Deutsche Ausgabe

Production Haskell (Deutsche Ausgabe)

Erfolg in der Industrie mit Haskell

Matt Parsons

Dieses Buch wird verkauft unter http://leanpub.com/production-haskell-de

Diese Version wurde veröffentlicht am 2024-08-13

Inhaltsverzeichnis

Einführung

Ein Meinungsfreudiger Reiseführer

Also, du hast Haskell gelernt. Du hast deinen Freunden etwas über Monaden beigebracht, hast einige Einsteiger-Lehrbücher durchgearbeitet und vielleicht mit einigen Open-Source-Projekten gespielt. Jetzt, wo du einen Vorgeschmack bekommen hast, willst du mehr: Du möchtest eine Anwendung in Haskell aus Spaß schreiben! Vielleicht möchtest du Haskell bei der Arbeit einsetzen!

Du setzt dich an deinen Computer und bist ratlos.

> Wie schafft es eigentlich jemand, mit dieser Sprache etwas zu erledigen?

Das ist eine häufige Frage.

Haskell hat immer eine Vielzahl an hochwertigen Lernmaterialien für fortgeschrittene Teile der Sprache genossen, wenn man sich nicht vor akademischen Artikeln scheut. In den letzten fünf Jahren haben viele Menschen fantastische Ressourcen für Anfänger geschaffen. Jedoch gibt es nicht viele Ressourcen, um die Sprache in der Produktion zu nutzen. Die Haskell-Ökosysteme können schwer zu navigieren sein. Es gibt viele Ressourcen unterschiedlicher Qualität mit unklaren Zielen und Werten. Den richtigen Rat zu finden, ist fast ebenso herausfordernd wie ihn überhaupt erst zu entdecken.

Haskell ist ein äußerst vielfältiges Umfeld. Es gibt viele regionale Gruppen: das Vereinigte Königreich, Skandinavien, das Festland Europas, Russland, die USA, Japan, China und Indien verfügen alle über florierende Haskell-Ökosysteme mit interessanten Dialekten und Unterschieden in Sitten und Gebräuchen.

Menschen kommen mit vielen verschiedenen Hintergründen zu Haskell. Einige haben Haskell erst spät in ihrer Karriere gelernt und hatten vorher eine lange Karriere mit Java, Scala oder C#. Einige kamen von dynamisch typisierten Sprachen wie LISP oder Ruby zu Haskell. Einige begannen früh in ihrer Programmierkarriere mit Haskell und nutzen es als Vergleichsbasis. Manche verwenden Haskell hauptsächlich in der akademischen Forschung, während andere es hauptsächlich in industriellen Anwendungen nutzen. Einige sind Hobbyisten und schreiben einfach gerne aus Spaß Haskell!

Dieses Buch richtet sich an Personen, die Haskell in der Industrie schreiben möchten. Die Kompromisse und Einschränkungen, denen industrielle Programmierer gegenüberstehen, unterscheiden sich von denen akademischer oder Hobby-Programmierer. Dieses Buch behandelt nicht nur technische Aspekte der Haskell-Sprache, sondern auch soziale und ingenieurtechnische Anliegen, die nicht "wirklich" mit Haskell zu tun haben.

Ein Teil dieses Buches wird objektiv sein. Ich werde dir beibringen, wie man einige interessante Techniken und Ideen nutzt, um die Entwicklung mit Haskell produktiver zu machen. Wir werden über Template Haskell, Typ-Level-Programmierung und andere spannende Themen lernen.

Jedoch ist dieses Buch größtenteils von Natur aus subjektiv. Da Haskell so vielen Ökosystemen dient, ist es unerlässlich zu erkennen, für welches Ökosystem etwas gedacht ist. Mehr als nur Rezepte zu geben - "Diese Bibliothek ist produktionsreif! Das ist ein Spielzeug!" - hoffe ich, meinen Denkprozess zu zeigen und dir zu ermöglichen, eigene Urteile zu fällen.

Letztendlich ist dies ein Buch über die soziale Realität des Software-Engineerings in einer Nischensprache.

Nach dem Lesen dieses Buches solltest du dich wohlfühlen:

- Große Softwareprojekte in Haskell zu schreiben
- Konkurrenzfähige Bibliotheken und Techniken zu bewerten
- Material von einer Vielzahl von Haskell-Nutzern produktiv zu lesen

Über den Autor

Ich bin Matt Parsons.

Ich begann im Januar 2014 mit der Informatik 101 an der Universität von Georgia mit dem Programmieren. Zu der Zeit arbeitete ich in der IT-Abteilung, installierte Windows und behebt Druckerprobleme. Mein Vorgesetzter mochte mich nicht und machte deutlich, dass er mich bei jeder Gelegenheit unter den Bus werfen würde. Ich war verzweifelt auf der Suche nach einer neuen Karriere und hatte eine Menge College-Credits von einem gescheiterten Versuch eines Biochemie-Studiums. Informatik schien die beste Option zu sein, um aus diesem Job herauszukommen.

CS101 brachte mir die Grundlagen der Java-Programmierung bei. Keines der lokalen Startups oder Programmierer nutzten oder mochten Java, also fragte ich, was ich lernen sollte, um schnell einen Job zu bekommen. JavaScript und Ruby waren die

Top-Optionen. Ich lernte JavaScript in jenem Sommer mit dem ausgezeichneten Buch Eloquent JavaScript[1], das Kapitel über funktionale Programmierung und objektorientierte Programmierung hatte. Ich fand das Kapitel über funktionale Programmierung intuitiver, also merkte ich mir, die funktionalste Sprache zu lernen, die ich finden konnte. Einige Monate später begann ich mit dem Lernen von Haskell und Ruby on Rails.

Ich kündigte meinen IT-Job im Dezember 2014, um Vollzeitstudent zu werden. Mitte Januar hatte ich ein Rails-Praktikum bei einem lokalen Startup - so viel zum Vollzeitstudium.

Mein Gehirn nahm Haskell schnell auf. Ich hatte kaum begonnen, imperativ und objektorientiert zu programmieren, sodass die schwierige Neuheit, neues Fachjargon und Konzepte zu lernen, erwartet wurde. Die Ruby-Sprache war bemerkenswert empfänglich für die Implementierung von Haskell-Ideen, obwohl die Community nicht so begeistert war. Die Konzepte, die ich in Haskell lernte, halfen mir, in Ruby leicht testbaren und zuverlässigen Code zu schreiben.

Im August 2015 begann ich ein Haskell-Praktikum, wo ich Webanwendungen und schnelle Parser bauen durfte. Ich durfte Haskell in meinem Kurs für Künstliche Intelligenz verwenden. In meinem letzten Semester am College nutzte ich Haskell in meiner Bachelorarbeit, um die Verbindung zwischen Kategorientheorie, Modallogik und verteilten Systemen zu untersuchen.

Ich habe das Glück, diese Möglichkeiten gehabt zu haben, da sie mich für den Erfolg mit Haskell vorbereitet haben. Mein erster Job nach dem Studium bestand darin, PHP-Anwendungen in Neuentwicklungen mit Haskell zu konvertieren, und seitdem arbeite ich Vollzeit mit Haskell. Ich habe in verschiedenen Kontexten gearbeitet: Einem Startup, das nicht zu 100% von Haskell überzeugt war, einem größeren Unternehmen, das von Haskell überzeugt war, aber mit sozialen und technischen Schwierigkeiten eines riesigen Code-Bestands und Entwicklungsteams zu kämpfen hatte, und einem Startup, das von Haskell überzeugt war und auf Wachstum hinarbeitete. Ich trage auch zu vielen Open-Source-Projekten bei und bin mit den meisten Ökosystemen vertraut. Alles in allem habe ich mit Millionen von Zeilen Haskell-Code gearbeitet.

Ich habe gesehen, wie Haskell scheitert. Ich habe gesehen, wie es erfolgreich ist. Ich möchte Ihnen helfen, mit Haskell erfolgreich zu sein.

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

Prinzipien

Dieser Abschnitt dokumentiert die Leitprinzipien für das Buch. Ich habe festgestellt, dass diese Kernideen wichtig sind, um erfolgreiche Haskell-Projekte zu verwalten.

- Komplexität
- Neuheit
- Kohäsion
- Empathie

Komplexität

Das Management von Komplexität ist die wichtigste und schwierigste Aufgabe bei Haskell-Projekten.

Das ist so wichtig, dass ich daraus das erste Prinzip mache, und ich sage es sogar zweimal:

Das Management von Komplexität ist die wichtigste und schwierigste Aufgabe bei Haskell-Projekten.

Genauso wie es "technische Schulden" gibt, gibt es ein "Komplexitätsbudget." Sie verwenden Ihr Komplexitätsbudget, indem Sie ausgefallene Technologien nutzen, und Sie verwenden Ihr Neuheitsbudget, indem Sie neue oder interessante oder andere Technologien auswählen. Sie können Ihr Budget erhöhen, indem Sie erfahrene Ingenieure und Berater einstellen. Im Gegensatz zu technischen Schulden haben diese Budgets einen realen und direkten Einfluss auf Ihr tatsächliches finanzielles Budget.

Komplexität ist ein Fat Tail

Es ist leicht, die Übel der Komplexität zu beklagen, wenn man nur über Komplexität spricht. Aber wir nehmen Komplexität nicht für sich allein auf. Codebasen übernehmen kleine Funktionen, clevere Tricks und Sicherheitsfunktionen langsam. Mit der Zeit akkumulieren sie sich zu hochkomplexen Systemen, die schwer zu verstehen sind. Das passiert sogar, wenn jedes zusätzliche Stück Komplexität scheinbar sein eigenes Gewicht trägt!

Wie passiert das?

Eine Codeeinheit steht nicht allein. Sie muss sich auf den Code beziehen, der sie verwendet, sowie auf den Code, den sie aufruft. Es sei denn, sie ist sorgfältig versteckt, muss die durch eine Codeeinheit eingeführte Komplexität von allen Codes, die sie verwenden, behandelt werden. Wir müssen die Beziehungen zwischen den Codeeinheiten sowie die Einheiten selbst berücksichtigen. Deshalb fügen sich zwei Stücke Komplexität nicht einfach zusammen - sie multiplizieren sich! Leider multiplizieren sich die Vorteile der Komplexität nicht - sie sind normalerweise nur additiv.

Ein System, das schwer zu verstehen ist, ist schwer zu bearbeiten. Schließlich kann ein System so schwer verständlich werden, dass es zu einer Blackbox wird, mit der man praktisch nicht arbeiten kann. In diesem Fall ist ein kompletter Neuanfang oft die angenehmste Option für das Projekt. Das tötet oft das Projekt, wenn nicht sogar das Unternehmen. Das müssen wir vermeiden.

Komplexität malt uns in eine Ecke. Sicherheitsfunktionen besonders schränken unsere Optionen ein und reduzieren die Flexibilität des Systems. Schließlich besteht der ganze Sinn von "Programmsicherheit" darin, ungültige Programme zu verbieten. Wenn sich die Anforderungen ändern und sich die Vorstellung eines "ungültigen Programms" ebenfalls ändert, können die Sicherheitsfunktionen hinderlich werden. Komplexität birgt ein Risiko bei jeder Änderung der Codebasis.

Die Kosten oder die Zeit, die erforderlich sind, um ein komplexes System zu ändern, vorherzusagen, ist schwierig. Die Varianz dieser Vorhersagen wächst mit der Komplexität des Systems. Aufgaben, die einfach erscheinen, könnten extrem schwierig werden, und es wird ebenso problematisch sein, Schätzungen über die verbleibende Zeit zur Fertigstellung einer Aufgabe abzugeben.

In der Messung betrachten wir Genauigkeit und Präzision als separate Konzepte. Eine präzise Messung oder Vorhersage ist hochgradig konsistent - für eine gegebene Wahrheit wird sie eine ähnliche Messung konsistent melden. Eine genaue Messung oder Vorhersage liegt nahe an der tatsächlichen Wahrheit. Wir können uns Vorhersagen vorstellen, die präzise, aber nicht genau sind, sowie genaue, aber nicht präzise.

Komplexe Systeme verschlechtern sowohl die Präzision als auch die Genauigkeit von Vorhersagen. Präzision ist das ernstere Problem. Unternehmen verlassen sich auf Prognosen und Regelmäßigkeit, um Pläne zu machen. Wenn die Vorhersage unpräzise wird, wird es schwieriger, das Geschäft aufrechtzuerhalten.

Ein hochkomplexes System ist dann eher katastrophalen Ausfällen ausgesetzt als ein einfaches System. Das gilt selbst dann, wenn das System in jeder anderen Hinsicht besser ist! Stellen Sie sich zwei Autos vor - eines fährt 100 Meilen pro Gallone, kann mit

200 Meilen pro Stunde fahren und nimmt Kurven wie ein Traum. Das andere ist viel schlechter: nur 40 Meilen pro Gallone und eine Höchstgeschwindigkeit von 60 Meilen pro Stunde. Natürlich gibt es einen Haken: Das erste Auto wird relativ oft und zufällig ausfallen, und es kann bis zu einer Woche dauern, es zu reparieren. Das zweite Auto ist nicht perfekt, aber es fällt einmal im Jahr zuverlässig aus, und es dauert immer einen Tag, es zu reparieren.

Wenn Sie ein Auto brauchen, um zur Arbeit zu kommen, und nur ein Auto haben können, dann wollen Sie das zweite Auto. Sicher, das erste Auto kann schneller fahren und kostet weniger, aber die wesentliche Qualität, die Sie bei einem Pendlerfahrzeug brauchen, ist Zuverlässigkeit.

Komplexität mindern

Wir kehren zu einem häufigen Thema in diesem Buch zurück: die Vielfalt der Ökosysteme. Können Sie sich eine Gruppe vorstellen, die das erste Auto bevorzugen würde? Hobbyisten! Und professionelle Rennfahrer, die Ersatzautos haben können! Und Ingenieure, die fortschrittliche Automobiltechnologie studieren!

Haskell dient in erster Linie der akademischen Forschung als funktionale Programmiersprache. Die industrielle Nutzung ist ein sekundäres Anliegen. Viele Haskeller sind auch Hobbyisten, die es hauptsächlich zum Spaß nutzen. Dies sind alles gültige Verwendungen von Haskell, aber akademische und hobbyistische Praktiker glauben normalerweise, dass ihre Techniken für die Industrie geeignet sind. Leider funktionieren sie oft nicht so gut, wie sie hoffen.

Wenn Sie in Haskell nach zwei Autos fragen, werden Sie oft hören, dass Leute das schnelle Auto empfehlen. Versuchen Sie, mehr über die betreffenden Personen zu erfahren. Sind sie tatsächlich das schnelle Auto gefahren? Als Pendler? Sind sie dafür verantwortlich, es zu reparieren, wenn es kaputt geht?

Leute werden Ihnen fantastische und wunderbare Lösungen für Ihre Probleme empfehlen. Genießen Sie diese mit Vorsicht. Es gibt nur wenige Codebasen in Haskell, bei denen eine Technik umfassend untersucht wurde. Nur wenige dieser Untersuchungen gehen in den allgemeinen Wissensschatz ein.

Der beste Weg, den Erfolg Ihres Haskell-Projekts zu garantieren, besteht darin, die Komplexität zu bewältigen und die einfachste mögliche Lösung zu bevorzugen.

Warum ist das schwierig?

Haskell selektiert für eine bestimmte Art von Person.

Hobby- und Industrieprogrammierer folgen einem Weg. Wenn Sie keine Freude an Neuheit und Schwierigkeit haben, werden Sie es schwer haben, eine so neuartige und komplexe Sprache überhaupt zu lernen. Die meisten Haskell-Entwickler lernen Haskell in ihrer Freizeit, indem sie persönliche Projekte verfolgen oder intellektuelles Wachstum anstreben. Die Lernmaterialien von Haskell, die sich in letzter Zeit stark verbessert haben, sind immer noch so schwierig, dass nur entschlossene Menschen mit einer großen Toleranz für Neuheit und Frustration es schaffen.

Akademische Programmierer tendieren dazu, einen anderen Weg zu folgen. Viele von ihnen lernen Haskell in Universitätskursen, mit einem Professor, Lehrassistenten und anderen Kommilitonen, die Unterstützung bieten. Sie verfolgen ihre Forschung und Studien, um die Grenzen von Programmiersprachen und der Informatik zu erweitern. Viel akademische Arbeit ist eher ein Machbarkeitsnachweis als eine robuste industrielle Implementierung. Die resultierenden Arbeiten sind oft ziemlich komplex und fragil.

Die Programmiersprache Haskell ist dafür auch teilweise verantwortlich. Starke Typen und funktionale Programmierung können Schutz bieten. Programmierer fühlen sich oft viel selbstbewusster, wenn sie mit diesen Sicherheiten arbeiten. Dieses Selbstvertrauen ermöglicht es den Entwicklern, nach größeren und komplexeren Lösungen zu streben.

Infolgedessen tendieren ein Großteil des Ökosystems und der Gemeinschaft dazu, weniger abgeneigt gegenüber Komplexität und Neuheit zu sein. Hobbyisten und Akademiker werden auch von einem anderen Satz von Anreizen angetrieben als Industrieprogrammierer. Komplexität und Neuheit sammeln sich schnell in Haskell-Projekten an, wenn man sie nicht aggressiv kontrolliert.

Neuheit

Neuheit ist die zweite Gefahr in einem Haskell-Projekt. Sie ist fast so gefährlich wie die Komplexität, und tatsächlich ist das Problem mit der Komplexität oft die Neuheit, die damit einhergeht.

Im Gegensatz zu einem Komplexitätsbudget, das durch den Einsatz von Geld für Expertise erhöht werden kann, ist Ihr Neuheitsbudget schwerer zu erhöhen. Neue Techniken sind in der Regel schwer einzustellen. Sie sind schwer zu erlernen und zu dokumentieren.

Wenn Sie Haskell als Anwendungssprache ausgewählt haben, haben Sie bereits einen Großteil Ihrer Komplexitäts- und Neuheitsbudgets ausgegeben. Sie werden wahrscheinlich grundlegende Bibliotheken für Ihre Domäne schreiben oder pflegen müssen - daher müssen Sie Bibliotheksingenieure einstellen (oder sich mit der Vergabe dieser Arbeiten

wohlfühlen). Sie müssen ein Verständnis für GHC entwickeln - sowohl den Compiler als auch das Laufzeitsystem. Expertise (und Beratung) zu diesen Themen ist schwieriger zu finden als das Tuning der JVM oder des CLR. Vieles dieses Verständnisses kann über die Prototypenphase hinausgeschoben werden - die Bibliothekssituation von Haskell ist für viele Domänen gut genug, und die Leistung von GHC ist von Haus aus so gut genug, dass Sie prototypen können und damit zurechtkommen.

Da Haskell ein großer Posten im Komplexitäts-/Neuheitsbudget ist, ist es wichtig, bei den restlichen Komponenten auf kostengünstige Entscheidungen zu setzen. Probieren Sie keine ausgefallene neue Graphdatenbank für Ihre App aus - bleiben Sie bei Postgres. Probieren Sie vor allem keine ausgefallenen In-Haskell-Datenbanken aus! Das Festhalten an Industriestandards und gängiger Technologie eröffnet ein breiteres und vielfältigeres Feld an Ingenieuren zur Einstellung.

Jede Anforderung, die Sie in Ihrer Stellenausschreibung für einen Entwickler stellen, erhöht die Schwierigkeit und die Kosten der Einstellung. Haskell ist eine seltene Fähigkeit. Jahre der Erfahrung mit Haskell im Produktionsbetrieb mit ausgefallenen Bibliotheken und Techniken sind noch seltener. Die Produktivitätsvorteile von Haskell sind real, aber diese gelten nur beim Schreiben, Lesen und Verstehen von Code. Dokumentation, Anforderungen und Qualitätssicherung nehmen genauso viel Zeit in Anspruch wie in anderen Sprachen.

Kohäsion

Zwei Ingenieure streiten sich wieder einmal über ihre persönlichen Vorlieben. Sie seufzen und bewerten die Argumente. Beide Lösungen sind in Ordnung. Sicher, sie haben Kompromisse, aber das hat alles.

Haskell-Ingenieure sind ungewöhnlich meinungsstark, sogar für Software-Ingenieure. Haskell selbst ist stark meinungsstark - rein funktionale Programmierung ist das einzige Paradigma, das die Sprache direkt unterstützt. Softwareentwickler, die funktionale Programmierung lernen möchten und nicht allzu meinungsstark darüber sind, lernen typischerweise mit JavaScript oder einer weniger extremen funktionalen Sprache wie OCaml, F# oder Scala. Wenn Sie erfolgreich Haskell lernen, sind Sie wahrscheinlich ziemlich meinungsstark darüber, wie man es macht!

Die Vielfalt im Haskell-Ökosystem führt zu vielen unterschiedlichen Praktiken und Konventionen. Der Haskell-Compiler GHC selbst hat viele verschiedene Formatierungsstile und Konzepte, und viele davon sind spezifisch für dieses Projekt. Ich habe Unterschiede im Stil bemerkt, die stark mit den kulturellen Zentren korrespondieren - die Ost-

und Westküste der Vereinigten Staaten unterscheiden sich, ebenso wie die Stile der Niederlande, Schottland und Schweden.

Vanilla Haskell ist flexibel genug. GHC Haskell, die faktische Standardimplementierung, erlaubt eine große Vielzahl semantischer und syntaktischer Variationen durch Sprach-Erweiterungen. `MultiWayIf`, `LambdaCase` und `BlockArguments` bieten syntaktische Änderungen der Sprache. Die Erweiterungen `MultiParamTypeClasses` + `FunctionalDependencies` können verwendet werden, um Typenprogrammierung durchzuführen, die weitgehend `TypeFamilies` entspricht, und welche zu verwenden ist oft eine Frage der persönlichen Vorliebe. Viele Probleme lassen sich genauso leicht mit entweder `TemplateHaskell` oder `Generic` ableiten lösen, aber die wirklichen Kompromisse werden oft zugunsten persönlicher Vorlieben ignoriert.

In der Zwischenzeit tragen die verschiedenen Ökosysteme alle konkurrierende Ideen dazu bei, wie man etwas machen kann. Es gibt oft viele konkurrierende Bibliotheken für grundlegende Hilfsprogramme, die jeweils einen leicht unterschiedlichen Ansatz bieten. Menschen entwickeln starke Meinungen zu diesen Hilfsprogrammen, oft unverhältnismäßig zu den tatsächlichen Kompromissen, die damit verbunden sind. Ich bin sicherlich schuldig dessen!

Ein Mangel an Kohäsion kann die Produktivität eines Projekts beeinträchtigen. Erfolgreiche Projekte sollten einige Anstrengungen darauf verwenden, Kohäsion aufrechtzuerhalten. Die Förderung von Kohäsion ist ein Sonderfall der Vermeidung von Neuheit - man wählt eine Methode, um Dinge zu tun, und widersteht dann dem Drang, mit einer anderen Methode zur Problemlösung weitere Neuheiten einzuführen.

Kohäsiver Stil

Haskells Syntax ist extrem flexibel. Bedeutender Leerraum ermöglicht wunderschön eleganten Code sowie schwierige Parser-Regeln. Vertikale Ausrichtung wird zur Kunstform, und die Struktur des Textes kann die Struktur der zugrunde liegenden Berechnung andeuten. Code wird nicht mehr nur gelesen, sondern wie ein Gedicht angeordnet. Leider kann diese Schönheit oft das Warten und Verstehen von Code beeinträchtigen.

Projekte sollten einen Stil-Leitfaden übernehmen und sie sollten automatisierte Werkzeuge verwenden, um die Einhaltung zu unterstützen. Es gibt viele Werkzeuge, die dabei helfen können, aber die Vielfalt der Haskell-Syntax macht es schwierig, sich auf eine vollständige Lösung festzulegen. Die Erkundung der Kompromisse eines bestimmten Codestils liegt außerhalb des Umfangs dieses Kapitels, aber ein konsistenter ist wichtig für die Produktivität.

Kohäsive Effekte

Haskeller haben eine enorme Menge an Gedanken und Anstrengungen in das Konzept von 'Effekten' gesteckt. Jede andere Sprache baut ihr Effektsystem in die Sprache selbst ein, und es ist normalerweise nur imperatives Programmieren mit unbegrenzter Mutation, Ausnahmen und einem gewissen impliziten globalen Kontext. In Haskell haben wir ein einziges 'Standard'-Effektsystem - den `IO`-Typ. Direkt in `IO` zu schreiben, fühlt sich unangenehm an, weil es weniger bequem ist als viele imperative Programmiersprachen, also erfinden wir Erweiterungen, die sich gut anfühlen. All diese Erweiterungen haben Kompromisse.

Wenn Sie ein exotisches Effektsystem in Ihrer Anwendung verwenden, sollten Sie es konsequent nutzen. Sie sollten bereit sein, neue Mitarbeiter darin zu schulen und sie zu lehren, wie man es verwendet, wie man es debuggt und wie man es bei Bedarf modifiziert. Wenn Sie ein Standard-Effektsystem verwenden, sollten Sie versuchen, neuartige Effektsysteme nicht einzubeziehen.

Kohäsive Bibliotheken

Es gibt über ein Dutzend Logging-Bibliotheken auf Hackage. Nicht-Logging-Bibliotheken (wie z.B. Datenbank- oder Webbibliotheken) verlassen sich oft auf eine einzige Logging-Bibliothek, anstatt diese Verantwortung auf die Anwendung zu abstrahieren. Infolgedessen ist es einfach, mehrere Logging-Bibliotheken in Ihrer Anwendung zu sammeln. Sie werden sich auf eine einzige Logging-Bibliothek standardisieren wollen und dann Adapter für die anderen Bibliotheken nach Bedarf schreiben.

Diese Situation zeigt sich in anderen Bereichen in Haskell. Es gibt viele mögliche Situationen, und einige zugrunde liegende Bibliotheken zwingen Sie dazu, mit mehreren umzugehen. Der Weg des geringsten Widerstands verwendet einfach, was auch immer die zugrunde liegende Bibliothek tut. Sie sollten dem widerstehen und sich stattdessen darauf konzentrieren, an einer einzigen Lösung festzuhalten.

Kohäsive Teams

Wenn Sie zwei Entwickler einstellen, die widersprüchliche Meinungen haben und keiner von beiden bereit ist, nachzugeben, werden Sie Konflikte in Ihrem Projekt erleben. Haskeller sind in meiner Erfahrung besonders eigenwillig in dieser Hinsicht. Es ist daher wichtig, die Standards Ihres Teams in Stellenanzeigen zu kommunizieren. Während Sie

neue Mitarbeiter interviewen, sollten Sie prüfen, wie meinungsstark sie sind und ob sie Ihre Meinungen teilen.

Glücklicherweise sind keine der starken Meinungen, die ein Haskeller haben könnte, auf rassische, geschlechtliche, sexuelle oder religiöse Linien ausgerichtet. Sich darauf zu konzentrieren, eine starke Teamkohäsion zu entwickeln, steht im Einklang mit der Einstellung einer vielfältigen Gruppe von Menschen.

Empathie

Das letzte Prinzip dieses Buches ist Empathie.

Softwareentwickler sind einigermaßen bekannt dafür, sich in großen Ego-Kämpfen zu engagieren. Betrachten Sie die Python vs. Ruby Flame Wars oder wie jeder JavaScript hasst. Über PHP schlecht zu reden, ist weit verbreitet und akzeptiert, und ich weiß, dass ich sicherlich schuldig bin. Wenn wir auf unsere eigene Perspektive beschränkt sind, kann es schwierig sein, zu verstehen, warum andere Menschen andere Entscheidungen treffen.

Die Realität ist komplexer. PHP bietet eine kurze Lernkurve, um produktive Websites in einer wichtigen Nische zu erstellen. Ruby löst reale Probleme, die echte Programmierer haben. Python löst andere reale Probleme, die echte Programmierer haben. JavaScript hat sich weit über sein ursprüngliches Nischenfeld hinaus entwickelt, und JavaScript-Entwickler arbeiten hart daran, ihre Probleme auf schöne Weise zu lösen.

Um effektiv zu kommunizieren, müssen wir zuerst unser Publikum verstehen. Um effektiv zuzuhören, müssen wir zuerst den Sprecher verstehen. Dies ist eine wechselseitige Angelegenheit, und es erfordert echten Einsatz von allen Seiten, damit gute Kommunikation stattfinden kann.

Wenn wir Software lesen oder bewerten, versuchen wir zuerst zu verstehen, woher sie kommt und welche Probleme sie löst. Dann versuchen wir, die Einschränkungen zu verstehen, die zu den getroffenen Entscheidungen führten, und vermeiden unnötig breite negative Bewertungen. Haskell wird von einer besonders breiten Gruppe von Menschen in verschiedenen Ökosystemen genutzt, hat aber gleichzeitig eine relativ kleine Gesamtanzahl von Menschen, die zu einem bestimmten Zeitpunkt daran arbeiten. Es ist leicht, Missverständnisse zu verursachen und Schaden anzurichten, daher müssen wir gezielt darauf achten, dies zu vermeiden.

Empathie: Für dich selbst

Haskell ist schwer zu erlernen. Es gibt nicht viele Ressourcen, um Haskell erfolgreich in der Industrie einzusetzen. Ich habe viele Fehler gemacht, und das wirst du auch tun. Es ist wichtig, dass ich Empathie für mich selbst habe. Ich weiß, dass ich mein Bestes gebe, um zu produzieren, zu lehren und zu helfen, auch wenn ich Fehler mache. Wenn ich diese Fehler mache - selbst wenn sie Schaden verursachen - versuche ich, den angerichteten Schaden zu erkennen. Ich verzeihe mir selbst und lerne dann, was ich kann, um diese Fehler in Zukunft ohne obsessive Beurteilung zu vermeiden.

Auch du wirst Fehler machen und Schaden anrichten, während du diese neue Welt erkundest. Das ist in Ordnung. Irren ist menschlich! Und sich auf unsere Fehler zu konzentrieren verursacht mehr Leiden und blockiert Heilung. Verzeihe dir selbst für deine Schwierigkeiten. Verstehe diese Schwierigkeiten. Lerne von ihnen und überwinde sie!

Empathie: Für dein vergangenes Ich

Dein vergangenes Ich war begeistert von einer neuen Technik und konnte es kaum erwarten, sie anzuwenden. Es fühlte sich so schlau und zufrieden mit der Lösung! Lass uns an ihr Glück erinnern und ihnen das Chaos verzeihen, das sie uns hinterlassen haben. Behalte das Gefühl von Frustration und Angst im Hinterkopf und verzeihe dir selbst, dass du ihnen begegnest. Diese Gefühle sind normal und ein Zeichen von Wachstum und Fürsorge.

Vielleicht haben sie etwas übersehen, das im Problemfeld jetzt für dich völlig offensichtlich erscheint. Sie taten damals ihr Bestes mit dem, was sie hatten. Schließlich hat es der Menschheit fast 9.000 Jahre gedauert, den Kalkül zu erfinden, den wir zuverlässig Teenagern beibringen können. Deine Frustration und dein Unglaube sind der Treibstoff, den du brauchst, um zu wachsen und deinem zukünftigen Ich mehr Empathie entgegenzubringen.

Empathie: Für dein zukünftiges Ich

> Jeder weiß, dass Debugging doppelt so schwer ist wie das Schreiben eines Programms. Wenn du also so clever bist, wie du kannst, wenn du es schreibst, wie wirst du es jemals debuggen?

- Brian Kernighan, The Elements of Programming Style, 2. Auflage, Kapitel 2

Dein zukünftiges Ich ist müde, gelangweilt und hat den gesamten Kontext dieser Codezeile nicht im Kopf. Schreibe Code, der für sie funktioniert. Schreibe Dokumentation, die offensichtlich und langweilig erscheint. Stell dir vor, du hast alles vergessen, was du weißt, und musst es neu lernen - was würdest du aufschreiben?

Software ist schwierig, und du kannst nicht immer 100% deines Gehirns und deiner Energie in alles stecken. Schreibe etwas, das leichter zu verstehen ist, als du denkst, dass es notwendig ist, selbst wenn du denkst, dass es einen schwerwiegenden Fehler hat.

Empathie: Für deine Teamkollegen

Gesunde Selbstempathie ist eine Voraussetzung für gesunde Empathie gegenüber anderen Menschen.

So schwierig es auch ist, mit sich selbst empathisch zu sein, ist es noch schwieriger, mit anderen empathisch zu sein. Du kennst deinen inneren Zustand und deine Gefühle. Vielleicht erinnerst du dich sogar an deine vergangenen Zustände. Und du kannst vielleicht (mit unterschiedlicher Zuverlässigkeit) vorhersagen, wie du auf etwas reagieren wirst.

All diese Intuitionen sind bei anderen Menschen viel schwächer. Wir müssen unser Verständnis und unsere Vergebung auch auf unsere Teamkollegen anwenden. Sie arbeiten mit uns zusammen und geben ihr Bestes.

Empathie: Für dein Publikum

Ich entschuldige mich im Voraus für jeglichen Schaden, den dieses Buch verursachen könnte. Ich hoffe, dass das Publikum meines Buches es für Erfolg und Glück in ihrer Karriere und ihren Geschäftsprojekten nutzen wird. Ich erkenne auch an, dass mein Rat - unvermeidlich - misskommuniziert oder falsch angewendet wird und Schaden und Leiden verursacht.

Ebenso musst du, wenn du Code schreibst, dein Publikum berücksichtigen. Du wirst Teil deines Publikums sein, daher werden dir die Lektionen, die du über dein Vergangenes-Ich und Zukunfts-Ich gelernt hast, hilfreich sein. Wenn du Code für eine Anwendung schreibst, dann bedenke alle Menschen, die ihn lesen könnten. Du wirst die Bedürfnisse von Anfängern, Neulingen, erfahrenen Entwicklern und Experten berücksichtigen wollen.

Dies vollständig zu tun, ist unmöglich, daher musst du die Kompromisse sorgfältig abwägen. Wenn du erwartest, dass dein Publikum hauptsächlich neu in Haskell ist, dann schreibe einfach und klar. Wenn du fortgeschrittene Fähigkeiten und ausgefallene Techniken benötigst, mach eine Notiz davon und schreibe Beispiele und Dokumentationen,

um zu demonstrieren, was vor sich geht. Es ist nichts falsch daran, ein Warnsignal oder eine Notiz anzubringen, um darauf hinzuweisen, dass etwas schwierig sein könnte!

Empathie: Für die Geschäftsleute

Das ist besonders schwer. Sie werden nie Ihren Code lesen. Und sie ändern oft die Anforderungen nach Belieben, ohne Rücksicht auf die Abstraktionen, die Sie entwickelt haben. Aber - letztendlich - sind wir hier, um Code zu schreiben, der es dem Unternehmen ermöglicht, profitabel zu sein.

Wir können Empathie für ihre Bedürfnisse haben, indem wir unseren Code ihren Launen offen lassen. Ein Projekt sollte sich entwickeln und ändern können, ohne neu geboren werden zu müssen.

Wenn das Unternehmen scheitert, sind wir alle ohne Job. Wenn genug Haskell-Projekte scheitern, werden wir nicht genug Haskell-Jobs für alle haben, die einen wollen. Und, am besorgniserregendsten, wenn *zu viele* Haskell-Projekte scheitern, wird Haskell keine tragfähige Wahl in der Industrie sein.

Ich glaube, dass alle Haskeller in der Industrie eine Verantwortung gegenüber ihrer Gemeinschaft haben, um ihren Projekten zum Erfolg zu verhelfen. Dieses Buch ist das Ergebnis dieses Glaubens.

Referenzen

- 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 Haskell-Teams aufbauen

1. Haskell verkaufen

Du möchtest Haskell bei der Arbeit einsetzen. Dein Chef ist skeptisch – ist Haskell nicht eine obskure, ausgefallene, akademische Programmiersprache? Hat es nicht schreckliche Build-Tools und miserable IDEs? Ist es nicht super schwer zu lernen und zu benutzen?

Haskell-Skeptiker haben viele gemeine Dinge über Haskell zu sagen. Wenn dein Chef ein Haskell-Skeptiker ist, wirst du wahrscheinlich Haskell in deinem aktuellen Job nicht einsetzen können. Wenn dein Chef offener dafür ist, Haskell auszuprobieren, und du die Aufgabe hast, die Eignung von Haskell für eine Aufgabe zu bewerten, dann wirst du Haskell *verkaufen*. Um Haskell effektiv zu verkaufen, müssen wir unsere Programmierer-Mentalität ablegen und die Geschäftsperspektive einnehmen.

Der kluge Geschäftsmann wird das tun, was den größten Gewinn bringt, und wird langfristige Vorteile gegen kurzfristige Kosten abwägen. Haskell kann das Ergebnis eines Unternehmens tatsächlich verbessern, und wenn du Haskell verkaufst, musst du wissen, wie du dafür argumentieren kannst.

1.1 Einschätzung der Aufnahmebereitschaft

Ist deine Firma ein Ruby-Shop? Hassen deine Kollegen statische Typen, lieben Monkeypatching und stören sich nicht an gelegentlichen Produktionsabstürzen durch `nil`-Fehler? Wenn ja, wirst du wahrscheinlich keinen großen Erfolg haben, ihnen Haskell schmackhaft zu machen. Um von Haskell überzeugt zu werden, ist es gut, wenn das Team die gleichen Werte teilt, die die Haskell-Sprache verkörpert.

Auf der anderen Seite hat ein solcher Shop am *meisten* von Haskell zu gewinnen. Gibt es ein Stück Kerninfrastruktur, das langsam und fehlerhaft ist und den Gewinn erheblich bremst? Wenn ja, könntest du diesen Teil der Infrastruktur neu schreiben und dem Unternehmen einen enormen Nutzen bieten. Meine Vorgänger bei einem früheren Job überzeugten das Management, Haskell für eine Neuschreibung zu verwenden, anstatt einen weiteren Versuch in PHP zu starten, und ich wurde eingestellt, um dies zu tun. Die Haskell-Version des Dienstes benötigte 1/10 der Cloud-Ressourcen, um zu laufen, und beseitigte einen Engpass, der es uns ermöglichte, unseren größeren Kunden mehr Geld zu berechnen.

Wenn dein Unternehmen bereits Entwickler beschäftigt, die mit statisch typisierten funktionalen Sprachen wie Scala oder F# vertraut sind, wirst du es leichter haben, sie von Haskell zu überzeugen. Vermutlich schätzen sie bereits die Stärken von Haskell, weshalb sie sich für eine funktionale Sprache entschieden haben. Es könnte jedoch sein, dass nicht genug Gewinn aus dem Schreiben eines Dienstes in Haskell zu ziehen ist – schließlich ist Scala ja schon fast dort. Die Entwickler könnten das Gefühl haben, dass der zusätzliche Aufwand, eine weitere Sprache in den Stack aufzunehmen, minimalen Nutzen bringt, da es so nah dran ist. In diesem Fall musst du sie von anderen Vorteilen überzeugen, die Haskell bietet.

1.2 Software-Produktivität

Wir wollen behaupten, dass Haskell die Produktivität von Softwareentwicklern steigern wird. Dies führt zu reduzierten Entwicklerkosten und erhöhtem Gewinn durch neue Funktionen. Wir müssen jedoch die Entwicklerproduktivität auf einem etwas nuancierten Niveau verstehen, um dies angemessen zu verkaufen.

Wie messen wir die Produktivität von Entwicklern? Es ist nicht einfach. Es gibt viele Studien, und alle sind schlecht. Unabhängig von deiner Position zu einer bestimmten Praxis (dynamische vs. statische Typen, Paarprogrammierung, formale Verifikation, Wasserfall, agil, usw.) wirst du eine Studie finden, die das unterstützt, was du denkst. Wir wissen einfach nicht, wie wir die wissenschaftliche Methode effektiv und genau nutzen können, um die Produktivität von Entwicklern zu messen.

Wie behaupten wir dann, dass Haskell sie verbessert? Wir können nur unsere Erfahrungen nutzen – anekdotische Evidenz. Ebenso können unsere Argumente – bestenfalls – Menschen dazu bringen, offen zu sein, unsere Erfahrungen zu teilen. Die erfahrungsbasierte Natur der Entwicklerproduktivität bedeutet, dass wir in den Ingenieuren, die wir überzeugen möchten, einen offenen Geist kultivieren müssen, und dann müssen wir sie dazu führen, die gleichen Erfahrungen zu machen.

Wir werden ein bisschen darüber im Kapitel “Lernen und Lehren von Haskell” erfahren.

1.3 Statistiken zur Produktivität

Das Management interessiert sich für Produktivität, aber es interessiert sich nicht nur dafür, wie schnell du eine Funktion herausbringen kannst. Es interessiert sich dafür, wie gut du vorhersagen kannst, wie lange eine Funktion dauern wird. Es interessiert sich

dafür, wie groß der Unterschied in der Produktivität zwischen den Teammitgliedern sein wird. Varianz ist für das Management wichtig.

Die Statistik gibt uns Werkzeuge, um über die Aggregation von Daten nachzudenken. Der Durchschnitt wird berechnet, indem alle Einträge summiert und durch die Anzahl geteilt werden. Es ist das häufigste statistische Maß, kann aber auch schrecklich irreführend sein. Das durchschnittliche Einkommen in den Vereinigten Staaten beträgt 72.000 $, und du könntest das hören und denken, dass die meisten Menschen in etwa diesen Betrag verdienen. Tatsächlich verdienen die meisten Menschen weniger als diesen Betrag.

Ein anderes Maß, der Median, ist angemessener. Der Median ist der Mittelpunkt der Verteilung, was bedeutet, dass die Hälfte aller Werte über dem Median und die Hälfte aller Werte unter dem Median liegt. Das Medianeinkommen eines Haushalts beträgt 61.000 $. Der Durchschnitt ist viel höher als der Median, was bedeutet, dass es eine kleine Anzahl von Menschen gibt, die eine enorme Menge an Geld verdienen. Um zu wissen, ob ein Durchschnitt oder ein Median für deinen Zweck angemessener ist, musst du die Verteilung deiner Daten kennen.

Ihr Softwareteam hat möglicherweise eine beeindruckende durchschnittliche Entwicklerproduktivität. Dies könnte daran liegen, dass Sie eine Gruppe von überdurchschnittlichen Entwicklern haben. Es kann auch daran liegen, dass Sie einen extrem produktiven Entwickler und eine Gruppe von unterdurchschnittlichen Entwicklern haben. Ein stark unausgewogenes Team stellt ein *Risiko* für das Unternehmen dar, da der Verlust eines einzelnen Entwicklers drastische Konsequenzen haben könnte. Das Management wird aus diesem Grund eine hohe Median-Entwicklerproduktivität der durchschnittlichen vorziehen.

Was das Management jedoch *wirklich* möchte, ist eine geringe Varianz zwischen den Entwicklern. Varianz ist der Durchschnitt der quadrierten Abweichung vom Durchschnitt. Die Abweichung wird quadriert, damit negative und positive Abweichungen gleichermaßen berücksichtigt werden. Ein Team mit hoher Varianz wird einige Entwickler deutlich über und unter dem Median haben. Dies ist riskant, da die genaue Zuweisung von Entwicklern drastisch beeinflussen kann, wie schnell und effektiv Software entwickelt wird. Ein Team mit geringer Varianz wird die meisten Entwickler relativ nah beieinander in Bezug auf ihre Fähigkeiten haben. Dies reduziert das Risiko, da ein einzelner Entwickler in den Urlaub gehen kann, ohne die durchschnittliche Fähigkeit des Teams signifikant zu verändern.

Größere Unternehmen neigen dazu, die Varianz in der individuellen Produktivität zu minimieren. Sie können es sich leisten, viele Software-Ingenieure einzustellen, und sie möchten, dass ihr Personal austauschbar und ersetzbar ist. Dies dient nicht ausschließlich dazu, die Mitarbeiter zu entmenschlichen und abzuwerten - es ist einfacher, Urlaub

oder Elternzeit zu nehmen, wenn man nicht der einzige ist, der die Arbeit erledigen kann. Es führt jedoch in der Regel zu einer reduzierten Produktivität für die Spitzenleister.

Es ist eine gültige Entscheidung, auf geringe Varianz zu setzen. Tatsächlich ergibt dies für Unternehmen viel Sinn. Elm und Go sind zwei neue Programmiersprachen, die Einfachheit und leichte Erlernbarkeit betonen. Sie opfern Abstraktion und Ausdruckskraft, um die Varianz der Entwicklung in der Sprache zu reduzieren. Das bedeutet, dass ein Elm-Experte nicht viel produktiver ist als ein Elm-Anfänger. Elm- oder Go-Programmierer können die Sprache schnell erlernen und so produktiv sein, wie sie es jemals sein werden. Das Management liebt dies, weil die Einarbeitung neuer Entwickler schnell geht, man nicht die besten und klügsten einstellen muss, und man zuverlässig Dinge erledigt bekommt.

Haskell ist keine Sprache mit geringer Varianz. Haskell ist eine Sprache mit extrem hoher Varianz. Es dauert relativ lange, um minimal kompetent zu werden, und das Potenzial in Bezug auf Fähigkeiten ist nach oben hin offen. Haskell wird aktiv in der Wissenschaft eingesetzt, um die Grenzen des Software-Engineerings zu erweitern und neue Techniken zu entdecken. Experten aus Industrie und Wissenschaft arbeiten hart daran, Haskell neue Funktionen hinzuzufügen. Der Unterschied in der Produktivität zwischen jemandem, der es seit Jahren verwendet, und jemandem, der es sechs Monate studiert hat, ist enorm.

Die Beachtung der Varianz ist entscheidend. Wenn Sie mit Haskell arbeiten, setzen Sie bereits auf das obere Ende der Varianzkurve. Wenn Sie fortgeschrittene oder ausgefallene Haskell-Bibliotheken und -Funktionen auswählen, erhöhen Sie die Varianz. Je schwieriger es ist, in Ihren Code einzusteigen, desto weniger werden Ihre Kollegen es genießen, und desto skeptischer werden sie gegenüber Haskell insgesamt sein. Aus diesem Grund ist es wichtig, die einfachsten Haskell-Materialien zu bevorzugen, die Sie verwenden können.

1.4 Kennen Sie Ihre Konkurrenz

Konkurrenz existiert in jedem Verkaufs- und Marketingproblem. Ihre Konkurrenz beim Verkauf von Haskell wird hart sein - mehrere andere Programmiersprachen werden ebenfalls überzeugende Vorteile haben. Haskell muss die Vorteile anderer Sprachen mit den technischen Gewinnen übertreffen, die erzielt werden können.

In einigen Bereichen, wie Compiler-Design oder Web-Programmierung, verfügt Haskell über ausreichende Bibliotheken und eine Community, die es Ihnen ermöglicht, schnell produktiv zu sein. Die Sprache ist gut positioniert, um einen überzeugenden Vorteil zu

bieten. Andere Bereiche haben nicht so viel Glück, und die Bibliothekssituation wird in einer anderen Sprache besser sein als in Ihrer.

Wenn Ihr Haskell-Projekt aus irgendeinem Grund scheitert, wird das Projekt in einer anderen Sprache neu geschrieben. Sie werden wahrscheinlich keine Gelegenheit bekommen, es "noch einmal zu versuchen." Unternehmen sind in der Regel nicht bereit, außerhalb ihrer Kernkompetenz Wetten abzuschließen, und die Wahl der Programmiersprache gehört wahrscheinlich nicht zu dieser Kernkompetenz. Deshalb sollten Sie darauf achten, Haskell als sichere, gewinnende Wahl zu positionieren, mit erheblichen Vorteilen gegenüber der Konkurrenz.

2. Haskell Lernen und Lehren

Wenn Sie möchten, dass Ihr Haskell-Projekt erfolgreich ist, müssen Sie neue Haskeller betreuen und unterrichten. Es ist möglich, eine Weile nur erfahrene Ingenieure einzustellen, aber schließlich werden Sie Juniors einstellen wollen. Neben der Ausbildung und dem Wachstum der Haskell-Community werden Sie neue Perspektiven und Erfahrungen gewinnen, die dazu beitragen, Ihren Code robuster zu machen.

2.1 Die Philologie von Haskell

Haskell zu lernen ist ähnlich wie das Erlernen jeder anderen Sprache. Der große Unterschied besteht darin, dass die meisten Erfahrungen mit dem "Erlernen einer Programmiersprache" professionelle Ingenieure sind, die eine weitere Sprache lernen, oft in einer ähnlichen Sprachfamilie. Wenn Sie Java kennen und dann C# lernen, wird die Erfahrung reibungslos verlaufen - Sie können C# praktisch lernen, indem Sie auf die Unterschiede zu Java achten. Von Java zu Ruby zu wechseln ist ein größerer Sprung, aber sie sind beide imperative Programmiersprachen mit viel eingebauter Unterstützung für objektorientierte Programmierung.

Lassen Sie uns ein wenig über die Geschichte der Programmiersprachen lernen. Dies ist nützlich zu verstehen, da es hervorhebt, wie anders Haskell im Vergleich zu anderen Sprachen wirklich ist.

Am Anfang gab es Maschinensprache - Assembler. Dies war fehleranfällig und schwer zu schreiben, also erfand Grace Hopper den ersten Compiler, der es Programmierern ermöglichte, höhere Programmiersprachen zu schreiben. Die 1950er und 1960er Jahre brachten uns viele grundlegende Programmiersprachen: ALGOL (1958) und FORTRAN (1957) waren frühe imperative Programmiersprachen. LISP (1958) wurde entwickelt, um KI zu studieren, und wird oft als die erste funktionale Programmiersprache anerkannt. Simula (1962) war die erste objektorientierte Programmiersprache, direkt inspiriert von ALGOL.

Smalltalk (1972) versuchte, die objektorientierte Programmierung von Grund auf neu zu gestalten. Die Programmiersprache C (1972) wurde entwickelt, um beim Betriebssystem UNIX zu helfen. In der Zwischenzeit führte Standard ML (1973) die moderne funktionale Programmierung ein, wie wir sie heute kennen. Prolog (1972) und SQL (1974) wurden

ebenfalls in diesem Zeitraum erfunden. Diese Sprachen definieren größtenteils die Sprachfamilien, die heute gebräuchlich sind.

C++ übernahm Lektionen von Simula und Smalltalk, um C mit objektorientiertem Verhalten zu erweitern. Java fügte C++ einen Garbage Collector hinzu. Ruby, JavaScript, Python, PHP, Perl usw. gehören alle zu dieser Sprachfamilie - imperative Programmiersprachen mit einem gewissen Grad an objektorientierter Unterstützung. Tatsächlich gehören fast alle gängigen Sprachen heute zu dieser Familie!

In der Zwischenzeit entwickelte sich Standard ML weiter, und Theoretiker der Programmiersprachen wollten die funktionale Programmierung genauer untersuchen. Die Programmiersprache Miranda (1985) war ein Schritt in diese Richtung - sie bietet träge Auswertung und ein starkes Typsystem. Das Haskell-Komitee wurde gegründet, um eine Sprache zu schaffen, die die Forschung in der trägen funktionalen Programmierung vereinheitlicht. Schließlich wurde 1990 die erste Version der Programmiersprache Haskell veröffentlicht.

Haskell wurde hauptsächlich als Vehikel für die Forschung zu funktionalen Programmierungstechnologien genutzt. Viele Menschen erhielten ihre Promotionen, indem sie Haskell oder GHC mit neuen Funktionen oder Techniken erweiterten. Haskell war bis Mitte der 2000er Jahre keine praktische Wahl für industrielle Anwendungen. Das Buch "Real World Haskell" von Don Stewart, Bryan O'Sullivan und John Goerzen zeigte, dass es endlich möglich war, Haskell zur Lösung von Industrieproblemen zu verwenden. Die GHC-Laufzeit war schnell und unterstützte hervorragendes Threading.

Zum Zeitpunkt dieses Schreibens ist es 2022. Haskell ist in einer Reihe von Bereichen erstklassig. Haskell - mehr als alles andere - ist radikal anders als andere Programmiersprachen. Die Sprache hat sich nicht mit der Industrie entwickelt. Die Akademie war für die Forschung, das Design und die Entwicklung verantwortlich. Fast 30 Jahre parallele Entwicklung fanden statt, um Haskell von Java zu unterscheiden.

2.2 Programmieren ist schwer zu lernen

Wenn Sie ein erfahrener Ingenieur mit zehn Jahren Erfahrung sind, haben Sie wahrscheinlich eine Menge Sprachen gelernt. Vielleicht haben Sie kürzlich Go, Rust oder Swift gelernt und fanden es nicht schwierig. Dann versuchen Sie, Haskell zu lernen, und plötzlich stehen Sie vor einer Schwierigkeit, die Sie lange nicht mehr gespürt haben. Diese Schwierigkeit ist die Herausforderung eines neuen Paradigmas.

Die meisten professionellen Programmierer beginnen mit dem Erlernen der imperativen Programmierung und nehmen später die objektorientierte Programmierung auf. Der

Großteil ihres Codes ist solide imperativ, mit einigen objektorientierten Elementen. Daran ist nichts falsch - dieser Stil von Code funktioniert wirklich und löst echte Geschäftsprobleme, unabhängig davon, wie komplex Programme werden, wenn sie wachsen. Viele Programmierer haben vergessen, wie schwierig es ist, die imperative Programmierung zu erlernen oder überhaupt wie eine Maschine zu denken.

Ich möchte mich in diesem Abschnitt auf das Prinzip der Empathie konzentrieren. Programmieren ist schwer zu lernen. Trivial bedeutet das, dass funktionale Programmierung schwer zu lernen ist.

Die Erfahrung, etwas Neues zu erlernen, kann oft unangenehme Gefühle hervorrufen. Frustration, Traurigkeit und Wut sind häufige Reaktionen auf diese Schwierigkeit. Wir müssen uns jedoch nicht in diesen Emotionen verlieren. Versuchen Sie, die Erfahrung positiv umzudeuten: Sie lernen! Genauso wie Muskelkater nach dem Sport ein Zeichen dafür ist, dass Sie stärker werden, ist leichte Frustration beim Lernen ein Zeichen dafür, dass Sie Ihren Horizont erweitern. Diese positive Einstellung wird Ihnen helfen, schneller und angenehmer zu lernen.

2.3 Lernmaterialien auswählen

Ich bevorzuge Haskell Programming from First Principles[1]. Chris Allen und Julie Moronuki haben hart daran gearbeitet, dass das Buch zugänglich ist und das Material an frischen Studenten getestet. Ich habe es persönlich verwendet, um vielen Menschen beim Lernen von Haskell zu helfen. Ich habe das Buch als Eckpfeiler des Haskell-Lehrplans und Ausbildungsprogramms bei Mercury verwendet, wo wir Leute darauf trainieren, in 2-8 Wochen produktive Haskell-Entwickler zu werden.

2.4 Schreiben Sie viel Code

Beim Lernen und Lehren von Haskell ist es wichtig, einen schnellen Feedback-Kreis zu ermöglichen. Ablenkungen zu minimieren ist ebenfalls wichtig. Aus diesem Grund empfehle ich, minimale Werkzeuge zu verwenden - ein einfacher Texteditor mit Syntaxhervorhebung ist ausreichend. Komplizierte IDEs und Plugins behindern oft den Lernprozess. Die Zeit, die Sie mit der Einrichtung Ihrer Editor-Situation verbringen, ist Zeit, die Sie *nicht* tatsächlich mit dem Lernen von Haskell verbringen.

Studenten sollten sich mit `ghci` vertraut machen, um Ausdrücke zu evaluieren und ihre Arbeit mit `:reload` neu zu laden, um schnell Rückmeldungen zu erhalten. Das Tool

[1] https://haskellbook.com/

`ghcid`[2] kann diesen Prozess automatisieren, indem es relevante Dateien überwacht und neu lädt, sobald eine geändert wird.

Wenn wir Haskell lernen, besteht die große Herausforderung darin, ein mentales Modell davon zu entwickeln, wie GHC funktioniert. Ich empfehle, ein "prädiktives Modell" von GHC zu entwickeln. Ändere zuerst eine Kleinigkeit im Code. Bevor du speicherst, sage voraus, was deiner Meinung nach passieren wird. Dann speichere die Datei und sieh dir das Ergebnis an.

Wenn dich das Ergebnis überrascht, ist das gut! Du hast die Gelegenheit, dein Modell zu verfeinern. Entwickele eine Hypothese, warum deine Vorhersage nicht eingetroffen ist. Teste dann diese Vorhersage.

Compiler-Fehler zeigen uns, dass etwas nicht stimmt. Wir sollten dann versuchen, eine Hypothese darüber zu entwickeln, was schiefgelaufen ist. Warum hat mein Code diesen Fehler? Was habe ich erwartet, dass er tut? Wie unterscheidet sich mein mentales Modell von Haskell von dem Verständnis, das GHC hat?

Programmieren ist stilles Wissen. Es reicht nicht aus, darüber zu lesen. Das Lesen informiert die analytischen und verbalen Teile unseres Gehirns. Aber Programmieren greift auf viel mehr zurück, was nur durch praktisches Tun wirklich trainiert wird. Wir müssen Code schreiben – viel davon – und wir müssen auf dem Weg viele Fehler machen!

2.5 Keine Angst vor dem GHC

Viele Studenten entwickeln eine Abneigung gegen Fehlermeldungen. Sie fühlen Urteil und Verurteilung durch sie – "Ach, ich bin nicht klug genug, um es richtig zu verstehen!" In anderen Programmiersprachen sind Compiler-Fehler oft wenig hilfreich, obwohl sie eine Bildschirmladung an Ausgabe erzeugen. Infolgedessen überspringen sie oft das Lesen der Fehlermeldungen vollständig. Studenten dazu zu bringen, *wirklich* die Compiler-Fehler von GHC zu lesen, hilft erheblich beim Lernen von Haskell. Die Fehlermeldungen von GHC sind oft hilfreicher als in anderen Sprachen, auch wenn sie anfangs schwer zu lesen sein können.

Wir wollen die Beziehung der Menschen zu ihren Compilern verbessern. Fehlermeldungen sind Geschenke, die der Compiler dir gibt. Sie sind eine Seite eines Gesprächs, das du mit einem Computer führst, um ein gemeinsames Ziel zu erreichen. Manchmal sind sie nicht hilfreich, besonders wenn wir nicht gelernt haben, zwischen den Zeilen zu lesen.

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

Eine Fehlermeldung bedeutet nicht, dass du nicht klug genug bist. Die Nachricht ist GHCs Art zu sagen: "Ich kann nicht verstehen, was du geschrieben hast." GHC ist kein perfektes Wesen, das in der Lage ist, jede vernünftige Idee zu verstehen – tatsächlich sind viele hervorragende Ideen durch Haskells Typsystem verboten! Eine Fehlermeldung kann als Frage von GHC gesehen werden, als Versuch, Klarheit zu gewinnen, um herauszufinden, was du wirklich gemeint hast.

2.6 Einfach anfangen

Beim Schreiben von Code für einen Anfänger versuche ich, so einfach wie möglich zu bleiben. "Einfach" ist ein vages Konzept, aber um präziser zu sein, meine ich etwas wie die kleinste transitive Hülle von Konzepten zu bevorzugen. Oder, "Ideen mit wenigen Abhängigkeiten."

Dies bedeutet, viele `case`-Ausdrücke und explizite Lambdas zu schreiben. Diese beiden Merkmale sind die grundlegenden Bausteine von Haskell. Studenten können diese Ausdrücke später vereinfachen, als Lernübung, aber darauf sollten wir uns nicht konzentrieren – stattdessen sollten wir uns darauf konzentrieren, tatsächliche Probleme zu lösen! Zusätzliche Sprachstrukturen können im Laufe der Zeit eingeführt werden, wenn der Student Komfort und Fähigkeiten mit den Grundlagen zeigt.

Als Beispiel, nehmen wir an, wir versuchen, `map` auf Listen neu zu schreiben:

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

Ich würde dem Studenten empfehlen, damit zu beginnen, einen `case` Ausdruck einzuführen.

```
1 map function list =
2     case ??? of
3         patterns ->
4             ???
```

Was können wir für diese `???` und `patterns` einsetzen? Nun, wir haben eine `list` Variable. Lassen Sie uns das einfügen:

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

Was sind die Muster für eine Liste? Wir können die Dokumentation einsehen und feststellen, dass es zwei Konstruktoren gibt, auf die wir einen Musterabgleich durchführen können:

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

Die Verwendung eines `case`-Ausdrucks hat unser Problem in zwei kleinere Probleme unterteilt. Was können wir zurückgeben, wenn wir eine leere Liste haben? Wir haben `[]` als möglichen Wert. Wenn wir eine nicht-leere Liste erstellen wollten, müssten wir einen `b`-Wert haben, aber wir haben keinen, also können wir ihn nicht einsetzen.

Für den Fall der nicht-leeren Liste haben wir `head :: a` und `tail :: [a]`. Wir wissen, dass wir die `function` auf `head` anwenden können, um ein `b` zu erhalten. Wenn wir uns unser "Werkzeugkasten" ansehen, ist der einzige Weg, wie wir ein `[b]` erhalten können, indem wir `map function` aufrufen.

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

Wir möchten mit einem relativ kleinen Werkzeugkasten von Konzepten beginnen. Funktionen, Datentypen und `case`-Ausdrücke werden uns lange Zeit gute Dienste leisten. Viele Probleme lassen sich leicht mit diesen grundlegenden Bausteinen lösen, und es ist wichtig, ein starkes grundlegendes Verständnis für ihre Leistungsfähigkeit zu entwickeln, insbesondere für Anfänger in Haskell.

Dies knüpft an unsere Prinzipien der Neuheit und Komplexität an. Wir möchten Konzepte langsam hinzufügen, um uns nicht zu überfordern. Während wir Konzepte einführen, müssen wir nicht nur das Konzept selbst betrachten, sondern auch, wie dieses Konzept mit jedem anderen uns bekannten Konzept interagiert. Dies kann leicht zu viel werden!

2.7 Reale Probleme lösen

Es dauert einige Zeit, bis man sich darauf vorbereitet hat, aber ein Anfänger kann lernen, den IO-Typ gut genug zu nutzen, um grundlegende Dienstprogramme zu schreiben, ohne alle Besonderheiten der Monaden zu verstehen. Betrachten Sie schließlich dieses Beispielprogramm in Java, Ruby und schließlich Haskell:

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:

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

Haskell:

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

Der Java-Code enthält viele Funktionen. Sie müssen nicht erklärt werden. In meinen Java 101 Kursen an der Universität wurde uns gesagt, wir sollten es einfach “kopieren

und einfügen" und eine Erklärung würde später folgen. Das hat für mich einigermaßen funktioniert. Schließlich werden Computer oft als Black Boxes mit mysteriöser magischer Kraft wahrgenommen: Programmiersprachen ähnlich zu behandeln, fühlt sich natürlich und normal an.

Ein Anfänger kann die übliche Haskell-Ausbildung durchlaufen, indem er `Functor`, `Monad`, `Monoid` usw. Instanzen für gängige Typen definiert und gleichzeitig grundlegende Dienstprogramme und Beispiele entwickelt.

2.8 Pair-Programmierung

Pair-Programmierung kann eine großartige Möglichkeit sein, die implizite Natur der Programmierung in Haskell zu zeigen. Der Fahrer kann Pairing auch als Gelegenheit nutzen, um anzugeben und sein eigenes Ego zu füttern, was dem Anfänger schadet. Der Lehrer muss große Sorgfalt walten lassen, um Empathie für den Lernenden zu haben.

Der Fahrer wird langsamer werden wollen und seinen Denkprozess erklären. Ich finde es hilfreich, verbale Erklärungen in "Beobachtung der Realität", "Bemerkung von Gefühlen" und "Diskussion von Strategien" zu unterteilen. Diese Technik stammt aus der Gewaltfreien Kommunikation. Ich werde auch mein Vorhersagemodell verbal erklären. Zum Beispiel, wenn ich ein Problem löse, könnte ich normalerweise ein paar Vorhersagen überspringen und eine größere Änderung vornehmen, wenn ich alleine programmiere. Beim Pairing werde ich stattdessen meine Vorhersage äußern, die Änderung vornehmen und das Ergebnis durchsprechen.

Der Schüler wird aufmerksam sein und Fragen stellen wollen. Hab keine Angst zu unterbrechen - der Zweck der Übung besteht hauptsächlich darin, Wissen und Praxis vom Fahrer zu übertragen. Ein großer Teil des Nutzens besteht jedoch darin, den Fahrer dazu zu bringen, klar über das nachzudenken, was sie tun! Der Fahrer sollte von einer guten Frage genauso profitieren wie der Schüler.

Leider ist „Pair-Programmierung" auf diese Weise eine implizite Übung, genau wie das Programmieren selbst. Ich kann meine Strategien und Techniken für eine erfolgreiche Sitzung beschreiben, aber der beste Weg, um zu lernen, ist durch Beobachtung und Teilnahme. Lassen Sie uns ein hypothetisches Beispiel durchgehen.

2.9 Ein Dialog

(Dinge, die ich vielleicht denke, stehen in Klammern. Ich werde sie tatsächlich nicht sagen, weil es wichtig ist, den Schüler nicht mit unnötigen Abschweifungen abzulenken.)

Schüler: Hey, stört es dich, wenn wir bei etwas zusammenarbeiten?

Matt: Sicher! Ich würde mich freuen.

S: Meine Aufgabe ist es, unsere Benutzerliste zu nehmen und herauszufinden, wie viele E-Mail-Konten zu jedem Hosting-Dienst gehören.

M: Okay, cool. Also im Grunde zählen, wie viele `@gmail.com` und `@yahoo.com` usw. es gibt?

S: Ja. Ich bin mir nicht sicher, wie ich anfangen soll! Ich weiß, dass ich es in SQL machen kann, aber ich würde lieber lernen, wie man es in Haskell macht.

M: Sicher! Okay, zuerst werde ich unsere Datenbanktypen überprüfen. Ich möchte meine Annahmen darüber überprüfen, wie unsere Daten strukturiert sind, da dies die Quelle unserer Informationen ist. Ich werde zu der Datei navigieren, die unsere Definition enthält. Hier ist der Typ:

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

S: Also möchten wir jeden `User` nehmen und die `EmailAddress` überprüfen. Wie sieht dieser Typ aus?

M: Gute Frage! Wenn ich die Datei nach `EmailAddress` durchsuchen, finde ich nichts. Also werde ich die Datei nach `Email` durchsuchen, da das näher dran ist. Das führt mich zur Importliste, wo ich sehe, dass wir ein Modul namens `Text.Email.Validate` importieren.

S: Woher kommt das?

M: Ich bin mir nicht sicher. Ich sehe dieses Modul nicht in unserem Projekt gelistet, was bedeutet, dass es in einer Abhängigkeit ist. Also werde ich jetzt meinen Browser öffnen und auf `stackage.org` nach `EmailAddress` suchen. Es gibt hier ein paar Ergebnisse, und das erste ist ein Paket namens `email-validate` in einem Modul `Text.Email.Parser`. Da es die gleiche `Text.Email.*` Struktur teilt und es eine Validierung hat, vermute ich, dass es das ist.

S: Ja, ich glaube nicht, dass es aus der Kryptobibliothek kommt, und wir verwenden `pushbullet` nicht, also ist das `pushbullet-types` wahrscheinlich nicht.

M: Gute Beobachtung!

S: OK! Ich glaube, ich weiß, was als nächstes kommt. Das Modul exportiert eine Funktion `domainPart :: EmailAddress -> ByteString`. Also können wir das verwenden, um die Domain für eine E-Mail zu bekommen!

M: Das ist auch mein Tipp. Jetzt, da wir unsere primitiven Typen verstanden haben, lasst uns die Signatur aufschreiben. Was ist deine Vermutung für eine erste Signatur?

S: Ich glaube, ich würde hier anfangen:

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

Die `ByteString`s sind der `domainPart`, und die `Int` ist unsere Zählung.

M: Das klingt gut. Ich würde wahrscheinlich etwas anderes wählen, aber lassen Sie uns das zuerst erkunden.

S: Warum?

M: Nun, immer wenn ich ein `[(a, b)]` sehe, denke ich sofort an ein `Map a b`. Aber wir können es einfach halten und einfach versuchen, dieses Problem zu lösen. Wenn es zu nervig wird, suchen wir nach einer alternativen Lösung.

S: Okay, das funktioniert für mich! Ich sehe wirklich nicht, wie ein `Map` gerade für uns funktioniert - wir machen keine Nachschlagevorgänge. Also das Erste, was ich tun möchte, ist, alle Benutzer zu erhalten.

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

Sobald ich die Benutzer habe, möchte ich die E-Mail-Adressen erhalten.

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

M: Gute Verwendung von `map` dort!

S: Als nächstes möchte ich die Domain-Teile extrahieren.

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

Und, äh, ich glaube, ich stecke hier fest. Ich möchte die Liste nach den Domains gruppieren. Aber ich weiß nicht, wie ich das machen soll.

(Ich werde dem Drang widerstehen, den Code prägnanter zu machen! Nur weil ich das als `map (domainPart . userEmail) <$> selectAllUsers` schreiben kann, bedeutet das nicht, dass es jetzt wichtig ist.)

M: Okay! Wir haben gerade ein `[ByteString]`. Wie könnte unsere Gruppierung aussehen?

S: Ich schätze ein `[[ByteString]]`?

(Na ja, ein `[NonEmpty ByteString]` wäre genauer, aber dahin kommen wir später.)

M: Klingt gut für mich. Nun, wir haben ein paar Möglichkeiten - wenn ich mir bei einer Funktionalität unsicher bin, schaue ich entweder in die relevanten Module oder suche bei Hoogle nach der Typsignatur. Wenn ich mir bei den relevanten Modulen nicht sicher bin, gehe ich direkt zu Hoogle. Also lass uns nach `[ByteString] -> [[ByteString]]` suchen.

S: Keines davon ist relevant! `text-ldap` ist nicht nah dran. `subsequences`, `inits`, `permutations`, `tails`, nichts davon hat mit Gruppierung zu tun.

M: Hmm. Ja. Hoogle lässt uns hier im Stich. Was, wenn wir nach `group` suchen?

S: Oh, dann bekommen wir `group :: Eq a => [a] -> [[a]]` zurück. Das ist genau das, was wir wollen!

M: Lass uns die Dokumentation lesen, nur um sicher zu sein. Springt etwas als potenzielles Problem ins Auge?

S: Ja - das gegebene Beispiel ist ein bisschen seltsam.

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

Ich würde erwarten, dass alle gleichen Elemente zusammen gruppiert werden, aber `s` erscheint zweimal. Ich denke, ich kann dies umgehen!

(Hmmm, wohin geht der/die Schüler/in? Sortiert er/sie die Liste?)

S: Wir können `group` aufrufen und dann die Größe der Listen erhalten.

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

Okay okay okay, also das ist die richtige Form, ABER, wir müssen sie auf eine besondere Weise verwenden!

M: Wie machen wir das?

S: Okay, nehmen wir an, wir suchen nach `gmail.com`. Wir würden die Ergebnisliste nach `gmail.com` filtern und dann die `Int`s summieren!

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

(Widerstehe dem Drang, eine `sum . map snd`-Umgestaltung vorzuschlagen!)

M: Schön! Das funktioniert für den Fall, dass wir die Domain kennen. Aber wenn wir nur eine Zusammenfassungsstruktur wollen, wie können wir unseren Code ändern, um das zu erreichen?

S: Hm. Wir könnten die Liste durchgehen und für jeden Namen `domainCount` berechnen, aber das ist ineffizient…

M: Das funktioniert! Aber du hast recht, das ist ineffizient. Ich denke, wir können es definitiv besser machen. Was fällt dir als Problem ein?

S: Nun, es gibt möglicherweise mehrere Gruppen für jede Domain. Wenn es nur eine einzige Gruppe für jede Domain gäbe, dann wäre das einfach.

M: Wie könnten wir das erreichen?

S: Nun, wir beginnen mit einem `[ByteString]`. Oh! Oh. Wir können es doch `sort`, oder? Dann wären alle Domains sortiert, nebeneinander, und die `group`-Funktion würde funktionieren!

M: Ja! Lass es uns versuchen.

S: LOS GEHT'S!

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

(muss dem Drang widerstehen, darüber zu sprechen, dass `head` unsicher ist...)

M: Gut gemacht! Wie machen wir nun ein Lookup von, sagen wir, `gmail.com`?

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

M: Ah, aber da ist `lookup` - deutet das nicht auf eine `Map` hin?

S: Eh, klar!

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

Aber das scheint nicht wirklich besser zu sein, oder? Ich schätze, wir haben ein effizienteres Nachschlagen, aber ich denke, wir machen zusätzliche Arbeit, um ein `Map` zu konstruieren.

M: Das tun wir, aber ein Großteil dieser Arbeit ist unnötig. Lass uns die Dokumentation des `Data.Map` Moduls ansehen, um `Map`s zu konstruieren. Anstatt die ganze Arbeit mit Listen zu machen, lass uns versuchen, stattdessen ein `Map` zu konstruieren.

S: Hm. Ich werde mit `foldr` anfangen, da man so eine Liste zerlegt.

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

M: Ein großartiger Start! Nur zur Auffrischung, was ist `acc` und `x` hier?

S: `acc` ist eine `Map` und `x` ist ein `ByteString`.

M: Richtig. Aber von was ist es eine `Map`? Erinnern Sie sich, wir hatten ein `[(ByteString, Int)]`.

S: Oh, `Map ByteString Int`.

M: Richtig. Also, was wollen wir mit dem `ByteString` machen?

S: Es in die `Map` einfügen? Hm, aber was sollte der Wert sein?

M: Wir verfolgen die Anzahl. Das deutet darauf hin, dass wir die `Map` aktualisieren möchten, anstatt einzufügen, wenn wir einen doppelten Schlüsseltreffer haben.

S: Ah! Okay. Schau dir das an:

```
1  func domains =
2      foldr (\domain acc ->
3          case Map.lookup domain acc of
4              Nothing ->
5                  Map.insert domain 1 acc
6              Just previousCount ->
7                  Map.insert domain (previousCount + 1) acc
8      ) Map.empty domains
```

M: Gut gemacht! Wir können es jedoch noch besser machen. Lass uns einen Blick darauf werfen, wie man in der Dokumentation einfügt. Scheint hier etwas vielversprechend zu sein?

S: Hmm. `insertWith` könnte es tun. Lass es mich versuchen:

```
1  func domains =
2      foldr (\domain acc ->
3          Map.insertWith
4              (\newValue oldCount -> newValue + oldCount)
5              domain
6              1
7              acc
8      ) Map.empty domains
```

M: Wunderschön.
Dies ist effizient und erfüllt perfekt unsere Bedürfnisse.
Und Sie haben gelernt, was `Map`s sind!

Haskell zu lehren bedeutet, *zu zeigen*, wie man die Aktion *durchführt*, ebenso sehr wie *zu erklären*, wie man die Konzepte *versteht*.

2.10 Referenzen

- Geschichte der Programmiersprachen[3]
- Generationenliste von Programmiersprachen[4]
- Stilles Wissen[5]

[3] https://en.wikipedia.org/wiki/History_of_programming_languages
[4] https://en.wikipedia.org/wiki/Generational_list_of_programming_languages
[5] https://commoncog.com/blog/tacit-knowledge-is-a-real-thing/

3. Haskeller einstellen

3.1 Das zweischneidige Schwert

Haskell ist ein zweischneidiges Schwert, wenn es um die Einstellung geht.
Dies ist eine durchgängige Erfahrung jedes Einstellungsmanagers, mit dem ich über Haskell gesprochen habe, sowie meine eigenen Erfahrungen beim Durchsehen von Lebensläufen und bei Vorstellungsgesprächen mit Kandidaten.
Eine offene Haskell-Stelle zieht ein fantastisches Verhältnis von hochqualifizierten Kandidaten an.
Unter ihnen befinden sich Doktoranden, erfahrene Haskeller, Senior-Entwickler in anderen Sprachen und einige aufgeregte Junioren, die enormes Potenzial zeigen.
Die Position ist möglicherweise "unter" den Bewerbern, aber Haskell ist ein solcher Vorteil, dass sie dennoch zufrieden sind.

Während die Qualität hoch sein wird, wird die Quantität enttäuschend sein.
Eine Java-Ausschreibung kann 1.000 Bewerbungen anziehen, von denen 25 großartig sind.
Eine Haskell-Ausschreibung kann 50 Bewerbungen anziehen, von denen 10 großartig sind.
Dies ist ein echtes Problem, wenn Sie ein großes Team einstellen müssen.
Haskells Produktivitätsvorteile verringern die Notwendigkeit für ein großes Team, aber das kann man nur so lange hinauszögern.

Sie können ein Haskell-Team ausschließlich durch das Training von Neueinsteigern in die Sprache aufbauen.
Dies erfordert mindestens einen Haskell-erfahrenen Ingenieur mit einer Vorliebe für Mentoring und die Zurückhaltung, den Code so einfach zu halten, dass man leicht damit anfangen kann.
Das ist eine große Herausforderung, aus dem gleichen Grund, dass Komplexität und Neuheit besonders schwierige Probleme in Haskell sind.
Wenn Sie dies lesen, bevor Sie Ihr Haskell-Team starten, dann bitte ich Sie - schreiben Sie Code, den Sie einem Junior beibringen können, ohne zu viel Stress.
Wenn Sie bereits einen komplexen Code haben, müssen Sie wahrscheinlich einen Senior einstellen.

3.2 Junioren und Senioren

Dieses Kapitel wird die Begriffe ‘Senior’ und ‘Junior’ verwenden.
Diese Begriffe sind etwas umstritten, mit einem gewissen Maß an Urteil, und ich möchte sie definieren, bevor wir fortfahren.

Ein Senior-Entwickler hatte die Zeit und Gelegenheit, mehr Fehler zu machen und aus ihnen zu lernen.
Senior-Entwickler sind in der Regel erfahren, abgeklärt und hoffentlich weise.
Senior-Entwickler kennen das Gelände und können im Allgemeinen entweder schwierige Situationen meistern oder sie ganz vermeiden.

Ein Junior-Entwickler ist klug, neugierig und hat noch nicht genug Fehler gemacht.
Junioren sind begeistert, lernen schnell und bringen wichtige neue Perspektiven in ein Projekt ein.
Sie sind keine Belastungen, die schnell zu Senioren werden - ihre neue Energie ist entscheidend für Experimente und die Herausforderung des möglicherweise veralteten Wissens Ihres Senior-Teams.
Das freudige Chaos eines talentierten Juniors kann Ihnen mehr über Ihre Systeme beibringen, als Sie vielleicht für möglich halten.

Eine Person kann zehn Jahre Erfahrung mit Java haben und ein Junior in Haskell sein.
Ein Senior-Haskell-Ingenieur könnte ein Junior in C# sein.
Ein Junior kann erheblich klüger sein als ein Senior.
Eine Person mit zwei Jahren Erfahrung kann erfahrener sein als eine Person mit acht Jahren.
Das relevante Merkmal - für mich - ist die Summe der gemachten Fehler und der gelernten Lektionen.

Ein großes Team und Projekt profitiert davon, sowohl Senioren als auch Junioren zu haben.
Die Haskell-Community als Ganzes profitiert davon, Juniorenrollen zu haben - wie sonst werden wir erfahrene Haskell-Entwickler bekommen, die neue Unternehmen gründen und überzeugende Projekte starten können?
Wenn mir das Praktikum nicht angeboten worden wäre, wäre ich heute kein professioneller Haskell-Entwickler.
Sie würden dieses Buch nicht lesen.
Wir müssen es weitergeben, um die Community zu vergrößern und den Erfolg dieser wunderbaren Sprache zu festigen.

Leider sind die meisten Haskell-Projekte, die ich erlebt habe, fast ausschließlich mit Senior-Entwicklern besetzt.

Es gibt einen Teufelskreis:

1. Alice, eine brillante Haskellerin, bekommt die Möglichkeit, ein Projekt zu starten. Sie ist einzigartig dafür geeignet - sie hat eine Menge Domänenerfahrung und kennt Haskell in- und auswendig.
2. Alice nutzt fortgeschrittene Funktionen und Bibliotheken, um das Projekt zu entwickeln.
   Alice nutzt alle Sicherheits- und Produktivitätsmerkmale, um das Projekt pünktlich, unter Budget und fehlerfrei zu liefern.
   Das Projekt ist ein durchschlagender Erfolg.
3. Das Projekt sammelt neue Funktionen und Verantwortungen an.
   Während Alice in der Lage ist, schnell genug Code zu schreiben, um dieses Wachstum abzudecken, beginnen die anderen Aspekte eines Projekts, einen weiteren Entwickler zu erfordern.
   Haskell macht das Schreiben von Dokumentationen nicht schneller.
4. Alice erstellt eine Liste von Anforderungen für einen neuen Entwickler.
   Um produktiv zu sein, muss der Ingenieur fortgeschrittene Haskell-Tricks verstehen.
   Es bleibt keine Zeit, einen Junior-Ingenieur auszubilden, um produktiv zu sein.

Dies schafft eine immer größere Nachfrage nach erfahrenen Haskellern. Wenn Sie ein erfahrener Haskeller sind, denken Sie vielleicht, dass das in Ordnung ist. Mehr Jobmöglichkeiten und mehr Gehaltswettbewerb!

Dies ist nicht nachhaltig. Das Unternehmen hat immer die Möglichkeit, Haskell abzuschaffen, eine Gruppe von Go/Java/C#-Entwicklern einzustellen und das Haskell-Projekt zu zerstören. Nicht nur wurde ein Haskell-Projekt zerstört, sondern ein weiterer Geschäftsmann hat echte Erfahrungen damit gemacht, dass Haskell gescheitert ist.

3.3 Einstellung von Senioren

Sie werden wahrscheinlich erfahrene Haskell-Ingenieure einstellen müssen. Nach rohen Haskell-Fähigkeiten zu suchen, ist verlockend, aber das ist nicht so notwendig, wie Sie vielleicht denken. Der ursprüngliche Haskell-Entwickler oder die Entwickler können alle schwierigen Teile handhaben, die der neue Mitarbeiter nicht versteht. Stattdessen sollten wir nach den vier Prinzipien dieses Buches suchen:

1. Komplexität: bevorzugt einfache Lösungen

2. Neuheit: bevorzugt traditionelle Lösungen
3. Kohäsion: wird sich nicht auf Stilfragen einlassen
4. Empathie: kann Mitgefühl für andere zeigen

Einstellung ist eine zweiseitige Straße, also schauen wir uns zuerst an, wie Sie die Wahrscheinlichkeit eines erfolgreichen Einstellungsprozesses verbessern können. Darüber hinaus geht es uns nicht nur um das Einstellungsereignis - wir sind auch an der Bindung interessiert.

Remote-freundlich

Wenn Ihr Unternehmen in San Francisco, New York City, Glasgow oder einer Handvoll anderer Haskell-Zentren ist, dann können Sie wahrscheinlich lokal einstellen. Andernfalls müssen Sie Ihre Suche auf vollständig remote Kandidaten ausweiten. Haskell-Entwickler sind weltweit verteilt, und Sie werden die Qualität und Quantität der Haskell-Entwickler erheblich steigern, wenn Sie nicht verlangen, dass sie in Ihre Stadt ziehen.

Die ersten paar Einstellungen sind eine großartige Gelegenheit, Ihre remote-freundlichen Arbeitsabläufe zu entwickeln. Diese Arbeitsabläufe funktionieren fantastisch für viele Unternehmen und Open-Source-Gemeinschaften. Darüber hinaus wird Remote-Arbeit durch die Förderung asynchroner Arbeitspraktiken die Produktivität steigern.

Dies ist kein Buch darüber, wie man ein Remote-Team erfolgreich verwaltet; dafür müssen Sie woanders nachsehen. Ich bin nur ein bescheidener Haskell-Entwickler und kann Ihnen nur sagen, was die Einstellung erheblich erleichtert.

Nicht sparen

Es gibt ein Missverständnis, dass Entwickler bereit sind, niedrigere Gehälter zu akzeptieren, um Haskell zu verwenden. Dies ist im Allgemeinen nicht wahr. Erfahrene Haskell-Ingenieure sind selten und wertvoll, und Sie bekommen, wofür Sie bezahlen. Mein Gehalt und meine Leistungen als Haskell-Ingenieur waren in der Regel wettbewerbsfähig mit dem Markt für meine Rolle und Erfahrung.

Wie bei jedem Missverständnis gibt es in einem bestimmten Kontext einen Kern Wahrheit. Einige Unternehmen zahlen ihren Ingenieuren außergewöhnlich gut. Google, Facebook, Netflix, Indeed usw. sind in der Lage, Gesamtvergütungspakete von über 500.000 USD pro Jahr anzubieten. Ich habe noch nie von einem einzigen Haskell-Entwickler gehört, der so viel verdient, obwohl ich von mindestens einem im Bereich von 300.000 USD gehört habe.

Sie könnten also in der Lage sein, einen Ex-Googler einzustellen, der es gewohnt ist, 400.000 USD zu verdienen, und ihn "nur" 250.000 USD zahlen, um Haskell zu verwenden. Aber Sie sollten nicht erwarten, einen erfahrenen Ingenieur einzustellen und unter 100.000 USD zu zahlen - der Rabatt funktioniert nicht so.

Sie könnten in der Lage sein, einen erfahrenen Scala- oder F#-Ingenieur einzustellen, der noch nie Haskell in der Produktion verwendet hat, zu einem reduzierten Preis. Obwohl dies gut funktionieren könnte, sind Haskell und Scala/F# ausreichend unterschiedlich, dass die Erfahrung nicht so stark übertragbar ist, wie Sie vielleicht erwarten. Produktions-Haskell hat genug Eigenheiten und Besonderheiten, dass bloße Sprachgewandtheit mit funktionalen Programmieridiomen Sie nicht weit bringen wird.

Die meisten Haskell-Unternehmen zahlen ihren erfahrenen Ingenieuren einen wettbewerbsfähigen Tarif. Und die meisten Haskell-Unternehmen, die nur Senioren einstellen, erfordern nicht so viel Produktionserfahrung, um eingestellt zu werden. Wenn Sie einen erfahrenen Scala-Entwickler einstellen, um Haskell zu einem starken Rabatt zu machen, wird er diese Erfahrung nutzen und schnell einen besser bezahlten Haskell-Job bekommen.

Dies zeichnet ein Bild der Haskell-Gehaltslandschaft als bimodal. Erfahrene Haskeller können wettbewerbsfähige Gehälter verdienen, wenn nicht wettbewerbsfähig mit FAANG[1]. Weniger erfahrene Haskeller können einen Gehaltsabzug akzeptieren, um Haskell im Job zu lernen, aber sie werden schnell aufsteigen und in diesen zweiten Eimer gelangen. Denken Sie daran, dass Haskell eine Sprache mit hoher Varianz ist - Sie wetten nicht auf Durchschnitte oder Mediane, Sie wetten darauf, die Kurve zu übertreffen. Statistisch gesehen müssen Sie *besser* als der Marktdurchschnitt zahlen, um aus dem oberen Ende der Kurve auszuwählen.

Wenn Sie einen Junior-Haskeller einstellen (der ansonsten ein Senior-Ingenieur ist), seien Sie bereit, ihm nach einem Jahr eine erhebliche Gehaltserhöhung zu geben oder bereiten Sie sich auf Fluktuation vor.

Kein Ködern und Wechseln

Erfahrene Haskell-Ingenieure kennen das nur allzu gut. Es gibt eine Stellenanzeige, die Haskell als gewünschte Fähigkeit aufführt. Oder vielleicht gibt es eine Haskell-Stelle, aber Sie müssen auch Java, PHP, Ruby und Go kennen. Leider macht Haskell nur einen winzigen Teil dessen aus, was der Entwickler erwartet wird zu tun, obwohl die Stelle als "Haskell-Job" verkauft wird.

[1] Facebook, Amazon, Apple, Netflix, Google. Ein gängiges Akronym für einige der großen Akteure in der Technologiebranche bei der Einstellung, mit einigen der höchsten Vergütungen. Obwohl selten, beschäftigen Facebook und Google einige Haskeller.

Tun Sie das nicht. Sie werden Entwickler frustrieren, die es durchhalten, und Sie werden Haskell-Talente nicht lange halten, wenn Sie sie die meiste Zeit in einer anderen Sprache schreiben lassen. Sie können ihre Leidenschaft für Haskell nicht entführen und sie auf dem gleichen Niveau PHP schreiben lassen. Wie oben, wird der Entwickler in der Lage sein, Haskell in der Produktion in seinen Lebenslauf aufzunehmen und zu einem anderen Unternehmen zu wechseln, um die meiste Zeit an Haskell zu arbeiten.

Das bedeutet nicht, dass Sie keinen polyglotten Tech-Stack haben können. Es ist nichts Falsches daran, Microservices in Go, Ruby und auch Haskell zu haben. Tatsächlich ist es eine gute Möglichkeit, Pragmatic Haskellers statt Puristen auszuwählen, wenn man gelegentlich von einem Haskeller verlangt, eine andere Sprache zu schreiben. Dies muss jedoch ehrlich und im Voraus kommuniziert werden. Wenn Sie erwarten, dass jemand 10% Haskell schreibt, dann bezeichnen Sie es nicht als Haskell-Job. Es ist ein Go-Job mit einer kleinen Haskell-Verantwortung.

Unreinheitsprüfung

Nicht-Haskell-Verantwortlichkeiten können sinnvoll sein, in die Stellenbeschreibung aufzunehmen.

Es gibt einen Typ von Haskell-Entwickler, den ich gesehen habe. Sie wollen nur mit Haskell arbeiten. 100% Haskell. Kein JavaScript, kein Ruby, kein Go, kein Java, kein Bash, kein PHP, keine Sysadmin-Verantwortlichkeiten, nichts! Nur Haskell.

Diese Entwickler sind oft großartig in Haskell und es ist verlockend, sie einzustellen. Das sollten Sie wahrscheinlich nicht tun. Obwohl Haskell eine fantastische Wahl für viele Anwendungen ist, wird jeder, der eine 100% Haskell-Erfahrung benötigt, zwangsläufig Haskell wählen, wo eine andere Wahl angemessener wäre. Das wollen Sie nicht in Ihrem Team. Schlimmer noch, sie könnten unerwünschte Komplexität und Neuheit in das Projekt bringen.

Denken Sie daran, dass das Ziel eines industriellen Softwareprojekts darin besteht, die Bedürfnisse des Unternehmens zu fördern. Haskell tut dies legitim. Wenn ich das nicht glaubte (basierend auf meinen Erfahrungen), würde ich kein Buch darüber schreiben, wie man es erfolgreich macht. Aber Haskell ist keine ausreichende Ursache für Erfolg. Zu wissen, wann man Haskell einsetzt und wann man auf ein anderes Werkzeug zurückgreift, ist entscheidend für jeden gut abgerundeten Haskell-Ingenieur.

Vielfalt umarmen

Haskellers sind seltsam. Sie werden nicht wie typische Programmierer aussehen oder handeln, weil sie es nicht sind! Wenn Sie zu streng auf zusätzliche “Kulturpassung”-

Qualitäten filtern, werden Sie eine Menge guter Ingenieure verpassen. Dies gilt doppelt für unterrepräsentierte Minderheiten.

Das bedeutet nicht, dass Haskell-Entwickler *besser* oder *schlechter* sind als andere, nur dass sie anders sind. Lehnen Sie sich in die Unterschiede hinein und umarmen Sie sie.

3.4 Juniors einstellen

Ein Junior für Haskell ist jemand, der noch nicht genug Fehler gemacht hat. Das kann jemand sein, der erst letztes Jahr angefangen hat zu programmieren und irgendwie Haskell ausgewählt hat, oder es könnte ein erfahrener Scala-Entwickler mit 10 Jahren Berufserfahrung sein, der gerade angefangen hat, Haskell zu lernen. Einen guten Kandidaten für eine Junior-Rolle auszuwählen, ist etwas anders als in anderen Programmiersprachen.

Haskell hat eine viel kleinere Population als andere Programmiersprachen. Die Population ist auf viele Gemeinschaften verteilt. Die insgesamt verfügbare Unterstützung für Juniors ist geringer als in anderen Sprachen. Das bedeutet, dass Sie den Mangel ausgleichen müssen.

In eine Kultur der Schulung und Mentorschaft zu investieren, ist eine großartige Möglichkeit, dies zu erreichen. Direktes Mentoring Ihrer Juniors fördert die Kohäsion im Projekt. Senior-Entwickler erhalten wertvolle Übung im Lehren und gewinnen Einblicke von den Juniors.

Die Auswahlwirkungen von Haskell bedeuten, dass Sie wahrscheinlich weniger Mentoring benötigen, als Sie vielleicht erwarten. Haskell ist ausreichend seltsam und anders, dass die meisten Menschen, die sich dafür begeistern, bereits selbständig sind und hervorragend darin sind, eigenständig zu recherchieren. Die Arbeit des Mentors besteht weniger darin, zu “lehren”, sondern mehr darin, zu “leiten”.

Alle obigen Ratschläge für die Einstellung von Senioren gelten auch für Juniors. Sie sollten definitiv versuchen, frühzeitig im Lebenszyklus eines Projekts einen Junior Haskeller einzustellen, aus einigen wichtigen Gründen. Ein Senior Haskell-Ingenieur wird dramatisch produktiver sein als in einer anderen Sprache, aber die Gesamtarbeitsbelastung eines Ingenieurs ist nur teilweise technisch. Junior-Ingenieure sind bemerkenswert gut geeignet für viele der Aufgaben im Software-Engineering, die nicht direkt mit technischer Kompetenz und Erfahrung zu tun haben. Diese Arbeit dient auch als ausgezeichnetes Training für den Junior.

Unterstützende Aufgaben

Ein erfahrener Haskeller kann Funktionen schneller bereitstellen als in jeder anderen Sprache, mit weniger Zeitaufwand für Fehlerbehebungen. Leider nimmt die Dokumentation nicht weniger Zeit in Anspruch, um geschrieben zu werden. Wenn Sie für die Fähigkeit einstellen, Funktionen zu liefern, dann werden Sie nicht die Personenstunden haben, um Dokumentation zu schreiben oder andere Formen der Unterstützung für das Projekt bereitzustellen.

Junior-Entwickler haben möglicherweise nicht die gleiche Fähigkeit, Code zu schreiben wie Senioren, aber sie sind vergleichsweise weniger benachteiligt beim Schreiben von Dokumentation und der Unterstützung des Projekts in anderer Weise. Das Schreiben dieser Dokumentation und die Unterstützung des Codes geben ihnen hervorragende Erfahrungen mit dem Code, was ihnen hilft, ihr Wissen und ihre Fähigkeiten weiterzuentwickeln.

Das soll nicht heißen, dass Senioren keine Dokumentation schreiben sollten. Sie sollten es unbedingt tun! Aber die Dokumentation eines Seniors könnte den Kontext vermissen, der für jemanden, der tief im Projekt eingebunden ist, nicht offensichtlich ist. Die Dokumentation, die ein Junior schreibt, wird oft umfassender sein und weniger über den Leser voraussetzen, es sei denn, der Senior ist ein besonders guter technischer Autor.

Der Prozess der Überprüfung dieser Dokumentation ist eine ausgezeichnete Gelegenheit für einen Senior, zusätzliche Informationen und Klarstellungen für den Junior bereitzustellen.

Konzepte klären

Erfahrene Haskeller werden die Komplexität eines Projekts natürlich erhöhen, es sei denn, sie wenden konsequent Anstrengungen an, dies zu vermeiden. Ein Senior Haskeller ist jedoch schlecht positioniert, um Entscheidungen darüber zu treffen, wie viel Komplexität genau eingeführt wird. Schließlich kennen und verstehen sie es oder haben bereits die Arbeit erledigt, um es herauszufinden. Der umgebende Kontext und die Annahmen sind im Hintergrund.

Der Akt, Entscheidungen und Konzepte einem Junior zu erklären, ist eine erzwungene Funktion, um die Komplexität zu identifizieren, die damit verbunden ist. Wenn die Idee für den Junior im Team zu kompliziert ist, um sie ohne erhebliche Anleitung zu verstehen, dann ist sie zu kompliziert!

Das soll nicht heißen, dass Junioren nicht in der Lage sind, komplexe Ideen zu verstehen, oder dass ein Junior ein Vetorecht über jedes Konzept in einer Codebasis haben sollte.

Junioren bieten eine kraftvolle neue Perspektive, die Entscheidungen informieren kann. Diese Perspektive zu respektieren ist entscheidend, aber ihr vollständig nachzugeben ist unnötig. Einen Junior dabei zu unterstützen, komplexere Ideen zu lernen und praktische Erfahrungen damit zu sammeln, ist Teil des Prozesses.

Institutionelles Wissen

Angenommen, Ihr Haupt-Haskell-Entwickler gewinnt im Lotto und kündigt. Sie müssen diese Person ersetzen. Ein Junior könnte sogar die Gelegenheit ergreifen und den kürzlich ausgeschiedenen Entwickler vollständig ersetzen. Wir sollten nicht erwarten oder Druck auf sie ausüben, dies zu tun.

Der Junior wird jedoch in einer ausgezeichneten Position sein, dieses institutionelle Wissen zu bewahren. Sie können helfen, neue Kandidaten zu interviewen und Einblicke darüber zu geben, was die Codebasis benötigt. Die Natur eines Junior-Entwicklers bietet ihnen einen ausgezeichneten Einblick, was helfen wird, die Codebasis zu erweitern und über die Zeit wartbar zu halten. Ein Senior-Ingenieur, der nicht in der Lage ist, einem Junior-Ingenieur etwas beizubringen oder zu erklären, wird keine großartige Ergänzung sein.

Junioren sind in einer besseren Position, um diese Transfers zu machen, weil sie weniger internalisierte Annahmen als Senioren haben. Dies gibt ihnen eine bessere Perspektive bei der Bewertung und Weitergabe von Wissen an neue Mitarbeiter.

4. Bewertung von Beratungsfirmen

Möglicherweise müssen Sie Berater einstellen, um an Ihrem Projekt zu arbeiten. Haskell-Beratungsfirmen können eine ausgezeichnete Quelle für tiefes Fachwissen sein. Aufgrund der Nischenart der Sprache gibt es nicht viele Haskell-Beratungsfirmen, und sie sind alle brillant. Ich erwarte nicht, dass dies für immer so bleiben wird, also werde ich teilen, wie ich Beratungsfirmen bewerte, um herauszufinden, wo sie am besten angewendet werden könnten.

4.1 Identifizierung des Ziels

Wenige Beratungsfirmen senden potenzielle Geschäfte an ihre Konkurrenz. Infolgedessen wird eine Beratungsfirma Ihr Geschäft gerne annehmen, selbst wenn Sie von einem anderen Unternehmen besser bedient werden könnten. Genau wie die Haskell-Sprache viele Gemeinschaften hat, sind diese Beratungsfirmen normalerweise besser für einige Gemeinschaften als für andere geeignet.

Alle Beratungsfirmen werden sagen, dass sie sich auf industrielles Haskell spezialisieren. Ihre Ansätze unterscheiden sich jedoch, und einige sind mehr oder weniger für verschiedene Anwendungsbereiche in dieser Nische geeignet.

Haskell-Beratungsfirmen werben über Open-Source-Portfolios und Blogbeiträge. Diese Portfolios bilden einen Beweis für die Arbeit und können analysiert werden, um die richtige Passform zu bestimmen. Viele der Techniken zur Bewertung von Beratungsfirmen erfordern die Bewertung von Bibliotheken, und ich behandle das nicht, bis Abschnitt 5 ("Interfacing the Real"). Wir können jedoch eine allgemeine Vorstellung vom Zielmarkt bekommen, ohne zu tief einzutauchen.

Zuerst schauen wir uns die Website, den Blog und andere Marketingmaterialien an. Beratungsfirmen richten sich im Allgemeinen an ihre Nischen, und wenn sie nicht auf Ihre Bedürfnisse eingehen, sind sie wahrscheinlich keine gute Passform. Beratungsfirmen erhalten Material für Blogbeiträge aus unabhängiger Forschung und aus Lektionen, die sie in der Beratungsarbeit gelernt haben, daher ist dies eine gute Möglichkeit zu sehen, wie sie mit auftretenden Problemen umgehen. Zusätzlich bekommen Sie ein Gefühl für ihren Kommunikationsstil (zumindest, wie er der Welt präsentiert wird).

Als nächstes wollen wir die Bibliotheken bewerten, die die Beratungsfirma unterstützt. Dies gibt uns wichtige Informationen darüber, wie sie Code schreiben und welche Ansätze sie unterstützen. Der einfachste Weg, dies zu tun, besteht darin, nach der Hauptquelle der Code-Repository-Organisation für die Beratungsfirma zu suchen (oft GitHub, aber auch GitLab und BitBucket sind Möglichkeiten). Wir werden auch die GitHub-Konten von Mitarbeitern ansehen wollen, wenn wir sie finden können. Beratungsfirmen stellen in der Regel Ingenieure wegen ihrer Open-Source-Beiträge ein - wenn Sie den Ingenieur einstellen, der die X-Bibliothek unterstützt, können Sie den Benutzern dieser Bibliothek Beratungsdienste anbieten.

Schließlich wollen wir versuchen, Erfahrungsberichte von Unternehmen zu finden, die diese Beratungsfirmen genutzt haben. Mitarbeiter, die an Projekten gearbeitet haben, die von Beratungsfirmen unterstützt oder abgeschlossen wurden, sind hier eine weitere wertvolle Ressource. Diese Informationen werden schwieriger zu beschaffen sein. Unternehmen wollen selten solche Details veröffentlichen, daher werden Sie sie wahrscheinlich eher durch Community-Beteiligung erhalten. Einzelpersonen werden keine negativen Erfahrungsberichte veröffentlichen, da Beratungsfirmen dazu neigen, einen übergroßen Einfluss auf die öffentliche Meinung in kleinen Gemeinschaften wie Haskell zu haben.

Lassen Sie uns einige der größeren Beratungsunternehmen untersuchen.

4.2 Well-Typed

Well-Typed präsentiert sich als “Die Haskell-Berater”. Mit Schwergewichten der Community wie Duncan Coutts, Andres Löh und Edsko de Vries haben sie sicherlich Anspruch auf diesen Titel. Das Unternehmen ist seit 2008 aktiv. Dies ist ein solides Fundament für Erfolg, und sie haben eine Erfolgsbilanz, die dies untermauert.

Fast jeder in der Mitarbeiterliste hat einen Abschluss in Informatik, und mehr haben einen Doktortitel als nur einen Bachelor-Abschluss. Der akademische Hintergrund bei Well-Typed ist gut vertreten.

Das GitHub unter `https://www.github.com/well-typed` listet eine Reihe von Repositories auf, die es zu untersuchen gilt. Ich werde hier einige auswählen:

- `optics`, eine alternative `lens`-Bibliothek, die stark verbesserte Fehlermeldungen bietet
- `generics-sop`, eine Alternative zu `GHC.Generics`
- `ixset-typed`, eine stark typisierte indizierte Datenstruktur

- `cborg`, eine Binärserialisierungsbibliothek

Darüber hinaus sehen wir eine Reihe von Beiträgen zum `cabal`-Repository von Well-Typed-Mitarbeitern, zusammen mit anderer wichtiger Haskell-Infrastruktur. Die Hauptverantwortlichen für die Servant-Webbibliothek sind bei Well-Typed beschäftigt. Die `acid-state`-Datenbank wird ebenfalls von Well-Typed-Mitarbeitern gewartet.

Ich habe direkt mit Well-Typed zusammengearbeitet, während ich bei IOHK angestellt war. Das starke theoretische Wissen war entscheidend für die Entwicklung vieler der hochtheoretischen Aspekte der Codebasis. Das ebenso starke technische Wissen für die Haskell-Entwicklung war hervorragend für die Entwicklung des Wallet-Teils der Codebasis.

Well-Typed liefert extrem starke theoretische Kenntnisse und Haskell-Expertise. Dies offenbart jedoch eine operative Schwachstelle: die Abhängigkeit von Haskell, wo andere Werkzeuge möglicherweise besser geeignet sind. Die Verwendung von `acid-state` bei IOHK war die Quelle zahlreicher Probleme, die im Datenbankkapitel dieses Buches dokumentiert sind. Darüber hinaus spiegelt sich die extrem hohe Kompetenz der Well-Typed-Mitarbeiter in der Komplexität und Schwierigkeit der gelieferten Lösungen wider.

Ich würde nicht zögern, Well-Typed für ein Projekt im industriellen Einsatz zu engagieren, insbesondere wenn das Projekt neuartige theoretische Erkenntnisse erfordert. Ich wäre jedoch vorsichtig, um sicherzustellen, dass die resultierende Lösung leicht vom Kernteam der Ingenieure verstanden werden kann. Die Schulungen von Well-Typed sind hervorragend, um einen Haskeller auf mittlerem oder fortgeschrittenem Niveau auf die nächste Stufe zu bringen.

4.3 FP Complete

FP Complete bezeichnete sich früher hauptsächlich als Haskell-Berater, hat sich jedoch in den letzten Jahren auch auf DevOps und Blockchain konzentriert. Michael Snoyman ist der Leiter der Technik, und abgesehen davon listet ihre Website keine Ingenieure auf. Ihr Blog enthält Beiträge zu vielen Themen, einschließlich Rust, DevOps, Container und Haskell.

Michael Snoyman und Aaron Contorer, die beiden treibenden Mitglieder von FP Complete, haben keinen umfangreichen Hintergrund in der akademischen Welt oder der Informatiktheorie. Michaels Abschluss ist in Versicherungsmathematik, während Aaron sich bei Microsoft auf aufkommende Technologien spezialisierte. Der Ansatz des Unternehmens wird in erster Linie von industriellen Bedürfnissen geprägt. Dies hilft,

ihre vielfältigere Ausrichtung zu erklären - Haskell spielt eine prominente Rolle, aber DevOps, Rust und andere Technologien sind wichtig für ihre Geschäftsstrategie und ihr Marketing.

Das GitHub unter `https://www.github.com/fpco` bietet ein paar weitere Hinweise. Es gibt mehrere Mitglieder der Organisation, die aufgeführt sind: Niklas Hambüchen, Sibi Prabakaran, Chris Done stechen als Haskell-Beitragende hervor. Das FPCo GitHub hat viele Repositories, die wir prüfen können:

- `safe-exceptions`, eine Bibliothek zur Unterstützung bei sicherem und vorhersehbarem Exception-Handling
- `stackage-server`, der Code, der das Stackage-Paketset hostet
- `weigh`, eine Bibliothek zur Messung von Speicherzuweisungen von Haskell-Funktionen
- `resourcet`, eine Bibliothek für sichere und zeitnahe Ressourcenverwaltung

Andere relevante Bibliotheken umfassen das Yesod-Web-Framework, die Persistent-Datenbankbibliothek und das `stack`-Build-Tool.

Ich habe nicht direkt mit FP Complete gearbeitet, habe jedoch umfangreiche Erfahrungen mit Yesod, Persistent und habe direkt mit Michael Snoyman zusammengearbeitet. Ich habe diese Bibliotheken verwendet, um schnell und effektiv funktionierende und wartbare Software in meinem ersten Job zu liefern, und der Fokus auf reale industrielle Anliegen führte zu meinem Erfolg dort. Die Bibliotheken sind in der Regel einfach zu handhaben, nehmen regelmäßig Beiträge von Neulingen an und sind nicht allzu streng in Bezug auf Codierungsstandards. Dies ist ein zweischneidiges Schwert - viele Bibliotheken werfen häufiger Ausnahmen, als es Programmierern lieb ist, anstatt mit typisierten Fehlerkanälen zu signalisieren. Template Haskell wird oft verwendet, um Boilerplate zu reduzieren und Typsicherheit zu bieten, eine Wahl, die pragmatisch ist, aber unter eher ‘reinen’ funktionalen Persönlichkeiten unpopulär.

Ich würde nicht zögern, FP Complete für den industriellen Einsatz zu engagieren, insbesondere wenn das Projekt keine neuartigen theoretischen Anforderungen hat. Die Schulung von FP Complete hat sich bewährt, um Junioren mit Haskell vertraut zu machen, und sie sind in der Lage, auch auf fortgeschrittene und knifflige GHC-Verhaltensweisen zu schulen.

Ausnahmen

FP Complete hat den definitiven Artikel über sicheres Ausnahmehandling in Haskell[1] geschrieben. Es mag nicht überraschen, dass ihre Bibliotheken dazu neigen, Laufzeit-

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

ausnahmen häufiger auszulösen, als man erwartet oder möchte. Einige Bibliotheken im Haskell-Ökosystem verwenden den `ExceptT`-Monadentransformator, um Ausnahmen anzuzeigen. FP Complete hält dies für ein Antimuster und hat einen Artikel geschrieben[2], der dies erklärt. Stattdessen kann man erwarten, dass `IO`-Funktionen in FP Complete Bibliotheken Laufzeitausnahmen auslösen.

`TemplateHaskell`

FP Completes Bibliotheken neigen dazu, `TemplateHaskell` extensiv für Funktionalität zu verwenden. Yesod verwendet einen `QuasiQuoter`, um Routen für die Web-App zu definieren. Shakespeare verwendet einen `QuasiQuoter`, um Werte zu interpolieren. `monad-logger` verwendet `TemplateHaskell`-Logging-Funktionen, um den Ort der Logzeile einzufügen. `persistent` verwendet einen `QuasiQuoter`, um Typen für die Interaktion mit der Datenbank zu definieren.

`TemplateHaskell` und `QuasiQuoter`s werden oft unter Haskellers kritisiert. Sie haben einige Nachteile. Jede Verwendung von `TemplateHaskell` in einem Modul erfordert, dass GHC einen Code-Interpreter startet - dies verlangsamt die Kompilierung mit einem konstanten Schlag von ein paar hundert Millisekunden auf meinem Laptop. Das Generieren des Codes ist jedoch in der Regel recht schnell. Wenn der resultierende generierte Code extrem groß ist, wird das Kompilieren dessen langsam sein.

`QuasiQuoter`s definieren eine separate Sprache, die in einen Haskell-Ausdruck geparst wird. Eine separate Sprache hat einige Vorteile: Man kann genau das definieren, was man will und braucht, ohne sich um die Beschränkungen von Haskell kümmern zu müssen. Leider muss man seine eigene Syntax und Parser erfinden. Diese Dinge müssen dokumentiert und diese Dokumente aktuell gehalten werden. Der vom `QuasiQuoter` generierte Code ist oft nicht zur Inspektion geeignet - man kann nicht "Zur Definition springen" bei einem Typ, der von `TemplateHaskell` generiert wird, noch kann man den Code leicht einsehen.

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

5. Invertiere Deine Mocks!

Mocking wird häufig in Diskussionen über das Testen von effektvollem Code in Haskell erwähnt. Ein Vorteil von `mtl` Typklassen oder `Eff` Freien Monaden ist, dass du Implementierungen austauschen und dasselbe Programm auf unterschiedlichen zugrunde liegenden Interpretationen ausführen kannst. Das ist cool! Allerdings ist es eine extrem schwergewichtige Technik mit einer Menge Komplexität.

Im vorherigen Kapitel habe ich empfohlen, mit dem `ReaderT` Muster zu arbeiten - so etwas wie das:

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

Wie würde ich nun vorgehen, um eine solche Funktion zu testen?

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

Wenn wir unsere `mtl`- oder `Eff`- oder OOP-Mocking-Hüte aufhaben, könnten wir denken:

> Ich weiß! Wir müssen unsere HTTP-, Datenbank- und Redis-Effekte simulieren. Dann können wir die Umgebung mit Mock-Implementierungen kontrollieren und überprüfen, ob die Ergebnisse stimmig sind!

Mocking ist schrecklich. Es verkompliziert jeden Aspekt unseres Codebasises und führt nicht einmal zu zuverlässigen Tests. Ich werde Techniken zum Mocking in einem späteren Teil des Buches behandeln, aber es wäre erheblich angenehmer, wenn wir es

nie tun müssten. Wer weiß - vielleicht müssen Sie es nie! Aber zuerst müssen wir Wege finden, unseren Code zu testen, ohne uns auf Mocking zu verlassen.

Lassen Sie uns einen Schritt zurücktreten und einige grundlegendere Techniken auf dieses Problem anwenden.

5.1 Effekte Dekomponieren

Das Erste, was wir tun müssen, ist zu erkennen, dass *Effekte* und *Werte* getrennt sind und versuchen, sie so weit wie möglich getrennt zu halten. Die Trennung von Effekten und Werten ist ein grundlegendes Prinzip der rein funktionalen Programmierung. Allgemein gesprochen sind Funktionen, die wie `doWork` aussehen, nicht funktional (im Sinne der "funktionalen Programmierung"). Schauen wir uns die Typsignatur an, um ein paar Hinweise zu finden.

```
1  doWork :: App ()
```

Unsere erste Warnung ist, dass diese Funktion keine Argumente hat. Das bedeutet, dass alle Eingaben in diese Funktion aus der `App`-Umgebung kommen müssen. Diese Eingaben sind *Effekte*.

Ebenso gibt diese Funktion `()` zurück - den Einheitstyp, was nichts bedeutet. Es gibt hier keinen sinnvollen Wert. Wenn diese Funktion *überhaupt* etwas tut, muss es ein Nebeneffekt sein.

Schauen wir uns also noch einmal an, was die Funktion tut. Wir müssen die Funktion zerlegen, bevor wir sie testen können.

```
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)
```

Wir erhalten eine Menge von Dingen - Eingaben -, die als Ergebnis eines *Effekts* erworben werden. Um dies direkt zu testen, müssen wir irgendwie den Effekt abfangen und einen anderen Wert bereitstellen. Dies ist in Haskell unangenehm zu tun.

Stattdessen lassen Sie uns dies in zwei Funktionen aufteilen. Die erste wird für die Durchführung der Eingabeeffekte verantwortlich sein. Die zweite akzeptiert die *Ergebnisse* dieser Eingabeeffekte als einen Parameter der reinen Funktion.

```
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)
```

Nun, um `doWorkHelper` zu testen, müssen wir die Effekte, die den `[User]` herausholen, nicht mocken. Wir können in unseren Tests beliebige `[User]` bereitstellen, ohne einen gefälschten HTTP-Dienst und eine Datenbank orchestrieren zu müssen.

Jetzt sind die einzigen verbleibenden Effekte in `doWorkHelper` `getSomething` und `runRedis`. Aber ich bin nicht zufrieden. Wir können `getSomething` loswerden, indem wir einen weiteren Helfer ausgliedern. Wir folgen demselben Muster: den Eingabeeffekt aufrufen, die Werte sammeln und sie als Eingaben für eine neue Funktion bereitstellen.

```
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)
```

Wir haben nun alle "Eingabeeffekte" extrahiert. Die Funktion `lookMaNoInputs` führt, wie der Name schon sagt, nur *Ausgabeeffekte* aus. Wenn wir dies testen wollen, können wir jede beliebige `[(Thing, User)]` bereitstellen.

Allerdings stecken wir immer noch mit unseren Ausgabeeffekten fest. Wenn wir dies testen wollen, müssten wir überprüfen, ob sich die `App`-Umgebung (oder die reale Welt) tatsächlich so verändert hat, wie wir es erwarten. Glücklicherweise haben wir dafür einen Trick in petto. Lassen Sie uns unseren Ausgabeeffekt untersuchen:

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

Es erwartet zwei Dinge:

1. Den Redis-Schlüssel des Benutzers
2. Das berechnete Ergebnis von `thing`.

Wir können den Redis-Schlüssel und das berechnete Ergebnis ziemlich einfach vorbereiten:

```
1  businessLogic :: (Thing, User) -> (RedisKey, Result)
2  businessLogic (thing, user) = (userRedisKey user, compute thing)
3
4  lookMaNoInputs :: [(Thing, User)] -> App ()
5  lookMaNoInputs users = do
6      for_ (map businessLogic users) $ \(key, result) -> do
7          runRedis (writeKey key result)
```

Toll! Wir haben die Kern-Geschäftslogik isoliert, und jetzt können wir schöne Unittests für diese Geschäftslogik schreiben. Das Tupel ist etwas irrelevant - die `userRedisKey`-Funktion und der `compute thing`-Aufruf sind völlig unabhängig. Wir können Tests für `compute` und `userRedisKey` unabhängig schreiben. Die *Komposition* dieser beiden Funktionen sollte *auch* in Ordnung sein, selbst ohne `businessLogic` selbst zu testen. Die gesamte Geschäftslogik wurde aus dem effektreichen Code herausgenommen, und wir haben die Menge des zu testenden Codes reduziert.

Nun, Sie möchten vielleicht dennoch Integrationstests für die verschiedenen effektreichen Funktionen schreiben. Es ist wichtig zu überprüfen, dass *diese* korrekt funktionieren. Allerdings möchten Sie sie nicht immer wieder testen. Sie möchten Ihre Geschäftslogik unabhängig von Ihrer effektreichen Logik testen.

5.2 Streaming-Zerlegung

Streaming-Bibliotheken wie `Pipes` und `Conduit` sind eine großartige Möglichkeit, große Datensätze zu verarbeiten und Effekte zu überlagern. Sie sind *auch* eine großartige Möglichkeit, Funktionen zu zerlegen und "inverted mocking" Möglichkeiten in Ihre Programme zu integrieren. Sie haben vielleicht bemerkt, dass unser Refactoring im vorherigen Abschnitt von einer einzigen Iteration über die Daten zu mehreren Iterationen geführt hat. Zuerst haben wir die `[User]` geholt, und für jeden `User` eine Anfrage gestellt und nach Redis geschrieben. Aber die endgültige Version iteriert über die `[User]` und paart sie mit der Anfrage. Dann iterieren wir erneut über das Ergebnis und schreiben gleichzeitig nach Redis.

Wir können `conduit` verwenden, um den zusätzlichen Durchlauf zu vermeiden, während wir unseren Code schön strukturiert und testbar halten.

Die meisten Conduits sehen so aus:

```
1  import Data.Conduit (runConduit, (.|))
2  import qualified Data.Conduit.List as CL
3
4  streamSomeStuff :: IO ()
5  streamSomeStuff = do
6      runConduit
7           $ conduitThatGetsStuff
8          .| conduitThatProcessesStuff
9          .| conduitThatConsumesStuff
```

Der Pipe-Operator (`.|`) kann wie eine Unix-Pipe betrachtet werden - "nehmen Sie die gestreamten Ausgaben vom ersten `Conduit` und stecken Sie sie als Eingaben in das zweite `Conduit` ein." Der erste Teil eines `Conduit` ist der "Produzent" oder die "Quelle." Dies kann aus einer Datenbankaktion, einer HTTP-Anfrage oder von einem Datei-Handle stammen. Sie können auch aus einer einfachen Liste von Werten produzieren.

Schauen wir uns `conduitThatGetsStuff` an - es produziert die Werte für uns.

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

-- Abbreviated
type ConduitT i o m r

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

`conduitThatGetsStuff` akzeptiert `()` als Eingabe. Dies signalisiert, dass es hauptsächlich genutzt wird, um Dinge zu *produzieren*, insbesondere im `monad`-Typ. Daher kann `conduitThatGetsStuff` `IO`-Effekte ausführen, um `ByteString`-Blöcke zu erzeugen. Wenn der Conduit seine Ausführung beendet hat, gibt er `()` zurück – oder, nichts Wichtiges.

Der nächste Teil des Conduits ist `conduitThatProcessesStuff`. Diese Funktion befindet sich genau hier:

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

Diese `ConduitT` akzeptiert `ByteString` als Eingabe, gibt `RealThing` als Ausgabe aus und arbeitet in `IO`. Wir beginnen, indem wir Werte in ein `Either` parsen. Der zweite Teil der Pipeline wirft eine Ausnahme, wenn der vorherige Schritt `Left` zurückgegeben hat, oder gibt das `Right` an den nächsten Teil der Pipeline weiter. `CL.map` führt eine Umwandlung durch, und `CL.filter` gibt nur `RealThing`s weiter, die eine Bedingung erfüllen.

Schließlich müssen wir tatsächlich *etwas* mit dem `RealThing` machen.

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

Dies `printet` jedes Element, bevor es an `makeHttpPost` übergeben wird, das schließlich an `saveToDatabase` weiterleitet.

Wir haben eine Menge kleiner, zerlegter Dinge. Unser `conduitThatProcessesStuff` ist es egal, woher es die `ByteString`s erhält, die es parst – Sie können es mit *jedem* `ConduitT i ByteString IO r` verbinden. Datenbanken, HTTP-Aufrufe, Datei-IO oder sogar einfach nur `CL.sourceList [example1, example2, example3]`.

Ebenso ist es dem `conduitThatConsumesStuff` egal, woher die `RealThing`s kommen. Sie können `CL.sourceList` verwenden, um eine Reihe von Fake-Input bereitzustellen.

Normalerweise arbeiten wir hier auch nicht direkt mit `Conduit`s – die meisten Funktionen werden `CL.mapM_`, `CL.filter` oder `CL.map` bereitgestellt. Das ermöglicht es uns, Funktionen zu schreiben, die einfache `a -> m b` oder `a -> Bool` oder `a -> b` sind, und diese sind wirklich einfach zu testen.

`doWork`: im Conduit-Stil

Oben hatten wir `doWork`, und wir haben es in mehrere kleine Funktionen zerlegt. Obwohl wir zuversichtlich sein können, dass es die Eingabeliste effizient verarbeitet, sind wir nicht garantiert, dass es in einer konstanten Menge an Speicher funktioniert. Die ursprüngliche Implementierung machte einen einzigen Durchlauf über die Benutzerliste. Die zweite macht konzeptionell drei: der erste `for_`, um die sekundären Eingaben zu erfassen, der Aufruf von `map businessLogic` und der abschließende `for_`, um den Ausgabe-Effekt auszuführen. Wenn es mehr Durchgänge gäbe und wir sofortige Effekte garantieren wollten, könnten wir einen Conduit verwenden.

Lassen Sie uns also `doWork` als `ConduitT` umschreiben. Zuerst möchten wir einen Produzenten, der unsere `User`-Datensätze nach unten weitergibt.

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

Nun definieren wir einen Kanal, der eine Sache für einen Benutzer erhält und sie weitergibt.

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

Eine andere Leitung berechnet das Ergebnis.

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

Der letzte Schritt in der Pipeline ist es, das Ergebnis zu verwenden.

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

Die zusammengestellte Lösung ist hier:

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

Dies hat die gleiche Effizienz wie die ursprüngliche Implementierung und verarbeitet die Dinge auch in der gleichen Reihenfolge. Wir konnten jedoch die Effekte extrahieren und sie trennen. Das `computeResult :: ConduitT _ _ _` ist *rein* und kann getestet werden, ohne irgendein `IO` auszuführen.

Selbst wenn man annimmt, dass `computeResult` *in* einfachem `IO` wäre, ist das leichter zu testen als ein potenziell komplexer `App`-Typ.

5.3 Einfachste Abstraktion

Denken Sie immer an die leichtesten und allgemeinsten Techniken in der funktionalen Programmierung:

1. Machen Sie es zu einer Funktion
2. Abstrahieren Sie einen Parameter

Diese werden Sie weit bringen.

Lassen Sie uns das `doWork` Geschäft oben noch einmal betrachten:

```
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)
```

Wir können dies *abstrakt* machen, indem wir konkrete Begriffe nehmen und sie zu Funktionsparametern machen. Die wörtliche Definition der lambda abstraction!

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

Es gibt einige interessante Dinge, die man über diese abstrakte Definition beachten sollte:

1. Sie ist über *jede* Monad parametrisiert. `Identity`, `State`, `IO`, was auch immer. Sie haben die Wahl!
2. Wir haben eine reine Spezifikation der Effektlogik. Diese kann *nichts* tun. Sie beschreibt nur, was zu tun ist, wenn die richtigen Werkzeuge gegeben sind.
3. Das ist im Grunde Dependency Injection im Extremfall.

Angesichts der oben genannten abstrakten Definition können wir die konkrete `doWork` leicht wiederherstellen, indem wir die nötigen Funktionen bereitstellen:

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

Wir können auch problemlos eine Testvariante erhalten, die die durchgeführten Aktionen protokolliert:

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

Alles, ohne sich mit Monad-Transformatoren, Typklassen oder sonst etwas schrecklich Kompliziertem herumschlagen zu müssen.

5.4 Zerlegen!!!

Letztendlich geht es darum, Programme in ihre kleinsten, am einfachsten testbaren Teile zu zerlegen. Diese kleinen Teile werden dann einzeln oder anhand von Eigenschaften getestet, um sicherzustellen, dass sie zusammenarbeiten. Wenn alle Teile unabhängig funktionieren, sollten sie auch in der Komposition zusammenarbeiten.

Ihre Effekte sollten idealerweise nicht in der Nähe Ihrer Geschäftslogik sein. Reine Funktionen von `a` nach `b` sind unglaublich einfach zu testen, besonders wenn Sie Eigenschaften ausdrücken können.

Wenn Ihre Geschäftslogik wirklich Effekte ausführen muss, versuchen Sie zunächst die einfachsten Techniken: Funktionen und Abstraktionen. Ich glaube, dass das Schreiben und Testen von Funktionen, die reine Werte verarbeiten, einfacher und leichter ist. Diese sind unabhängig davon, *woher* die Daten kommen, und müssen überhaupt nicht gemockt werden. Diese Transformation ist typischerweise einfacher als die Einführung von `mtl`-Klassen, Monad-Transformatoren, `Eff` oder ähnlichen Techniken.

5.5 Was, wenn ich *muss*?

Manchmal kann man es einfach nicht vermeiden, effektvollen Code zu testen. Ein häufiges Muster, das mir aufgefallen ist, ist, dass Leute Dinge auf einer viel zu niedrigen Ebene abstrakt machen wollen. Sie möchten die Abstraktion so *schwach* wie möglich machen, um sie *leicht* mocken zu können.

Betrachten Sie den häufigen Fall, die Datenbank mocken zu wollen. Das ist verständlich: Datenbankaufrufe sind extrem langsam! Eine Mock-Datenbank zu implementieren, ist jedoch eine extrem schwierige Aufgabe – im Grunde müssen Sie eine Datenbank implementieren. Wo sich das Verhalten der Datenbank von Ihrem Mock unterscheidet, werden Sie einen Test-/Produktionsmismatch haben, der irgendwann explodieren wird.

Gehen Sie stattdessen eine Ebene höher - schaffen Sie eine neue Indirektionsebene, die sowohl von der Datenbank als auch von einem einfach zu implementierenden Mock erfüllt werden kann. Sie können dies mit einer Typklasse tun oder einfach durch konkrete Abstraktion der relevanten Funktionen. Die Abstraktion der relevanten Funktionen ist die einfachste und unkomplizierteste Technik, aber es ist auch nicht unvernünftig zu schreiben:

```
1  data UserQuery
2      = AllUsers
3      | UserById UserId
4      | UserByEmail Email
5
6  class Monad m => GetUsers m where
7      runUserQuery :: UserQuery -> m [User]
```

Dies ist eine *weitaus* tragfähigere Schnittstelle zu implementieren als eine SQL-Datenbank! Schreiben wir unsere Instanzen, eine für die `persistent`[1] Bibliothek und eine andere für ein Mock, das den `Gen`-Typ von QuickCheck verwendet:

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

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

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

Alternativ können Sie einfach Funktionen manuell übergeben, anstatt den Typeclass-Mechanismus zu verwenden, der dies für Sie übernimmt.

Oh, warte, nein! Diese `GetUsers Gen`-Instanz hat einen Fehler! Können Sie erraten, was es ist?

Im Fall von `UserById` und `UserByEmail` testen wir nie den "leere Liste"-Fall - was, wenn dieser Benutzer nicht existiert?

Eine korrigierte Variante sieht so aus:

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

Ich habe einen Fehler gemacht, als ich ein super einfaches Generator geschrieben habe. Stellen Sie sich vor, wie viele Fehler ich gemacht hätte, wenn ich versucht hätte, etwas Komplexeres zu modellieren!

6. Das Problem mit typisierten Fehlern

Wir Haskell-Entwickler mögen keine Laufzeitfehler. Sie sind schrecklich und widerlich! Man muss sie debuggen, und sie sind nicht in den Typen repräsentiert. Stattdessen verwenden wir gerne `Either` (oder etwas Isomorphes), um Dinge darzustellen, die fehlschlagen könnten:

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

`Either` hat eine `Monad`-Instanz, daher können Sie eine `Either l r`-Berechnung mit einem `l`-Wert kurzschließen oder sie an eine Funktion auf dem `r`-Wert binden. Die Namen des Typs und der Konstruktoren sind nicht willkürlich. Wir haben zwei Typvariablen: `Either left right`. Die `left`-Typvariable befindet sich im `Left`-Konstruktor, und die `right`-Typvariable befindet sich im `Right`-Konstruktor.

Also nehmen wir unsere unsicheren, zur Laufzeit fehlerhaften Funktionen:

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

und wir verwenden informative Fehlertypen, um ihre möglichen Fehler darzustellen:

```
1  data HeadError = ListWasEmpty
2
3  head :: [a] -> Either HeadError a
4
5  data LookupError = KeyWasNotPresent
6
7  lookup :: k -> Map k v -> Either LookupError v
8
9  data ParseError
```

```
10      = UnexpectedChar Char String
11      | RanOutOfInput
12
13  parse :: String -> Either ParseError Integer
```

Außer, dass wir Typen wie `HeadError` oder `LookupError` eigentlich nicht verwenden. Es gibt nur eine Möglichkeit, wie `head` oder `lookup` fehlschlagen könnten. Deshalb verwenden wir stattdessen einfach `Maybe`. `Maybe a` ist genau wie `Either () a` zu verwenden - es gibt nur einen möglichen `Left ()`-Wert, und es gibt nur einen möglichen `Nothing`-Wert. (Wenn Sie nicht überzeugt sind, schreiben Sie `newtype Maybe a = Maybe (Either () a)`, leiten Sie alle relevanten Instanzen ab, und versuchen Sie, einen Unterschied zwischen diesem `Maybe` und dem Standard-`Maybe` zu erkennen).

Aber, `Maybe` ist nicht ideal - wir haben Informationen verloren! Angenommen, wir haben eine Berechnung:

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

Jetzt probieren wir es mit einem Eingabewert aus, und es gibt uns `Nothing` zurück. Welche Schritt ist fehlgeschlagen? Tatsächlich können wir das nicht wissen! Alles, was wir wissen können, ist, dass *etwas* fehlgeschlagen ist.

Also, versuchen wir `Either` zu verwenden, um mehr Informationen darüber zu bekommen, was fehlgeschlagen ist. Können wir das einfach so schreiben?

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

Leider ergibt dies einen Typfehler. Wir können sehen, warum, indem wir uns den Typ von `>>=` ansehen:

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

Die Typvariable `m` muss eine Instanz von `Monad` sein, und der Typ `m` muss *genau derselbe* sein sowohl für den Wert auf der linken Seite als auch für die Funktion auf der rechten Seite. `Either LookupError` und `Either ParseError` sind nicht derselbe Typ, und daher schlägt dieser Typencheck fehl.

Stattdessen benötigen wir eine Möglichkeit, diese möglichen Fehler zu akkumulieren. Wir werden eine Hilfsfunktion `mapLeft` einführen, die uns dabei hilft:

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

Nun können wir diese Fehlertypen kombinieren:

```
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))
```

Da! Jetzt können wir genau wissen, wie und warum die Berechnung fehlgeschlagen ist. Leider ist dieser Typ ein bisschen ein Monster. Er ist weitschweifig und der ganze `mapLeft`-Standardmäßiger Code ist nervig.

An diesem Punkt werden die meisten Anwendungsentwickler einen “Anwendungsfehler”-Typ erstellen und einfach alles hineinschieben, was schiefgehen kann.

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

Nun, dies räumt den Code etwas auf:

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

Allerdings gibt es ein ziemlich großes Problem mit diesem Code. `foo` behauptet, dass es alle möglichen Fehler "werfen" kann - es ist ehrlich in Bezug auf Parserfehler, Nachschlagefehler und Head-Fehler, aber es behauptet auch, dass es werfen wird, wenn Dateien nicht gefunden werden, "was auch immer" passiert, und `usw`. Es gibt keine Möglichkeit, dass ein Aufruf von `foo` zu `Datei nicht gefunden` führt, weil `foo` nicht einmal `E/A` machen kann! Das ist absurd. Der Typ ist zu groß! Das vorherige Kapitel diskutiert, wie wichtig es ist, die Typen klein zu halten und wie wunderbar es sein kann, um Fehler zu beseitigen.

Angenommen, wir wollen `foo`'s Fehler behandeln. Wir rufen die Funktion auf und schreiben dann einen Fallausdruck wie gute Haskeller:

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

```
11                  handleHeadError
12              _ ->
13                  error "impossible?!?!?!"
```

Leider ist dieser Code anfällig für Refaktorisierung! Wir behaupten, alle Fehler zu behandeln, aber in Wirklichkeit behandeln wir viele davon nicht. Wir "wissen" momentan, dass dies die einzigen Fehler sind, die auftreten können, aber es gibt keine Garantie durch den Compiler, dass dies der Fall ist. Jemand könnte später `foo` ändern, um einen anderen Fehler auszulösen, und dieser Fallausdruck würde fehlschlagen. Jeder Fallausdruck, der ein Ergebnis von `foo` auswertet, muss aktualisiert werden.

Der Fehlertyp ist zu groß, und so besteht die Möglichkeit, ihn falsch zu behandeln. Es gibt ein weiteres Problem. Angenommen, wir wissen, wie man einen oder zwei Fälle des Fehlers behandelt, aber wir müssen den Rest der Fehlerfälle nach oben weiterleiten:

```
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
```

Wir *wissen*, dass `AllParseError` von `bar` gehandhabt wurde, weil - schauen Sie es sich einfach an! Der Compiler hat jedoch keine Ahnung. Wann immer wir den Fehlerinhalt von `bar` überprüfen, müssen wir entweder a) einen Fehlerfall „behandeln", der vielleicht zweifelhaft bereits behandelt wurde, oder b) den Fehler ignorieren und verzweifelt hoffen, dass kein zugrundeliegender Code jemals den Fehler auslöst.

Sind wir mit den Problemen dieses Ansatzes fertig? Nein! Es gibt keine Garantie, dass ich den *richtigen* Fehler werfe!

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

Dieser Code wird typgeprüft, aber er ist *falsch*, weil `LookupError` nur von `lookup` ausgelöst werden soll! In diesem Fall ist es offensichtlich, aber in größeren Funktionen und Codebasen wird es nicht so klar sein.

6.1 Monolithische Fehlertypen sind schlecht

Ein monolithischer Fehlertyp hat also viele Probleme. Ich werde hier eine Behauptung aufstellen:

> Alle Fehlertypen sollten einen einzigen Konstruktor haben

Das heißt, Fehler sollten keine Summentypen sein. Der Name des Typs und der Name des Konstruktors sollten gleich sein. Die Ausnahme sollte *tatsächliche Werte* enthalten, die nützlich wären, um einen Komponententest zu schreiben oder das Problem zu debuggen. Ein `String`-Nachricht mitzuschleppen ist ein No-Go.

Fast alle Programme können auf mehrere potenzielle Arten fehlschlagen. Wie können wir dies darstellen, wenn wir nur einen einzigen Konstruktor pro Typ verwenden?

Vielleicht können wir sehen, ob wir `Either` angenehmer verwenden können. Wir definieren ein paar Helfer, die den nötigen Schreibaufwand reduzieren:

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

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

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

Nun, lassen Sie uns diesen unansehnlicheren `Either`-Code mit diesen neuen Hilfsmitteln refaktorisieren:

```
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))
```

Nun, die Syntax ist schöner. Wir können über das verschachtelte `Either` im Fehlerzweig mit `case` gehen, um einzelne Fehlerfälle zu eliminieren. Es ist einfacher sicherzustellen, dass wir nicht behaupten, Fehler auszulösen, die wir nicht auslösen - schließlich wird GHC den Typ von `foo` korrekt ableiten, und wenn GHC eine Typvariable für irgendein `+` ableitet, können wir annehmen, dass wir diesen Fehlerplatz nicht verwenden und ihn löschen können.

Leider gibt es immer noch den `mapLeft`-Standardcode. Und Ausdrücke, von denen man *wirklich* möchte, dass sie gleich sind, sind es nicht –

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

Die Werte `x` und `y` sind *isomorph*, aber wir können sie in einem `do` Block nicht verwenden, da sie nicht exakt gleich sind. Wenn wir Fehler hinzufügen, müssen wir *alle* `mapLeft`-Codes sowie alle `case`-Ausdrücke, die die Fehler überprüfen, überarbeiten. Glücklicherweise sind dies vollständig vom Compiler geführte Refaktorisierungen, sodass die Wahrscheinlichkeit, dabei Fehler zu machen, gering ist. Sie tragen jedoch erheblich zu Boilerplate, Lärm und Fleißarbeit in unserem Programm bei.

6.2 Boilerplate ade!

Nun, es stellt sich heraus, dass wir die Abhängigkeitsreihenfolge und Boilerplate mit Typklassen loswerden *können*! Der erste Ansatz besteht darin, “classy prisms” aus dem `lens`-Paket zu verwenden. Lassen Sie uns unsere Typen von konkreten Werten in prismatische umwandeln:

```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
```

Nun, Typklasseneinschränkungen kümmern sich nicht um die Reihenfolge - `(Foo a, Bar a) => a` und `(Bar a, Foo a) => a` sind genau dasselbe, soweit es GHC betrifft. Die `AsXXX` Typklassen werden *automatisch* die `mapLeft` Sachen für uns bereitstellen, so dass unsere `foo` Funktion jetzt viel übersichtlicher aussieht:

```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)
```

Dies scheint eine deutliche Verbesserung gegenüber dem zu sein, was wir vorher hatten! Und der meiste Standardcode mit den `AsXXX`-Klassen wird über Template Haskell erledigt:

```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
```

Allerdings müssen wir unseren eigenen Boilerplate schreiben, wenn wir schließlich diese Typen konkret behandeln wollen. Wir könnten am Ende einen riesigen `AppError` schreiben, in den all diese Fehler eingespeist werden.

Es gibt einen großen, fatalen Fehler bei diesem Ansatz. Obwohl es sich gut zusammensetzt, zerfällt es überhaupt nicht! Es gibt keine Möglichkeit, einen einzelnen Fall zu erfassen und sicherzustellen, dass er behandelt wird. Die Mechanik, die uns Prismen bieten, erlaubt es nicht, eine einzelne Einschränkung herauszulösen, sodass wir keinen Pattern Match auf einen einzelnen Fehler durchführen können.

Unsere Typen werden erneut immer größer, mit all den damit verbundenen Problemen.

6.3 Typklassen zur Rettung!

Was wir *wirklich* wollen, ist:

- Unabhängigkeit von der Reihenfolge
- Kein Boilerplate
- Einfache Zusammensetzung
- Einfache Zerlegung

In PureScript oder OCaml können Sie offene Variantentypen verwenden, um dies fehlerfrei zu tun. Haskell hat keine offenen Varianten, und die Versuche, sie nachzubilden, sind in der Praxis recht umständlich zu verwenden.

Glücklicherweise können wir Typklassen und Einschränkungen verwenden, um etwas Ähnliches zu erreichen. Oben hatten wir eine Menge Probleme mit dem "verschachtelten `Either`"-Muster - `Either (Either (Either A B) C) D`. Dies ermöglicht es uns, den Ausnahme-Typ zu vergrößern und zu verkleinern, was es uns erlaubt,

Fälle zu behandeln und neue einzuführen. Aber die Benutzerfreundlichkeit ist ziemlich schlecht.

Der Grund ist, dass `Either _ _` einen *binären Baum* von Typen darstellt. Wir wollen keinen binären Baum - wir wollen eine `Menge`. Aber eine `Menge` von Typen wird am besten als Typklasse-Einschränkung modelliert. Wir brauchen also eine Möglichkeit zu sagen, dass `A`, `B`, `C` und `D` alle im Typ 'enthalten' sind.

Obwohl ich dieses Thema gerne vollständig im Buch aufnehmen würde, wäre es unehrlich. Ich habe die Technik nicht in der Produktion verwendet und kann sie daher nicht voll empfehlen. Wenn Sie daran interessiert sind, mehr über experimentelles Material zu lesen, dann empfehle ich den Blogbeitrag "Plucking Constraints"[1], sowie die `plucky`[2] Proof-of-Concept Bibliothek und das super experimentelle `prio`[3] Repository, das die `plucky` Technik mit `IO`-basierten Ausnahmen verwendet.

6.4 Die Tugend von ungetypten Fehlern

Wir haben gesehen, dass getypte Fehler eine Reihe von Problemen haben. Es ist schwierig, Fehlerfälle zu entfernen. Der Boilerplate ist intensiv. Die Buchhaltung ist selten ergonomisch oder benutzerfreundlich.

Getypte Fehler haben *viele* Probleme und erfordern *viel* Arbeit. Währenddessen haben ungetypte Fehler *viele* Probleme, erfordern aber *wenig* Arbeit. Deshalb denke ich, dass es am besten ist, bei ungetypten Ausnahmen zu bleiben, bis etwas Robusteres kommt.

```
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

7. Template Haskell ist nicht beängstigend

7.1 Ein Anfängertutorial

Dieses Tutorial richtet sich an Personen, die Anfänger bis fortgeschrittene Haskeller sind und die Grundlagen von Template Haskell erlernen möchten.

Ich habe über die Macht und den Nutzen der Metaprogrammierung in Ruby gelernt. Ruby-Metaprogrammierung wird durchgeführt, indem Quellcode durch Zeichenkettenverkettung konstruiert wird und der Interpreter ihn ausführt. Es gibt auch einige Methoden, die verwendet werden können, um Methoden, Konstanten, Variablen usw. zu definieren.

In meiner Squirrell[1] Ruby-Bibliothek, die entwickelt wurde, um das Kapseln von SQL-Abfragen etwas einfacher zu machen, habe ich ein paar Elemente der Metaprogrammierung, um einige Annehmlichkeiten bei der Definition von Klassen zu ermöglichen. Die Idee ist, dass Sie eine Abfrageklasse so definieren können:

```
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
```

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

```
14    end
15
16    def has_post?
17      post_id ? "AND posts.id = #{post_id}" : ""
18    end
19  end
```

und indem Sie `requires` mit den Symbolen angeben, die Sie benötigen möchten, wird es für Sie eine Instanzvariable und einen Attributleser definieren und Fehler auslösen, wenn Sie den erforderlichen Parameter nicht übergeben. Das zu erreichen war ziemlich einfach. Der Aufruf von `requires` erledigt etwas Buchhaltung mit den erforderlichen Parametern und ruft dann diese Methode mit den übergebenen Argumenten auf:

```
1  def define_readers(args)
2    args.each do |arg|
3      attr_reader arg
4    end
5  end
```

Man kann es ein wenig wie ein Makro lesen: Nimm die Argumente und rufe `attr_reader` mit jedem auf. Die Magie passiert später, wo ich die `initialize`-Methode überschrieben habe:

```
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
```

Wir iterieren über die an `new` übergebenen Argumente, und wenn erforderliche fehlen, tritt ein Fehler auf. Andernfalls setzen wir die mit dem Argument verbundene Instanzvariable und entfernen es aus dem Hash.

Ein anderer Ansatz besteht darin, einen String zu nehmen und ihn im Kontext der Klasse, in der Sie sich befinden, auszuwerten:

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

Diese Codezeile definiert eine Methode mit einem Namen Ihrer Wahl und einem String, der im Kontext der ausführenden Klasse gedruckt wird.

7.2 Moment mal, das ist nicht Haskell, was mache ich hier

Metaprogrammierung in Ruby basiert hauptsächlich auf einem textuellen Ansatz von Code. Man verwendet Ruby, um einen String von Ruby-Code zu generieren, und lässt Ruby dann den Code auswerten.

Wenn Sie aus einem solchen Hintergrund kommen (wie ich), wird Ihnen Template Haskell anders und seltsam vorkommen. Sie denken vielleicht: “Oh, ich weiß, ich benutze einfach `QuasiQuoters`, und alles wird gut funktionieren.” Nein. Sie müssen anders über Metaprogrammierung in Template Haskell nachdenken. Sie werden keine Strings zusammensetzen, die zufällig gültigen Code ergeben. Das ist Haskell, wir werden eine Kompilierzeitprüfung durchführen!

7.3 Konstruktion eines AST

In Ruby haben wir einen String erstellt, den der Ruby-Interpreter dann geparst, in einen abstrakten Syntaxbaum umgewandelt und interpretiert hat. In Haskell überspringen wir den String-Schritt. Wir bauen den Abstrakten Syntaxbaum (AST) direkt mit Standarddatenkonstruktoren. GHC wird überprüfen, ob beim Aufbau des Syntaxbaums alles in Ordnung ist, und den Syntaxbaum dann in unseren Quellcode einfügen, bevor das Ganze kompiliert wird. So erhalten wir zwei Ebenen der Kompilierzeitprüfung - dass wir eine korrekte Vorlage erstellt haben und dass wir die Vorlage korrekt verwendet haben.

Eines der unangenehmsten Dinge an textueller Metaprogrammierung ist das fehlende Garantie, dass Ihre Syntax korrekt ist. Das Debuggen von Syntaxfehlern im generierten Code kann schwierig sein. Die Überprüfung der Korrektheit unseres Codes ist einfacher, wenn wir direkt in einen AST programmieren. Die QuasiQuoter sind eine bequeme Ergänzung zur AST-Programmierung, aber ich bin der Meinung, dass man zuerst das AST-Zeug lernen sollte und dann in die Quoter eintauchen sollte, wenn man eine gute Vorstellung davon hat, wie sie funktionieren.

Also, lassen Sie uns mit unserem ersten Beispiel beginnen. Wir haben eine Funktion `bigBadMathProblem :: Int -> Double` geschrieben, die zur Laufzeit viel Zeit in Anspruch nimmt, und wir möchten eine Nachschlagetabelle für die häufigsten Werte erstellen. Da wir sicherstellen wollen, dass die Laufzeitgeschwindigkeit super schnell ist und wir keine Probleme haben, auf den Compiler zu warten, werden wir dies mit Template Haskell tun. Wir geben eine Liste von häufigen Zahlen ein, führen die Funktion für jede aus, um sie vorab zu berechnen, und übergeben dann schließlich die Funktion, wenn wir die Zahl nicht zwischengespeichert haben.

Da wir etwas Ähnliches wie die `makeLenses`-Funktion tun möchten, um eine Menge Deklarationen für uns zu generieren, schauen wir uns zuerst den Typ dieser Funktion in der `lens`-Bibliothek an. Wenn wir zu den Lens-Dokumenten[2] springen, können wir sehen, dass der Typ von `makeLenses Name -> DecsQ` ist. Wenn wir zu den Template Haskell-Dokumenten[3] springen, ist `DecsQ` ein Typsynonym für `Q [Dec]`. `Q` scheint ein Monad für Template Haskell zu sein, und ein `Dec`[4] ist der Datentyp für eine Deklaration. Der Konstruktor zur Erstellung einer Funktionsdeklaration ist `FunD`. Damit können wir loslegen!

Wir beginnen mit der Definition unserer Funktion. Sie nimmt eine Liste von häufig verwendeten Werten, wendet die Funktion auf jeden an und speichert das Ergebnis. Schließlich benötigen wir eine Klausel, die den Wert an die Mathematikfunktion weitergibt, falls wir ihn nicht zwischengespeichert haben.

```
1 precompute :: [Int] -> DecsQ
2 precompute xs = do
3     -- .......
4     return [FunD name clauses]
```

Da `Q` eine Monad ist und `DecsQ` ein Typsynonym dafür ist, wissen wir, dass wir mit `do` beginnen können. Und wir wissen, dass wir eine Funktionsdefinition zurückgeben

[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

werden, die gemäß der `Dec`-Dokumentation ein Feld für den Namen der Funktion und die Liste der Klauseln hat. Jetzt liegt es an uns, den Namen und die Klauseln zu generieren. Namen sind einfach, also machen wir das zuerst.

Wir können einen Namen aus einem String mithilfe von `mkName` erhalten. Dies konvertiert einen String in einen unqualifizierten Namen. Wir werden `lookupTable` als den Namen unserer Nachschlagetabelle wählen, sodass wir diesen direkt verwenden können.

```
1 precompute xs = do
2     let name = mkName "lookupTable"
3     -- ...
```

Nun müssen wir jede Variable in `xs` auf die Funktion namens `bigBadMathProblem` anwenden. Dies wird im `[Clause]`-Feld stehen, also schauen wir uns an, was eine `Clause` ausmacht. Laut der Dokumentation[5] ist eine Klausel ein Datenkonstruktor mit drei Feldern: einer Liste von `Pat`-Mustern, einem `Body` und einer Liste von `Dec`-Deklarationen. Der Body entspricht der tatsächlichen Funktionsdefinition, die `Pat`-Muster entsprechen den Mustern, auf die wir Eingabeargumente abgleichen, und die `Dec`-Deklarationen sind das, was wir möglicherweise in einer `where`-Klausel finden.

Lassen Sie uns zuerst unsere Muster identifizieren. Wir versuchen, direkt auf die `Int`s abzugleichen. Unser gewünschtes Ergebnis wird ungefähr so aussehen:

```
1 lookupTable 0 = 123.546
2 lookupTable 12 = 151626.4234
3 lookupTable 42 = 0.0
4 -- ...
5 lookupTable x = bigBadMathProblem x
```

Also brauchen wir eine Möglichkeit, die `Int`s in unserer `xs`-Variablen in ein `Pat`-Muster zu bringen. Wir brauchen eine Funktion `Int -> Pat`... Lassen Sie uns die Dokumentation[6] für `Pat` ansehen und sehen, wie sie funktioniert. Das erste Muster ist `LitP`, das ein Argument vom Typ `Lit` annimmt. Ein `Lit` ist ein Summen-Typ, der einen Konstruktor für die primitiven Haskell-Typen hat. Es gibt einen für `IntegerL`, den wir verwenden können.

So können wir von `Int -> Pat` mit der folgenden Funktion kommen:

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

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

Was wir auf die Anfangsliste abbilden können, um unser `[Pat]` zu erhalten!

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

Unsere `lookupTable`-Funktion wird nur ein einziges Argument entgegennehmen, daher möchten wir unsere `Pat`-Ganzzahlen in `Clause` umwandeln, indem wir von `[Pat] -> [Clause]` gehen. Das wird die `clauses`-Variable nutzen, die wir benötigen. Von oben ist eine Klausel wie folgt definiert:

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

Also, unser `[Pat]` ist einfach - wir haben nur einen literalen Wert, auf den wir passen. `Body` ist definiert als entweder ein `GuardedB`, das Musterwächter verwendet, oder ein `NormalB`, das dies nicht tut. Wir könnten unsere Funktion in Bezug auf eine einzelne Klausel mit einem `GuardedB` Körper definieren, aber das klingt nach mehr Arbeit, also verwenden wir einen `NormalB` Körper. Der `NormalB` Konstruktor nimmt ein Argument vom Typ `Exp`. Also lassen Sie uns in die Dokumentation von `Exp`![7] eintauchen.

Hier gibt es viel. Wie oben erwähnt, möchten wir wirklich eine einzige Sache - einen Literal! Den vorkomputierten Wert. Es gibt einen `LitE` Konstruktor, der einen `Lit` Typ nimmt. Der `Lit` Typ hat einen Konstruktor für `DoublePrimL`, der einen `Rational` nimmt, also müssen wir ein wenig Umwandlung durchführen.

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

Wir können die `Body`s für die `Clause`s erhalten, indem wir diese Funktion über die Liste der Argumente abbilden. Die Deklarationen werden leer sein, also sind wir bereit, unsere `clauses` zu erstellen!

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

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

Es gibt noch eine Sache, die hier zu tun ist. Wir müssen eine weitere Klausel mit einer Variablen `x` erstellen, die wir an die Funktion delegieren. Da `mkName` es erlaubt, dass die Variable überschattet wird und dies eine Warnung im generierten Code erzeugen könnte, möchten wir `newName` verwenden, um einen hygienischen Namen für die Variable zu erstellen. Wir müssen ein wenig komplizierter mit unserem `Body`-Ausdruck werden, da eine Anwendung auf eine Funktion stattfindet.

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

Zurückkehrend zum `Exp` Typ, suchen wir nun nach etwas, das die Idee der Anwendung erfasst. Der `Exp` Typ hat einen Datenkonstruktor `AppE`, der zwei Ausdrücke nimmt und den zweiten auf den ersten anwendet. Genau das brauchen wir! Er hat auch

einen Datenkonstruktor `VarE`, der ein `Name` Argument nimmt. Das ist alles, was wir brauchen. Los geht's.

```
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]
```

Um den Namen für `bigBadMathProblem` zu erhalten, haben wir ein Template Haskell Zitat verwendet. Das Zeichen `'` erstellt einen `Name` aus einem Wert, während zwei Apostrophe einen `Name` aus einem Typ erstellen. Dies sieht man oft beim Ableiten: `deriveJSON ''MyType`.

Wir haben es geschafft! Wir haben etwas Template Haskell zusammengestellt und uns eine Nachschlagetabelle geschrieben. Jetzt möchten wir sie mit der Spleiß-Syntax `$()` in die oberste Ebene unseres Programms einfügen:

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

Wie es der Zufall will, ist GHC intelligent genug, um zu erkennen, dass ein Ausdruck auf oberster Ebene mit dem Typ `Q [Dec]` ohne die explizite Splicing-Syntax eingefügt werden kann. Wir hätten also auch schreiben können:

```haskell
module X where

import Precompute (precompute)

precompute [1..1000]
```

Haskell-Ausdrücke mit den Datenkonstruktoren zu erstellen, ist wirklich einfach, wenn auch ein wenig umständlich. Schauen wir uns ein etwas komplizierteres Beispiel an.

7.4 Boilerplate Ade!

Wir freuen uns, die ausgezeichnete `users`-Bibliothek mit dem `persistent`-Backend für die Webanwendung, an der wir gerade arbeiten, zu verwenden (Quellcode hier, falls Sie neugierig sind[8]). Es erledigt alle möglichen Dinge für uns und kümmert sich um eine Menge Standardcode und benutzerbezogenen Code. Es erwartet als erstes Argument einen Wert, der entpackt und verwendet werden kann, um eine Persistent-Abfrage auszuführen. Es operiert außerdem in der `IO`-Monade. Derzeit ist unsere Anwendung so eingerichtet, dass sie eine benutzerdefinierte Monade `AppM` verwendet, die folgendermaßen definiert ist:

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

Um die Funktionen in der `users`-Bibliothek tatsächlich zu nutzen, müssen wir diese lustige Angelegenheit erledigen:

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

Das wird schnell lästig, also beginnen wir, spezielle Funktionen für unsere Monad zu schreiben, die wir aufrufen können, anstatt das ganze Lifting und Wrapping selbst zu machen.

[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, total mechanischer Code. Einfach nur Standardcode. Das ist genau die Art von Sache, die ich in Ruby metaprogrammiert hätte. Also lass uns das in Haskell metaprogrammieren!

Zuerst wollen wir den Ausdruck vereinfachen. Lass uns `listUsers` als Beispiel nehmen. Wir machen es so einfach wie möglich - keine Infix-Operatoren, keine `do`-Notation, etc.

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

Schön. Um das Betrachten des AST ein wenig einfacher zu machen, können wir noch einen Schritt weiter gehen. Zeigen wir alle Funktionsanwendungen explizit, indem wir Klammern hinzufügen, um alles so deutlich wie möglich zu machen.

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

Die allgemeine Formel, die wir anstreben, ist:

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

Wir beginnen damit, unsere `deriveReader`-Funktion zu erstellen, die als erstes Argument den `backend`-Funktionsnamen übernimmt.

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

Dies ist unser erster Teil der speziellen Syntax. Das einfache Anführungszeichen in `'destroyUserBackend` gibt den `Name` für `destroyUserBackend` zurück. Im Gegensatz zu `mkName "destroyUserBackend"` ist dies jedoch ein *global qualifizierter Name*. Dieser `Name` funktioniert auch dann, wenn das Modul, das den Code einfügt, den Code, aus dem er stammt, nicht importiert. Wenn Sie sich auf Namen beziehen, die außerhalb des von Ihnen generierten Codes existieren, müssen Sie diese Form verwenden. Andernfalls müssen Ihre Benutzer eine Menge Module importieren, um die Anforderungen Ihres Makros zu erfüllen.

Nun, was wir brauchen, ist eine Funktion `decForFunc`, die die Signatur `Name -> Name -> Q Dec` hat.

Um dies zu tun, müssen wir einige Informationen über die Funktion erhalten, die wir ableiten möchten. Insbesondere müssen wir wissen, wie viele Argumente die Quellfunktion annimmt. Es gibt einen ganzen Abschnitt in der Template Haskell Dokumentation über 'Abfragen des Compilers'[9], den wir gut nutzen können.

Die Funktion `reify` gibt einen Wert vom Typ `Info` zurück. Für Typklassenoperationen hat sie den Datenkonstruktor `ClassOpI` mit den Argumenten `Name`, `Type`, `ParentName` und `Fixity`. Keiner dieser hat direkt die Stelligkeit der Funktion...

Ich denke, es ist an der Zeit, ein wenig erkundendes Programmieren im REPL zu machen. Wir können `GHCi` starten und mit den folgenden Befehlen einige Template Haskell-Sachen machen:

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

Wir können auch den folgenden Befehl ausführen, und es wird den gesamten generierten Code ausgeben, den es erstellt:

```
λ: :set -ddump-splices
```

Lassen Sie uns nun `reify` auf etwas Einfaches ausführen und das Ergebnis sehen!

```
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
```

Hm.. Keine Show-Instanz. Glücklicherweise gibt es eine Umgehungslösung, die Dinge im Q-Monaden ausdrucken kann:

[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)"
```

Ich habe es ein wenig formatiert, um es etwas leserlicher zu machen. Wir haben den `Name`, den `Type`, einen `Nothing`-Wert, der immer `Nothing` ist, und die Fixität der Funktion. Der `Type` scheint ziemlich nützlich zu sein... Schauen wir uns die `reify`-Ausgabe für eine der Klassenmethoden an, mit denen wir arbeiten wollen:

```
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, das ist eine Menge Text! Glaub es oder nicht, ich habe es formatiert, um es ein wenig leserlicher zu machen. Wir interessieren uns hauptsächlich für die `Type`-Deklaration, und wir können viele Informationen darüber erhalten, welche Datenkonstruktoren verwendet werden, aus der Dokumentation[10]. Genau wie `AppE` ist, wie wir einen Ausdruck auf einen Ausdruck anwenden, ist `AppT` die Art und Weise, wie wir einen Typ auf einen Typ anwenden. `ArrowT` ist der Funktionspfeil in der Typsignatur.

Nur als Übung gehen wir die folgende Typsignatur durch und verwandeln sie in etwas Ähnliches wie das oben:

[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))
```

Okay, jetzt sind alle unsere `(->)`s in Präfixform geschrieben. Wir werden die Pfeile durch `ArrowT` ersetzen, explizite Klammern hinzufügen und die `ApplyT`-Konstruktoren von den innersten Ausdrücken nach außen einfügen.

```
~ (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))
```

Das ist ziemlich aus dem Ruder gelaufen und sieht unordentlich aus. Aber wir haben jetzt eine gute Vorstellung davon, wie wir von einer Darstellung zur anderen gelangen können.

Ausgehend von unserer Typsignatur sieht es so aus, als könnten wir herausfinden, wie wir die benötigten Argumente aus dem Typ erhalten können! Wir werden einen Musterabgleich an der Typsignatur durchführen, und wenn wir etwas sehen, das wie die Fortsetzung einer Typsignatur aussieht, werden wir eins zu einer Zählung hinzufügen und tiefer gehen. Andernfalls werden wir aussteigen.

Die Funktionsdefinition sieht so aus:

```
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
```

Toll! Wir können diese genauso wie gewöhnliche Haskell-Werte pattern matchen. Nun, sie *sind* gewöhnliche Haskell-Werte, also ergibt das vollkommen Sinn.

Zuletzt brauchen wir eine Funktion, die den Typ aus einem `Info` erhält. Nicht alle `Info` haben Typen, daher kodieren wir das mit `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
```

Gut, wir sind bereit, mit der `decForFunc`-Funktion zu beginnen! Füllen wir aus, was wir wissen, dass wir tun müssen:

```
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) []])
```

Stelligkeit erfasst. Jetzt möchten wir eine Liste neuer Variablennamen erstellen, die den Funktionsargumenten entsprechen. Wenn wir hygienisch mit unseren Variablennamen

sein wollen, verwenden wir die Funktion `newName`, die einen völlig einzigartigen Variablennamen mit dem davor angefügten String erstellt. Wir möchten `(Stelligkeit - 1)` neue Namen, da wir den gebundenen Wert aus der Leserfunktion für den anderen verwenden werden. Wir benötigen auch einen Namen für den Wert, den wir aus dem Lambda binden werden.

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

Als Nächstes steht der neue Funktionsname an. Um eine konsistente API beizubehalten, verwenden wir denselben Namen wie im eigentlichen Paket. Dies erfordert, dass wir das andere Paket qualifiziert importieren, um einen Namenskonflikt zu vermeiden.

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

`nameBase` hat den Typ `Name -> String` und erhält den nicht qualifizierten Namensstring für einen gegebenen `Name`-Wert. Dann verwenden wir `mkName` mit dem String, was uns einen neuen, nicht qualifizierten Namen mit demselben Wert wie die ursprüngliche Funktion gibt. Könnte das eine schlechte Idee sein? Wahrscheinlich möchten Sie eine eindeutige Kennung bereitstellen. Allerdings kann es hilfreich sein, die Namen konsistent zu halten, um die Entdeckung zu erleichtern.

Als nächstes möchten wir die `(>>=)` Funktion auf den `reader` anwenden. Dann möchten wir eine Funktion erstellen, die den `bound` Ausdruck auf eine Lambda anwendet. Lambdas haben einen LamE[11] Konstruktor im `Exp` Typ. Sie nehmen ein `[Pat]` zum Abgleichen und ein `Exp`, das den Lambda-Körper darstellt.

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

Also ist `AppE bound . LamE [VarP b]` genau dasselbe wie `(>>=) reader (\b -> ...)`! Cool.

Als nächstes müssen wir `VarE`-Werte für alle Variablen erstellen. Dann müssen wir alle Werte auf den Ausdruck `VarE fn` anwenden. Funktionsanwendung bindet nach links, also haben wir:

[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)
```

Das sieht aus wie ein linker Fold! Sobald wir das haben, werden wir den vollständig angewandten `fn`-Ausdruck auf `VarE 'liftIO` anwenden und schließlich an die Lambda binden.

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

Dies erzeugt unseren Ausdruck `(>>=) reader (\b -> fn b arg1 arg2 ... argn)`.

Das letzte, was wir tun müssen, ist, unsere Muster zu erhalten. Dies ist die Liste von Variablen, die wir zuvor erstellt haben.

```
varPat = map VarP varNames
```

Und nun das Ganze:

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

```
14          , 'deleteUser
15          ]
16
17  decForFunc :: Name -> Name -> Q Dec
18  decForFunc reader fn = do
19      info <- reify fn
20      arity <-
21          case getType info of
22              Nothing -> do
23                  reportError "Unable to get arity of name"
24                  return 0
25              Just typ ->
26                  pure $ functionLevels typ
27      varNames <- replicateM (arity - 1) (newName "arg")
28      b <- newName "b"
29      let fnName     = mkName . nameBase $ fn
30          bound      = AppE (VarE '(>>=)) (VarE reader)
31          binder     = AppE bound . LamE [VarP b]
32          varExprs   = map VarE (b : varNames)
33          fullExpr   = foldl AppE (VarE fn) varExprs
34          liftedExpr = AppE (VarE 'liftIO) fullExpr
35          final      = binder liftedExpr
36          varPat     = map VarP varNames
37      return $ FunD fnName [Clause varPat (NormalB final) []]
```

Und wir haben nun eine Menge Boilerplate metaprogrammiert!

Wir haben die Dokumentation für Template Haskell durchgesehen, herausgefunden, wie man Werte in Haskells AST konstruiert, und erarbeitet, wie man einige Arbeiten zur Kompilierzeit erledigt sowie etwas Boilerplate automatisiert. Ich bin gespannt darauf, mehr über die Magie der Definition von Quasiquoters und fortgeschritteneren Template-Haskell-Konstrukten zu lernen, aber selbst ein super grundlegender Ansatz, der “Ausdrücke und Deklarationen mit Datenkonstruktoren baut”, ist nützlich.