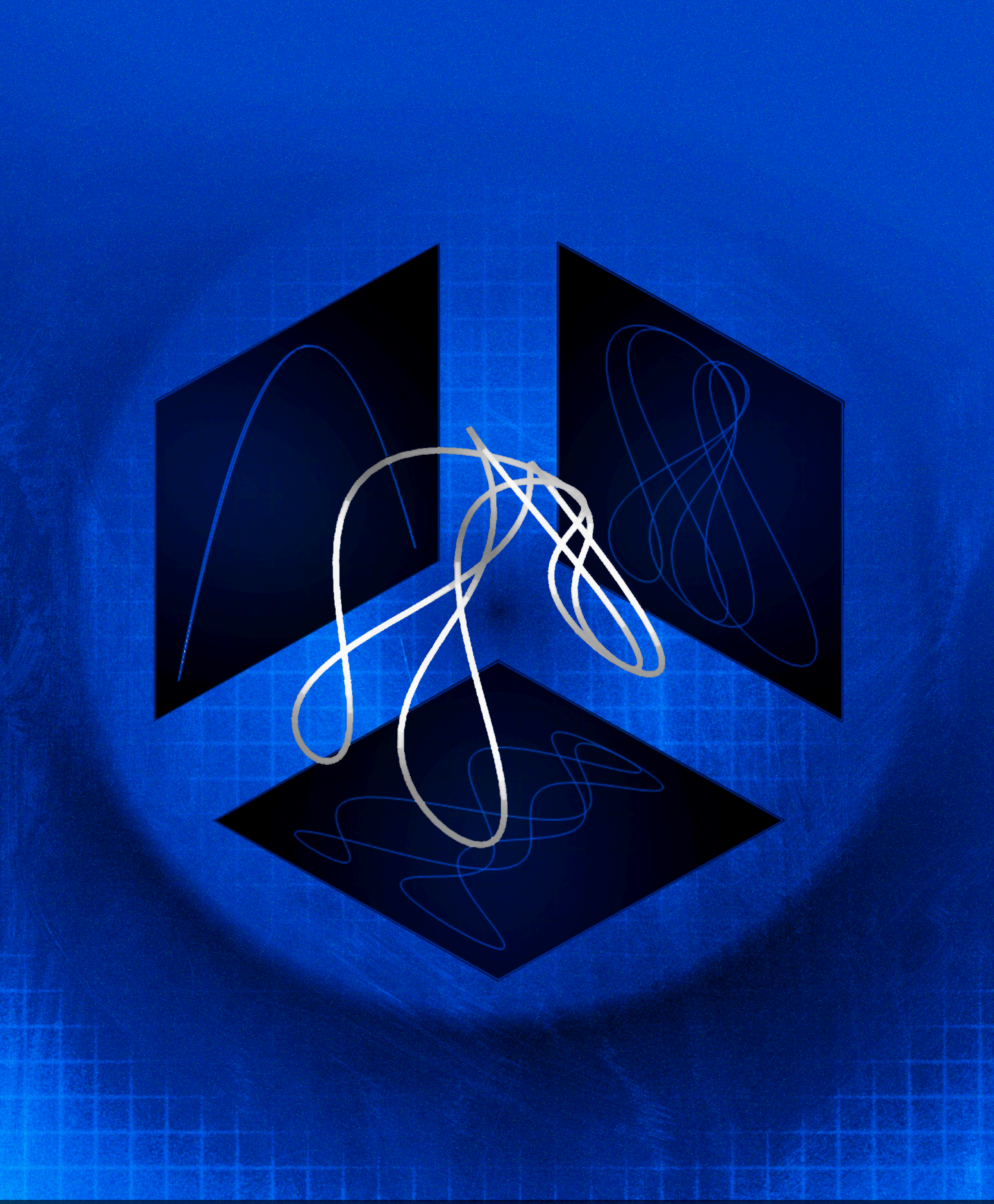

Algebra-Driven Design

Elegant Software from Simple Building Blocks

Sandy Maguire

Algebra-Driven Design

Elegant Software from Simple Building Blocks

Sandy Maguire

Cofree Press

First published 2020

Version 1.2 / 2023-11-09

Are you quite sure
that all those bells and whistles,
all those wonderful facilities of your so-called
“powerful” programming languages,
belong to the solution set
rather than to the problem set?

EDSGER W. DIJKSTRA

Contents

Foreword

The Restoring of Broken Parts

Algebra. We all studied it at school. We learned "algebraic laws" such as commutativity

$$x + y = y + x$$

and associativity

$$(x + y) + z = x + (y + z)$$

so well that applying them became second nature, and we could use them in long reasoning about equalities, without any need for more complicated proof techniques such as proof-by-cases, proof-by-induction, or proof-by-contradiction.

We may think of algebra in connection with proofs, but it is not only useful for reasoning. It also lets us *abstract away* from unimportant details. Every time we write $a + b + c + d$ without worrying where the brackets should go, we are taking advantage of the associative law—the second equation above—and thinking at a higher level of abstraction. This matters to mathematicians, and it also matters to programmers. Every time you sum an array in a loop,

```
int sum(int a[], int n)
{ int sum=0;
  for(int i=0;i<n;i++) sum = sum + a[i];
  return sum;
}
```

you are relying on associativity, to guarantee that it doesn't matter which *order* you combine the array elements in. When laws break down, there are problems. For example, *the code above is actually incorrect* for summing an array of floats! Floating point addition is *not* associative, and if you add up a million floats using a loop like this one, *you will get the wrong answer!* Instead you can use a clever algorithm called 'Kahan summation' to get a much better one (look it up; next time you have a million floats to add up, you'll thank me). But the lesson is this: when algebraic laws fail, the ground wobbles under our feet.

Computer scientists have been interested in the "*algebra of programs*" for more than half a century. In his classic 1966 paper 'The next 700 programming languages', Peter Landin wrote:

> For most programming languages there are certain statements of the kind, 'There is a systematic equivalence between pieces of program like this, and pieces like that,' that nearly hold but not quite. ... At first sight it might appear pedantic to quibble about such untidiness—'What's the point of having two different ways of doing the same thing anyway? Isn't it better to have two facilities than just one?' The author believes that *expressive power should be by design rather than accident*, and that there is great point in equivalences that hold without exception.

The desire for "equivalences that hold without exception" is one of the strong motivations for functional programming; that $x - x = 0$, but

```
getchar() - getchar() == 0
```

will usually be false in C, is a great impediment to algebraic reasoning.

However, the algebra of programs extends far beyond the usual algebra of numbers. In his 1978 Turing Award lecture, John Backus argued that the very *constructions of a programming language*, which he called “functional forms”, should be chosen to support algebraic reasoning:

> “One chooses only those functional forms that not only provide powerful programming constructs, but that also have attractive algebraic properties: one chooses them to maximize the strength and utility of the algebraic laws that relate them to other functional forms of the system.”

Backus advocated high-level combining forms such as `map` and `reduce` (fold), rather than low-level constructions such as sequencing or iterating statements, which is echoed in the “point-free” programming style popular in Haskell today.

Backus favoured algebra, over other kinds of reasoning, because it is *easy* and *practical*:

> The algebra of the programs described below is the work of an amateur in algebra, and I want to show that it is a game amateurs can profitably play and enjoy, a game that does not require a deep understanding of logic and mathematics. In spite of its simplicity, it can help one to understand and prove things about programs in a systematic, rather mechanical way.

Indeed, we can reason *algebraically* about programs, that one program is the same as another, without *any* external notations or tools. All we need is some middle school mathematics.

We can go on to apply algebra not only to numeric expressions and to programming language constructions, but to *all* our APIs! In their classic 1978 paper 'The Algebraic Specification of Abstract Data Types', Guttag and Horning argue that algebraic laws are the right way to *specify* the behaviour of an API:

> The set of axioms defines the meaning of the operations by stating their relationships to one another.
>
> They are easy to read and comprehend, thus facilitating informal verification of the fact that they do indeed conform to the intent of their creator.

When an API satisfies a rich set of laws, then the *algebra gives us freedom*:

- We're free to think at a higher level of abstraction, without worrying about trivial differences that the algebra assures us don't matter.
- We're free to optimize one piece of code to another with better performance, without risking introducing a bug, because the algebra assures us the two are equivalent.

Algebraic laws are a powerful approach to optimization, a key part of Burstall and Darlington's program transformation method, in their classic 1977 paper 'A Transformation System for Developing Recursive Programs'. Today you can even give performance-enhancing algebraic laws about an API to the Glasgow Haskell compiler, to be applied automatically by the optimizer to speed up client programs whenever opportunity arises.

If algebraic laws are so useful, then clearly it is highly *desirable* that an API should satisfy many of them. That means that algebra can serve as a *touchstone* for good design. In 1982 Peter Henderson invented "functional geometry", a simple API for describing complex pictures, with a beautifully simple algebra. Henderson later wrote: "It seems there is a positive correlation between the simplicity of the rules and the quality of the algebra as a description tool." You will learn all about Henderson's algebra later in this book; remember his advice—when choosing between design alternatives, choose the one with the better algebra! If your algebra is strong enough, you may even be able to *calculate the implementation* using it.

I experienced all this myself in the early 90s, when working on a library for writing pretty-printers in Haskell. Almost every data-structure needs a pretty-printer, and I was fed up writing the same kind of code over and over again, and making the same kinds of mistakes over and over again—getting the layout of pretty-printed output right is surprisingly tricky. So I designed an API for pretty-printers, but it wasn't crystal clear what each operator should do, with the result that my pretty-printers were *still* buggy in weird cases! But now that I had an API, I could look for algebraic laws—and tweak the meanings of the operators to satisfy them. That clarified the design wonderfully, and eliminated my bugs. I was even able to use my algebra to *calculate* a highly-efficient implementation, resulting in code that I could never have written by hand—with confidence that the algebra would ensure that all the weird corner cases were correctly handled. The paper I wrote as a result, 'The Design of a Pretty-printing Library' (1995) has spawned an entire mini-field of algebraically-based pretty-printing.

Of course, when you try to apply algebra to your own code in practice, you will immediately ask yourself questions like

- How do I know my code really satisfies this law?
- How can I figure out *which laws* my code satisfies?

Fortunately, today there are tools for Haskell that can answer these questions for you—QuickCheck to test that a particular law is satisfied, and QuickSpec to *find* a set of laws in the first place. So we are in a much better position, today, to apply algebraic methods to our code in practice, than the pioneers I quoted above.

But to apply these methods yourself, there is much to learn—and this is what this lovely book will teach you. What algebra is, how to apply it, what to look for, how to let it guide your designs, how to calculate your code, how to use the tools that are now available to support the approach. It's all illustrated with a new take on Peter Henderson's functional geometry, and a much larger-scale and more realistic game application, with a down-to-earth approach that you can put straight into practice. I am so glad that Sandy has written it—it provides an excellent introduction to so many ideas that I love.

Algebra enabled me to turn my pretty-printing code from a useful-but-buggy mess into a thing of beauty. The word 'algebra' itself comes from the work of Persian mathematician Muhammad ibn Musa al-Khwarizmi in the ninth century, and means "the restoring of broken parts". I think it's quite appropriate.

May this book teach you how to restore broken parts.

John Hughes
Gothenburg, Sweden, September 2020.

Preface

For the last seven years, I've been fascinated by what I see to be the central question in computing — "why isn't functional programming more popular?" In my eyes, functional programming (FP) is easier to get right, requires less effort to produce, and comes with stronger maintainability guarantees than the more conventional paradigms afford. But if FP is so fantastic, why hasn't it taken over the world yet?

I can see only three possibilities.

The most obvious one is that functional programming *simply isn't* all that much better. But this flies directly in the face of my experience and that of my colleagues in the FP community. There are numerous stories about people coming from procedural paradigms and falling in love with FP; but very few in which people go the other direction. Common themes are that functional programming is more "elegant," and "easier to reason about," and that it "expands our way of thinking."

Perhaps instead, it's that the market doesn't reward what functional programming brings to the table. It seems a little far-fetched, but maybe software doesn't live and die by the speed at which it's written, its correctness, and its maintainability. By being smaller than its procedural and object-oriented peers, functional programming languages boast significantly fewer *libraries,* which is likely part of the issue. It's not that the market doesn't reward speed, merely that until we achieve library parity, the mainstream inertia will keep its adherents. There is probably some truth to this. But I don't think this explains the whole story.

However, the third option is that we FP-people are just not very good at applying functional principles in large-scale, real-world applications. That is to say, maybe the problem isn't with functional programming; it's that we collectively aren't yet good enough with it. It's a common argument that functional programming works well in the small, but in the large, you actually need to deal with external systems and real-world complexity. It's in this interaction with reality that the cracks in FP begin to show.

I think FP's inability to take over the world is this last point. I believe that, as a community, we're not very good at scaling up functional thinking. There is no blame here; after all, we all have significantly more experience engineering procedural systems than we do functional ones. The issue is that it takes a lot of false starts to move forward. In the mainstream world, these false starts were already taken by our predecessors. But with functional programming only now just starting to gain widespread attention, we stand on our own, with little conventional knowledge to fall back on.

Fortunately, we're not alone in this endeavor. This book presents a fundamentally different approach for thinking about and writing software, one which plays to our strengths. It's not a novel idea by any means — while researching this book, I found that most of my discoveries were first unearthed in the mid-70s in the academic community. Academic researchers don't have the best track record for communicating their research outside of the ivory tower, and I fear that's what has happened here. An idea is only as good insofar as it can be acted upon. But it's important to note that the material presented here is in no way my own research.

To paraphrase Gwern Branwen: if this book has done anything meritorious, it was perhaps simply putting more work into than someone else would have. My goal has always been to help communicate better ideas than the ones I come up with. That's the natural progression of learning, and after a year and two complete rewrites of this book, boy have I ever learned a lot. I hope you do too.

Sandy Maguire
Victoria, BC, Canada
September 2020

Chapter 1

Overview

1.1 Abstraction

This book is about abstractions — how to think about them, find good ones, and wield them effectively. At first blush, this sounds like a question of style; certainly, abstraction is abstraction is abstraction, right? No, not only is this not right; it is not even wrong. When asked what abstraction *is,* many programmers will give one of the following answers:

1. "It's refactoring: pulling out duplicated code into a reusable function."
2. "It's indirection: providing thin wrappers around calls, and opening the doors for future extensibility."
3. "It's parameterization: putting adjustable knobs onto code and solving the more general problem."

Each of these is a means to a noble goal, but none of them is the abstraction tackled here. Instead, we will take a broader, more encompassing, easier-to-measure definition of what abstraction is. Instead, to quote Dijkstra (1972):

> The purpose of abstraction is not to be vague, but to

> create a new semantic level in which one can be absolutely precise.

To us, abstraction is only that which creates new, precise semantic domains. But what does this mean? A helpful abstraction is one which gives us an interpretation of the world that we accept as being *true.* All ideas are necessarily metaphors — reality is just too complicated for human brains. No idea is, therefore, a ground truth, but an excellent abstraction is one that never reminds you of its falsehood. A useful abstraction fundamentally changes the way you think and operate.

Spotting good abstractions is challenging, as they're often mistaken for the way that the world *really is.* Perfect abstractions are invisible. As an example, the computer hardware world makes excellent abstractions. Software doesn't actually proceed one instruction at a time: a modern CPU is decoding many hundreds of instructions at a time, and invisibly parallelizing them. But short of extremely-precise out-of-band timing attacks, you can't ever observe your CPU is doing this. This one-instruction-at-a-time abstraction is so persuasive that we operate as though it's true, and thus it never violates our implicit assumptions about how the code works. An excellent abstraction is out of sight and out of mind.

Likewise, are the fundamental building blocks of your computer really logic gates? No — logic gates don't *really* exist; they are an abstraction describing how transistors behave when placed in specific patterns. But the concept of logic gates is so persuasive that we collectively forget that are they nothing but useful figments of our collective imagination. Treat with suspicion anyone who says abstractions are fundamentally leaky; maybe these people are just *bad at abstraction.*

Other excellent abstractions are TCP/IP, Boolean algebra, Newtonian physics — even mathematics and logic themselves. These are good examples, not because they're accurate reflections of reality — but because we *forget that they aren't.* Contrast this sort of abstraction against what we usually encounter in software contexts. We're all too familiar with wrappers promising to unify disparate database interfaces, but which inevitably throw exceptions when one of the backends doesn't support an operation. Or consider how web-servers encourage us to think of HTTP headers as key/value pairs, but that a newline character in either key or value will compromise its entire security model. These are leaky abstractions, which is to say, worthless ones.

The goal of abstraction is to shield us from the reality beneath. If the real world somehow manages still to poke through, the abstraction-wielder must be aware now of both ground truth and the artificial, semantic layer on top. A careful practitioner now has two sets of invariants he must respect. Simultaneously he can't be sure of how the abstraction maps to reality or whether the leaks indicate a broken invariant somewhere. In essence, this wrong abstraction has doubled its practitioner's workload and her burden of understanding. Take a moment to appreciate just how common this is when writing software. Any system which gives you a backdoor to escape the abstraction is necessarily one which admits its incompetence. Computer systems give us no escape hatch to turn off instruction pipelining, nor can our programs opt-out of logic gates and instead work directly with transistors. Good abstractions don't require escape hatches.

Our discussion on abstraction is merely foreplay to set the stage for this book's main contribution: that *code is the wrong abstraction for doing programming.* There are infinitely many computer programs, the astronomical majority of which we don't want. There are so many that we *can't want* most of them. The argument is that unconstrained, implementation-first programming is too expressive as it's usually done. Code is just too powerful and too low-level for even the most diligent among us to understand truly any non-trivial program. Instead, we need better abstractions and tools for the decomposition, understanding, and solving of problems.

If you take away nothing else from this book, it should be that code is a uniquely terrible tool for thought. The traditional thought patterns taught in algorithms class — and sought out during software interviews — is fundamentally detrimental to the problems we're trying to solve. It is a testament to human heroics and industriousness that we can accomplish so much in the world of software despite these handicaps. Unfortunately, it's much more complicated than it needs to be, reflected in part by how normalized bugs are. Debugging is considered part of the job, and the vast majority of software has security flaws. The cost of software also reflects this pain: maintaining software is much more expensive than writing it in the first place. In most industries, the up-front costs dominate.

Why is this? My answer is that all software tends towards large systems and that large codebases are impossible for humans to understand in full. Worse, there are not any widespread tools to aid in that understanding. In an ideal world, the knowledge from the original design is *reusable,* and can be reliably shared with others. Imagine a world in which we all understood a codebase as well as its original author. Or if we could ask the compiler to ensure that our invariants always hold. Imagine if the abstractions never leaked and needing to debug an underlying library were a thing of the past. Imagine if the code were a byproduct of the understanding, and wrote itself.

Algebra-Driven Design is a framework for making that future *the future.*

1.2 What is Algebra-Driven Design?

> Programs are meant to be read by humans and only incidentally for computers to execute.
>
> –Harold Abelson

Writing software is hard — likely one of the most difficult challenges that individual humans have ever undertaken. Our brains aren't wired for it. We aren't well-adapted for thinking about subtle state interactions that accumulate over hundreds of thousands of lines of code. Code that is pedantic, written to describe, in excruciating detail, tasks which are self-evident to humans. Computers are, by and large, idiots, and the vast majority of a software engineer's job is understanding problems so well that you can explain them to these uncomprehending machines.

It is this comprehension that is of paramount importance. As this book argues, the software engineer's ability to understand problems is her primary employable skill. The code is merely a byproduct, serving only to explain this understanding to the computer — uninteresting in its own right.

While it might sound like a truism to suggest we focus on the understanding of problems rather than the *programming* of solutions, consider just how difficult this is with conventional tools and best practices. Our programming languages, the primary tool for thought for many software engineers, give us no support in this department. Our only facilities for writing code that is "easily understood" are to use descriptive variable names, write explanatory comments, and to optimize our code "to be read" — whatever that means.

But notice that these are all *code-centric* improvements. At their core, they still privilege the program as the fundamental unit of study — the very thing that is first and foremost an artifact for a computer to execute. The program is not our understanding of the problem and its solution; it is the end product of that understanding. If we're lucky, the program comes with comments that are simultaneously insightful and true, though ensuring this continues to be the case over time is probably asking too much. For better or worse (but mostly just for the worse,) documentation is our only tool for preserving institutional understanding of a codebase. Unfortunately, there is no tool in existence to ensure our documentation stays in sync with the system it purports to document. Indeed, tools like `doctest` can help show our examples produce the output, but there are no tools that check the *prose* of our comments.

Of course, the program still exists, and in some sense, is the source of truth of meaning. When documentation goes stale, we are not stuck; we can always reverse-engineer understanding from the program itself. Unfortunately, this state of affairs is almost synonymous with "programming," at least in most professional spheres. Over time, software engineers get quite good at this detective work — sussing out the "what" from the "how" — but it is essential to remember that this is inherently a lossy procedure. Rather than being able to page in the original author's understanding of the code, we must reinterpret it, reading between the lines. This is more of an exercise of psychology than of engineering, as the goal is to recreate another human's state of mind.

Civilizationally-speaking, we are remarkably successful in this endeavor of reverse-psychology. It's a testament to all of the heroic maintenance programmers out there who manage to keep these software systems up and running. But let's not mince words, this is truly a Herculean undertaking. Programming is a fundamentally challenging undertaking, yes, but it shouldn't be nearly as hard as it is. Nor need it be.

The irony here is in our rush to automate away other professions by providing better tools, by and large, we've forgotten to apply this same mindset to our own field. Rather than trying to solve underlying problems, which we are reasonably blind to, we usually find a convenient workaround. For example, why do we still represent and edit our source code as strings? Syntactically valid programs are a vanishingly small subset of all possible strings. The number of valid edits to a program is minuscule compared to the number of possible ways we can manipulate a string. Yes, source code does need eventually to be stored as bytes somewhere on a filesystem. But memory too is just a series of bytes, and unless you're a low-level C programmer, you almost certainly never think of memory like that. We don't manipulate instances of objects by explicitly twiddling their bytes in memory, so why do we still think about the raw representation of bytes when writing source code? Continually improving text editors are our convenient workaround here — but fundamentally, we are still thinking in terms of bytes, as indicated by our frustration when these tools "incorrectly" change our formatting.[1]

My point here is to illustrate that thinking about source code as a string of bytes is merely working at the wrong level of abstraction. It isn't difficult to imagine vastly better program editors[2] than today's best. Tools that edited programs as a tree and which would show only options that type-checked or were otherwise sane. Tools that would obsolete style arguments by displaying us code in whatever presentation we preferred. Tools that could automatically test the code we write to ensure it will never crash (or at least, only crash expectedly) when given garbage inputs. Imagine just how spectacular our tooling could be if we stopped insisting on seeing our code as bytes instead of structured objects that could be manipulated, transformed, inspected, and generated.

A core theme of Algebra-Driven Design is the insistence on working at the proper level of abstraction and on creating new levels if the available ones aren't sufficient. This book isn't here to harp on about how source code shouldn't be represented — or, for that matter, experienced — in bytes. No, the argument presented is that *programs themselves are the wrong level of abstraction,* and what we can do about that. This book is about learning to focus on the understanding and offloading most of the coding to our computer tool.

In the same way that having a structured model of source code would enable us to perform exciting transformations that are infeasible in the land of bytes, having a *structured model of understanding* affords us a great deal of flexibility. By reifying our knowledge of what our programs are, we can pass along *machine-checked* doc-

[1]Lisp is the notable exception to this point.

[2]It seems more natural to say "vastly better text editors," but that would be to miss the point entirely.

umentation that not only describes our understanding of what's going on, but is guaranteed to stay relevant (see chapter 8). The same machinery allows us to *automatically generate* thousands of unit tests — not only for properties we understand but also for emergent interactions between components that are necessary for human understanding of the system at large (chapter 7). By having our understanding explicitly modeled, it becomes an object of study in itself, and we can play around with the formulation to find better asymptotics or more elegant designs. Perhaps the most exciting feature of this approach is that it allows us directly to derive implementations from our understanding, and to discover mind-bending optimizations that are unlikely to be found by intuition alone.

In short, Algebra-Driven Design is an entirely new way of thinking about software engineering. To quote Elliott (2009), whose thinking on this topic has greatly influenced this book:

> Adopting the discipline illustrated [here] requires additional up-front effort in clarity of thinking, just as static typing does. The reward is that the resulting designs are simple and general, and sometimes have the feel of profound inevitability.

If you are happy toiling in the mire of source code, resigned to semantic games of "Telephone" between you and all other technicians who have touched a project over the years, then this is not the book for you. But if the above arguments have piqued your interest, if you've been dissatisfied with the way things are done in software, maybe this book can help.

This approach has three primary benefits over the usual intuition-driven, cowboy-coding that often dominates our profession. The first is that it allows us to focus on the fun part of software design, which is the *design* and understanding of the problem. It spares us most of the work of dealing with the incidental complexity, interfacing with clunky libraries, figuring out how exactly to decompose our dependency injection, and writing unit tests. These things are necessary for a working, shippable program, but they aren't software engineers' comparative advantage: understanding and dealing with complex systems.

As a byproduct of reifying our thinking, we find the second benefit: that we can offload a great deal of our work to the computer tool. By having machine-checkable artifacts, the computer can tell us when our laws combine in nonsensical ways, when our reference implementation doesn't do what the laws say, or when our real implementation doesn't line up with the reference. By doing this thinking outside of our heads, we can ask the computer to help suggest laws we might have missed, and generate thousands of unit tests following our specification — testing for edge cases that no human could ever consider "manually."

Having saved the best for last, the final benefit of ADD is that it allows us to derive implementations, if not "for free," then at least "greatly discounted." The equality laws guiding us through this entire process are excellent at finding the best "carve" of implementation through the design. The resulting programs are often beautiful, and more often than not, feel *discovered* rather than *engineered.* Through this approach, programs are elegant in their simplicity and generality, and playful manipulation of the laws is eerily good at finding asymptotic improvements. In essence, this means that the approach is a reliable generator of insights — both in design work and in coming up with intelligent optimizations.

In a genuine sense, Algebra-Driven Design is about designing programs *for humans.* The unreasonable effectiveness of mathematics seems related to the fact that it models things that humans can understand formally. Algebra is not a study of the world; it is a study of psychology. It is the study of those systems we humans can think clearly about.

With all of these significant advantages of Algebra-Driven Design, it's necessary to realize what it is *not.* ADD is incredibly helpful in finding reusable abstractions, which is to say, it's good when designing libraries. Applications are particular instantiations of reusable library components, and if you insist on thinking of an application as completely non-reusable code, ADD cannot help you. Instead, it encourages separating the core logic — the bits that make your program *your* program — from the chaff of programming tasks necessary to connect your program to the outside world. Strong adherence to ADD pushes library code to the forefront and resigns applications to thin wrappers around library functionality.

If you were never particularly good at math in school, take heart! Despite having the word "algebra" in its title, Algebra-Driven Design is not about the sort of algebra that causes nightmares. The algebra here has nothing to do with numbers, nor with arcane rules handed down by fiat from on high. The word "algebra" specifies that we are working algebraically, which is to say, thinking about what equations should hold, and manipulating those rules to find answers. Those answers? The implementations of the programs we want to write.

Furthermore, Algebra-Driven Design has nothing at all to do with visual aesthetics. It is not about "designing a good user experience" or with "designing a pleasing webpage." ADD is about designing excellent *software* — which is to say, software that is easy to understand, guaranteed to do what it says on the tin, and flexible enough to be used in more ways than its original authors ever could have intended.

Algebra-Driven Design isn't particularly prescriptive. It's an interactive process, a discussion between your tools and your intuition, helping refine ideas until they're beautiful.

Finally, Algebra-Driven Design need not be a solo endeavor. The entire point of giving laws and models is to share your understanding with others. Doing a formal review with your team of the design is a great way to find missed opportunities for additional structure or more elegant decompositions. Furthermore, the design acts as a living document, as machine-verified documentation, which gives future maintenance programmers (perhaps including yourself) a direct-link into your mind at the time you created the system, as well as to any improvements which have happened over the years.

1.3 Conventions

This book uses a few conventions throughout its pages. While it's not necessary to discuss the full "hows and whys" here and now — I'm sure you're hungry to jump into the thick of Algebra-Driven Design — we will need to cover the basics.

1.3.1 Why Haskell?

If you are already comfortable with the basic syntax of Haskell, and roughly how its type system works, feel free to skip to chapter 1.3.4.

This book presents its examples in Haskell and uses many common Haskell idioms. This might seem like a strange choice to many readers — why have I not picked a better-known language? Haskell is an odd duck of a language: it doesn't do object-oriented programming; it doesn't have mutable variables; its type-system is famously complicated, and its syntax is not inspired by C. So why use it?

First and foremost, Haskell is a fantastic tool for thought. Its seeming lack of "modern" features is not a flaw in the language; indeed, their absence can help us see design details that are often obscured by more conventional programming languages.

Haskell is considered a difficult language to learn, which is certainly true if you come from traditional procedural languages. But rest assured, this is not a *Haskell book.* You won't need more than a passing understanding of the language's syntax and a few high-

Part I

Designing Algebras

Chapter 2

Tiles

Doing Algebra-Driven Design requires a tremendous shift in perspective; we want to sculpt our software in thought, having it mostly worked out before we ever write a line of code. The reason behind this is that cajoling computers into executing our ideas takes effort, and empirically, programmers are likely to continue sinking costs on a lousy implementation, rather than starting again once they've figured out what they're building. Working out designs "in code" is akin to taking on technical debt before you've even started.

Instead, we should reason through our design's building blocks, fully working out their properties and interrelationships. We will hold off on writing any code for quite some time, which can be disconcerting; there will be nothing to "get our hands on" and interact with. We can't "get it running, then get it right after." Thus, all of the playing must happen in our minds, which is not a fair thing to ask of beginning practitioners. So we will start slowly, and use an incredibly visual first example. Hopefully, this will help convince you that it is possible to understand everything about a library before even a single line of code is written to implement it.

Our example algebra follows closely from Henderson (2002), and is one for constructing images out of recursively-subdivided images, any of which might be modified by a simple spatial transformation. For example, our algebra can describe images like figure 2.1, figure 2.2 and figure 2.3.

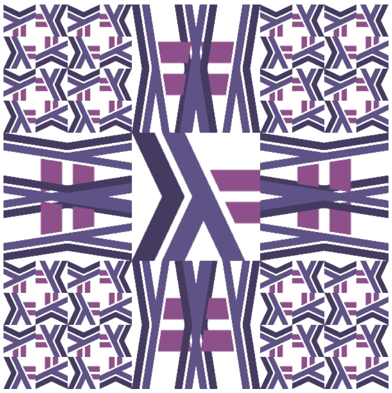

Figure 2.1: Example 1

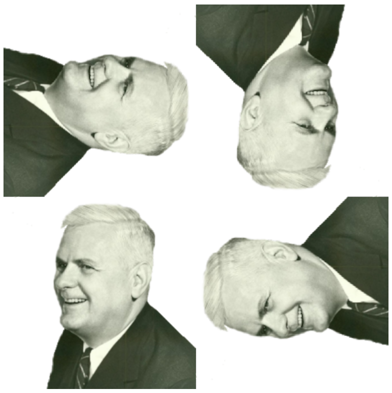

Figure 2.2: Swirling mathematicians

The eventual algebra we will find is small and powerful — with only modest additions, Henderson (2002) uses it to recreate some of M. C. Escher's artwork.

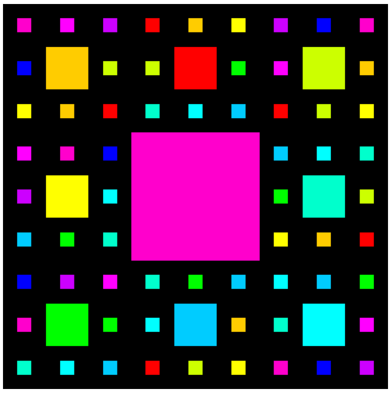
Figure 2.3: Sierpinski Carpet

2.1 Basic Building Blocks

We begin by noting that there exists a type for tiles, which appropriately we will name `Tile`:

```haskell
data Tile
```

We can illustrate our tile-algebra with two *terminal* constructors: `haskell` and `church`. Our final implementation will involve means for a user to get custom images into the system. Still, for now, we choose not to worry ourselves about image formats or input/output and simply provide these two tiles by fiat.

```haskell
haskell :: Tile
church :: Tile
```

When rendered, these tiles look like figure 2.4 and figure 2.5.
All terms in an algebra are built from terminal constructors (like `haskell` and `church` above) and *inductive* constructors: ones which

Figure 2.4: haskell

Figure 2.5: church

"derive" new terms based on existing terms. For example, if we have a `Tile` we might want to rotate it 90 degrees clockwise, as shown in figure 2.6.

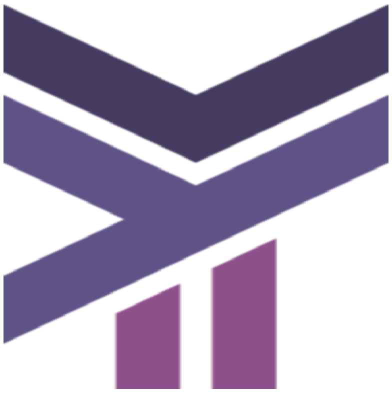
Figure 2.6: cw haskell

Let's call this operation `cw` ("clockwise"). Because `cw` is an operation that consumes a `Tile` and produces a new one, its type is:

```
cw :: Tile -> Tile
```

Because `cw` works on *any* `Tile`, we can apply it to the result of itself as in figure 2.7.
However, an interesting thing happens when we call `cw` on itself *four* times — we somehow get back to where we started. If you don't believe me, compare figure 2.4 and figure 2.8! Because we all have decades' worth of experience in the domain of geometry, this result isn't particularly startling — which is always the danger when analyzing a simple structure. But when you stop to think about it, this property of "four times around gets you back where you started" is quite the *defining characteristic* of rotation by 90 degrees. If the number of times we needed to call `cw` to get back

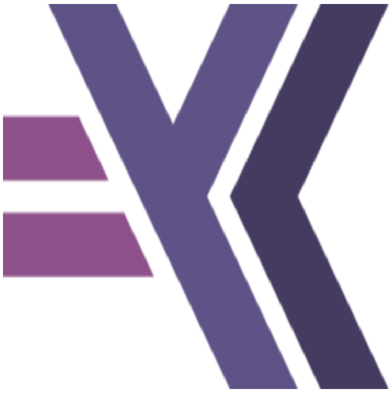

Figure 2.7: cw (cw haskell)

to the original `Tile` were not four but instead n, our rotation *must instead* be by $\frac{360}{n}$ degrees. Thus, we can closely specify what it is we're talking about by requiring the following law always holds true:

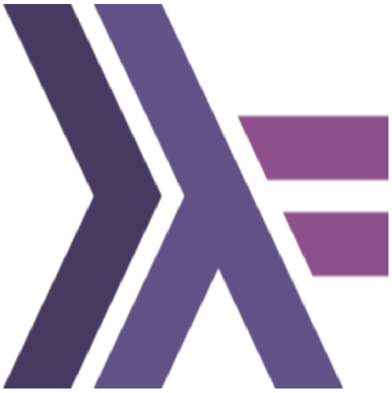

Figure 2.8: cw (cw (cw (cw haskell)))

Law: "cw/cw/cw/cw"

```
∀ (t :: Tile).
  cw (cw (cw (cw t))) = t
```

It's important to understand where this law comes from. Because of our problem's underlying geometry, this law is *foisted upon us;* we have no choice but to accept it. Our understanding of the problem itself has uncovered this law, and there is nothing we can do about it, short of changing our minds about what `cw` should do. This is always the case when designing algebras; because an algebra must have consistent semantics, decisions we make ripple throughout the design, forcing constraints upon us. Our final design will consist of dozens of equations like these. *Any implementation of the design is necessarily a solution to that system of equations.*

As you work through this or any other algebra, keep in mind the slogan,

It's the equations that really matter!

The laws are of critical importance, as they are what foist meaning upon our otherwise empty syntactic constructs.

Let's return to our algebra. Of course, there is no reason to privilege one direction of rotation over another. Let's also provide a `ccw` ("counterclockwise") constructor with the same type, as illustrated in figure 2.9.
We don't require both `cw` and `ccw` — having both gives us no additional expressiveness than having only one. To see this for yourself, note that rotating counterclockwise once is equivalent to rotating clockwise three times, and vice versa. If we valued extreme parsimony, we could certainly get by having only one of these combinators, but we will provide both because they are both useful. However, having two ways of getting the same result gives us twice

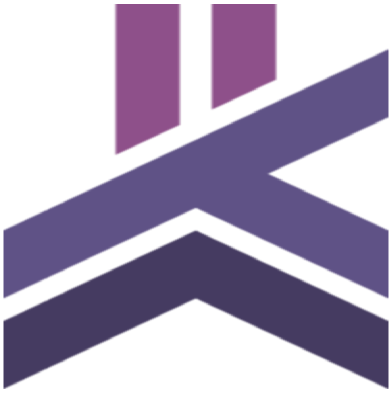

Figure 2.9: ccw haskell

as many chances to write a bug. But we can subvert such a problem by simply giving a law relating `cw` to `ccw`; so long as the law holds, there can be no bug here.

The most obvious relationship between `cw` and `ccw` is that they are inverses of one another. Performing either after the other is equivalent to doing nothing.

Law: "ccw/cw"

```
∀ (t :: Tile).
  ccw (cw t) = t
```

Law: "cw/ccw"

```
∀ (t :: Tile).
  cw (ccw t) = t
```

Again, these laws are nothing we have control over; they are required to hold for any `cw` and `ccw` that could correspond to our informal notions that these operations should rotate their arguments 90 degrees. In time, you will learn to analyze software in terms of which laws it satisfies, that is to say, you will be able to visualize software via its equations.

Our equations don't just look like the sort of algebra you did in grade-school; indeed, we can use them in just the same way. For example, we can derive the fact that rotating clockwise three times is equivalent to going counterclockwise once, via simple algebraic manipulation. We start with the fact that rotating clockwise four times is equivalent to doing nothing, then use `ccw` on both sides of the equation, and then use the fact that `ccw` eliminates a `cw`:

```
  ccw t
= . . . . . . . . . . . . . . . . . . . . .   (via "cw/cw/cw/cw")
  ccw (cw (cw (cw (cw t))))
= . . . . . . . . . . . . . . . . . . . . . . . . .   (via "ccw/cw")
  cw (cw (cw t))
```

This sort of algebraic manipulation is extraordinarily useful when it comes to solving your algebra's system of equations (that is to say: actually implementing it.)

So far, our algebra isn't very powerful. Let's introduce some new constructors. We will add the capability to mirror a tile horizontally — illustrated by figure 2.10.

```haskell
flipH :: Tile -> Tile
```

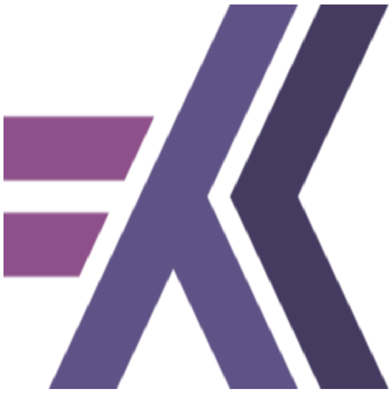

Figure 2.10: flipH haskell

Mirroring a tile is a fundamentally new idea in our algebra; it simply can't be expressed in terms of `cw` or `ccw`. Its major governing law is that `flipH` is its own inverse:

Law: "flipH/flipH"

```haskell
∀ (t :: Tile).
  flipH (flipH t) = t
```

However, we note that mirroring a tile, rotating it twice, and then mirroring it back is equivalent to just turning it twice. This equation helpfully relates `flipH` to `cw`, entangling their semantics. Again, this law is nothing other than a logical conclusion if `flipH` flips about its X-axis and `cw . cw` simultaneously flips its X- and

Y-axes. This is a fact about our problem and is the only reasonable conclusion if `flipH` and `cw` behave like our minds' eyes say they do.

Law: "flipH/cw/cw/flipH"

```
∀ (t :: Tile).
  flipH (cw (cw (flipH t)) = cw (cw t)
```

Exercise Prove `flipH . cw^{2*n} . flipH = cw^{2*n}`, where the `^` operation means repeated composition. For example, `cw^4 = cw . cw . cw . cw`.

A little mental geometry shows us that horizontally flipping a clockwise rotation is equivalent to rotating counterclockwise a horizontal flip. This is a delightful law that relates `cw` to `ccw` under the `flipH` transformation: to say, that

Law: "x-symmetry"

```
∀ (t :: Tile).
   flipH (cw t) = ccw (flipH t)
```

Exercise Find a way of recreating figure 2.11, using only `cw`, `ccw` and `flipH`.

The operation carried out in figure 2.11 is equivalent to flipping a `Tile` vertically. Rather than require our users to perform the

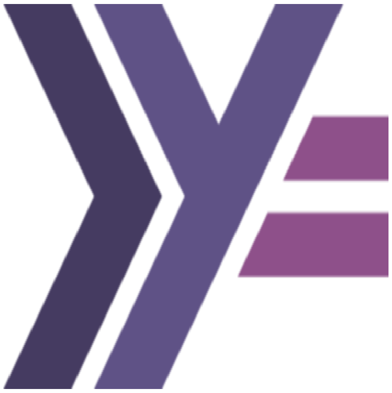

Figure 2.11: Recreate this tile

complex set of operations to create it by hand, we will offer this effect as a constructor in its own right.

```
flipV :: Tile -> Tile
```

Of course, `flipV` is also its own inverse, but two other interesting equations as well – that we can derive it from `cw`, `ccw` and `flipH`, and that performing both flips is equivalent to doing two rotations.

Law: "flipV/flipV"

```
∀ (t :: Tile).
  flipV (flipV t) = t
```

Law: "ccw/flipH/cw"

```
∀ (t :: Tile).
  flipV t = ccw (flipH (cw t))
```

Law: "flipV/flipH"

```
∀ (t :: Tile).
  flipV (flipH t) = cw (cw t)
```

Exercise Derive the fact that `flipV` is its own inverse, using any of the *other* laws we've given for our algebra.

Solution

```
  flipV (flipV t)
= . . . . . . . . . . . . . . . . . . . . .   (via "flipV")
  flipV (ccw (flipH (cw t)))
= . . . . . . . . . . . . . . . . . . . . .   (via "flipV")
  ccw (flipH (cw (ccw (flipH (cw t)))))
=  . . . . . . . . . . . . . . . . . . .   (via "cw/ccw")
  ccw (flipH (flipH (cw t)))
= . . . . . . . . . . . . . . . . . . .   (via "flipH/flipH")
  ccw (cw t)
=  . . . . . . . . . . . . . . . . . . .   (via "ccw/cw")
  t
```

Exercise Derive a proof that `flipV . flipH = cw . cw`

Solution

```
  flipV (flipH t)
= · · · · · · · · · · · · · · · · · · · · · ·  (via "flipV")
  ccw (flipH (cw (flipH t)))
= · · · · · · · · · · · · · · · · · · · · · ·  (via "ccw")
  cw (cw (cw (flipH (cw (flipH t)))))
= · · · · · · · · · · · · · · · · · ·  (via "x-symmetry")
  cw (cw (flipH (ccw (cw (flipH t)))))
= · · · · · · · · · · · · · · · · · · · ·  (via "ccw/cw")
  cw (cw (flipH (flipH t)))
= · · · · · · · · · · · · · · · · · · · ·  (via "flipH/flipH")
  cw (cw t)
```

2.2 Subdividing Space

It's time to add some intrigue to our algebra. As great as transforming square tiles is, it's just not enough to capture our imaginations for long. We will now introduce our killer feature: being able to compose multiple tiles together. The most exciting of these is `beside`, which lays out one tile beside another.

Because our tiles are always square, we need to determine how to *close under* this operation; recall, every operation in an algebra must take valid inputs to valid outputs. Simply sticking one square tile beside another would result in a rectangular image, which would not be a square tile! Instead, we decide that to maintain closure, we will first subdivide our square into two rectangular halves, and then fill each half, stretching the tiles to cover the space. Our new operation is illustrated in figure 2.12.

```
beside :: Tile -> Tile -> Tile
```

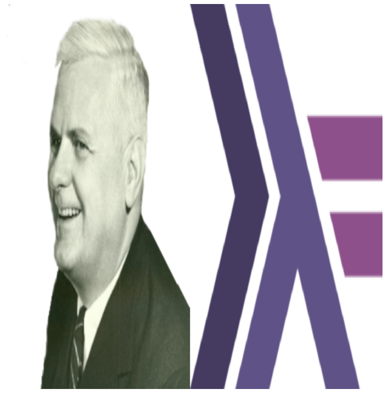

Figure 2.12: beside church haskell

Of course — because of the closure property, we can freely nest calls to `beside`, as in figure 2.13 and figure 2.14.

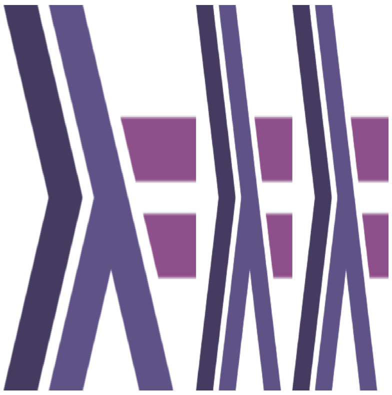

Figure 2.13: beside haskell (beside haskell haskell)

As you might expect, we should look for some laws relating `beside` to our other constructors. This should always be our modus

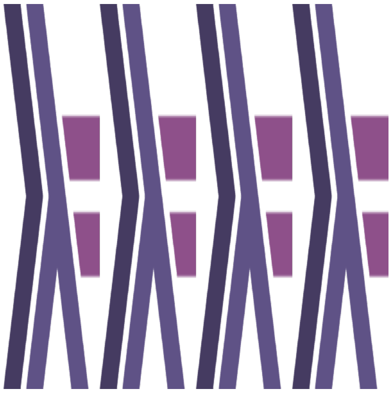

Figure 2.14: beside (beside haskell haskell) (beside haskell haskell)

operandi when working with algebras; for every new constructor you add, look for a way to connect it to other things in your algebra. Over time, this web of connections will strengthen and often help us find properties that are too hard to deduce by intuition alone. In this case, because `beside` is aligned along the X-axis, we should search for equalities that preserve the X-axis. Interestingly, some mental manipulation shows us that `flipH` distributes through `beside`, but in doing so, flips the order of its arguments:

Law: "flipH/beside"

```
∀ (t1 :: Tile) (t2 :: Tile).
  flipH (beside t1 t2) = beside (flipH t2) (flipH t1)
```

Exercise Prove `flipH (flipH (beside t1 t2)) = beside t1 t2` in two separate ways.

By some clever manipulation — reminiscent of how we derived `flipV`, we can position one tile above another — as shown in figure 2.15.

Exercise Recreate figure 2.15, using `beside`, `cw` and `ccw`.

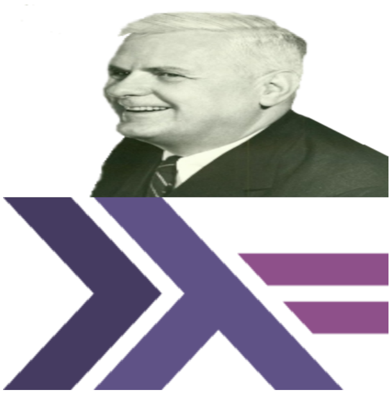

Figure 2.15: Recreate this tile

Again, rather than make our users jump through hoops, we will just provide `above` as its own constructor.

```haskell
above :: Tile -> Tile -> Tile
```

Law: "above"

```
∀ (t1 :: Tile) (t2 :: Tile).
  above t1 t2 = cw (beside (ccw t1) (ccw t2))
```

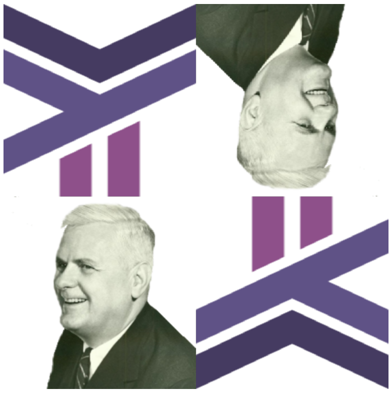

Figure 2.16: above (beside (cw haskell) (cw (cw church))) (beside church (ccw haskell))

Intuitively, we can also rewrite an `above` of `beside`s as a `beside` of `above`s, so long as we swap the top-right and bottom-left tiles when we do so.

Law: "above/beside"

```
∀ (a :: Tile) (b :: Tile) (c :: Tile) (d :: Tile).
  above (beside a b) (beside c d) =
    beside (above a c) (above b d)
```

The construction of four tiles in a square — as in figure 2.16 — turns out to be a particularly common pattern. Let's call it `quad`:

```
quad :: Tile -> Tile -> Tile -> Tile -> Tile
```

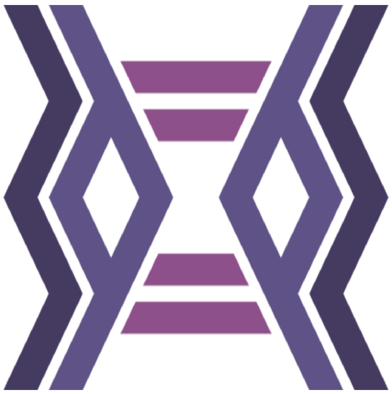
Figure 2.17: quad haskell (flipH haskell) (flipV haskell) (flipV (flipH haskell))

Law: "quad"

```
∀ (a :: Tile) (b :: Tile) (c :: Tile) (d :: Tile).
  above (beside a b) (beside c d) = quad a b c d
```

As an even more special case, we can rotate one tile as we move through a quad, creating a sort of `swirl` effect as in figure 2.18. This operation is given by:

```
swirl :: Tile -> Tile
```

and is subject to the law:

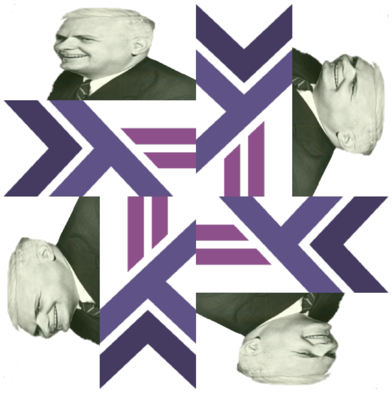
Figure 2.18: swirl (above church haskell)

Law: "swirl"

```
∀ (t :: Tile).
  quad t (cw t) (ccw t) (cw (cw t)) = swirl t
```

We now come to final spatial operation, `behind`, which allows us to layer one tile on top of another as in figure 2.19:

```
behind :: Tile -> Tile -> Tile
```

As great as all of these spatial constructions are, it will be helpful to have another terminal constructor — one which fills the space with a given color. Let's call it `color`, and give it the type:

```
color
    :: Double  -- ^ red
```

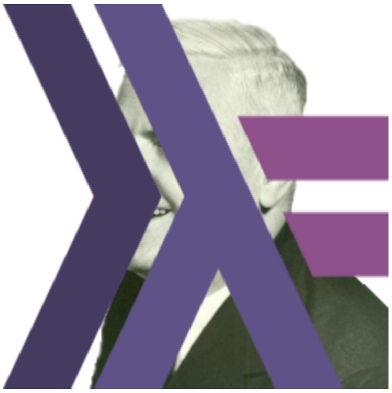

Figure 2.19: behind church haskell

```
  -> Double  -- ^ green
  -> Double  -- ^ blue
  -> Double  -- ^ alpha
  -> Tile
```

Each of these channels should be within the closed interval `[0,1]`. Of course, nothing in the typesystem requires this to be the case, so we will need to constrain it with a law:

Law: "clamp channels"

```
∀ (r :: Double) (g :: Double) (b :: Double)
      (a :: Double).
  color r g b a =
    color (clamp 0 1 r)
          (clamp 0 1 g)
          (clamp 0 1 b)
          (clamp 0 1 a)
```

Figures figure 2.20, figure 2.21 and figure 2.22 give some illustrations of the different channels, and how alpha blending works.

Figure 2.20: color 1 0.8 0 1

The `color` combinator has the interesting property that is unaffected by `cw` and `flipH`:

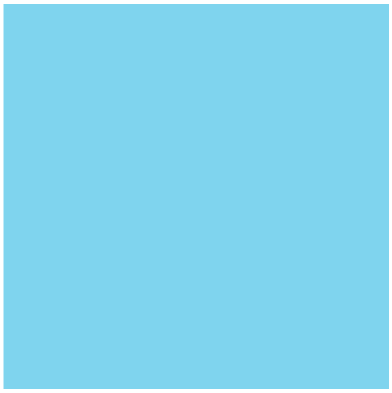

Figure 2.21: color 0 0.67 0.87 0.5

Law: "cw/color"

```
∀ (r :: Double) (g :: Double) (b :: Double)
      (a :: Double).
  cw (color r g b a) = color r g b a
```

Law: "flipH/color"

```
∀ (r :: Double) (g :: Double) (b :: Double)
      (a :: Double).
  flipH (color r g b a) = color r g b a
```

The semantics of `color` are "obvious" to a human, but it's not entirely clear how to *specify* such an operation. Of particular challenge is its interaction with `behind` as demonstrated in figure 2.22, where we should expect alpha compositing to occur. We can attempt a partial specification by noting what happens with extreme alpha values. If the alpha channel of the `color` in front is fully set, it doesn't matter what was behind it:

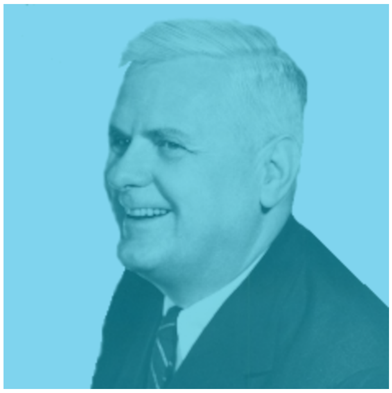

Figure 2.22: behind church (color 0 0.67 0.87 0.5)

Law: "opaque"

```
∀ (t :: Tile) (r :: Double) (g :: Double) (b :: Double).
  behind t (color r g b 1) = color r g b 1
```

Likewise, if there is zero alpha, it's the same as having not invoked `behind` at all.

Part II

Deriving Implementations

Chapter 5

Tile Implementation

In chapter 2, we specified the algebra for an image tiling algebra. As a quick refresher, our algebra allows us to build images by subdividing space into “tiles,” which can be rotated and flipped. Our public interface eventually ended up looking like this:

```
data Tile a
instance Functor Tile
instance Applicative Tile

rasterize :: Int -> Int -> Tile a -> [[a]]

cw    :: Tile a -> Tile a
ccw   :: Tile a -> Tile a
flipH :: Tile a -> Tile a
flipV :: Tile a -> Tile a

quad  :: Tile a -> Tile a -> Tile a -> Tile a -> Tile a
swirl :: Tile a -> Tile a

beside :: Tile a -> Tile a -> Tile a
above  :: Tile a -> Tile a -> Tile a
```

```
empty  :: Monoid a => Tile a
behind :: Monoid a => Tile a -> Tile a -> Tile a
```

We are left now with the task of *actually implementing* this thing. Unfortunately, computers are unable to execute our specifications directly, so we must translate our design into code. Our path for getting there will be incremental, done in three distinct phases. First, we will implement a solution that is naive but obviously-correct. Second, we will use the naive implementation to generate a broad suite of regression tests automatically. Finally, we will give an innovative, optimized implementation, using our new test suite to ensure we haven't introduced any bugs along the way.

5.1 The Initial Encoding

By construction, every algebra gives rise to an "obvious" implementation called its *initial encoding.* You might be familiar with initial encodings under the object-oriented concept "interpreter pattern." If not, the initial encoding is a fancy name for implementing each of our algebra's constructors as a *data constructor directly in the implementation language.*

The idea is this: we will build an explicit data structure representing our tile. Later, when we go to "interpret" it — that is, `rasterize` it — the drawing routines can traverse this tree, doing different drawing operations for every different sort of node. An initial encoding is characterized by creating a tree of data whose nodes correspond precisely with the algebraic constructors. For this reason, we might refer to "trees" and "nodes" in the following section, but these refer only to terms and constructors, respectively.

To illustrate the initial encoding, one of the infinitely many ways of implementing `Tile` is as follows:

```
data Tile a where
  Cw    :: Tile a -> Tile a
  Ccw   :: Tile a -> Tile a
  FlipH :: Tile a -> Tile a
  FlipV :: Tile a -> Tile a
  Quad  :: Tile a -> Tile a -> Tile a -> Tile a -> Tile a
  Swirl :: Tile a -> Tile a
  Beside :: Tile a -> Tile a -> Tile a
  Above  :: Tile a -> Tile a -> Tile a
  Empty  :: Monoid a => Tile a
  Behind :: Monoid a => Tile a -> Tile a -> Tile a
  Fmap :: (a -> b) -> Tile a -> Tile b   · · · · · · · ·   ❶
  Pure :: a -> Tile a   · · · · · · · · · · · · · · · · ·   ❷
  Ap   :: Tile (a -> b) -> Tile a -> Tile b   · · · · · ·   ❸
```

The constructors marked by ❶ , ❷ and ❸ correspond respectively to the (implicit) ways of building tiles via `fmap`, `pure`, and `(<*>)`.

You might be wondering why `rasterize` doesn't appear here. It doesn't, by virtue of not being a constructor. It's an observation over our algebra. Rasterizing is our only way to get data out of the system; it cannot help us get information in.

With the definition of `Tile` in place, we can give a trivial implementation for every constructor in our algebra:

```
cw :: Tile a -> Tile a
cw = Cw

ccw :: Tile a -> Tile a
ccw = Ccw

flipH :: Tile a -> Tile a
flipH = FlipH
```

```
-- etc
```

By giving a Haskell constructor for every constructor in our algebra, we can build a one-to-one mapping between the terms in our algebra and a "syntax tree" for our algebra in Haskell. There is absolutely no computational power here; all we have managed to compute is a tree in memory that corresponds precisely with a term in our algebra. We can now imagine implementing the `rasterize` observation by pattern matching on this tree — doing the "right thing" in each case.

But this one-to-one mapping is not the whole story; it doesn't necessarily satisfy the laws that it is supposed to. For example, **"flipH/flipH"** states that `flipH . flipH = id`, but this is decidedly not true given the definition of `flipH` above. We can force the laws to hold by fiat, simply by performing some pattern matching in the definition of `flipH`:

```
flipH :: Tile a -> Tile a
flipH (FlipH t) = t  . . . . . . . . . . . . . . . . . . . . .   ❶
flipH t = FlipH t  . . . . . . . . . . . . . . . . . . . . . .   ❷
```

This new definition, at ❶ , checks a term to see if its root is already a `FlipH` node, and if so, removes it — satisfying the law that `flipH . flipH = id`. If this is *not* the case, as in ❷ , we instead *add* a `FlipH` node. As another example, we expect that `cw . cw . cw . cw = id`, and can encode it as follows:

```
cw :: Tile a -> Tile a
cw (Cw (Cw (Cw t))) = t  . . . . . . . . . . . . . . . . . .   ❶
cw t = Cw t
```

Because it's non-obvious, we should note that it's vital that there are only three `Cw` data constructors at ❶ . The fourth comes from the definition of `cw` itself.

You can imagine how a patient and assiduous implementer could go through every equation in our algebra and write them, one by one, as pattern matches. Such an approach would certainly ensure that every law holds, but it quickly becomes an untenable amount of work for even moderately-sized algebras.

Instead of doing all this work by hand, we can employ our equations themselves to simplify the task. The high-level idea here is that we can pick a core set of "primitive" operations in our algebra, and use our laws to rewrite every other combinator in terms of the primitives. Instead of the massive thirteen-constructor definition of `Tile` above, we can distill it down into only these five:

```haskell
data Tile a where
  Cw     :: Tile a -> Tile a
  FlipH  :: Tile a -> Tile a
  Above  :: Tile a -> Tile a -> Tile a
  Pure   :: a -> Tile a
  Ap     :: Tile (a -> b) -> Tile a -> Tile b
```

Where did these five come from, you might wonder. The answer is that they come from trial and error, and I have saved my readers from the trial on this particular example. Picking primitives is more of an art than a science; we can allow our intuition and implementation experience to suggest constructors to use as primitives. There is little consequence for picking a lousy set of primitives; if it's not minimal, you'll have to do a bit more work. If it's insufficient to implement the entire algebra, you'll get stuck within five minutes and can then backtrack.

In the case of `Tile` above, our primitive data constructors correspond with constructors in our algebra, but this is not a requirement. We will look at an example of a drastically different representation in chapter 6.

We still should give the pattern matching by-fiat laws when implementing the algebra's constructors in terms of these data constructors. But by having fewer syntactical forms in the mix, we have fewer combinations of laws we need to enforce manually. Let's first give the remaining primitive implementations:

```haskell
above :: Tile a -> Tile a -> Tile a
above = Above

instance Applicative Tile where
  pure = Pure
  (<*>) = Ap
```

The trick is now to find derivations of the other, non-primitive constructors in terms of these primitives. To illustrate this, we'd like to find a derivation of `ccw`:

```
  ccw t
=  . . . . . . . . . . . . . . . . . . . . . .   (via "cw/cw/cw/cw")
  ccw (cw (cw (cw (cw t))))
= . . . . . . . . . . . . . . . . . . . . . . . . . . .   (via "ccw/cw")
  cw (cw (cw t)))
```

Since we have the implicit primitive law that `Cw = cw`, we have derived an implementation of `ccw` in terms of our primitives. We choose to use `cw` here rather than the data constructor `Cw` directly because `cw` performs the automatic simplification of **"cw/cw/cw/cw"**. This derivation maps directly to an implementation:

```
ccw :: Tile a -> Tile a
ccw t = cw (cw (cw t))
```

or, in its eta-reduced form:

```
ccw :: Tile a -> Tile a
ccw = cw . cw . cw
```

In general, it's imperative to be very careful in our use of the primitive forms. We have chosen to implement `ccw` here in terms of the constructor `cw` rather than the primitive `Cw`. Why is this? It's to ensure that our by-fiat laws get a chance to simplify the expression. If we instead implemented `ccw` in terms of `Cw`, it's easy to see how `ccw . ccw` would produce six nested `Cw` nodes — which is a state impossible under the implementation of `cw`. In this case, it probably doesn't matter — but it can lead to obscure bugs. Thus, it is prudent to use only a primitive form to implement exactly one constructor, if at all possible. Doing so will save you many headaches down the line.

To look at a more complicated derivation, let's take `flipV`:

```
∀ (t :: Tile).
  flipV t
= . . . . . . . . . . . . . . . . . . . . . . .   (via "rotated flipH")
  ccw (flipH (cw t))  . . . . . . . . . . . . . . . . . . .  ❶
= . . . . . . . . . . . . . . . . . . . . . .   (via "cw/cw/cw/cw")
  ccw (cw (cw (cw (cw (flipH (cw t))))))
= . . . . . . . . . . . . . . . . . . . . . . . . . . .   (via "ccw/cw")
  cw (cw (cw (flipH (cw t))))
```

This derivation goes all the way to our primitive forms `cw` and `flipH`, which is technically the right way to go about things. However, you'll notice that the line of the proof marked by ❶ is already a perfectly good implementation, as the remainder of the proof is simply expanding out the implementation of `ccw`. So long as we approach these derivations in a topologically sorted manner — that is, being careful to use only those constructors we have already implemented — we can shave off a good deal of the derivation work. Working "all the way through" a derivation is more important when you haven't enforced any equations in your implementation by fiat. We can thus give an implementation for `flipV`:

```
flipV :: Tile a -> Tile a
flipV = ccw . flipH . cw
```

The remainder of the implementations follow in this pattern of using previously-implemented constructors. Each is an easy, one-step derivation. Let's start with `above`:

```
  beside t1 t2
=  . . . . . . . . . . . . . . . . . . . . . . . .   (via "above")
  ccw (above (cw t1) (cw t2))
```

```
beside :: Tile a -> Tile a -> Tile a
beside t1 t2 = ccw (above (cw t1) (cw t2))
```

Nice. Again, the implementation follows directly from the laws. This is also the case for `quad`:

```
  quad t1 t2 t3 t4
=  . . . . . . . . . . . . . . . . . . . . . . . . .   (via "quad")
  above (beside t1 t2) (beside t3 t4)
```

```haskell
quad :: Tile a -> Tile a -> Tile a -> Tile a -> Tile a
quad t1 t2 t3 t4 = above (beside t1 t2) (beside t3 t4)
```

and again for `swirl`:

```
  swirl t
= . . . . . . . . . . . . . . . . . . . . . . . .   (via "swirl")
  quad t (cw t) (ccw t) (cw (cw t))
```

```haskell
swirl :: Tile a -> Tile a
swirl t = quad t (cw t) (ccw t) $ cw $ cw t
```

In order to give an implementation of `fmap`, we can turn to the applicative laws:

```
  fmap f t
= . . . . . . . . . . . . . . . . . . . . . . .   (via "pure/ap")
  pure f <*> t
```

which in turn is the delightful (and applicable for every applicative functor) instance:

```haskell
instance Functor Tile where
  fmap f t = pure f <*> t
```

Finally, `empty` and `behind` turn out to be specific cases of the applicative operations when generalized to monoids:

```
empty :: Monoid a => Tile a
empty = pure mempty

behind :: Monoid a => Tile a -> Tile a -> Tile a
behind = flip (liftA2 (<>))
```

Et voila! We automatically have a lawful set of implementations for our initial encoding. That said, there is still no code that generates an image — our observations are still unimplemented. But this is a problem easily fixed; we need only give an interpreter for the `Tile` syntax tree. Here again, the laws suggest another implementation, one which operates directly over two-dimensional lists. Such a thing seems like it would be notoriously slow and inefficient — because it will be — but remember, we're looking only for an obviously-correct implementation, not a *good* one. That will come later.

We proceed by implementing `rasterize :: Int -> Int -> Tile a -> [[a]]` as a piece-wise fashion, pattern matching on each data constructor of `Tile`. The `Pure` case is simplest; it corresponds to a pixel matrix with a constant value in every cell. Recalling that the result of `rasterize` should be row-major, we can construct a row with the right width of pixels via `replicate`, and then replicate that to get the correct height.

```
rasterize w h (Pure a) = replicate h $ replicate w a
```

When designing the algebra, we took great care also to provide a `rasterize'` observation which is equivalent to `rasterize`, but which is in a form more amenable to applicative homomorphisms. We can exploit that machinery now in order to implement `Ap`:

```
rasterize w h (Ap f a) =
  coerce (rasterize' w h f <*> rasterize' w h a)
```

Horizontally flipping an image is also easy; we simply rasterize the underlying tile and then `reverse` each of its rows:

```
rasterize w h (FlipH t) = fmap reverse $ rasterize w h t
```

Rotation of a tile is a little trickier. Intuition tells us that rotating a non-square matrix will swap the width and height dimensions, and since we would like the *result* of `cw` to have the specified dimensions, we must swap the width and height that we give when rasterizing the inner tile:

```
rasterize w h (Cw t) = rotate2d $ rasterize h w t
  where
    rotate2d = fmap reverse . transpose
```

Our only remaining primitive to interpret is `Above`, which conceptually builds two half-height tiles and glues them vertically. This definition again maps directly to an implementation:

```
rasterize w h (Above t1 t2) =
    rasterize w (div h 2) t1 <>
    rasterize w (h - div h 2) t2   · · · · · · · · · · · ·   ❶
```

At ❶ we use `h - div h 2` as the second width in case the desired height is odd — we wouldn't want to drop a pixel accidentally. And just like that, we have plucked a fully working implementation seemingly out of thin air.

Finally, let's give some instances for `Tile`. It admits obvious `Semigroup` and `Monoid` instances, simply by lifting an instance from the "pixel" type:

```haskell
instance Semigroup a => Semigroup (Tile a) where
  (<>) = liftA2 (<>)


instance Monoid a => Monoid (Tile a) where
  mempty = pure mempty
```

Additionally, we'll need a `Show` instance in order for QuickCheck to test our properties later. This instance unfortunately can't be automatically derived due to the existential type in `Ap`, but we can write it by hand without much effort:

```haskell
instance Show a => Show (Tile a) where
  show (Cw t) = "cw (" ++ show t ++ ")"
  show (FlipH t) = "flipH (" ++ show t ++ ")"
  show (Above t1 t2)
    = "above (" ++ show t1 ++ ") (" ++ show t2 ++ ")"
  show (Pure a) = "pure (" ++ show a ++ ")"
  show (Ap _ _) = "ap _ _"  . . . . . . . . . . . . . . . .  ❶
```

There is no `Show` instance for the existentially quantified type in `Ap`, so we ignore those two arguments when printing `ap` terms at ❶ . This isn't a perfect solution by any means, but it's practical and gets the job done.

5.2 Generating Tests

In the previous section, we followed our algebra's rules to derive an implementation automatically. The resulting code, while inefficient and naive, is overwhelmingly simple and is guaranteed to follow our specification. The next step is to ensure that the specification corresponds with our *intuition*, as the specification is only useful insofar as it helps us solve problems we are interested in.

Here again, we can automate a great deal of the tedium involved in checking the implementation. Most systems are verified using hand-written unit tests: little checks that the software behaves predictably in particular scenarios. Unit tests are a good start but suffer from the problem that they are boring and written by humans. To quote Hughes (2016):

> Imagine writing a suite of unit tests for software with, say, n different features. Probably you will write 3-4 test cases per feature. This is perfectly manageable — it's a linear amount of work. But, we all know you will not find all of your bugs that way, because some bugs can only be triggered by a pair of features interacting. Now, you could go on to write test cases for every pair of features — but this is a quadratic amount of work, which is much less appealing.

Claessen and Hughes (2000) present *property tests* as an alternative to unit tests. Property tests can be thought of as templates for generating unit tests; by specifying exactly how input and output should be related, a property testing system can create randomly generated inputs and ensure that the property always holds. If we generate ten thousand random inputs, and the property holds for each one, we should be pretty confident that our code is working as intended. Of course, equality is undecidable in general, but after ten thousand tests, if the two haven't been shown yet to be unequal, they probably never will.

Property testing often requires a better understanding of the software under test, as its authors must be able to describe classes of correctness, rather than instances of correctness. However, this additional effort is well rewarded; property tests can be used to stamp out an arbitrary number of unit tests, driving our confidence of the system asymptotically up to 100%.

But in this section, we will take the automation ladder one rung higher, and *automatically generate our property tests.* How is such a thing possible? By giving a reference implementation and a description of the constructors of our algebra to QuickSpec (Smallbone et al. (2017)) — a theorem searching program — it can simply try every possible term, and use property testing to see which ones are equal.

By treating the matching terms found by QuickSpec as new equations of our algebra, we are, in essence, discovering property tests that must be true of any correct implementation. The result? An automatically generated suite of regression tests.

We will present just enough of QuickCheck and QuickSpec in this section to get our work done, but each is given a more robust treatment in chapter 7 and chapter 8, respectively.

Our first step is to write a *generator* for tiles, which is a way of creating random tiles. Generators run in the `Gen` monad provided by the `QuickCheck` library, and come with a few primitive actions for picking elements at random. Generators in `QuickCheck` are usually given via an `Arbitrary` instance, which defines an `arbitrary` function used to produce random values.

```
instance (CoArbitrary a, Arbitrary a)
      => Arbitrary (Tile a) where
  arbitrary = sized $ \n ->   . . . . . . . . . . . . . . .   ❶
    case n <= 1 of
      True -> pure <$> arbitrary   . . . . . . . . . . . .   ❷
      False -> frequency   . . . . . . . . . . . . . . . . .   ❸
        [ (3,) $ pure <$> arbitrary   . . . . . . . . . .   ❹
        , (9,) $ beside <$> decayArbitrary 2   . . . .   ❺
                        <*> decayArbitrary 2
        , (9,) $ above <$> decayArbitrary 2
                       <*> decayArbitrary 2
        , (2,) $ cw <$> arbitrary
        , (2,) $ ccw <$> arbitrary
        , (4,) $ flipV <$> arbitrary
        , (4,) $ flipH <$> arbitrary
        , (6,) $ swirl <$> decayArbitrary 4
        , (3,) $ quad <$> decayArbitrary 4
                      <*> decayArbitrary 4
                      <*> decayArbitrary 4
                      <*> decayArbitrary 4
        , (2,) $ (<*>)
              <$> decayArbitrary @(Tile (a -> a)) 2
              <*> decayArbitrary 2
        ]
```

```
decayArbitrary :: Arbitrary a => Int -> Gen a
decayArbitrary n = scale (`div` n) arbitrary
```

Every generator has access to an implicit size parameter provided by the testing engine, which roughly corresponds to how complicated the generated term should be. At ❶ we get access to the size parameter via the `sized` function. If the size is less than or equal to one, we simply return a `pure` tile whose color is itself `arbitrary`. This check ensures that our arbitrary tile eventually terminates.

At ❸ , we use the `frequency` combinator to assign random weights to the different possibilities of `Tile` constructors. In ❹ we also build an arbitrary `pure` colored tile, with a weight of 3. At ❺ however, we give a weight of 9 — meaning three times more likely — to the `beside` constructor, because it results in more complex tiles. Rather than filling in the parameters of `beside` with `arbitrary` tiles directly, we instead use the `decayArbitrary 2` combinator, which asks for a tile that is half as complicated as the one we are being asked to generate. In this way, we are "splitting our complexity budget" between the two sub-tiles. Due to polymorphism, the call to `arbitrary` at ❺ creates a `Tile a`, but at ❹ it creates just a bare `a`.

Notably missing from the definition of `arbitrary` are the `behind` and `empty` constructors. This is a technical limitation; those constructors require `a` to be a monoid, but nothing else in the algebra does. If we included them in the list, we'd only generate tiles of monoids, which is somewhat annoying for testing purposes. We console ourselves with the understanding that `empty` and `behind` are specializations of `pure` and `(<*>)` respectively, both of which get generated by this instance. In a real codebase, you'd probably want to do some trickery to allow both instances to exist, but such a thing is out of this book's scope.

The remainder of the instance carries on in this way, listing the constructors with relative weights, and building them out of `arbitrary` smaller pieces. There is nothing of real interest here; every generator you write should have the same shape, containing:

1. a check of the size parameter, terminating in a simple constructor if required.
2. a list of every constructor of the algebra, each lifted into the `Gen` monad, with arbitrary children.

It's **extremely important** that your generator use the algebra's constructors, and not the data constructors (that is, it should use `cw` instead of `Cw`.) Recall that our algebraic constructors possibly contain pattern matching that implements by-fiat laws; thus, building terms out of the data constructors directly is likely to break your invariants.

As it happens, this `Arbitrary` instance (and one for `Color`, elided here) is all that's required for us to generate random tiles — some samples of which are given in figures 5.1, 5.2, 5.3.

Our next course of action is to teach the testing program about what equality means. Recall that we are explicitly not using whatever notion of equality Haskell gives us, instead choosing to reason via **"obs eq"**, reproduced here:

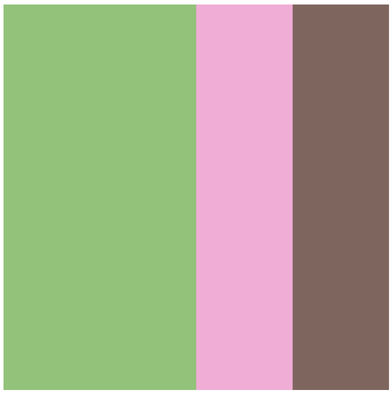
Figure 5.1: Random Tile 1

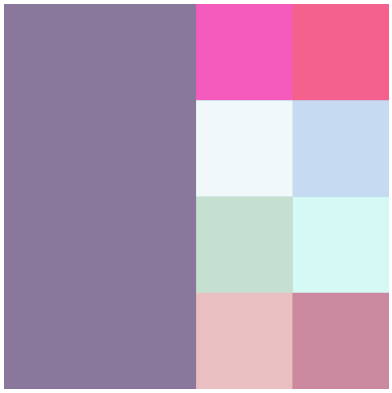
Figure 5.2: Random Tile 2

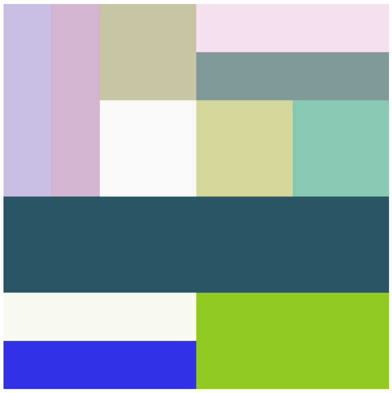

Figure 5.3: Random Tile 3

Law: "obs eq"

```
∀ (t1 :: Tile) (t2 :: Tile).
  (∀ (w :: Int) (h :: Int).
    rasterize w h t1 == rasterize w h t2) => t1 = t2
```

This law states that two tiles `t1` and `t2` are equal *if and only if* they rasterize to the same pixels for every imaginable width and height. Regardless of the actual memory layout for our tiles, this is the metric by which we'd like to consider two tiles equal. This notion can be encoded by giving an instance of the `Observe` typeclass from the `quickspec` library.

`Observe` allows us to describe observational equality between terms — possibly requiring quantified arguments. As an interface, it requires us to fill in one function, `observe`, which has a parameter for the quantified arguments, and another for the term we'd like to observe. These arguments are generated randomly via the property testing `Arbitrary` machinery, and two terms require observed equality for every input thrown at them.

We'd like to write an instance that looks like this:

```
instance Observe
            (Small Int, Small Int)  · · · · · · · · · ·   ❶
            [[a]]  · · · · · · · · · · · · · · · · · · ·   ❷
            (Tile a) · · · · · · · · · · · · · · · · · ·   ❸
            where
  observe (Small w, Small h) t
    = rasterize (max 1 w) (max 1 h) t   · · · · · · · ·   ❹
```

This says that given two small integers (❶) corresponding to the width and height, we can observe a `Tile a` value (❸) as a `[[a]]` (❷). We implement the `observe` function as `rasterize` at ❹ , ensuring that the width and height are both at least one.

While this instance is what we want conceptually, it doesn't work as written. The issue is that pesky universally quantified type `a`; we have no guarantees that it can be observed. The fix is a little mind-bending: we must introduce an `Observe` constraint for `a`, and observe our matrix of pixels in terms of the observation on `a`. The working instance is listed below, but it's important to keep in mind that this is an implementation detail, not a moral one.

```
instance Observe test outcome [[a]]
      => Observe
            (Small Int, Small Int, test)
            outcome
            (Tile a) where
  observe (Small w, Small h, x) t
    = observe x (rasterize (max 1 w) (max 1 h) t)
```

Our new `Observe` instance gives us access to the `(=~=)` operator, which creates a property test showing observational equality between two terms. For example, we can now test **"cw/cw/cw/cw"** experimentally:

```
> quickCheck $ cw @Bool . cw . cw . cw =~= id
+++ OK, passed 100 tests.
```

We need to specify `@Bool` to tell Haskell which sort of tiles to generate — left to its own devices it will pick the single-valued type `()` which is always equal to itself, and thus particularly unhelpful for testing equality. The `quickCheck` function then generated 100 different random tiles and checked to see that each was observationally equal to rotating the tile four times clockwise. In essence, it created one hundred unique unit tests, and our implementation passed each!

Compare the amount of code we wrote in this section to the amount of code we would have needed to write to create one hundred unit tests. When you take into account that the number one hundred is arbitrary and could just as easily have been ten thousand (use `quickCheckWith stdArgs {maxSuccess = 10000}` instead), perhaps the power of this approach becomes more apparent to you.

More amazingly, this is just the tip of the iceberg. While we certainly could go and write property tests for every law we discovered in chapter 2, it feels like a wasted effort. After all, every one should pass, since our implementation is derived from those same laws. Of course, just because they *should* pass doesn't mean they *will,* and we must still exercise prudence. But rather than write these tests ourselves, let's get the computer to do it for us.

Here is where the `quickspec` library comes in. We need only to write a *signature* of our algebra, describing what constructors and types are available, and it will do the rest. By enumerating every possible well-typed expression up to a maximum size and comparing them observationally to one another, QuickSpec will find *every law* that holds for our implementation. We can then

```
reassoc :: (a, (b, c)) -> ((a, b), c)
reassoc (x, (y, z)) = ((x, y), z)
```

Fundamentally, these laws govern what "combining two containers" means. Consider the two following applicative functors over lists:

```
unit :: [()]
unit = [()]

zap :: [a] -> [b] -> [(a, b)]
zap = cartesianProduct
```

and

```
unit :: [()]
unit = repeat ()   . . . . . . . . . . . . . . . . . . . . .   ❶

zap :: [a] -> [b] -> [(a, b)]
zap = zip   . . . . . . . . . . . . . . . . . . . . . . . . .   ❷
```

where the `repeat` function at ❶ creates an infinitely long list, and `zip` at ❷ combines two lists element-wise, truncating to whichever list is shorter.

Exercise Show that both `([()], cartesianProduct)` and `(repeat (), zip)` form applicative functors over lists.

In Haskell, applicative functors are usually expressed in terms of the equivalent operations `pure` and `(<*>)` — pronounced "ap" — given by:

```
pure :: a -> T a
pure a = fmap (const a) unit

(<*>) :: T (a -> b) -> T a -> T b
tf <*> ta = fmap (uncurry ($)) zap tf ta
```

This (<*>) operation enables a particular Haskell idiom, which lifts function application over "pure values" into function application over applicative functors — thus the name. Examples of this are scattered throughout chapter 4 and chapter 6, where we lift expressions of the form:

```
step i (both c1 c2) = both <$> step i c1 <*> step i c2
```

Back Matter

Acknowledgements

Writing a book is a serious investment of time and energy, and this one couldn't have happened without the support of many, many fantastic people. I want to thank everyone for their support, their patronage, and their enthusiasm. Some of the exceptionally instrumental people, however, require further accolades. In particular:

Reed Mullanix, for tirelessly and enthusiastically helping me work through the many, many mathematical difficulties I came across while researching this book.

Barry Moore, for spending so many of his weekend hours beta-testing every chapter and each of their revisions. I'm genuinely sorry for just how many "part twos" I put you through.

Jonathan Lorimer, for his unending enthusiasm and our many late-night strategy sessions trying to whip tricky chapters into shape.

Louisa Edelmann, for playfully teasing that writing only a single book doesn't really warrant calling oneself a writer. Without you, this project probably would have never happened.

Kenneth Bruskiewicz, for telling me what I needed to hear — even though I didn't want to listen.

Intellectually, this book stands on the shoulders of Conal Elliott and John Hughes, whose ideas will forever continue to inspire me.

Thanks to Nick Smallbone for all of his hard work on QuickSpec, and for being so patient with the deluge of issues and questions I sent his way.

Furthermore, this book wouldn't have been possible without the generous financial support of Bob Ippolito, codygman, Adam Conner-Sax, Gabriele Lana, Daniel Tebbutt, Jan Hrček, Vladimir Ciobanu, Samuel Evans-Powell, Edmund Cape, Pavlos Pantatakis, Fintan Halpenny, Jonathan Lorimer, Alexey Nesterov, Jakub Zarybnicky, Imre Gulyas, Andor Penzes, Shane Sveller, Reynaldo Cordero, Sam Raker, Andrei Orlov, Henry Laxen, Michael Alan Dorman, David Burkett, Michael Hood, Barry Moore, Markus Läll, Adnaan Mukadam, pluton, Daniel Gasienica, Andre Moelle, Adam Flott, Martin Allard, Patrick Weemeeuw, Joel McCracken, Ian, Chris Penner, Forest Walk, Dawid Furman, Angel Vanegas, Christian Lavoie, Alexander Granin, Csongor Kiss, and Sergey Kintsel.

Algebra-Driven Design uses icons made by Freepik, turkkub, and Becris, from www.flaticon.com. Additionally, the maps in chapter 4 are from Google. The cover uses elements from Greg Egan's Lissajous generator, from https://gregegan.net.

Thank you all most sincerely, from the very bottom of my heart.

Bibliography

10 Bjarnason, Runar. 2015. “Constraints Liberate, Liberties Constrain.” Scala World. https://www.youtube.com/watch?v=GqmsQeSzMdw.

Böhm, Corrado, and Alessandro Berarducci. 1985. “Automatic Synthesis of Typed λ-Programs on Term Algebras.” *Theoretical Computer Science* 39: 135–54. https://doi.org/10.1016/0304-3975(85)90135-5.

Claessen, Koen, and John Hughes. 2000. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.” In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming - ICFP ’00*, 268–79. Not Known: ACM Press. https://doi.org/10.1145/351240.351266.

———. 2002. “Testing Monadic Code with QuickCheck.” *ACM SIGPLAN Notices* 37 (12): 47. https://doi.org/10.1145/636517.636527.

Dijkstra, Edsger W. 1972. “The Humble Programmer.” *Communications of the ACM* 15 (10): 859–66. https://doi.org/10.1145/355604.361591.

———. 1982. “Why Is Software so Expensive?” In *Selected Writings on Computing*. Springer-Verlag. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD648.html.

Elliott, Conal. 2009. “Denotational Design with Type Class Morphisms (Extended Version).” 2009-01. LambdaPix. http://conal.net/papers/type-class-morphisms.