What I learned about functional programming

while writing a book about it

Sergei Winitzki

Scale by the Bay 2021

2021-10-28

Why I wrote a book about functional programming. I

My background: theoretical physics

- I used to write academic publications looking like this:

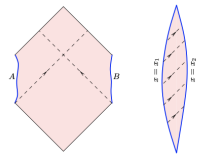


Figure 9: Left: The domain of Minkowski spacetime between two causally separated observers A, B moving with a constant proper acceleration in opposite directions. The dashed lines show that the two observers cannot see each other. Right: A closed Minkowski universe. The thick lines represent the identified boundaries $x = x_{1,2}$. A lightray (dashed line) crosses the line $x = x_{1,2}$ infinitely many times.

- An asymptotically flat spacetime with two stars collapsing into two black holes; the line of sight crosses the two star centers (Fig. 10).
- The future part of a de Sitter spacetime with a star collapsing into a black hole (Fig. 11).
- A maximally extended Schwarzschild-de Sitter spacetime (Fig. 12). It is interesting to note that this diagram is usually drawn with all Schwarzschild and de Sitter regions having the same size, which makes the figure unbounded (Fig. 12, top) despite the intention to represent the spacetime by a *finite* figure in the fiducial Minkowski plane. To adhere to the definition of a conformal diagram as a bounded figure, one can use a suitable conformal transformation reducing the diagram to a finite size (e.g. Fig. 12, bottom).

III. DIAGRAMS FOR ETERNALLY INFLATING SPACETIMES

I now apply the method of lightrays to spacetimes resulting from generic models of eternal inflation. In these models, inflation ends in a particular region of spacetime but continues in nearby regions. The process is analogous to the nucleation of non-inflating bubbles in a de Sitter spacetime. More precisely, the bubble interior inflates at a progressively slower rate until the inflation stops and the interior reheats. In the meantime, the bubble continues to expand, annexing new regions that stop inflating somewhat later. The bubble wall moves with a

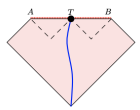


Figure 10: A spacetime with two collapsing stars. The point T is a future timelike infinity reached by an observer (thick curve) remaining between the two black holes. The lines AT and TB represent BH singularities and the dashed lines are the BH horizons.

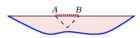


Figure 11: The future part of a de Sitter spacetime with a collapsing star. The line AB represents the BH singularity; the dashed line is the BH horizon.

constant acceleration and its worldline quickly becomes almost lightlike. As was noted in Ref. [11], the interior of the bubble appears to be an infinite FRW universe to an interior observer, even though the bubble occupies a finite interval $x_1 < x < x_2$ of the comoving volume in the background de Sitter spacetime (see Fig. 13). In particular, the hypersurfaces of equal temperature (identified as the equal-time surfaces by interior observers) have an infinite proper volume.

In an eternally inflating spacetime, all Hubble-sized re-

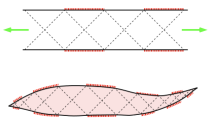


Figure 12: Diagrams for a maximally extended Schwarzschild-de Sitter spacetime. The conventional, unbounded diagram (top) and an equivalent diagram having a finite extent (bottom) are related by a conformal transformation. The thick dotted lines represent Schwarzschild singularities. Both diagrams contain infinitely many Schwarzschild and de Sitter regions.

integrals and shift the integration variable τ by δt:

$$\begin{aligned}
&\int_{\delta t}^{\infty} d\tau\, e^{\lambda\tau} P_+(\phi_0, \tau) \\
&= \int_{\delta t}^{\infty} d\tau\, e^{\lambda\tau} \int_{-\infty}^{b(\phi_0)} d\xi\, p(\xi)\, P_+(\phi_0 + \delta\phi, \tau - \delta t) \\
&= \int_{-\infty}^{b(\phi_0)} d\xi\, p(\xi) \int_0^{\infty} d\tau\, e^{\lambda(\tau+\delta\tau)}\, P_+(\phi_0 + \delta\phi, \tau) \\
&= \int_{-\infty}^{b(\phi_0)} d\xi\, p(\xi)\, e^{\lambda\delta t} f(\lambda; \phi_0 + \delta\phi).
\end{aligned} \tag{37}$$

Note that the top line in Eq. (37) is slightly different from the definition of $f(\lambda; \phi_0)$: The integration proceeds from $\tau = \delta t$ rather than from $\tau = 0$ in order to allow the subtraction $\tau - \delta t$ in the argument of P_+. The difference,

$$\int_0^{\delta t} d\tau\, P_+(\phi_0, \tau) \approx \delta t\, P_+(\phi_0, \delta t), \tag{38}$$

is negligible as long as ϕ_0 is at least a few e-foldings away from reheating. This is so because $P_+(\phi_0, \delta t)$ is equal to the (exponentially small) probability of jumping from $\phi = \phi_0$ directly to $\phi = \phi_*$ in one Hubble time δt. Therefore, we may replace the top line in Eq. (37) by $f(\lambda; \phi_0)$ and finally obtain the equation

$$f(\lambda; \phi_0) = \int_{-\infty}^{b(\phi_0)} d\xi\, p(\xi)\, e^{\lambda\delta t} f(\lambda; \phi_0 + \delta\phi(\phi_0, \xi)). \tag{39}$$

This is the basic equation describing the function $f(\lambda; \phi_0)$; as expected, it is nonlocal in ϕ.

It is not possible to approximate Eq. (39) by a diffusion equation (as is the normal procedure while deriving FP equations) because the integration in Eq. (39) proceeds over a ϕ-dependent range. Rather than trying to solve Eq. (39) directly, we will approximate the solution of Eq. (39) by an adiabatic ansatz [Eq. (46) below].

Up to now we have been using the proper time as the variable t. If a different time parameterization is desired, such as

$$\tilde{t} = \int^t A(\phi(t))dt, \tag{40}$$

where $A(\phi)$ is a known function, then the coefficients D, v, and δt must be modified as follows,

$$\tilde{v} = \frac{v}{A}, \quad \tilde{D} = \frac{D}{A}, \quad \delta\tilde{t} = A\delta t, \tag{41}$$

while the dimensionless coefficient $b(\phi)$ is unchanged. For instance, passing to the e-folding time

$$N = \int^t H dt \tag{42}$$

At the end of the calculation, we will only need to evaluate $f(\lambda = 0; \phi_0)$. As already mentioned above, $f(0; \phi_0)$ is the fraction of trajectories that never violate the NEC among all trajectories $\phi(t)$ starting at $\phi = \phi_0$. We note that for ϕ_0 in the fluctuation-dominated regime, the probability $f(0; \phi_0)$ rapidly decreases with growing ϕ_0 because there is a significant probability of violating the NEC at every Hubble time step at those ϕ_0. On the other hand, the probability of violating the NEC in the no-diffusion regime is exponentially small, and hence $f(0; \phi_0)$ is nearly constant and almost equal to 1 for ϕ_0 in that regime. Therefore, we expect that $f(\lambda; \phi_0)$ has exponentially strong dependence on ϕ_0. Moreover, $P_+(\phi_*, \tau) = \delta(\tau)$; this can be shown by considering

$$\int_0^{\infty} P_+(\phi_*, \tau) d\tau = 1, \tag{43}$$

$$P_+(\phi_*, \tau) = 0 \text{ for } \tau > 0, \tag{44}$$

which holds because trajectories starting with $\phi = \phi_*$ immediately reheat and have zero duration. Therefore

$$f(\lambda; \phi_*) = \int_0^{\infty} e^{\lambda\tau} P_+(\phi_*, \tau) d\tau = 1. \tag{45}$$

Motivated by these considerations, we represent the exponential behavior of $f(\lambda; \phi_0)$ and the boundary condition (45) by the ansatz

$$f(\lambda; \phi_0) = \exp\left[-\int_{\phi_*}^{\phi_0} W(\lambda; \phi) d\phi\right], \tag{46}$$

where $W(\lambda; \phi)$ is a new unknown function such that $W(0; \phi) > 0$. We then divide Eq. (39) through by $f(\lambda; \phi_0)$ and expand $f(\lambda; \phi + \delta\phi)$ to first order in $\delta\phi$:

$$\begin{aligned}
1 &= \int_{-\infty}^{b(\phi_0)} d\xi\, p(\xi)\, e^{\lambda\delta t} \exp\left[-\int_{\phi_0}^{\phi_0+\delta\phi} W(\lambda; \phi) d\phi\right] \\
&\approx \int_{-\infty}^{b(\phi_0)} d\xi\, p(\xi)\, e^{\lambda\delta t} \exp\left[-W(\lambda; \phi_0)\delta\phi(\phi_0, \xi)\right] \\
&= e^{(\lambda - vW)dt} \int_{-\infty}^{b} \frac{d\xi}{\sqrt{2\pi}} \exp\left[-\frac{\xi^2}{2} - \sqrt{2D\delta t} W \xi\right] \\
&= e^{(\lambda - vW + DW^2)\delta t} \left[\frac{1}{2} + \frac{1}{2}\operatorname{erf}\left(\frac{b + W\sqrt{2D\delta t}}{\sqrt{2}}\right)\right],
\end{aligned} \tag{47}$$

where we suppressed the argument ϕ_0 in the last line, $b \equiv b(\phi)$ was defined in Eq. (35), while $\operatorname{erf} x$ is the standard error function

$$\operatorname{erf} x \equiv \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt. \tag{48}$$

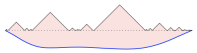


Figure 4.3: Conformal diagram of eternally inflating spacetime showing the future of an initial spacelike Cauchy hypersurface (bottom curve). There is a fractal arrangement of eternal points; regions between eternal points are reheated and are assumed to become asymptotically flat in the future.

criterion for the presence of eternal inflation. In this section we will give an introduction to the methods developed by the author for the analysis of this issue.

Our focus is the probability $X(\phi)$ of having at least one eternal point in an initial Hubble-size region where the field has value ϕ. As we will now show, this probability satisfies the equation[7]

$$\hat{L}_\phi X - 3H(1 - X)\ln(1 - X) = 0 \tag{4.121}$$

with zero boundary conditions, $X(\phi_*) = 0$ at the reheating boundary and $X(\phi_P = 0)$ at the Planck boundary (if any). Here

$$\hat{L}_\phi \equiv D(\phi)\partial_{\phi\phi} + v(\phi)\partial_\phi \tag{4.122}$$

is the Fokker–Planck operator used in the *backward* FP equation. Note that Eq. (4.121) is nonlinear due to the presence of the logarithmic term; this equation is of reaction-diffusion type.

We remark that $X(\phi) \equiv 0$ is trivially a solution of Eq. (4.121). However, this solution is not physically relevant if eternal inflation is present, since in that case the probability of having eternal points is certainly nonzero. As will be demonstrated below, the presence of eternal inflation is *equivalent* to the existence of a nontrivial solution $X(\phi) \not\equiv 0$ of Eq. (4.121). We will begin by deriving that equation and showing its independence of time parametrization.

It will be convenient to work with the complementary probability $\bar{X}(\phi) = 1 - X(\phi)$ and to use the proper time t as the time variable. We recall that the random walk of the inflaton field ϕ is described by a Langevin equation which can be written as a difference equation (4.7),

$$\phi(t + \Delta t) = \phi(t) + v(\phi)\Delta t + \xi\sqrt{2D(\phi)\Delta t}. \tag{4.123}$$

The evolution of ϕ in a horizon-size domain with field value $\phi = \phi_0$ at time $t = 0$ gives rise to independent random walks of ϕ in each of the "daughter" horizon-size subdomains that were formed out of the original domain. After

[7] This equation was first derived in [188]. A similar equation was recently rederived using a different approach in [208].

- Repented and turned to software engineering in 2010

I have been studying FP since 2008 (OCaml, Haskell, Scala)

- Learning from papers, online tutorials, and books
- Attending the SBTB conference since 2014
- Using Scala at my day job since 2015

Why I wrote a book about functional programming. II

I found the FP community to be unlike other programmers' communities

- Others are focused on a chosen programming language (Java, Python, JavaScript, etc.), and on designing and using libraries and frameworks
  - *"setup this YAML config, override this method, use this annotation"*
- The FP community talks in a very different way
  - *"referential transparency, algebraic data types, monoid laws, parametric polymorphism, free applicative functors, monad transformers, Yoneda lemma, Curry-Howard isomorphism, profunctor lenses, catamorphisms"*
    - ★ A glossary of FP terminology (more than 100 terms)
  - From SBTB 2018: *The Functor, Applicative, Monad talk*
    - ★ By 2018, everyone expects to hear these concepts mentioned
  - An actual Scala error message:

```
found   : Seq[Some[V]]
required: Seq[Option[?V8]] where type ?V8 <: V (this is a GADT skolem)
```

Why I wrote a book about functional programming. III

Main questions:

- Which theoretical knowledge will actually help write Scala code?
- Where can one learn about this, with definitions and examples?

What I did *not* want to see:

- Theory for the sake of theory, with no applications
  - "Monad is just a monoid in the category of endofunctors"
  - Lawvere theories as an alternative to monads
  - *The Book of Monads*: "monads from adjunctions" are never used
- Heuristic explanations without proofs
  - Most FP books have no proofs and few rigorous definitions
  - A couple of books (*Introduction to functional programming using Haskell* and *Functional programming in Scala*) include only a few simplest proofs
  - Even *The Book of Monads* does not prove the laws for any monads!

Why I wrote a book about functional programming. IV

Reading various materials has given me more questions than answers

- Monads
  - P. Wadler, "*Monads for functional programming*" (1995)

> Often, monads are defined not in terms of *unit* and $\star$, but rather in terms of *unit*, *join*, and *map* [10, 13]. The three monad laws are replaced by the first seven of the eight laws above. If one defines $\star$ by the eighth law, then the three monad laws follow. Hence the two definitions are equivalent.

  - Wikipedia: *In functional programming, a monad is an abstraction that allows structuring programs generically. ... Category theory also provides a few formal requirements, known as the monad laws, which should be satisfied by any monad and can be used to verify monadic code.*

Cannot understand this

Why I wrote a book about functional programming. V

Reading various materials has given me more questions than answers

- Applicative functors
  - P. Chuisano and R. Bjarnason, *Functional programming in Scala*

EXERCISE 12.2

Hard: The name *applicative* comes from the fact that we can formulate the `Applicative` interface using an alternate set of primitives, `unit` and the function `apply`, rather than `unit` and `map2`. Show that this formulation is equivalent in expressiveness by defining `map2` and `map` in terms of `unit` and `apply`. Also establish that `apply` can be implemented in terms of `map2` and `unit`.

```
trait Applicative[F[_]] extends Functor[F] {

  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B]
  def unit[A](a: => A): F[A]

  def map[A,B](fa: F[A])(f: A => B): F[B]
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[A]
}
```

Define in terms of **map2** and **unit**.

Define in terms of **apply** and **unit**.

  - Wikipedia: *In functional programming, an applicative functor ... is an intermediate structure between functors and monads. ... Applicative functors are the programming equivalent of lax monoidal functors with tensorial strength in category theory.*

Cannot understand this

Why I wrote a book about functional programming. VI

Reading various materials has given me more questions than answers

- Free monads
  - *Functional programming in Scala*: *The* `Return` *and* `FlatMap` *constructors witness that this data type is a monad for any choice of* `F`*, and since they're exactly the operations required to generate a monad, we say that it's a free monad.*
  - Wikipedia: *Sometimes, the general outline of a monad may be useful, but no simple pattern recommends one monad or another. This is where a free monad comes in; as a free object in the category of monads, it can represent monadic structure without any specific constraints beyond the monad laws themselves. ... For example, by working entirely through the Just and Nothing markers, the Maybe monad is in fact a free monad.*

  Cannot understand this

Why I wrote a book about functional programming. VII

Most resources for modern FP are either too academic or too limited to questions of practical usage

- But see "Haskell Wikibooks" and "Functional Programming in Scala"

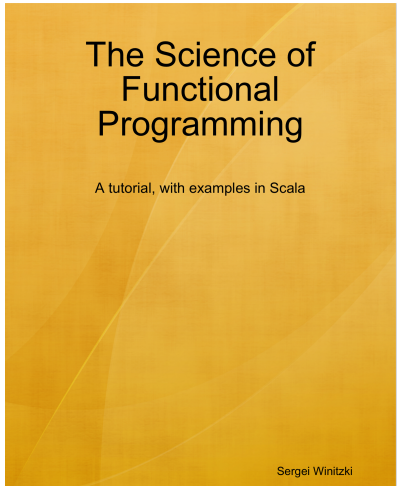
After several years of study, I found *systematic* ways of:

- finding the practice-relevant parts of FP theory
- organizing the required knowledge
- verifying theoretical statements through mathematical derivations

Then I started writing a new book to answer all my FP questions

The book explains (with code examples and exercises):

- theory and applications of major design patterns of FP
- techniques for deriving and verifying properties of types and code (typeclass laws, equivalence of types)
- practical motivations for (and applications of) these techniques



What I learned. I. Questions that have rigorous answers

In FP, a programmer encounters certain questions about code that can be answered rigorously

- The answers are *not* a matter of opinion or experience
- The answers are found via mathematical derivations and reasoning
- The answers will guide the programmer in designing the code

Examples of reasoning tasks. I

1. Are the types `Either[Z, R => A]` and `R => Either[Z, A]` equivalent? Can we compute a value of type `Either[Z, R => A]` given a value of type `R => Either[Z, A]` and conversely? (`A`, `R`, `Z` are type parameters.)

```
def f[Z, R, A](r: R => Either[Z, A]): Either[Z, R => A] = ???
def g[Z, R, A](e: Either[Z, R => A]): R => Either[Z, A] = ???
```

- We can implement `g` and there is only one way:

```
def g[Z, R, A](e: Either[Z, R => A]): R => Either[Z, A] =
  r => e.map(f => f(r))       // Scala 2.12
```

- It turns out that `f` *cannot* be implemented
- Programmers need to develop intuition about why this is so
- These results are rigorous (programmers do not need to write tests)
  - The Curry-Howard isomorphism and the LJT algorithm

Examples of reasoning tasks. II

2. How to use `for` / `yield` with `Either[Z, A]` and `Future[A]` together?

```
val result = for { // This code will not compile; need to combine...
  a <- Future(...) //  ... a computation that is run asynchronously,
  b <- Either(...) // a computation whose result may be unavailable,
  c <- Future(...) // and another asynchronous computation.
} yield ???     // Continue computations when results are available.
```

   Should `result` have type `Either[Z, Future[A]]` or `Future[Either[Z,A]]`?
   How to combine `Either` with `Future` so that we can use `flatMap`?

- It turns out that `Either[Z, Future[A]]` is wrong (cannot implement `flatMap` correctly). The correct solution is `Future[Either[Z, A]]`.
- Programmers need to develop intuition about why this is so
- This is a rigorous result (programmers do not need to test it)
  - The theory of monad transformers and their laws

Examples of reasoning tasks. III

3. Can we implement `flatMap` for the type constructor `Option[(A, A, A)]`?

```
def flatMap[A, B](fa: Option[(A, A, A)])(f: A => Option[(B, B, B)])
    : Option[(B, B, B)] = ???
```

- It turns out that `flatMap` *can* be implemented but fails the monad laws
- Programmers need to develop intuition about why this is so
  - How should we modify `Option[(A, A, A)]` to make it into a monad?
- This is a rigorous result (programmers do not need to test it)
  - The theory of monads and their laws

Examples of reasoning tasks. IV

4. Different people define a "free monad" via different sets of case classes. Are these definitions equivalent? What is the difference?
   - Three different implementations of the free monad: a blog post by Gabriel Gonzalez (2012), a talk given by Rúnar Bjarnason (2014), and a talk given by Kelley Robinson (2016) — but no rigorous definitions

- The free monad on a functor is less code than the free monad on a non-functor
- The free monad's encoding that assumes the monad laws is less code than an encoding without assumed laws
- These are rigorous results (programmers do not need to test them)
  - The theory of "free" inductive typeclasses and their encodings
- Programmers need to get intuition about implementing free monads
  - How to define a free monad on a `Pointed` functor (a functor that already has the `pure` method)?

What I learned. I. Questions that have rigorous answers

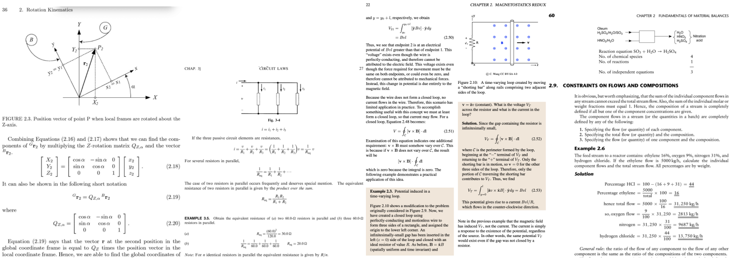
- FP allows programmers to ask and rigorously answer questions like these
  - These questions are relevant to writing code
  - These questions are not about the program's business logic or domain
- In this aspect of the programmer's work, it is *engineering*
  - Writing code via experience and best practices is *artisanship*
  - Code for scientific computing, data science, or aerospace control is usually artisanal — even though the corresponding business logic is mathematically rigorous

What I learned. II. Software engineers and software artisans

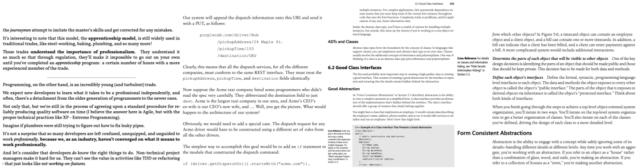
- FP is similar to engineering in a number of ways
  - Mechanical, electrical, chemical engineering are based on calculus, classical and quantum mechanics, electrodynamics, thermodynamics
  - FP is based on category theory, type theory, logic proof theory
- Engineers use special terminology
  - Examples from mechanical, electrical, chemical engineering: rank-4 tensors, Lagrangians with non-holonomic constraints, Fourier transform of the delta function, inverse Z-transform, Gibbs free energy
  - Examples from FP: rank-N types, continuation-passing transformation, polymorphic lambda functions, free monads, hylomorphisms
- As in engineering, the special terminology in FP is *not* self-explanatory
  - A "lambda function"?
  - A "free monad"?

Functional programming looks like engineering, while other approaches to software resemble artisanship

Books on mechanical, electrical, chemical engineering design:



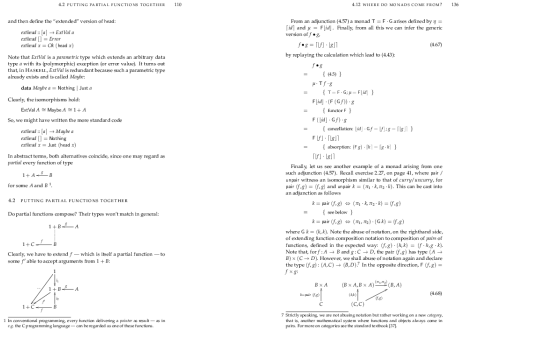
Books on software design and architecture:



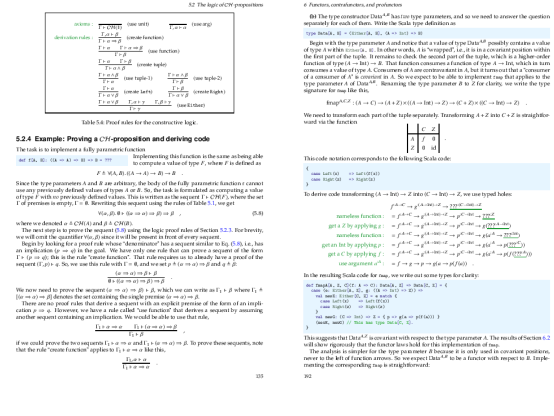
What I learned. II. Software engineers and software artisans

There are currently two books being written on the applied science of FP

Sample pages from *Program Design by Calculation*



Sample pages from *The Science of Functional Programming*



What I learned. III. The science of `map` / `filter` / `reduce`

The `map` / `filter` / `reduce` programming style — iteration without loops

- Compute the list of all integers n between 1 and 100 that can be expressed as $n = p * q$ (with $2 \leq p \leq q$) in exactly 4 different ways

```
scala> (1 to 100).filter { n =>
     |   (2 to n).count(x => n % x == 0 && x * x <= n) == 4
     | }
res0: IndexedSeq[Int] = Vector(36, 48, 80, 100)
```

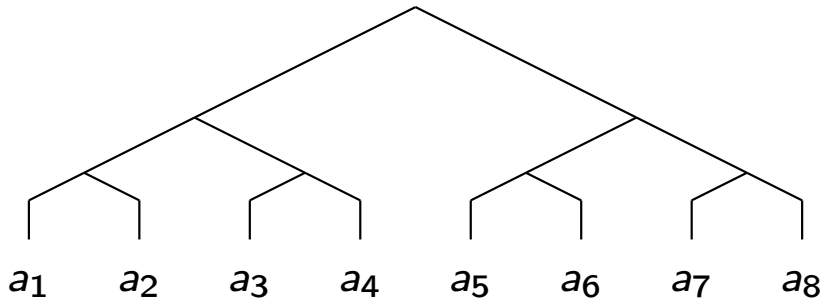
The `map` / `filter` / `reduce` programming style is an FP success story

- Nameless functions ("lambdas", "closures") are widely used
  - and have been added to most programming languages by now
- Essential methods: `map`, `filter`, `flatMap`, `zip`, `fold`
- Similar techniques work with parallel and stream processing (`Spark`)
- Similar techniques work with relational databases (`Slick`)

What I learned. III. The science of `map` / `filter` / `reduce`

Essential methods: `map`, `filter`, `flatMap`, `zip`, `fold`

- What data types other than `Seq[A]` can support these methods?
  - Algebraic data types?
  - Trees and other recursive types?
  - Perfect-shaped trees?



  - Which methods can be defined for `MyData[A]`?

```
type MyData[A] = String => Option[(String, A)]
```

What I learned. III. The science of `map` / `filter` / `reduce`

A systematic approach to understanding FP via `map` / `filter` / `reduce`:

- Determine the required laws of `map`, `filter`, `flatMap`, `zip`, `fold`
  - The laws express the programmers' expectations about code behavior
  - Define the corresponding typeclasses
    - ★ `map` — `Functor`, `filter` — `Filterable`, `flatMap` — `Monad`, `zip` — `Applicative`, `fold` — `Traversable`
- Find all type constructions that preserve the typeclass laws
  - If `P[A]` and `Q[A]` are filterable functors then so is `Either[P[A], Q[A]]`
  - If `P[A]` is a contravariant functor then `P[A] => A` is a monad
  - If `P[A]` is a monad then so is `Either[A, P[A]]`
  - If `P[A]` and `Q[A]` are applicative then so is `Either[P[A], (A, Q[A])]`
    - ★ I found many more type constructions of that sort
  - Sometimes it becomes necessary to define additional typeclasses
    - ★ Contravariant functor, contravariant filterable, contravariant applicative
- Develop intuition about implementing lawful typeclass methods
- Develop intuition about data types that can have those methods
  - ... and about data types that *cannot* (and reasons why)
- Develop notation and proof techniques for proving the laws

What I learned. IV. The logic of types

FP is not just "programming with functions": types play a central role
Most of FP use cases are based on only six type constructions:

- Unit type — `Unit`
- Type parameters — `[A]`
- Product types — `(A, B)`
- Co-product types ("disjunctive union" types) — `Either[A, B]`
- Function types — `A => B`
- Recursive types — `Fix[A, S]` where `S[_, _]` is a "recursion scheme"

```
final case class Fix[A, S[_, _]](unfix: S[A, Fix[A, S]])
```

Going through all possible type combinations, we can enumerate essentially all possible typeclass instances

- all possible functors, filterables, monads, applicatives, traversables, etc.
- in some cases, we can generate typeclass instances automatically

What I learned. IV. The logic of types

- Unit, product, co-product, and function types correspond to logical propositions `(true)`, `(A and B)`, `(A or B)`, `(if A then B)`
- Not all programming languages support all of these type constructions
  - The logic of types is *incomplete* in those languages
- Languages that do not support co-products will make you suffer

```
fileOpened, err := os.Open("filename.txt")     // go-lang has you
if err != nil { log.Fatal(err) }  // doomed to write this forever
```

- Returning a pair (both a result and an error) instead of a disjunction (either a result or an error) promotes many ways of making hard-to-find mistakes
  - In Scala, just return `Either[Error, Result]`

What I learned. V. Miscellaneous surprises

My approach forced me to formulate and prove every statement
Each chapter gave me at least one surprise

- What I believed and tried to prove turned out to be incorrect
- What seemed to be intuitively unexpected turned out to be true

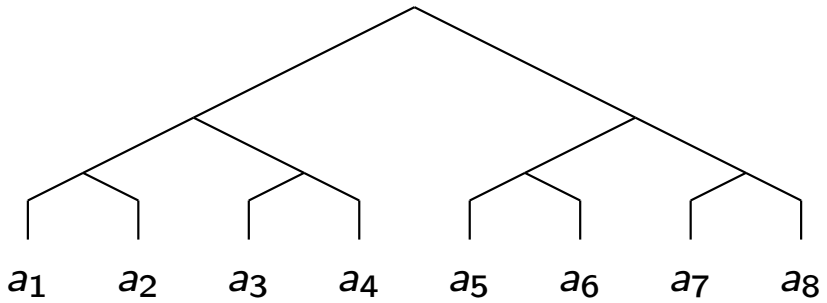
What I learned. V. Miscellaneous surprises

Chapters 1 to 3:

- Nameless functions are used in mathematics too, just hidden

$\sum_{n=1}^{100} n^2$	(1 to 100).map { n => n * n }.sum
$\int_0^1 \sin(x^3)\, dx$	integrateNumerically(0, 1) { x => math.sin(x * x * x) }

- Variables, shadowing, and lexical scoping are the same in math usage
- Many algorithms require non-tail-recursive code (map for a tree)
- Perfect-shaped trees *can* be defined via recursive ADTs



What I learned. V. Miscellaneous surprises

Chapters 4 and 5: a practical application of the Curry-Howard isomorphism

- "Type inference" — determining type signature from given code
- "Code inference" — determining code from given type signature
- The `curryhoward` library uses the LJT algorithm for code inference

```
import io.chymyst.ch._

scala> def in[A, B](a: A, b: Option[B]): Option[(A, B)] = implement
def in[A, B](a: A, b: Option[B]): Option[(A, B)]

scala> in(1.5, Some(true))
val res0: Option[(Double, Boolean)] = Some((1.5,true))

scala> def h[A, B]: ((((A => B) => A) => A) => B) => B  =  implement
def h[A, B]: ((((A => B) => A) => A) => B) => B

scala> println(h.lambdaTerm.prettyPrint)
a ⇒ a (b ⇒ b (c ⇒ a (d ⇒ c)))

scala> def g[A, B]: ((((A => B) => B) => A) => B) => B  =  implement
error: type ((((A ⇒ B) ⇒ B) ⇒ A) ⇒ B) ⇒ B cannot be implemented
```

What I learned. V. Miscellaneous surprises

Chapters 6 to 8:

- Subtypes / supertypes are not always the same as supersets / subsets
- Functions of type ADT => ADT can be manipulated via matrices
  - Matrix code notation is useful in symbolic proofs

```
val p: Either[A, B] => Either[C, D] = {
    case Left(x)   => Right(f(x))
    case Right(y)  => Left(g(y))
}
```

	C	D
A	$\mathbb{0}$	$x \to f(x)$
B	$y \to g(y)$	$\mathbb{0}$

- Typeclasses can be viewed as partial functions from types to values
- All non-parameterized types have a monoid structure

What I learned. V. Miscellaneous surprises

Chapters 9 to 12:

- "Filterable functors" are a neglected typeclass with useful properties
- Data types `Option[(A, A)]`, `Option[(A, A, A)]`, etc., *cannot* be monads
- Monads need "runners" to be useful, but some monads' runners do not obey the laws or cannot exist (`State`, `Continuation`)
- Without some laws, `flatMap` is *not* equivalent to `map` with `flatten`
  - It is not enough to write `_.flatten == _.flatMap(identity)` and `_.flatMap(f) == _.map(f).flatten`, we need to prove an isomorphism
- Almost all monads are non-commutative, but almost all applicative functors are commutative
- All contravariant functors are applicative (if defined using the six standard type constructions)
- Breadth-first traversal of trees *can* be defined via `fold` and `traverse` (not only depth-first traversal)

What I learned. V. Miscellaneous surprises

Chapter 13 (free typeclass constructions):

- Not all typeclasses have a "free" construction: there is free functor, filterable, applicative, etc.,; but no free foldable or free traversable
- *"Tagless final" is just a Church encoding of the free monad, what is the problem?*
- A complete proof of the correctness of the Church encoding is *hard*
  - My book uses relational parametricity together with some results from unpublished talk slides to prove that the Church encoding works
  - ... but programmers do not need to study those proofs

What I learned. V. Miscellaneous surprises

Chapter 14 (monad transformers):

- Monad transformers likely exist for all explicitly definable monads, but there is no general method or scheme for defining the transformers
- *Monad transformers are just pointed endofunctors in the category of monads, what is the problem?*
- Some monad transformers are incomplete and unusable for practical coding (`Continuation`, `Codensity`)

Conclusions

- Functional programming has a steep learning curve
  - Programmers can already benefit from the simplest techniques
    - ★ ... and mostly stop there (`map` / `filter` / `fold`, ADTs, `for` / `yield`)
  - Full *ab initio* derivations and proofs take 800 pages
  - The difficulty is at the level of undergraduate calculus / algebra
- Much of the theory is directly beneficial for coding
- Using FP techniques makes programmers' work closer to *engineering*
- Full details in the free book — https://github.com/winitzki/sofp