way of allowing easy extensibility. So you're going to accept something of kind Type or possibly a polymorphic kind variable

to allow people to define their own types, and their own instances of these types. For example, if I want to associate a type with a string, I can write:

```
type family TypeString (sym :: Symbol) :: Type

type instance TypeString "Int" = Int
type instance TypeString "Char" = Char
```

And that lets me run programs at the type level, that end users can extend. Much like you can write a type `class` and end users can extend your functionality.

21.3 The Bridge Between Worlds

Ultimately, you need to do **something** at the value level. Which means you need to take some type information and translate it to the value level. This is precisely what type classes do - they are morally "a function from a type to a value." We can write a super basic function, like:

```
typeStringProxy :: Proxy sym -> Proxy (TypeString sm)
typeStringProxy _ = Proxy
```

But this is **still** not useful without further classes. The Default class assigns a special value to any type, and we could do something like this:

```
typeStringDefault
    :: forall sm. Default (TypeString sm)
    => Proxy sm -> TypeString sm
typeStringDefault _ = def @(TypeString sm)
```

Since associating a type class and an open type family is so common, it's almost always better to use an associated type unless you know that the type family is going to be shared across multiple type classes.

"So how do you associate a closed type family with values?" That's a great question. We can do the same trick with Proxy functions:

```
type family Closed (a :: Bool) where
    Closed 'True = Int
    Closed 'False = Char

closed :: Proxy b -> Proxy (Closed b)
closed _ = Proxy
```

But, until we know what `b` is, we can't figure out what `Closed b` is. To pattern match on a type, we need a type class.

```
class RunClosed (b :: Bool) where
  runClosed :: Proxy b -> Closed b

instance RunClosed 'True where
    runClosed _ = 3

instance RunClosed 'False where
    runClosed _ = 'a'
```

This is an interesting turn of events. We have a *closed* type family, but we need an *open* type class in order to do anything useful with it, at the value level. We could collapse these two things together this with the following class:

```
class RunClosed (b :: Bool) where
    type Closed b :: Type
    runClosed :: Proxy b -> Closed b

instance RunClosed 'True where
    type Closed 'True = Int
    runClosed _ = 3

instance RunClosed 'False where
    type Closed 'False = Char
    runClosed _ = 'a'
```

But often you want to do more interesting things with types, and closed type families allow you to split out those interesting intermediate computations into separate reusable units.

21.4 Data Families

A data family is **like** a type family, but instead of allowing you to refer to **any** type, you have to specify the constructors inline. To take a simplified example from the `persistent` database library,

```
data family Key a

newtype instance Key User
    = UserKey { unUserKey :: UUID }

newtype instance Key Organization
    = OrganizationKey { unOrganizationKey :: UUID }
```

An advantage of this is that, since you specify the constructors, you can know the type of `Key a` by knowing the constructor in use - the **value** specifies the **type**. `OrganizationKey :: UUID -> Key Organization`. Likewise, because I can only have a single instance of `Key User`, I know that I can pattern match on the `UserKey` constructor.

It looks **a lot** like an "open type family," and in fact is completely analogous. But we don't call them "open data families," even though that's an appropriate name for it. It should make you wonder - is there such a thing as a **closed** data family?

The answer is "yes", but we call them `GADTs` instead.

The nice thing about an "open data family" is that you can learn about types by inspecting values - by knowing a value (like `OrganizationKey uuid`), I can work 'backwards' and learn that I have a `Key Organization`. But, I can't write a `case` expression over all `Key a` - it's open! - and case only works on closed things. So this code does not work:

```haskell
whatKey :: Key a -> Maybe UUID
whatKey k = case k of
  UserKey uuid -> Just uuid
  OrganizationKey uuid -> Just uuid
  _ -> Nothing
```

Indeed, we need a type class to allow us to write `get :: Key a -> SqlPersistT m (Maybe a)`.

A GADT - as a closed data family - allows us to work from a value to a type, and since it is exhaustive, we can write case expressions on them.

```haskell
data Key a where
    UserKey
        :: { unUserKey :: UUID }
        -> Key User
    OrganizationKey
        :: { unOrganizationKey :: UUID }
        -> Key Organization
```

If I have this structure, then I can actually write `get` without a type class.

```haskell
get :: Key a -> SqlPersistT IO (Maybe a)
get k = case k of
    UserKey uuid -> do
        [userName, userAge] <-
            rawSql
                "SELECT name, age FROM users WHERE id = ?"
                [toPersistValue uuid]
        pure User {..}
    OrganizationKey uuid -> do
        [organizationName, organizationPrimaryUser] <-
            rawSql (Text.unlines
                [ "SELECT name, primary user"
                , "FROM organizations"
                , "WHERE id = ?"
                ])
```

```
                [toPersistValue uuid]
        pure Organization {..}
```

A GADT is 'basically' a closed type family that gives you constructor tags for applying that type family. If we look at `Closed`, we can inline this:

```
type family ClosedTy (b :: Bool) where
    ClosedTy True = Int
    ClosedTy False = Char

data ClosedData (a :: Type) where
    ClosedData :: Proxy b -> ClosedData (ClosedTy b)

-- inlining:
data Closed (a :: Type) where
    ClosedTrue :: Proxy 'True -> Closed Int
    ClosedFalse :: Proxy 'False -> Closed Char
```

When we case on a `Closed` value, we get:

```
runClosed :: Closed a -> a
runClosed closed =
    case closed of
        ClosedTrue (Proxy :: Proxy 'True) -> 3
        ClosedFalse (Proxy :: Proxy 'False) -> 'a'
```

At this point, that we are dealing with an "input" of a `True` or `False` almost seems spurious. The `Proxy`'s type is duplicated in the name of the constructor, and we can't even polymorphically construct a `Closed` from an unknown `Proxy (b :: Bool)` without a type class.

```haskell
class ToClosed a b | a -> b where
    close :: Proxy a -> Closed b

instance ToClosed 'True Int where
    close _ = ClosedTrue

instance ToClosed 'False Char where
    close _ = ClosedFalse
```

There's a lot of juggling between type classes, type families, data families, and GADTs when you are doing type level programming in Haskell.

21.5 Conclusion

- Open type family + type class = extensible, open programming, but no exhaustivity.
- Closed type family + GADT + functions = exhaustive handling of types, but not extensible
- An open type family + a GADT isn't much fun.
- A closed type family + a type class isn't much fun

22. Trade-offs in Type Programming

Haskell supports two mostly equivalent means of programming at the type-level. The first uses type classes with multiple parameters and functional dependencies to relate them. The second uses type families.

Even more confusing, Haskell gives you two options for dealing with related types in type classes: functional dependencies and associated types.

This chapter will use ideas and concepts presented in the "Basic Type Level Programming" chapter, so if anything here is confusing, check that first.

22.1 MPTCs

`MPTC` is an initialism for `MultiParamTypeClasses`. This language extension allows you to have type classes with more than one parameter. We can use it to write a generic `Cast` class to allow conversion between datatypes.

```
1  class Cast from to where
2      cast :: from -> to
```

Unfortunately, it's not really as useful as you might want it to be. You'd want to define a generic instance for types that are the same:

```
1  instance Cast a a where
2      cast a = a
```

GHC immediately complains that we need the `FlexibleInstances` language extension, after which GHC is happy. We can then write a few instances for common types:

```
instance Cast Char Int where
    cast c = Data.Char.ord c

instance Cast Int Double where
    cast i = fromRational (toRational i)

instance Cast Char Double where
    cast c = cast (cast c :: Int)
```

As a natural pattern matcher, you note that the instance `Cast Char Double` should be generalizable. You put on your Polymorphism Hat and write the following code:

```
instance (Cast from middle, Cast middle to) => Cast from to where
    cast from = cast (cast from :: middle)
```

This gives us a bunch of errors. We can get GHC to stop complaining by enabling the extensions blindly (`UndecidableInstances` and then `AllowAmbiguousTypes`). GHC whines about `No instance for Cast from middle0`, and our Advanced Haskeller Intuition recognizes that errors like this usually means we need to turn on `ScopedTypeVariables`. Then we get a sea of entirely new errors - overlapping instance declarations?! What is going on?!

Well, this gets into how GHC does instance resolution for type classes. Type classes have the following shape:

```
instance (Context) => ClassName (Instance Head) where
    (implementation)
```

So when GHC sees a requirement for a type class instance, it looks at the instance head first. As soon as it finds an instance match, it stops. And *then* it checks the context. So our above instances *really* look like the following:

```
instance Cast a a
instance Cast Char Int
instance Cast Int Double
instance Cast Char Double
instance Cast from to
```

Turns out, `instance Cast from to` overlaps with *everything else* because it can unify with any of the above! Then GHC gets upset about the overlapping instances.

We can "fix" this by adding an `{-# OVERLAPPABLE #-}` pragma to that instance.

```
instance
    {-# OVERLAPPABLE #-}
    (Cast from middle, Cast middle to)
  => Cast from to
  where
    cast from = cast (cast from :: middle)
```

Now, finally, GHC does not complain. We delete our 'manual' instance of `Cast Char Double`, and then we go to use it:

```
blargh :: Char -> Double
blargh = cast
```

GHC is not happy about this. GHC is never happy about this.

```
/examples/src/Fundeps.hs:27:10: error:
  • Overlapping instances for Cast middle0 Double
      arising from a use of ‘cast’
    Matching instances:
      instance [overlappable] (Cast from middle, Cast middle to) =>
                              Cast from to
        -- Defined at src/Fundeps.hs:21:3
      instance Cast a a -- Defined at src/Fundeps.hs:10:10
      instance Cast Int Double -- Defined at src/Fundeps.hs:16:10
    (The choice depends on the instantiation of ‘middle0’
     To pick the first instance above, use IncoherentInstances
     when compiling the other instance declarations)
  • In the expression: cast
    In an equation for ‘blargh’: blargh = cast
  |
27 | blargh = cast
  |          ^^^^
```

Now

GHC is asking you to enable `IncoherentInstances` in the *module* that contains the instance declaration! This is an extension you should *not* enable. Unlike `FlexibleInstances` and `UndecidableInstances` (which are both almost always safe), this one can cause dangerous and difficult to find bugs. There is basically no good reason to ever use it.

Multiparameter type classes aren't all that useful on their own. It's difficult to provide an API that has nice type inference properties. Note that we even needed to write type annotations to suggest the right return type for `cast` in `cast (cast a :: Int)`. Otherwise, we'd get an ambiguous error there, as GHC needs to be able to pick the right type for *every* call. If you're writing a feature and think "Ah, I need a `MultiParamTypeClass`," but you *don't* want functional dependencies, then you should probably pick a different avenue.

Fortunately, we have a bunch of options to make this better.

22.2 MPTCs + Fundeps

"Fundeps" is an abbreviation for `FunctionalDependencies`. This style of type-level programming is based on a relational or logical model of

computation. If you're familiar with Prolog, it's like that. If you're familiar with SQL, it's like computed/derived columns.

Logically speaking, a functional dependency forms an implication. Relationally speaking, we have a functional dependency from column `A` to column `B` if the value of column `A` uniquely determines the value of column `B`.

The syntax in Haskell for this is:

```
class ClassName a b | a -> b
```

We can read the above declaration as:

> Create a new type class named `ClassName`. It has two type variables, `a` and `b`. The type `a` uniquely determines the type `b`.

We can put multiple type variables on either side of the arrow, and have multiple arrows.

```
class Invert a b | a -> b, b -> a

class Multiply a b c | a b -> c
```

We'd read these as:

> The class `Invert` has two type variables `a` and `b`. `a` uniquely determines `b` and `b` uniquely determines `a`.

> The class `Multiply` has three type variables. The two variables `a` and `b` uniquely determine the third, `c`.

This is all a bit abstract, so let's write a type level function that tells us the cardinality of a given type. Cardinality refers to how many values for a type exist.

```
class Cardinality (typ :: *) (result :: Nat) | typ -> result
```

This instance declaration requires the `KindSignatures` language extension, and the `Nat` type comes from `GHC.TypeLits`. Let's write some simple instances:

```
instance Cardinality Void 0
instance Cardinality () 1
instance Cardinality Bool 2
```

The `Void` datatype has 0 values, so it has a cardinality of `0`. `()` (unit) has a single member, so it has cardinality 1. `Bool` has two members, `True` and `False`, so it has cardinality 2.

Let's consider a slightly more complicated example: `Maybe`. We learned in the "Basic" chapter that `Maybe` has cardinality `a + 1`, where `a` is the cardinality of the type contained in `Maybe a`.

```
instance
    (Cardinality a ac)
  => Cardinality (Maybe a) (ac + 1)
```

Except, oops, this fails. We get the GHC error "Illegal type synonym family application". This means that we can't use type families (like +) in type class instances. Let's work around this by defining `Add` on our own `Natural` type. We'll use `DataKinds` and use the ol' "successor" model.

```
data Nat = Z | S Nat

type Zero  = Z
type One   = S Z
type Two   = S One
type Three = S Two
type Four  = S Three
```

Now, we'll define our class.

```
class Add x y result
```

When defining functions, we take our inputs and then make them *uniquely determine* our "output" type. Convention dictates that the `result` type is the last one, but there's nothing requiring this to be true.

```
class Add x y result | x y -> result
```

Now, we write "instances" to "pattern match" on the types. Let's start with our `Z` instances. If we add `Zero` to anything, we get `Zero` back.

```
instance Add Zero y y
instance Add x Zero x
```

Oops, this blows up. Turns out, GHC will infer that the types in a type class must be of kind * (or kind `Type`). We can solve this in a few ways:

- Enable `PolyKinds`, which will infer polymorphic kind variables unless * (or `Type`) can be inferred.
- Enable `KindSignatures`, and provide explicit signatures.

`PolyKinds` is awfully convenient, but can lead to some devastatingly bad error messages when kinds don't match. Don't use it unless you really need kind polymorphism. Instead, we'll toss some kind signatures on:

```
class Add (x :: Nat) (y :: Nat) (result :: Nat) | x y -> result

instance Add Zero y y
instance Add x Zero x
```

This works! Note that something kinda weird is going on here. We've defined two "equations" of `Add` by setting up relationships. In value-level terms, this would look like:

```
add :: Nat -> Nat -> Nat
add Z y = y
add x Z = x
```

But we don't have any equals signs here. Instead, we've declared that the third type variable *is* our result. Let's proceed to the recursive case:

```
instance (Add a b r) => Add (S a) b (S r)
```

OK. OK. This is neat. First, we require that we know the answer to `Add a b r`, where `r` is the result of adding `a` and `b`. Then, if we're adding `S a` to `b`, the answer is `S r`. Let's test this out. To test out MPTCs+FunDeps, we create a `Proxy` that contains the "result" type, and then we ask GHCi for the type of the value. GHCi will evaluate the functional dependencies as far as it can.

```
λ> :set -XFlexibleContexts
λ> five = Proxy :: (Add Three Two r) => Proxy r
λ> five
Proxy
λ> :type five
five :: Proxy ('S ('S ('S ('S ('S Z)))))
```

That's one, two, three, four, five `S`s - it works!

Alright, let's get back to business and write that Cardinality instance for Maybe.

```
instance (Cardinality a ac, Add ac One r)
  => Cardinality (Maybe a) r
```

GHC is happy, so let's test it out.

```
maybeBool :: Cardinality (Maybe Bool) r => Proxy r
maybeBool = Proxy
```

GHC is again unhappy, suggesting that we enable `MonoLocalBinds`. This extension is implied by both `GADTs` and `TypeFamilies`, so don't be surprised if you don't see it much in the wild - those are both common extensions to have when doing type level programming. We can enable it, and then inspect the type in `ghci`:

```
λ> :t maybeBool
maybeBool :: Proxy ('S ('S ('S 'Z)))
```

Cool, that works!

As an exercise, write the instances of `Cardinality (a, b) r` and `Cardinality (Either a b) r`.

Running Backwards

One thing that's neat about the relational model of programming is that you can run functions *backwards*. Sometimes. Depending on how the class is defined. Let's take `Add` that we defined above. If we know one of the inputs and the answer, then we can determine the other input:

```
whatPlusThreeIsFive
    :: Add res (S (S (S Z))) (S (S (S (S (S Z)))))
    => Proxy res
whatPlusThreeIsFive = Proxy
```

If we evaluate this, we get:

```
λ> :t whatPlusThreeIsFive
whatPlusThreeIsFive :: Proxy ('S ('S 'Z))
```

Which is great. But if we flip the argument order, then it doesn't work anymore.

```
whatPlusThreeIsFive'
    :: Add (S (S (S Z))) res (S (S (S (S (S Z))))) =>
       Proxy res
whatPlusThreeIsFive' = Proxy

λ> :t whatPlusThreeIsFive'
whatPlusThreeIsFive'
    :: Add ('S ('S ('S 'Z))) res ('S ('S ('S ('S ('S 'Z))))) =>
       Proxy res
```

So this can be somewhat fragile.

22.3 Associated Types

Unless you're experienced with Prolog, writing code using functional dependencies probably feels awkward. Using and testing code in that style is awkward, as well. We can replace functional dependencies with associated types. The algorithm to do so is:

1. Delete the `result` type parameter and the functional dependency.
2. Add an associated type that accepts the left-hand-side of the `->` and has the same kind as the stuff on the right.

Here's `Add` written in this style:

```
class Add (x :: Nat) (y :: Nat) where
    type Result x y :: Nat

instance Add x Z where
    type Result x Z = x

instance Add Z y where
    type Result Z y = y

instance (Add a b) => Add (S a) b where
    type Result (S a) b = S (Result a b)
```

Except, well, this doesn't compile. GHC gives us a conflicting instance declaration:

```
error:
    Conflicting family instance declarations:
      Result x 'Z = x
                      -- Defined at Fundeps.hs:61:8
      Result ('S a) b = 'S (Result a b)
                      -- Defined at Fundeps.hs:67:8
   |
61 |   type Result x Z = x
   |        ^^^^^^
```

Turns out, we have a redundant case. We can delete the `Add x Z` instance. This fixes the problem.

22.4 Comparisons

Alright, so we have four ways of writing type-level computations:

1. Closed `TypeFamilies`
2. Open `TypeFamilies`
3. `MultiParamTypeClasses` + `FunctionalDependencies`
4. Type Class with an Associated Type

The trade-offs are pretty big, so let's get into it.

Value Associations

If you want to have value-level correspondence with your type-level computation, then you need a type class. This is going to get into the theory of the Lambda Cube a bit, which is a fun and somewhat mystical sounding bit of computer science theory.

Basically, you start off with the simply typed lambda calculus. The core idea is the lambda, which has a function type. This function has the shape `Value -> Value`. You can extend the lambda calculus in three distinct ways, mostly by adding new kinds of functions:

1. `Type -> Type`
2. `Value -> Type`
3. `Type -> Value`

If we allow functions from types to types, that gives us type-level programming and type constructors. If we allow functions from values to types, that gives us dependent types. If we allow functions from types to values, that gives us parametric polymorphism and type classes.

In a real sense, type classes are a means of dropping information from the type sky to the value earth. To get a feel for this, let's look at the `Monoid` type class.

```
class (Semigroup a) => Monoid a where
    mempty :: a
```

The fully qualified type of `mempty` is something like:

```
λ> :t mempty
mempty :: forall a. Monoid a => a
```

`forall a` introduces an implicit "type variable argument." And `Monoid a =>` introduces an implicit argument for a `Monoid` dictionary for the type `a`. GHC will figure these things out as best it can. But, in a different world, where we had to apply all of the types manually, we'd always write:

```
mempty @(Sum Int)
```

We're passing a *type level* argument to a function and receiving a *value* back. That's the intuition for why a type class is a function from type to value.

If you buy that argument, it becomes clear why you want canonical instances. A type class with non-canonical instances is no longer a function - it is indeterminate.

```
instance Semigroup Int where (<>) = (+)
instance Monoid Int where mempty = 0

instance Semigroup Int where (<>) = (*)
instance Monoid Int where mempty = 1
```

Some people want this. Apparently you can get behavior like this in Scala. I don't want it. I like pure functions, and adding non-determinism to type class resolution makes a difficult process even more confusing.

Now that we know the theory, let's dig into an example. Let's say you've defined the classic inductive natural for type level computation:

```
data Nat = Z | S Nat
```

You can define addition, multiplication, and other operations on the `DataKinds` lifted `Nat` kind. But you can't take that type information and turn it into a *value* without a type class. We can write a type class that demotes our type-computed number like this:

```
class TypeToValue (a :: Nat) where
    typeToValue :: Proxy a -> Nat
```

We've got some name overloading going on here. Note that `(a :: Nat)` means "a type variable `a` with the kind `Nat`", but `Proxy a -> Nat` means "A function from a value of type `Proxy a` to a value of type `Nat`." So let's define the instances:

```
instance TypeToValue 'Z where
    typeToValue Proxy =
        Z

instance TypeToValue n => TypeToValue (S n) where
    typeToValue (Proxy :: Proxy (S n)) =
        S (typeToValue (Proxy :: Proxy n))
```

Defaulting

Associated types are the only form that permit default instances. This is a type class `Record` that carries an associated type `Fields` that returns a list of the type of fields that the record contains.

```
class Record typ where
    type Fields typ :: [Type]
    type Fields typ = GFields (Rep typ)
```

`GFields` is a type family that computes based on the `Generic` type class's `Rep` associated type.

```
type family GFields rep where
    GFields (D1 _ (C1 _ xs)) = EnumerateTypes xs

type family EnumerateTypes rep where
    EnumerateTypes (a :*: b) =
       EnumerateTypes a ++ EnumerateTypes b
    EnumerateTypes (S1 _ (Rec0 a)) =
       '[a]
```

This allows us to omit the definition if the default is acceptable. It also allows us to override it.

```
data User = User { name :: String }
    deriving stock Generic

-- can use default since is Generic
instance Record User

data Dog = Dog { name :: String }

-- does not need the Generic instance, can provide directly
instance Record Dog where
    type Fields Dog = '[String]
```

Closed type families require you to specify all the cases at once, so defaulting doesn't make sense.

Open type families are strict about possible overlaps.

```
type family Foo a

type instance Foo a = Int
```

With these two lines, you can no longer implement any other instances without getting a conflicting instance declaration. Even if you move the type family into a type class and associated type, GHC will complain about the messages. So setting a default is *only* possible on the `class` definition itself.

This is useful if you have a convention, but want the ability to break from that convention at times. Consider a database access class that assumes a `UUID` key with a phantom type parameter for the record you're retrieving.

```
class GetById record where
    getById :: UUID record -> Database (Maybe record)
```

However, you run into a problem: you want to use this on a type that doesn't have a `UUID` all on it's own. It has a different sort of primary key. The modification that preserves backwards compatibility is to add an associated type with a default.

```
class GetById record where
    type UniqueId record :: Type
    type UniqueId record = UUID record

    getById :: UniqueId record -> Database (Maybe record)
```

No existing instances need to change. But you're now able to define your new instance:

```
instance GetById WeirdType where
    type UniqueId WierdType = (UUID User, UUID Donut)

    getById (userId, donutId) = ...
```

Open vs Closed

Closed type families are the only remotely closed option, so if you want exhaustiveness, then these are your best choice. However, the closed aspect of type families has less to do with "exhaustiveness" and more to do with "no one else can add a case." Consider how we get exhaustiveness in value-level programming - a `case` expression. But there is no such thing in type level programming.

In value level programming, if you miss a case, then you get a run-time error. But in type level programming, if you miss a case, you get a "stuck type family." A stuck type family does not have an equation that it can use to proceed, and will propagate until some other type error happens. Stuck type families can cause all sorts of problems.

Bidirectional Dependencies

Above, I mentioned that `FunctionalDependencies` allow you to run type-level computation backwards. We aren't describing a linear flow of information - we're describing a set of relationships. So if I know two of three variables, I can solve for the third.

The extension `TypeFamilyDependencies` provides this for type families. To use it, we write:

```
{-# language TypeFamilyDependencies #-}

type family Foo x = r | r -> x
```

This open type family `Foo` is now saying that the *result* type can inform the input. So if I know the result of `Foo`, then I can infer the input.

This ends up being pretty limiting. Suppose you write:

```
type family PrimaryKey record = typ | typ -> record
```

This requires that the `typ` is *uniquely* given for a record. I can't use `UUID` or `Int64` for multiple different types. As a result, this open type family isn't all *that* useful - I need to define a separate datatype for every single type I want to put in the database.

The shortcut is to use a *data family*, which combine the type-family relationship with a data constructor declaration.

```
data family PrimaryKey record

data instance PrimaryKey User = UserKey UUID
data instance PrimaryKey Dog = DogKey UUID

-- instead of,
type family PrimaryKey record = typ | typ -> record

newtype UserKey = UserKey UUID
type instance PrimaryKey User = UserKey

newtype DogKey = DogKey UUID
type instance PrimaryKey Dog = DogKey
```

This sort of dependency is much more useful on closed type families. Consider `Not` - an operation that flips a `Bool` type.

```
type family Not (a :: Bool) :: Bool where
    Not 'True = 'False
    Not 'False = 'True
```

As it happens, if we know `Not a` $\sim$ `True`, then GHC can know that `a` $\sim$ `False`. With `TypeFamilyDependencies` enabled, we can write this:

```
type family Not (a :: Bool) = (r :: Bool) | r -> a where
    Not 'True = 'False
    Not 'False = 'True
```

This mostly ends up being useful in providing good type inference. Many things are not bijective. For example, addition is not.

```
type family Add (a :: Nat) (b :: Nat) = (r :: Nat) | r -> a b
```

This definition cannot be satisfied. The reason is that we can't know from `4` whether it was `4 + 0`, or `2 + 2`, or `3 + 1`.

23. Case Study: Prairie

Haskell records are Kind Of Bad. It's a known thing, and it's awfully unfortunate. By default, a record definition is rather underpowered.

Haskell record syntax looks like this:

```
data User = User
    { name :: String
    , age :: Int
    }
```

Defining a datatype with curly braces defines "field labels." Field labels can be used in one of three ways:

```
-- 1. Record Accessor Function
getUserName :: User -> String
getUserName user = name user

-- 2. Record Creation Syntax
example1 :: User
example1 = User { name = "Alice", age = 30 }

-- 3. Record Update Syntax
example2 :: User
example2 = example1 { name = "Bob" }
```

Unfortunately, the record creation and update syntax is not "first class." You can't pass record labels around.

```
nope field record =
  record { field = "Hello, world!" }
```

Also,

record fields are *monomorphic* - you can't define two fields in the same module with the same label.

```
data User = User { name :: String }
data Dog = Dog { name :: String }
```

You can get around this with `DuplicateRecordFields`, but that extension (as of this writing) has awful UX. You need to insert type signatures in unpredictable places to use the fields, or import everything qualified. Unfortunately, in GHC 9.4, the situation has been made much worse - you will receive a warning when doing a record update if duplicate names are in scope[1].

Inspired by the `EntityField`[2] type in `persistent`, I decided to fix the problem. This chapter is written in a "stream of thought" style, trying out a few dead ends and poor choices before settling on the good stuff. Learning is about making mistakes - here are some of mine!

23.1 Problem Statement:

I want to reify the fields on a Haskell record into first-class entities. Let's review the status quo:

```
data User = User
    { userName :: Text
    , userAge :: Int
    , userId :: Int
    }
```

This creates "record fields," which are sorta special, but also sorta not. We're able to use them as accessor functions:

[1] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/duplicate_record_fields.html
[2] https://www.stackage.org/haddock/lts-16.17/persistent-2.10.5.2/Database-Persist-Class.html#t:EntityField

```
>>> userName (User "Bob" 40 3)
"Bob"
```

We're able to use them in record creation:

```
>>> userId User { userName = "Foo", userAge = 32, userId = 1 }
1
```

and also in record update:

```
>>> let someUser = User ...
>>> userName someUser { userName = "blargh" }
"blargh"
```

But this is all quite monomorphic. There's no "higher order record field" syntax. I can't write:

```
alas :: RecordField -> Record -> Record
alas someField someRecord =
    someRecord { someField = Nothing }
```

Ultimately, I want to be able to write the following types and functions:

```
data Update record

instance ToJSON (Update record)
instance FromJSON (Update record)

update :: [Update record] -> record -> record
```

The JSON instances are going to tricky.

So, how can we get there?

23.2 Prior Art

`EntityField`

The `persistent` database library has something that is close to what I want. On the `PersistEntity` class, there's a `data family` called `EntityField` which has the following shape:

```
class PersistEntity entity where
    data EntityField entity :: * -> *

    fieldLens :: EntityField entity typ -> Lens' entity typ
```

Instances are supposed to be a GADT:

```
instance PersistEntity User where
    data EntityField User a where
        UserName :: EntityField User Text
        UserAge :: EntityField User Int
        UserId :: EntityField User Int

    fieldLens fld = case fld of
        UserName ->
            lens userName (\u s -> u { userName = s })
        UserAge ->
            ...
        UserId ->
            ...
```

This approach is nice because we can exhaustively cover all the fields of a record. We even get `fieldLens`, which lets us turn the `EntityField` value into a getter/setter for the record itself.

This is the approach I'm most interested in, because I've used it a lot and it works well. However, the `persistent` codebase is pretty old. There may be a better approach available now.

Generics

`GHC.Generics` can be used to collect a list of symbols corresponding to the field names of a record.

Lenses

The `Control.Lens.mkFields` TemplateHaskell function gives us the following:

```
mkFields ''User
-- =====>
class HasName u s | u -> s where
    name :: Lens' u s

instance HasName User Text where
    name :: Lens' User Text
    name = lens userName (\u s -> u { userName = s })

class HasAge u s | u -> s where
    age :: Lens' u s

instance HasAge User Int where
    age = ...

-- etc, you get it
```

However, `age` in this is "just" a lens. We can use `age` to set or view the `userAge` in a user, but it's not a "field" necessarily.

`generic-lens` generalizes the above pattern to (roughly speaking):

```
deriving stock instance Generic User

class HasField sym s t a b where
    field :: Lens s t a b

-- the following are automatically inferred:
instance HasField "userName" User User Text Text where
    field = lens userName (\u n -> u { userName = n })

instance HasField "userAge" User User Int Int where ...
instance HasField "userId" User User Int Int where ...
```

23.3 The GADT Approach

```
class Record a where
    data Field a :: Type -> Type

    fieldLens :: Field a r -> Lens' a r

instance Record User where
    data Field User a where
        UserName :: Field User Text
        UserId :: Field User Int
        UserAge :: Field User Int

    recordFieldLens rf =
        case rf of
            UserName -> lens userName (\u n -> u { userName = n })
            UserId -> lens userId (\u n -> u { userId = n })
            UserAge -> lens userAge (\u n -> u { userAge = n })
```

This is easy enough. Working with the GADT type is a bit tricky, because we can't "just" derive a lot of stuff we might want to. Eq and Show derive how you might expect:

```
deriving stock instance Show (Field User a)
deriving stock instance Eq (Field User a)
deriving stock instance Ord (Field User a)
```

However, `Eq` works a bit odd. We can't actually write `UserName == UserId` because types don't match. But we *can* write `UserAge == UserId`, which returns `False`. Likewise, we can't have a bare `[Field User a]` that has *all* the user fields - it can only contain user fields that have the same field type.

`Show` is easy because it destroys information. `ToJSON` is similarly easy:

```
instance ToJSON (Field User a) where
    toJSON = toJSON . show
```

Parsing raw data into a `Field` is going to be a challenge. Let's start with the naive approach:

```
instance FromJSON (Field User a) where
    parseJSON = withText "Field<User>" $ \txt ->
        case txt of
            "UserName" ->
                pure UserName
            "UserId" ->
                pure UserId
            "UserAge" ->
                pure UserAge
            _ -> fail
                $ "Couldn't parse Field<User>, got: "
                <> Text.unpack txt
```

This fails with a type error.

```
/home/matt/Projects/recupd/src/Record/GADTField.hs:52:25-35: error:
    • Couldn't match type ‘Int’ with ‘Text’
      Expected type: aeson-1.5.4.0:Data.Aeson.Types.Internal.Parser
                       (Field User Text)
        Actual type: aeson-1.5.4.0:Data.Aeson.Types.Internal.Parser
                       (Field User Int)
    • In the expression: pure UserId
      In a case alternative: "UserId" -> pure UserId
      In the expression:
        case txt of
          "UserName" -> pure UserName
          "UserId" -> pure UserId
          "UserAge" -> pure UserAge
          _ -> fail
             $ "Couldn't parse Field<User>, got: "
             <> Text.unpack txt
   |
52 |             "UserId" -> pure UserId
   |                         ^^^^^^^^^^
/home/matt/Projects/recupd/src/Record/GADTField.hs:53:26-37: error:
    • Couldn't match type ‘Int’ with ‘Text’
      Expected type: aeson-1.5.4.0:Data.Aeson.Types.Internal.Parser
                       (Field User Text)
        Actual type: aeson-1.5.4.0:Data.Aeson.Types.Internal.Parser
                       (Field User Int)
    • In the expression: pure UserAge
      In a case alternative: "UserAge" -> pure UserAge
      In the expression:
        case txt of
          "UserName" -> pure UserName
          "UserId" -> pure UserId
          "UserAge" -> pure UserAge
          _ -> fail
             $ "Couldn't parse Field<User>, got: "
             <> Text.unpack txt
   |
53 |             "UserAge" -> pure UserAge
```

```
38    |                                ^^^^^^^^^^^^
```

All the branches in a case expression must evaluate to the same type. So. We can't have one branch returning a `UserAge :: Field User Int` and another returning `UserName :: Field User Text`. What if we try to monomorphize?

```haskell
instance FromJSON (Field User Int) where
    parseJSON = withText "Field<User>" $ \txt ->
        case txt of
            "UserId" -> pure UserId
            "UserAge" -> pure UserAge
            _ -> fail
                $ "Couldn't parse Field<User>, got: "
                <> Text.unpack txt

instance FromJSON (Field User Text) where
    parseJSON = withText "Field<User>" $ \txt ->
        case txt of
            "UserName" -> pure UserName
            _ -> fail
                $ "Couldn't parse Field<User>, got: "
                <> Text.unpack txt
```

This works. But man is it kinda gross - we have to have a separate instance for every field type in the record. And it's not particularly useful. We can't parse an input string into *any* field - we have to know the type ahead of time.

So we can introduce an existential wrapper that hides the field type:

```haskell
data SomeField rec where
    SomeField :: Field rec a -> SomeField rec
```

This lets us write a good `FromJSON` instance:

```
instance FromJSON (SomeField User) where
    parseJSON = withText "SomeField<User>" $ \txt ->
        case txt of
            "UserName" -> pure $ SomeField UserName
            "UserId" -> pure $ SomeField UserId
            "UserAge" -> pure $ SomeField UserAge
            _ -> fail
                $ "Couldn't parse SomeFieldUser, got: "
                <> Text.unpack txt
```

We can even have lists of these:

```
allFields :: [SomeField User]
allFields = [SomeField UserName, SomeField UserAge, SomeField UserId]
```

Which can be encoded and decoded:

```
instance ToJSON (SomeField User) where
    toJSON (SomeField f) = toJSON f

roundTripAllFields =
    decode (encode allFields) == Just allFields
```

... Except, we don't have an instance of `Eq (SomeField User)`. Let's write it.

```
instance  Eq (SomeField User) where
    SomeField UserName == SomeField UserName = True
    SomeField UserAge == SomeField UserAge = True
    SomeField UserId == SomeField UserId = True
    _ == _ = False
```

Now `roundTrpAllFields` evaluates to `True`.

OK, so we can serialize the fields of a record. Let's write a type that contains a *value* of the record field, too.

```
data UpdateRecord rec where
    SetField :: Field rec a -> a -> UpdateRecord rec
```

Now, we can write our `updateRecord` function!

```
updateRecord :: Record rec => [UpdateRecord rec] -> rec -> rec
updateRecord updates original =
    foldr
        (\(SetField field newValue) ->
            set (recordFieldLens field) newValue)
        original
        updates
```

But that's not enough. We need to be able to serialize and deserialize them. This is where things become Tricky. Here's a naive first attempt:

```
instance  ToJSON (UpdateRecord User) where
    toJSON (SetField recField newVal) =
        object
            [ "field" .= recField
            , "value" .= newVal
            ]
```

GHC won't accept this.

```
/home/matt/Projects/recupd/src/Record/GADTField.hs:92:15-31: error:
  • No instance for (ToJSON a) arising from a use of '.='
    Possible fix:
     add (ToJSON a) to the context of the data constructor 'SetField'
  • In the expression: "value" .= newVal
    In the first argument of 'object', namely
      '["field" .= recField, "value" .= newVal]'
    In the expression: object ["field" .= recField, "value" .= newVal]
   |
92 |             , "value" .= newVal
   |
```

No instance of `ToJSON a` - but! We know that `a` has an instance of `ToJSON` for every single field!

If we want to communicate that information to GHC, we'll have to pattern match on each constructor manually.

```
instance  ToJSON (UpdateRecord User) where
    toJSON (SetField recField newVal) =
        object
            [ "field" .= recField
            , "value" .= case recField of
                UserAge -> newVal
                UserId -> newVal
                UserName -> newVal
            ]
```

This also fails - we've *already* pattern matched on the `SetField` constructor, which fixes the type variables. So we have to take the pattern match further out.

```
instance  ToJSON (UpdateRecord User) where
    toJSON sf = case sf of
        SetField UserAge newVal ->
            object [ "field" .= UserAge, "value" .= newVal ]
        SetField UserName newVal ->
            object [ "field" .= UserName, "value" .= newVal ]
        SetField UserId newVal ->
            object [ "field" .= UserId, "value" .= newVal ]
```

Kinda nasty, but it works.

Let's try and parse it, too.

```
instance FromJSON (UpdateRecord User) where
    parseJSON = withObject "UpdateRecord<User>" $ \o -> do
        someRecField <- o .: "field"
        case someRecField of
            SomeField fld ->
                case fld of
                    UserAge ->
                        SetField fld <$> o .: "value"
                    UserId ->
                        SetField fld <$> o .: "value"
                    UserName ->
                        SetField fld <$> o .: "value"
```

This works. And with Template Haskell, it wouldn't have so much boilerplate - we could easily generate all of this with a call to `mkRecordFields ''User`. It's somewhat inelegant though. Can we make these instances a bit more polymorphic?

Polymorphic Instances

First, let's tackle `SomeField`'s instances.

```
instance ToJSON (SomeField record) where
    toJSON (SomeField field) =
        toJSON field
```

This fails, as you might expect, with:

```
/home/matt/Projects/recupd/src/Record/GADTField.hs:72:9: error:
    • No instance for (ToJSON (Field record a))
        arising from a use of ‘toJSON’
    • In the expression: toJSON field
      In an equation for ‘toJSON’:
          toJSON (SomeField field) = toJSON field
      In the instance declaration for ‘ToJSON (SomeField record)’
   |
72 |         toJSON field
   |
```

Can we just ask for that constraint?

```
instance (ToJSON (Field record a)) => ToJSON (SomeField record) where
    toJSON (SomeField field) = toJSON field
```

Unfortunately, no.

```
/home/matt/Projects/recupd/src/Record/GADTField.hs:70:10: error:
 • Could not deduce (ToJSON (Field record a0))
   from the context: ToJSON (Field record a)
     bound by an instance declaration:
                 forall record a.
                 ToJSON (Field record a) =>
                 ToJSON (SomeField record)
     at src/Record/GADTField.hs:70:10-63
   The type variable ‘a0’ is ambiguous
 • In the ambiguity check for an instance declaration
   To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
   In the instance declaration for ‘ToJSON (SomeField record)’
   |
70 | instance (ToJSON (Field record a)) => ToJSON (SomeField record)
   |           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Why not? Well, `a` in this context isn't the same as `a` that is getting unpacked from the GADT. Let's write out explicit type variable introduction to see why.

```
instance forall a record.
    (ToJSON (Field record a))
  =>
    ToJSON (SomeField record)
  where
    toJSON (SomeField (field :: Field record a0)) =
        toJSON field
```

Pattern matching on `SomeField` "unpacks" the `a0` type variable. This type variable is untouchable from the outside, so we can't require the constraint like this. Fortunately, there's a trick: `QuantifiedConstraints` lets us say something a bit stronger:

```
instance
    (forall a. ToJSON (Field record a))
  =>
    ToJSON (SomeField record)
  where
    toJSON (SomeField field) = toJSON field
```

The two instance declarations look similar, but there's a subtle difference here. Let's get rid of some noise and explicitly introduce type variables:

```
-- 1.
instance forall r a. (C (Rec r a)) => C (SomeField r)
-- 2.
instance forall r.   (forall a. C (Rec r a)) => C (SomeField r)
```

The first version says something like this:

> For all types `r` and `a` such that there's an instance of `C (Rec r a)`, we have an instance of `C (SomeField r)`

A `forall` in this position means the user gets to pick it. So the user might try to instantiate `r ~ User` and `a ~ Int`. But that doesn't mean that we won't *actually* be carrying a `r ~ User` and `a ~ Text`. So we have to make a *stronger* claim. The second says this:

> For all types `r`, such that, for any type `a`, there's an instance of `C (Rec r a)`, we have an instance of `C (SomeField r)`.

It's like the difference in these two functions:

```
idEach  :: forall a. (a -> a) -> (a, a) -> (a, a)
idEach  idFn (x, y) = (idFn x, idFn y)

idEach' :: (forall x. x -> x) -> (a, b) -> (a, b)
idEach' idFn (a, b) = (idFn a, idFn b)
```

The user of the function does not get to pick the type `x`, which means I can choose it to work with both the `a` and `b` types. However, in the first one, the user gets to pick the `a` type, which means I can't have two different types in the tuple.

That's serializing. Let's deserialize. Parsing is always more difficult, because we're refining and creating information. We're creating information that exists in the type system, which makes it even more fun.

Let's review the `FromJSON (SomeField User)` type that we weant to generalize.

```
instance FromJSON (SomeField User) where
    parseJSON = withText "SomeField<User>" $ \txt ->
        case txt of
            "UserName" -> pure $ SomeField UserName
            "UserId" -> pure $ SomeField UserId
            "UserAge" -> pure $ SomeField UserAge
            _ -> fail
                $ "Couldn't parse SomeFieldUser, got: "
                <> Text.unpack txt
```

Abstraction, as a practice, means removing the concrete and accepting it as a variable instead.

```
-- concrete:
5 + 6

-- abstract 5:
(\x -> x + 6)
```

So we're going to abstract the details of `User` from this.

```
instance Record record => FromJSON (SomeField record) where
    parseJSON = withText "SomeField" $ \txt ->
        case txt of
            ??? ->
                pure $ ???
            _ -> fail
                $ "Couldn't parse SomeField, got: "
                <> Text.unpack txt
```

How do we approach this, programmatically?

The original case expression is a lookup table, essentially. So we could make another class method on `Record` which contains such a table.

```
class Record a where
    data Field a :: * -> *

    recordFieldLens :: Field a b -> Lens' a b

    fieldLookup :: Map Text (SomeField a)

instance Record User where
    data Field User a where
        UserName :: Field User Text
        UserId :: Field User Int
        UserAge :: Field User Int

    recordFieldLens rf =
        case rf of
```

```
            UserName -> lens userName (\u n -> u { userName = n })
            UserId -> lens userId (\u n -> u { userId = n })
            UserAge -> lens userAge (\u n -> u { userAge = n })

    fieldLookup =
        Map.fromList $ map
            (\sf -> (Text.pack (show sf), sf))
            [ SomeField UserName
            , SomeField UserAge
            , SomeField UserId
            ]
```

This allows us to write our `FromJSON` instance nicely:

```
instance Record record => FromJSON (SomeField record) where
    parseJSON = withText "SomeField" $ \txt ->
        case Map.lookup txt (fieldLookup) of
            Nothing ->
                fail
                    $ "Couldn't parse SomeField, got: "
                    <> Text.unpack txt
            Just fld ->
                pure fld
```

OK, on to updates, starting with `ToJSON` first because it's easier to serialize than parse. As a refresher, here's the original code:

```
instance ToJSON (UpdateRecord User) where
    toJSON sf = case sf of
        SetField UserAge newVal ->
            object [ "field" .= UserAge, "value" .= newVal ]
        SetField UserName newVal ->
            object [ "field" .= UserName, "value" .= newVal ]
        SetField UserId newVal ->
            object [ "field" .= UserId, "value" .= newVal ]
```

This is a bit tricky. The concrete type `User` brings with it a few things:

1. The knowledge of which `Field User x` constructors we can pattern match on.
2. When pattern matching on the constructor, we know what `x` type is, and because `x` gets fixed to a concrete type, we know that it has a `ToJSON` instance.

Serializing the label is easy - we covered that in `Record record => ToJSON (SomeField rec)`.

```
instance
    (forall a. ToJSON (Field rec a), Record rec)
  =>
    ToJSON (UpdateRecord rec)
  where
    toJSON sf = case sf of
        SetField label newVal ->
            object [ "field" .= label ]
```

But now we want to serialize `newVal`, too. For that, we need an instance of `ToJSON a`. But it's not enough to say `ToJSON a` - we need to somehow say "For all `a` that might possibly be in a `Field rec a`, we need na instance of `ToJSON a`".

Well, `UpdateRecord` is a GADT, which means we can put constraints there and unpack them later. Let's just do that.

```
data UpdateRecord rec where
    SetField :: ToJSON a => Field rec a -> a -> UpdateRecord rec

instance
    (forall a. ToJSON (Field rec a), Record rec)
  =>
    ToJSON (UpdateRecord rec)
  where
    toJSON sf = case sf of
        SetField label newVal ->
            object [ "field" .= label, "value" .= newVal ]
```

This works. But packing constraints in GADTs always makes me nervous. Let's try parsing. Here's our naive initial attempt:

```
instance Record rec => FromJSON (UpdateRecord rec) where
    parseJSON = withObject "UpdateRecord" $ \o -> do
        someField <- o .: "field"
        case someField of
            SomeField (fld :: Field rec a) ->
                SetField fld <$> o .: "value"
```

Now we have two errors.

```
/home/matt/Projects/recupd/src/Record/GADTField.hs:122:17: error:
  • Could not deduce (ToJSON a) arising from a use of ‘SetField’
    from the context: Record rec
      bound by the instance declaration
      at src/Record/GADTField.hs:117:10-50
    Possible fix:
      add (ToJSON a) to the context of the data constructor ‘SomeField’
  • In the first argument of ‘(<$>)’, namely ‘SetField fld’
    In the expression: SetField fld <$> o .: "value"
    In a case alternative:
       SomeField (fld :: Field rec a) -> SetField fld <$> o .: "value"
    |
122 |                 SetField fld <$> o .: "value"
    |                 ^^^^^^^^^^^^

/home/matt/Projects/recupd/src/Record/GADTField.hs:122:34: error:
  • Could not deduce (FromJSON a) arising from a use of ‘.:’
    from the context: Record rec
      bound by the instance declaration
      at src/Record/GADTField.hs:117:10-50
    Possible fix:
      add (FromJSON a) to the context of the data constructor ‘SomeField’
  • In the second argument of ‘(<$>)’, namely ‘o .: "value"’
    In the expression: SetField fld <$> o .: "value"
    In a case alternative:
```

```
26          SomeField (fld :: Field rec a) -> SetField fld <$> o .: "value"
27      |
28  122 |                   SetField fld <$> o .: "value"
29      |                                    ^^^^^^^^^^^^
```

Can't deduce either `ToJSON a` or `FromJSON a`. Wait. Why does it need `ToJSON a`? We're trying to parse a value, not serialize it.

Welp. When we added the constraint to the `SetField` constructor, that means we can't actually *call* the constructor unless the `ToJSON a` instance is in scope. This has me doubting whether or not it's reasonable to put the constraint in the constructor.

What were we abstracting in the original code? Here it is again for a refresher:

```
instance FromJSON (UpdateRecord User) where
    parseJSON = withObject "UpdateRecord<User>" $ \o -> do
        someRecField <- o .: "field"
        case someRecField of
            SomeField fld ->
                case fld of
                    UserAge ->
                        SetField fld <$> o .: "value"
                    UserId ->
                        SetField fld <$> o .: "value"
                    UserName ->
                        SetField fld <$> o .: "value"
```

Again - the constructors, which, upon pattern matching, brought a concrete type into scope for the `a`, which GHC could do a lookup and determine that they are indeed instances of `ToJSON` and `FromJSON`.

This is going to be a little tricker. For `SomeField rec`, we just needed to be able to lookup the actual value. But for `Update rec`, we also need to bring the `FromJSON` dictionary into scope.

Perhaps we can collect a list of the types on the record and require the constraint on all of them. Will that work? Let's try it.

```haskell
class Record a where
    type FieldTypes a :: [Type]
    data Field a :: Type -> Type

    -- ...

instance Record User where
    type FieldTypes User = '[Text, Int]

    -- ...

type family All (c :: k -> Constraint) (xs :: [k]) :: Constraint where
    All _ '[] = ()
    All c (x ': xs) = (c x, All c xs)

instance
    (All FromJSON (FieldTypes rec), Record rec)
  => FromJSON (UpdateRecord rec) where
```

Unfortunately, this also fails, with the same error messages as before. This actually didn't do anything to bring the relevant instances into scope! GHC doesn't know that the `a` type that is unpacked in `SomeField (fld :: Field rec a)` is present in the `FieldTypes rec` list. And, it doesn't know that, because we haven't actually proven it. So we need something a bit more interesting - a dictionary lookup function based on the field.

```haskell
class FieldDict rec c where
    getDict :: Field rec a -> Dict (c a)
```

`Dict` is a type from the `constraints`[3] package. It packages up a type class instance so it can be transmitted as a value. It uses `GADTs` and `ConstraintKinds` to accomplish this trick.

[3] https://hackage.haskell.org/package/constraints

```
data Dict (c :: Constraint) where
    Dict :: c => Dict
```

Now, we can make the claim we need. First, let's rip out the `ToJSON a` constraint in the `SetField` constructor. Then we'll fix `ToJSON`:

```
data UpdateRecord rec where
    SetField :: Field rec a -> a -> UpdateRecord rec

instance
    ( forall a. ToJSON (Field rec a)
    , Record rec
    , FieldDict rec ToJSON
    )
  =>
    ToJSON (UpdateRecord rec)
  where
    toJSON sf = case sf of
        SetField (label :: Field rec a) (newVal :: a) ->
            case getDict label of
                (Dict :: Dict (ToJSON a)) ->
                    object [ "field" .= label, "value" .= newVal ]
```

OK, so we're doing some fancy type level stuff here. Let's go through this line by line.

```
case sf of
    SetField (label :: Field rec a) (newVal :: a) ->
```

At this point, we've unpacked `SetField`, and used the type signatures to bring those type variables into scope. Haskell's type langauge allows us to use the same variable multiple times with introduction, and it infers them to be the same type. So this case expression pattern matches on `sf` and introduces:

- The value `label`

- The type `a`
- The value `newVal`

Next up, we pattern match on `getDict`:

```
case getDict label of
    (Dict :: Dict (ToJSON a)) ->
```

Since we have requested `FieldDict rec ToJSON` in the instance head, we can now call `getDict :: Field rec a -> Dict (c a)` at the type `c ~ ToJSON`. This gives us a value `Dict (ToJSON a)`. If we unpack that GADT, we get `Dict :: ToJSON a => Dict`. Now, we've introduced the `ToJSON a` dictionary into scope. This means we can finally do our value-level magic:

```
object [ "field" .= label, "value" .= newVal ]
```

While writing this code, it occured to me that the API for summoning dictionaries was a bit awkward. So I swapped the type variables on the class `FieldDict` and that enabled a nicer syntax with `TypeApplications`:

```
class FieldDict c rec where
    getDict :: Field rec a -> Dict (c a)

instance
    ( forall a. ToJSON (Field rec a)
    , Record rec
    , FieldDict ToJSON rec
    )
  =>
    ToJSON (UpdateRecord rec)
  where
    toJSON sf = case sf of
        SetField (label :: Field rec a) (newVal :: a) ->
            case getDict @ToJSON label of
                Dict ->
                    object [ "field" .= label, "value" .= newVal ]
```

Now we can just write `getDict @ToJSON label` and the unpacked dictionary just knows what we need. Nice! The pattern matching is a bit annoying, too. For a lot of "existential wrapper" types like this, it can be a convenience to write a "continuation passing style" variant.

```
withFieldDict
    :: forall c rec a r
     . FieldDict c rec
    => Field rec a
    -> (c a => r)
    -> r
withFieldDict l k =
    case getDict @c l of
        Dict -> k
```

The function `getDict` is equivalent to saying "Give me a `Field rec a` and I will return a proof that `c a` is true." A value of type `Dict (c a)` is theoretically equivalent to a proof of the proposition that the type `a` satisfies the class `c`. In order to use a proof, you must pattern match on it. `withFieldDict` says something slightly different. Instead of returning the proof directly, we accept a value that assumes the proof to be true.

That allows us to rewrite our `ToJSON` instance as:

```
instance
    ( forall a. ToJSON (Field rec a)
    , Record rec
    , FieldDict ToJSON rec
    )
  =>
    ToJSON (UpdateRecord rec)
  where
    toJSON sf = case sf of
        SetField (label :: Field rec a) (newVal :: a) ->
            withFieldDict @ToJSON label $
                object [ "field" .= label, "value" .= newVal ]
```

No more awkward pattern match on `Dict`.

`FromJSON` follows in much the same way.

```
instance
    ( Record rec
    , FieldDict FromJSON rec
    )
  =>
    FromJSON (UpdateRecord rec)
  where
    parseJSON = withObject "UpdateRecord" $ \o -> do
        someField <- o .: "field"
        case someField of
            SomeField (fld :: Field rec a) ->
                withFieldDict @FromJSON fld $
                    SetField fld <$> o .: "value"
```

Now, if we actually try to use these with our `User` type, then we'll run into some errors.

```
roundTripUpdates =
    decode (encode updates) == Just updates
  where
    updates =
        [ SetField UserAge 3
        , SetField UserId 2
        , SetField UserName "Bob"
        ]
```

We need instances of `FieldDict FromJSON User`, `FieldDict ToJSON User`, and `Eq (UpdateRecord User)`. Let's write them. `FieldDict` is going to be really easy, if a bit odd looking.

```
instance FieldDict ToJSON User where
    getDict f = case f of
        UserAge -> Dict
        UserName -> Dict
        UserId -> Dict

instance FieldDict FromJSON User where
    getDict f = case f of
        UserAge -> Dict
        UserName -> Dict
        UserId -> Dict
```

Actually, we can do something a little more interesting. We can be polymorphic in the constraint, provided that it is applied to all of our types.

```
instance (c Text, c Int) => FieldDict c User where
    getDict f = case f of
        UserAge -> Dict
        UserName -> Dict
        UserId -> Dict
```

We can use the `All` and `FieldTypes` machinery that we tried to use earlier:

```
instance (All c (FieldTypes User)) => FieldDict c User where
    getDict f = case f of
        UserAge -> Dict
        UserName -> Dict
        UserId -> Dict
```

Nice. Now let's write that `Eq` instance.

```
instance Eq (SomeField rec) => Eq (UpdateRecord rec) where
    SetField lbl a == SetField lbl' b =
        SomeField lbl == SomeField lbl'
```

This works, but it is unsatisfactory - if the two labels are equal, then we *also* want to compare the values. So we're going to need to bring some types into scope:

```
instance Eq (SomeField rec) => Eq (UpdateRecord rec) where
    (==)
        (SetField (lbl :: Field r0 a0) a)
        (SetField (lbl' :: Field r1 a1) b) =
            SomeField lbl == SomeField lbl'
            && case eqT @a0 @a1 of
                Just Refl ->
                    a == b
                Nothing ->
                    False
```

`eqT` comes from `Data.Typeable`. You pass it two type arguments using `TypeApplications` syntax. If the types are equal, you get `Just Refl`, which means that the two types can be considered equal within the case branch. Otherwise, you get `Nothing` back.

This code mostly works, but we get some problems: GHC can't deduce that `a0` or `a1` types are `Typeable`, and it can't deduce an instance of `Eq a0`. We'll need to use the `FieldDict` machinery again to bring those dictionaries into scope.

```
instance
    ( Eq (SomeField rec)
    , FieldDict Eq rec
    , FieldDict Typeable rec
    )
  =>
    Eq (UpdateRecord rec)
  where
    SetField (lbl :: Field r a0) a == SetField (lbl' :: Field r a1) b =
        withFieldDict @Eq lbl $
        withFieldDict @Eq lbl' $
        withFieldDict @Typeable lbl $
        withFieldDict @Typeable lbl' $
```

```
        SomeField lbl == SomeField lbl'
        && case eqT @a0 @a1 of
            Just Refl ->
                a == b
            Nothing ->
                False
```

GHC compiles this, and `roundTripUpdates` evaluates to `True`.

We can improve `FieldTypes`. Defining it manually is kind of a drag, since we can derive it generically.

```
data User = ...
    deriving stock Generic

type FieldTypes rec = GFieldTypes (Rep rec)

type family GFieldTypes rep where
    GFieldTypes (D1 _ (C1 _ xs)) = EnumerateTypes xs

type family EnumerateTypes rep where
    EnumerateTypes (S1 _ (Rec0 a) :*: rest) =
        a ': EnumerateTypes rest
    EnumerateTypes (S1 _ (Rec0 a)) =
        '[a]
```

However, requiring a `Generic` instance for this is a bit of a drag. We will end up using `TemplateHaskell` anyway to generate the instances, so rather than requiring a `Generic` implementation of a type family, let's just stick with the manual approach.

23.4 Improvements?

This works. But can it be better? Yeah. We can generalize `Eq` on `SomeField` so that it relies on the `Eq` instance for the underlying `Field`, which is derivable.

```
instance
    ( FieldDict Typeable rec
    , forall a. Eq (Field rec a)
    )
  =>
    Eq (SomeField rec)
  where
    SomeField (a :: Field rec a) == SomeField (b :: Field rec b) =
        withFieldDict @Typeable a $
        withFieldDict @Typeable b $
        case eqT @a @b of
            Just Refl ->
                a == b
            Nothing ->
                False
```

We can also write a generic diffing utility on records. It's super simple - no recursion or anything, just reports what changed on the new record.

```
diffRecord
    :: forall rec. (FieldDict Eq rec, Record rec)
    => rec
    -> rec
    -> [UpdateRecord rec]
diffRecord old new =
    foldr
        (\x acc ->
            case x of
                SomeField field ->
                    withFieldDict @Eq field $
                    let getter = view (recordFieldLens field)
                        newValue = getter new
                     in
                        if getter old == newValue
                        then acc
                        else SetField field newValue : acc
        )
```

```
        []
        (fieldLookup @rec)
```

I'm pretty happy with this. Let's try another approach - just because I've found success doesn't mean a better way isn't coming!

23.5 Symbols

One pattern for dealing with records involves `DuplicateRecordFields` and then using `generic-lens` with the `field` lens.

```
{-# language DuplicateRecordFields #-}

import Control.Lens (view)
import Data.Generics.Product (field)

data User = User
    { name :: Text
    , age :: Int
    , id :: Int
    }
    deriving stock Generic

data Dog = Dog
    { name :: Text
    , age :: Int
    }
    deriving stock Generic
```

Without `DuplicateRecordFields`, the above declaration would be forbidden, as `name` and `age` would have conflicting definitions. `DuplicateRecordFields` allows the definition and does some horrifying hacks to make actually using it work out. Unfortunately, it rarely does, when there's a conflict. You must provide type annotations at surprising times if a duplicate name is in scope. The UX around field selectors as accessor functions is pretty awful. The pattern that has

emerged at the work codebase that used it was to import types with qualified names:

```
import qualified Types as Foo(Foo(..))
import qualified Types as Bar(Bar(..))

idList foo bar = [Foo.id foo, Bar.id bar]
```

We can solve this with `generic-lens`, which uses type clases to get a satisfactory disambiguation.

The `field` lens takes a `TypeApplications` `Symbol` and returns a lens that works on *any* record that has a field with that name.

```
field         :: HasField sym s t a b    => Lens s t a b
field @"name" :: HasField "name" s t a b => Lens s t a b
```

With `getField :: HasField sym s s a a => s -> a`, you don't even need to know lenses to write code with this. The following line works exactly like you'd want it to:

```
idList foo bar = [getField @"id" foo, getField @"id" bar]
```

This suggests to me that there might be a good way to do this with symbols. Let's iterate.

```
class Record rec where
    type RecordField rec :: Symbol -> Type

    recordFieldLens :: Lens' rec (RecordField rec sym)

    fieldMap :: Map Text (SomeRecordField rec)
```

This is subtly different from the `GADT` variation, in that our record field is now the symbol - so we don't need a datatype for it.

```
instance Record User where
    type RecordField User = (??? :: Symbol -> Type)
```

GHC doesn't allow us to have a closed type family here. If we want one, we'll have to delegate.

```
type family UserFieldFn sym where

type family UserFieldFn sym where
    UserFieldFn "name" = Text
    UserFieldFn "age" = Int
    UserFieldFn "id" = Int

instance Record User where
    type RecordField User = UserFieldFn
```

GHC is unhappy with this, too.

```
src/Record/SymbolField.hs:61:8: error:
  • The type family 'UserFieldFn' should have 1 argument,
    but has been given none
  • In the type instance declaration for 'RecordField'
    In the instance declaration for 'Record User'
   |
61 |   type RecordField User = UserFieldFn
   |        ^^^^^^^^^^^
```

We're not allowed to partially apply type families. If we try to pass the `sym` parameter directly, we get another error:

```
instance Record User where
    type RecordField User sym = UserFieldFn sym
```

```
src/Record/SymbolField.hs:61:8: error:
  • Number of parameters must match family declaration; expected 1
  • In the type instance declaration for ‘RecordField’
    In the instance declaration for ‘Record User’
   |
61 |   type RecordField User sym = UserFieldFn sym
   |        ^^^^^^^^^^^
```

To fix this, we need to change the definition of `RecordField`.

```
- type RecordField rec      :: Symbol  -> Type
+ type RecordField rec (sym :: Symbol) :: Type
```

This compiles. Now we need to implement `recordFieldLens`. Here's a first try:

```
    recordFieldLens :: forall sym. Lens' User (RecordField User sym)
    recordFieldLens =
        field @sym
```

This *should* work. If `field` and `RecordField` could communicate each other's assumptions and proofs, then this *would* work. But they can't. We have to refine our implementation by case-matching on the string literal. This implementation gets us closer:

```
    recordFieldLens
        :: forall sym. KnownSymbol sym
        => Lens' User (RecordField User sym)
    recordFieldLens =
        case eqT @sym @"age" of
            Just Refl ->
                field @"age"
            Nothing ->
                case eqT @sym @"name" of
                    Just Refl ->
                        field @"name"
```

```
                Nothing ->
                    case eqT @sym @"id" of
                        Just Refl ->
                            field @"id"
                        Nothing ->
                            error "impossible"
```

GHC complained a lot about `KnownSymbol` not being inferrable, so we want to attach it. But this still gives us an error:

```
/home/matt/Projects/recupd/src/Record/SymbolField.hs:63:22: error:
  • Couldn't match type ‘UserFieldFn sym’ with ‘UserFieldFn sym0’
    Expected type: (RecordField User sym -> f (RecordField User sym))
                   -> User -> f User
      Actual type: (RecordField User sym0 -> f (RecordField User sym0))
                   -> User -> f User
    NB: ‘UserFieldFn’ is a non-injective type family
    The type variable ‘sym0’ is ambiguous
  • When checking that instance signature for ‘recordFieldLens’
      is more general than its signature in the class
      Instance sig: forall (sym :: Symbol).
                    KnownSymbol sym =>
                    Lens' User (RecordField User sym)
        Class sig: forall (sym :: Symbol).
                   KnownSymbol sym =>
                   Lens' User (RecordField User sym)
    In the instance declaration for ‘Record User’
   |
63 |   recordFieldLens
   |       :: forall sym. KnownSymbol sym
   |       => Lens' User (RecordField User sym)
   |          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

The tip `NB: 'UserFieldFn' is a non-injective type family` may be useful. An injective type family is one where the result type can determine the input type. However, we don't have an injective type family. `Int` is the result of both `UserFieldFn "age"` and `UserFieldFn "id"`.

Part of the problem here is the "open universe" expectation of `Symbol`s. Any string literal is a symbol. We have an `error "impossible"` case that is guaranteed because the type family `RecordField User sym` will get stuck if it isn't any of the symbols we've checked for, but this is uncheckable.

Hm. I think we will need a different approach. We had a type family in the GADT approach that pulled out all the types for a record, based on Generics. Maybe we can make a type-level list of labels to types, and use that instead.

```
type RecordFields rec = GRecordFields (Rep rec)

type family GRecordFields  rep where
    GRecordFields (D1 _ (C1 _ xs)) = EnumerateTypes xs

type family EnumerateTypes rep where
    EnumerateTypes (a :*: b) =
        EnumerateTypes a ++ EnumerateTypes b
    EnumerateTypes (S1 ('MetaSel ('Just sym) _ _ _) (Rec0 a)) =
        '[ '(sym, a)]
```

This also means we don't have to define it - we just need a `Generic` instance. A standard `Lookup` function and now we have this definition:

```
type family Lookup sym xs where
    Lookup sym ('(sym, a) ': _) =
        a
    Lookup sym (_ ': xs) =
        Lookup sym xs

class Record rec where
    recordFieldLens
        :: forall sym a. (a ~ Lookup sym (RecordFields rec))
        => Lens' rec a
```

We're still running into errors with this. But, something tells me this is unnecessary. After all, this signature is essentially `field'` from

`generic-lens`, which only requires a `Generic` instance. Can we implement `fieldMap` directly with `Generic`, piggy-backing on top of `generic-lens`? Let's see!

```
class FieldMap rec where
    fieldMap :: Map Text TypeRep

instance (Generic a, GFieldMap (Rep a)) => FieldMap a where
    fieldMap = gfieldMap @(Rep a)

class GFieldMap f where
    gfieldMap :: Map Text TypeRep

instance GFieldMap constr => GFieldMap (D1 meta constr) where
    gfieldMap = gfieldMap @constr

instance GFieldMap fields => GFieldMap (C1 meta fields) where
    gfieldMap = gfieldMap @fields

instance
    (Typeable typ, KnownSymbol fieldLabel)
  =>
    GFieldMap (S1 ('MetaSel (Just fieldLabel) a b c) (Rec0 typ))
  where
    gfieldMap =
        Map.singleton
            (Text.pack (symbolVal (Proxy @fieldLabel)))
            (typeRep (Proxy @typ))

instance (GFieldMap a, GFieldMap b) => GFieldMap (a :*: b) where
    gfieldMap = gfieldMap @a <> gfieldMap @b

-- λ> fieldMap @User
-- fromList [("age",Int),("id",Int),("name",Text)]
```

Well. Nice! That does what we want. Let's implement our `update` and `diff` method to see if it's as powerful as we want. `update` ends up being surprisingly simple:

```
data Update rec where
    SetField :: HasField' sym rec a => Proxy sym -> a -> Update rec

updateRecord :: [Update rec] -> rec -> rec
updateRecord upds rec =
    foldr
        (\(SetField (psym :: Proxy sym) newVal) ->
            set (field' @sym) newVal
        )
        rec
        upds

-- λ> updateUser
-- User {name = "Bob", age = 20, id = 4}
updateUser =
    updateRecord
        [ SetField (Proxy @"age") 20
        , SetField (Proxy @"name") "Bob"
        ]
        User { name = "Jane", age = 30, id = 4 }
```

However, I suspect we're going to have a difficult time with `diff`, because we're constructing the `Update` instead of consuming them. We'll need to know how to bring the relevant dictionaries into scope.

```
diffRecord :: forall rec. FieldMap rec => rec -> rec -> [Update rec]
diffRecord old new =
    foldr k [] (Map.toList $ fieldMap @rec)
  where
    k :: (Text, TypeRep) -> [Update rec] -> [Update rec]
    k (fieldLabel, fieldType) acc =
        undefined
```

This is our skeleton. I doubt we're going to have what we need from a `Text` and `TypeRep` to get that `HasField` dictionary in scope. So let's rework the `FieldMap` class to instead pack up that dictionary. We'll make a new `SomeField` type to carry it. We'll need to carry the original `rec` type in our type class, so there's a few changes there.

```haskell
data SomeField rec where
    SomeField :: Dict (HasField' sym rec a) -> SomeField rec

class FieldMap rec where
    fieldMap :: Map Text (SomeField rec)

instance (Generic a, GFieldMap a (Rep a)) => FieldMap a where
    fieldMap = gfieldMap @a @(Rep a)

class GFieldMap rec f where
    gfieldMap :: Map Text (SomeField rec)

instance GFieldMap rec constr => GFieldMap rec (D1 meta constr) where
    gfieldMap = gfieldMap @rec @constr

instance GFieldMap rec fields => GFieldMap rec (C1 meta fields) where
    gfieldMap = gfieldMap @rec @fields

instance
    ( Typeable typ
    , KnownSymbol fieldLabel
    , HasField' fieldLabel rec typ
    )
  =>
    GFieldMap rec (S1 ('MetaSel (Just fieldLabel) a b c) (Rec0 typ))
 where
    gfieldMap =
        Map.singleton
            (Text.pack (symbolVal (Proxy @fieldLabel)))
            (SomeField (Dict :: Dict (HasField' fieldLabel rec typ)))

instance
    (GFieldMap rec a, GFieldMap rec b)
  =>
    GFieldMap rec (a :*: b)
  where
    gfieldMap = gfieldMap @rec @a <> gfieldMap @rec @b
```

Diffing then looks like this:

```
diffRecord :: forall rec. FieldMap rec => rec -> rec -> [Update rec]
diffRecord old new =
    foldr k [] (fieldMap @rec)
  where
    k :: SomeField rec -> [Update rec] -> [Update rec]
    k (SomeField (dict :: Dict (HasField' sym rec a))) acc =
        let getter = view (field' @sym)
            newVal = getter new
        in
            if getter old == newVal
            then acc
            else SetField (Proxy @sym) newVal : acc
```

And we get the same issue as before - we need to bring the `Eq` dictionary into scope for this. The prior solution used a `FieldDict` type class, which relied on the `Field rec typ` data family. Can we replicate that?

```
class FieldDict c rec where
    getDict :: Field rec sym a -> Dict (c a)
```

We need a suitable `Field` type. And we'll probably want to rewrite `Update` and `SomeField` in terms of this new type. A `Field` in this formulation is kind of like a `Dict (HasField' sym rec a)`. So let's try that at first.

```
data Field rec sym a where
    Field :: Dict (HasField' sym rec a) -> Field rec sym a
```

Patching `SomeField` and `FieldMap` is straightforward. And writing `diffRecord` works mostly like the GADT approach:

```haskell
diffRecord
    :: forall rec. (FieldDict Eq rec, FieldMap rec)
    => rec -> rec -> [Update rec]
diffRecord old new =
    foldr k [] (fieldMap @rec)
  where
    k :: SomeField rec -> [Update rec] -> [Update rec]
    k x acc = case x of
     SomeField (field@(Field (Dict :: Dict (HasField' sym rec a)))) ->
         withFieldDict @Eq field $
             let oldVal = view (field' @sym) old
                 newVal = view (field' @sym) new
              in if oldVal == newVal
                 then acc
                 else SetField field newVal : acc
```

We need to type-annotate `Field` so we can bring into scope the `sym` type variables. But, this works!

Now we need to define `FieldDict` for our `User` type so we can actually do this. This may be tricky. Working with advanced type-level Haskell stuff often means that you can *define* something that looks amazing and compiles just fine, but you can't actually ever use it. I'm worried that may happen here.

```haskell
instance (All c (FieldTypes User)) => FieldDict c User where
    getDict (Field (Dict :: Dict (HasField' sym rec a))) =
        undefined
```

OK, this is our skeleton. In the `GADT` variation, we had a closed universe we could pattern match on, that'd give us the type. But we don't here. `HasField` has a functional dependency, so knowing `sym rec` tells us `a`. We know `rec` $\sim$ `User`, so we should be able to figure it out just based on the `sym` type variable.

We can't pattern match on `Symbol` directly here. It'd be nice to write:

```
instance (All c (FieldTypes User)) => FieldDict c User where
    getDict (Field (Dict :: Dict (HasField' "age" User Int))) =
        Dict
```

So, instead, we have to do something a bit nasty. Another `eqT` chain - like we did with `recordFieldLens` before we made it just `field'`.

```
instance (All c (FieldTypes User)) => FieldDict c User where
    getDict (Field (Dict :: Dict (HasField' sym User a))) =
        case eqT @'(sym, a) @'("age", Int) of
            Just Refl ->
                Dict :: Dict (c Int)
            Nothing ->
                undefined
```

This has GHC complaining about `Could not deduce 'KnownSymbol sym' arising from a use of 'eqT'`. Dang. We can either pack the `KnownSymbol` constraint into the GADT, or we can use the `FieldDict` trick again. Let's try the constructor. `KnownSymbol` is a pretty innocuous constraint. GHC then complains about `Typeable`. Once we add those to the `Field` constructor, we're set:

```
data Field rec sym a where
    Field
        :: (KnownSymbol sym, Typeable a)
        => Dict (HasField' sym rec a)
        -> Field rec sym a
```

This compiles now. We can finish the implementation:

```
instance (All c (FieldTypes User)) => FieldDict c User where
    getDict (Field (Dict :: Dict (HasField' sym User a))) =
        case eqT @'(sym, a) @'("age", Int) of
            Just Refl ->
                Dict :: Dict (c Int)
            Nothing ->
                case eqT @'(sym, a) @'("name", Text) of
                    Just Refl ->
                      Dict
                    Nothing ->
                        case eqT @'(sym, a) @'("id", Int) of
                            Just Refl ->
                                Dict
                            Nothing ->
                                error "Impossible"
```

This is unsatisfying - the `error "Impossible"` is just begging to get tripped somehow. Probably by adding a field to the `User` type. So this should be generated by Template Haskell to keep it safe.

Let's serialize and deserialize.

```
instance KnownSymbol sym => ToJSON (Field rec sym a) where
    toJSON _ = toJSON (symbolVal (Proxy @sym))

instance
    (HasField' sym rec a, Typeable a, KnownSymbol sym)
    => FromJSON (Field rec sym a) where
    parseJSON = withText "Field" $ \txt -> do
        case someSymbolVal (Text.unpack txt)  of
            SomeSymbol (Proxy :: Proxy sym') ->
                case eqT @sym @sym' of
                    Just Refl ->
                        pure $ Field Dict
                    Nothing ->
                        fail "Nope"
```

Nice. The rest of the code is pretty similar to the GADT approach:

```haskell
instance
    (forall sym a. ToJSON (Field rec sym a))
    => ToJSON (SomeField rec) where
    toJSON (SomeField field) = toJSON field

instance FieldMap rec => FromJSON (SomeField rec) where
    parseJSON = withText "SomeField" $ \txt ->
        case Map.lookup txt (fieldMap @rec) of
            Just fld ->
                pure fld
            Nothing ->
                fail "nope"

instance
    (FieldDict ToJSON rec, forall sym a. ToJSON (Field rec sym a))
    => ToJSON (Update rec) where
    toJSON (SetField field newVal) =
        withFieldDict @ToJSON field $
        object
            [ "field" .= field
            , "value" .= newVal
            ]

instance (FieldDict FromJSON rec, FieldMap rec) => FromJSON (Update rec) where
    parseJSON = withObject "Update" $ \o -> do
        field <- o .: "field"
        case field of
            SomeField (field :: Field rec sym a) ->
                withFieldDict @FromJSON field $
                    SetField field <$> o .: "value"

input = "[{\"field\":\"name\",\"value\":\"New Name\"}]"

route = do
    updates <- decode input
    pure $ updateRecord updates (User "Bob" 1 2 )
```

`route` evaluates to `Just (User { name = "New Name", age = 1, id = 2 })` - so all the machinery works here.

23.6 Compare and Constrast

The GADT field approach has some *really* nice properties - namely, that we have a closed universe of fields, and we can simply pattern match on them. This is most evident in the `FieldDict` instances:

```
-- GADT:
instance All c (FieldTypes User) => FieldDict c User where
    getDict c = case c of
        UserAge  -> Dict
        UserName -> Dict
        UserId   -> Dict

-- Symbol:
instance (All c (FieldTypes User)) => FieldDict c User where
    getDict (Field (Dict :: Dict (HasField' sym User a))) =
        case eqT @'(sym, a) @'("age", Int) of
            Just Refl ->
                Dict :: Dict (c Int)
            Nothing ->
                case eqT @'(sym, a) @'("name", Text) of
                    Just Refl ->
                        Dict
                    Nothing ->
                        case eqT @'(sym, a) @'("id", Int) of
                            Just Refl ->
                                Dict
                            Nothing ->
                                error "Impossible"
```

`getDict` is not only much uglier in the symbol version, it also has that nasty `error` case. It's *also* less efficient! It requires two runtime type

checks *per field* in the worst case. The GADT version is a simple lookup table.

On the other hand, in the Symbol version, we can derive Eq, Show, and Ord for `Field rec sym a`. We don't have to have a separate standalone deriving clause for each table.

The total code size is smaller for the GADT fields. My file for GADTField is about 230 lines, while Symbol fields are 260 lines. Not a huge difference.

It's more annoying to *use* the symbol fields. With the GADT, you can write the field and you're set. With the symbol, you need to write `Field Dict` and give it the right type.

```
userAge :: Field User "age" Int
userAge = Field Dict
```

However, you *can* write generic fields.

```
_age :: (HasField "age" rec a, Typeable a) => Field rec "age" a
_age = Field Dict
```

This is actually pretty neat. It means we can construct a generic Update too:

```
updateAge :: (HasField' "age" rec a, Typeable a) => a -> Update rec
updateAge newVal = SetField _age newVal
```

And with OverloadedLabels, we can even simplify further

```
instance (RecordField rec sym a) => IsLabel sym (Field rec sym a) where
    fromLabel = mkField @sym

updateAge newVal = SetField #age newVal
```

Okay, I have to admit - I'm excited about this. But I'm also really dissatisfied by the downsides! Can we recover polymorphic updates without the gross eqT staircasing to "pattern match" on fields?

23.7 Identify the Issue

In the GADT world, we have a closed universe of constructors representing our fields. GADTs are monomorphic, so we can't represent a field that is polymorphic. In the Symbol world, we have an open universe of symbols, and we can't pattern match on them.

What we want is a way to say:

```
generalize :: RecordField rec a -> Symbol
```

This is a type-level function. Which means it needs to be a type family (or class):

```
type Generalize :: RecordField rec a -> Symbol
```

But for `Generalize` to work, that means we need to have a *type* of *kind* `RecordField rec a`. But we can't promote a data constructor in a data family instance. So, we need to make it *not* a data family. What if the datatype is defined separately, and then the association is made through a type family?

23.8 Generalize a GADT

Let's change our definitions in the original GADT file.

```
class Record a where
    type Field a = (r:: Type -> Type) | r -> a

    recordFieldLens :: Field a b -> Lens' a b

    fieldLookup :: Map Text (SomeField a)

data UserField a where
    UserName :: UserField Text
```

```
    UserId :: UserField Int
    UserAge :: UserField Int

instance Record User where
    type Field User = UserField

  {- unchanged -}
```

GHC complains - a lot - about `Illegal type synonym family aplication 'Field r' in instance:`. Turns out we can't write something like:

```
instance (forall a. Show (Field r a)) => Show (SomeField r) where
    show (SomeField f) = show f
```

Well, we can't use a type family. What about a functional dependency?

23.9 Fundeps

```
data User = User
    { name :: Text
    , id   :: Int
    , age  :: Int
    }
    deriving stock (Show, Generic)

data UserField a where
    UserName :: UserField Text
    UserAge :: UserField Int
    UserId :: UserField Int

class
    Record (rec :: Type) (field :: Type -> Type)
        | rec -> field
        , field -> rec
```

```
    where
      fieldLens :: field a -> rec -> Lens' rec a

instance Record User UserField where
    fieldLens field rec =
        case field of
            UserName -> field' @"name"
            UserAge -> field' @"age"
            UserId -> field' @"id"
```

This works! Now, let's make that symbol association. We can't use functional dependencies in type family instances. There's just not a way to introduce the constraints.

So our association will *also* be a type class with a fundep.

```
class
    Record rec field
  =>
    FieldToSymbol rec field sym
      | rec field -> sym
```

This isn't quite what we want - we don't want `field :: Type -> Type` as the type variable, we want `fieldConstr :: field` - the promoted constructor. But how do we grab ahold of one of those?

We can just ask for it, apparently.

```
class
    (Record rec (field :: Type -> Type), KnownSymbol sym)
  =>
    FieldToSymbol rec (constr :: field a) sym
        | rec constr -> sym
        , rec sym -> constr
        , sym constr -> rec

instance FieldToSymbol User UserName "name"
instance FieldToSymbol User UserAge "age"
instance FieldToSymbol User UserId "id"
```

The functional dependencies will allow us to work backwards: if we know any two variables, we can recover the third.

Update looks a bit different:

```
data Update rec where
    SetField :: (Record rec field) => field a -> a -> Update rec

updateRec :: [Update rec] -> rec -> rec
updateRec upds rec = foldr k rec upds
  where
    k (SetField field newVal) =
        set (fieldLens field) newVal
```

We don't need the `Record rec field` constraint on `updateRec`, because we *actually* have the constraint packed inside the `Update` data constructor. We need to do that so we can make the association between `rec` and `field`.

Now, to serialize it, we're going to need the `FieldDict` stuff again. Except, we have to put the field type in the `FieldDict` class definition to make it available.

```
class (Record rec f) => FieldDict c rec f where
    getDict :: f a -> Dict (c a)

withFieldDict
    :: forall c rec a f r. FieldDict c rec f
    => f a -> (c a => r) -> r
withFieldDict l k =
    case getDict @c l of
        Dict -> k

instance All c (FieldTypes User) => FieldDict c User UserField where
    getDict c = case c of
        UserAge  -> Dict
        UserName -> Dict
        UserId   -> Dict
```

The `field` type parameter is becoming ubiquitous, which is annoying! Unfortunately, the next step reveals a problem:

```
instance
    (FieldDict ToJSON rec field, forall a. ToJSON (field a))
  =>
    ToJSON (Update rec)
  where
    toJSON (SetField field newVal) =
        withFieldDict @ToJSON field $
        object
            [ "field" .= field
            , "value" .= newVal
            ]
```

GHC complains here. It can't figure out that the `field` type mentioned in the context is the same as the `field` type variable that is packed inside the `SetField` constructor. Huh.

I guess we could backtrack and add `field` to the Update constructor. This fixes the issue, at the cost of another duplication of `Constr rec field`. The ergonomics are bad enough that I feel comfortable writing this approach off for now.

We can write `FromJSON`:

```
instance
    ( FieldDict FromJSON rec field
    , forall a. FromJSON (field a)
    , FieldMap rec field
    )
  =>
    FromJSON (Update rec field)
  where
    parseJSON = withObject "Update" $ \o -> do
        field <- o .: "field"
        case Map.lookup field (fieldMap @rec) of
            Nothing ->
```

```
13                 fail "field not found"
14             Just (Some a) ->
15                 withFieldDict @FromJSON a $
16                 SetField a <$> o .: "value"
```

So, this works. Cool.

It looks like, using the functional dependency, we have to include both types. This is redundant and a bit annoying, all so we can get an associated symbol.

23.10 More Class Please

Do we even really need the promoted constructor? I bet we don't.

Let's get back to the GADT field file:

```
1  class Record rec => SymbolToField sym rec a | rec sym -> a where
2      symbolToField :: Field rec a
3
4  instance SymbolToField "name" User Text where
5      symbolToField = UserName
6
7  instance SymbolToField "age" User Int where
8      symbolToField = UserAge
9
10 instance SymbolToField "id" User Int where
11     symbolToField = UserId
12
13 instance (SymbolToField sym rec a) => IsLabel sym (Field rec a) where
14     fromLabel = symbolToField @sym
15
16 updateAge :: (Num a, SymbolToField "age" rec a) => UpdateRecord rec
17 updateAge = SetField #age 2
```

See, we can even implement `OverloadedLabels` for this. Let's just go with it.

23.11 Refining the Class

`tabulateRecord`

Ollie Charles referred to "representable functors" on Twitter[4], and it seemed prudent to add this to the class.

```
tabulateRecord :: (forall ty. Field rec ty -> ty) -> rec
```

This method serves as evidence that you can construct a `rec` by specifying values for each field. It would be implemented like this:

```
tabulateRecord fromField =
    User
        { name = fromField UserName
        , age = fromField UserAge
        , id = fromField UserId
        }
```

`recordFieldLabel`

All the stuff with `FieldMap` is kind of silly. Let's scrap it and simplify. What we really want is a function from `Field rec a` to `Text`.

```
recordFieldLabel :: Field rec a -> Text
```

Adding this to the class and changing `fieldMap` to `allFields` simplifies matters greatly.

All told, the class ends up looking like this:

[4] https://twitter.com/acid2/status/1312054202382852096

```haskell
data User = User
    { name :: String
    , age :: Int
    , id :: Int
    }

class Record User where
    data Field User a where
        UserName :: Field User String
        UserAge :: Field Age Int
        UserId :: Field Age Int

    tabulateRecord fromField =
        User
            { name = fromField UserName
            , age = fromField UserAge
            , id = fromField UserId
            }

    recordFieldLens field =
        case field of
            UserName -> lens name (\u n -> u { name = n })
            UserAge -> lens age (\u a -> u { age = a })
            UserId -> lens id (\u a -> u { id = a })

    allFields =
        [SomeField UserName, SomeField UserAge, SomeField UserId]

    recordFieldLabel field =
        case field of
            UserName -> "name"
            UserAge -> "age"
            UserId -> "id"

instance (c Int, c String) => FieldDict c User where
    getFieldDict field =
        case field of
```

```
            UserName -> Dict
            UserAge -> Dict
            UserId -> Dict

instance SymbolToField "name" User String where symbolToField = UserName
instance SymbolToField "age" User String where symbolToField = UserAge
instance SymbolToField "id" User String where symbolToField = UserId
```

That's it! And now you can access all the functionality in the library.

23.12 Template Haskell

It's not a lot of boilerplate. But it *is* boilerplate. It'd be nicer to not need to write it. And it's all so so so mechanical. Let's generate the code with `TemplateHaskell`.

Some folks like to think real hard and write beautiful code. Sometimes, I like to do that, but more often, I'll just jump right in and write whatever garbage I need. So let's start.

```
mkRecord :: Name -> DecsQ
```

This is the most common signature when getting started with a Template Haskell function. Give me a `Name` and I'll give you a bunch of declarations. `DecsQ` is a synonym for `Q [Dec]`. I like to start at both ends - I know I'm going to `reify` the `Name` into something useful, and I also know I am going to be generating an instance fo `Record`, `FieldDict`, and several `SymbolToField` instances.

```
mkRecord :: Name -> DecsQ
mkRecord u = do
    ty <- reify u
    ???
    pure $
        [ recordInstance
        , fieldDictInstance
        ]
        ++ symbolToFieldInstances
```

Let's start from the demand side. We need a `recordInstance :: Dec`. So let's look up the constructors for Dec[5]. `InstanceD` is what we need.

```
    let recordInstance =
            InstanceD
                _maybeOverlap
                _cxt
                _type
                _recordDecs
```

We shouldn't need any overlap or context information, since instances should all be monomorphic. `Nothing` and `[]` can go in there. The `typ :: Type` is the whole type of the instance. So if we're making `instance Record User`, that's `Record User`.

```
    let recordInstance =
            InstanceD
                Nothing
                []
                (ConT ''Record `AppT` ConT typeName)
                []
```

So we know we're going to need `typeName` to make this work. Let's dig that out of `reify`.

[5] https://hackage.haskell.org/package/template-haskell-2.16.0.0/docs/Language-Haskell-TH.html#t:Dec

```
    typeName <-
        case ty of
            TyConI dec ->
                case dec of
                    DataD _cxt name [] _mkind [con] _derivs ->
                        pure name
                    NewtypeD _cxt name [] _mkind con _derivs ->
                        pure name
                    _ ->
                        fail "unsupported data structure"
            _ ->
              fail "unsupported type"
```

We're only going to support data declarations (with a single constructor) and newtypes. Anything else can't really be called a record. We could improve the error messages, but they're fine for now. Let's get back to our `Record` instance. We're going to start filling in the `[Dec]` it needs.

```
    [ fieldDataFamilyInstance
    , recordFieldLensDec
    , mkAllFields
    , mkTabulateRecord
    , mkRecordFieldLabel
    ]
```

Let's start with the data family instance. That'll certainly demand a lot of good stuff.

```
    fieldDataFamilyInstance =
        DataInstD cxt maybeTyvars ty maybeKind cons derivs
```

As above, `cxt = []`, there shouldn't be any type variable binders, there shouldn't be a kind signature, and we also shouldn't have any deriving clauses. This means we can focus on the `ty` and `cons`.

```
        DataInstD
            []
            Nothing
            (ConT ''Field `AppT` ConT typeName `AppT` VarT (mkName "a"))
            Nothing
            fieldConstrs
            []
```

Technically, it's bad hygiene to use `mkName "a"` here. To be totally proper, we'd want to use `tyVar <- newName "a"`. `newName` generates a totally fresh name that is guaranteed to not have any collisions. `mkName` generates a name that literally is what you pass. It's safe to use in cases where name overlap doesn't matter at all. Since we're introducing a type variable name here, and we won't even be referring to it again, it's safe. Now we need to figure out our `fieldConstructors`. They're going to be a `GadtC` constructor.

```
        fieldConstrs =
            map mkConstr fields
        mkConstr =
            GadtC (_ :: [Name]) (_ :: [BangType]) (_ :: Type)
```

`GadtC` takes a list of `[Name]` because you can define multiple constructors that share a signature. We know we're going to be iterating over a list of (`fields :: [???]`) to make this work out. Finally, we need to assign the type of the constructor at the end.

```
        mkConstr (fieldName, typ) =
            GadtC [fieldName] []
                (ConT ''Field `AppT` ConT typeName `AppT` typ)
```

We need the list to have shape `[(Name, Type)]` for this to work. Which means we need the list of fields on the record, along with their types. That information is stored on the `Con` for a record, as `RecC Name [VarBangType]`. So now we need to modify our match on `ty` to bring the record constructor into scope as well.

```
    (typeName, con) <-
        case ty of
            TyConI dec ->
                case dec of
                    DataD _cxt name [] _mkind [con] _derivs ->
                        pure (name, con)
                    NewtypeD _cxt name [] _mkind con _derivs ->
                        pure (name, con)
        {- snip -}

    names'types <-
        case con of
            RecC conName varBangTypes ->
                pure $ map (\(n, _b, t) -> (n, t))
            _ -> fail "unsupported constructor"
```

The `Name` given is the field name, like `name` or `age`. We want `UserName`. But we also want to ignore the type name prefix, in case our users have written something like `userName` for the field.

```
    let mkConstrFieldName fieldName =
            mkName
                $ nameBase typeName
                <> upperFirst (nameBase (stripTypeName fieldName))
        stripTypeName n =
            maybe n (mkName . lowerFirst) $
                List.stripPrefix
                    (lowerFirst (nameBase typeName))
                    (nameBase n)
```

If we can't strip the type name prefix, then we just use the name as-is. We can finally define `fieldConstructors`:

```
      fieldConstructors =
          map (\ (n, t) -> (mkConstrFieldName n, t)) names'types
```

And, with that, our data family instance is complete. `mkAllFields` should be a fun next step. As a plain value, we can use `ValD` to construct it. This accepts a `Pat` pattern, a `Body` definition, and a `[Dec]` that would act as a `where` clause. We're looking for a value like `[SomeField UserName, SomeField UserAge]`. Abstractly, it's `map (\constr -> SomeField constr) constrs`, but with the usual TemplateHaskell noise.

```
    mkAllFields =
        ValD (VarP 'allFields) (NormalB $ ListE $ someFields) []
    someFields =
        map (AppE (ConE ''SomeField) . ConE . fst) fieldConstructors
```

Next up is `mkRecordFieldLabel`. Here we want to make a `Text` for each field. We're going to work with a `case` expression here, which uses the `CaseE` constructor.

```
mkRecordFieldLabel <- do
    fieldName <- newName "fieldName"
    let body =
          CaseE (VarE fieldName)  $
              flip map names'types $ \(n, _) ->
                  let constrFieldName =
                          mkConstrFieldName n
                      pat =
                          ConP constrFieldName []
                      bdy =
                          AppE (VarE 'Text.pack)
                          $ LitE $ StringL
                          $ nameBase $ stripTypeName n
                   in
                      Match pat (NormalB bdy)  []
    pure $
        FunD 'recordFieldLabel
            [ Clause [VarP fieldName] (NormalB body) []
            ]
```

It can be nice to use do blocks like this to preserve variable scoping. `fieldName` is only in scope in this do block, so I don't need to worry about the name escaping.

There's a bit of redundancy here - we're reusing the `names'types` variable because we need the original field name. It may be fruitful to convert `names'types` into a list of a custom datatype that carries the various things we care about from the field definition - the constructor name, field name, type-stripped field name, field type, etc. But let's ignore that for now and just keep chugging.

`recordFieldLens` is going to be a little trickier. We're going to match on each constructor of the GADT and make a lens for it.

```
    let recordFieldLensDec =
            FunD 'recordFieldLens [fieldLensClause]
```

`fieldLensClause` is a bit more clumsy to write, since we need fresh variable names for the lambdas. So we'll kinda staircase the scopes. First we'll worry about making our pattern matches.

```
    fieldLensClause <- do
        let mkMatch (fieldName, _typ) = do
              recVar <- newName "rec"
              newVal <- newName "newVal"
              let constrPattern =
                      ConP (mkConstrFieldName fieldName) []
                  expr =
                      VarE 'lens
                      `AppE` VarE fieldName  -- getter
                      `AppE` setter
                  setter =
                      LamE
                          [VarP recVar, VarP newVal]
                          (RecUpdE
                            (VarE recVar)
                            [(fieldName, VarE newVal)]
                          )
              pure $ Match constrPattern (NormalB expr) []
```

To make a `Match`, we need the `constrPattern` - which is `mkConstrFieldName fieldName` applied to the relevant `ConP` constructor. There are no variables to match on, so we provide the empty list. Then we need our expression. In Haskell, it's `lens getter setter`, where `setter` is some record update on the field. In `TemplateHaskell`, we need the `VarE` and `AppE` noise to make it work out. The `setter` is the interesting bit. We construct a lambda (`LamE`) which takes a list of patterns. We'll use those fresh variables we made up to ensure we don't shadow any names accidentally. Then, we begin making the `RecUpd :: Exp -> [(Name, Exp)]`.

Fortunately, the expression isn't complicated, so the lambda can be pretty simple.

```
1       arg <- newName "field"
2       body <- CaseE (VarE arg) <$> traverse mkMatch names'types
3       pure $ Clause [VarP arg] (NormalB body) []
```

The answer, as usual, is `traverse`.

I haven't used many QuasiQuotes. I don't tend to like them. They're often fragile and they don't work, and I can never predict what'll break. Wouldn't it have been nice if we could have written the lambda expression as:

```
1       [e| (\ rec newVal -> rec { $(fieldName) = newVal }) |]
```

Alas, this gives us a parse error. Even TemplateHaskell doesn't think record fields are first class! Fortunately, we can use them to make our `SymbolToField` instances well enough

```
    symbolToFieldInstances <-
        fmap concat $ for names'types $ \(fieldName, typ) -> do
            let sym = litT (strTyLit (nameBase fieldName))
                conTypeName = conT typeName
            [d|
              instance SymbolToField $(sym) $(conTypeName) $(pure typ)
                where
                  symbolToField =
                      $(conE (mkConstrFieldName fieldName))
              |]
```

Most of the constructors in the Template Haskell library have corresponding "lifted" constructors that have a lowercase initial letter. So `ConT :: Name -> Type`, and `conT :: Name -> Q Type`.

When QuasiQuoters work, they're nice. But they just break often enough that I rarely reach for them first, unless I know for sure that it'll work. I'm certain a good bit of the code I've written so far could be cleaned up with QuasiQuotes.

Exercise:

Write the code for `tabulateRecord`'s TemplateHaskell definition. If you're unsure how to proceed, the code is available on GitHub[6].

Remember, the structure looks like:

```
tabulateRecord fromField =
    User
        { name = fromField UserName
        , age = fromField UserAge
        }
```

So you'll want to iterate over the record fields to create a `RecConE`.

[6]https://github.com/parsonsmatt/prairie/blob/4ef22f23a30774df9698fcd0009d2104d228dfad/src/Prairie/TH.hs

23.13 Conclusion

And, with that, I've explored a few approaches to an advanced Haskell library feature. I picked one and implemented it. After picking up a lot of boilerplate, I wrote a `TemplateHaskell` helper to make it all nice and easy.