

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:

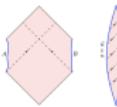


Figure 6: Left: The domain of differentiable spacetime between two causally connected states, A, B meeting at a contact point. Right: Two causally connected states, A and B meeting at a contact point such that the two observers meet each other. Right: A dashed line separates them, and a shaded region between them is labeled 'B'. A light-grey shaded line region for the BB horizon.



A spacelike flat spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.

$$\begin{aligned} & \text{integrate and shift the integration variable } \tau \text{ by } \delta t \\ & \int_{\tau_0}^{\tau_0 + \delta t} d\tau e^{i\omega \tau} P_n(\phi_\tau, \tau) \\ &= \int_{\tau_0}^{\tau_0 + \delta t} d\tau e^{i\omega \tau} \int_{\tau_0}^{\tau_0 + \delta t} d\tau' P_n(\phi_\tau + \delta t, \tau - \tau') \\ &= \int_{\tau_0}^{\tau_0 + \delta t} d\tau P_n(\phi_\tau) \int_{\tau_0}^{\tau_0 + \delta t} d\tau' e^{i\omega (\tau - \tau')} P_n(\phi_\tau + \delta t, \tau') \\ &= \int_{\tau_0}^{\tau_0 + \delta t} d\tau P_n(\phi_\tau) e^{i\omega \delta t} P_n(\phi_\tau + \delta t, \tau). \end{aligned} \quad (27)$$

Note that the way we split up Eq. (26) is slightly different from the definition of $J(\phi_\tau)$. The integration proceeds from $\tau = \tau_0$ to $\tau = \tau_0 + \delta t$ because that is in order to allow the subtraction of δt in the argument of P_n . The following,

$$\int_{\tau_0}^{\tau_0 + \delta t} d\tau P_n(\phi_\tau) e^{i\omega \delta t} P_n(\phi_\tau + \delta t, \tau), \quad (28)$$

is negligible as long as δt is at least a few Planck times. The probability of having at least one causal violation of the NEC owing to δt is roughly equal to the probability of jumping from $\tau = \tau_0$ directly to $\tau = \tau_0 + \delta t$, in one Hubble time. It follows, $P_n(\phi_\tau) \approx J(\phi_\tau)$, so we can drop Eq. (28) and finally obtain the equation

$$J(\phi_\tau) \phi_\tau = \int_{\tau_0}^{\tau_0 + \delta t} d\tau P_n(\phi_\tau) e^{i\omega \delta t} P_n(\phi_\tau + \delta t, \tau). \quad (29)$$

This is the first equation describing the function $J(\phi_\tau)$ as it is expected to be contained in the theory.

It is not possible to approximate Eq. (29) by a different equation, because the integration in Eq. (29) goes from τ_0 to $\tau_0 + \delta t$ and it is not possible to find a function $J(\phi_\tau)$ which satisfies Eq. (29) as an ordinary integral equation. To make things worse, we will apply the same trick again to the variable τ . If a different time parameterization is chosen, so that

$$\tilde{\tau} = \int_{\tau_0}^\tau dt J(t), \quad (40)$$

where $J(t)$ is a known function, then the coefficients D_i , v_i must be modified as follows,

$$i = \frac{D}{v}, \quad \tilde{D} = \frac{D}{v}, \quad R = Adt, \quad (41)$$

while the deformation coefficient $K(t)$ is modified. Furthermore, passing to the $\tilde{\tau}$ -folking time

$$N = \int_{\tau_0}^\tau M dt, \quad (42)$$

we are led to Eq. (35), while we let the standard

At the end of the calculation, we will only need to evaluate $J(\phi = 0, \phi_0)$. As already mentioned above, $J(\phi_\tau)$ is the probability of having at least one causal violation of the NEC owing to a separation δt starting at $\phi = \phi_0$. The probability of having a causal violation of the NEC is rapidly decreasing with growing δt , because there is a significant probability of violating the NEC in the first place. On the other hand, the probability of violating the NID is very small. In fact, $J(\phi_\tau)$ is nearly constant and almost equal to 1 for δt in that regime. Therefore, we expect that $J(\phi_\tau)$ is approximately constant and equals unity. We also note that $P_n(\phi_\tau) \approx J(\phi_\tau)$ because we are summing

$$\int_{\tau_0}^{\tau_0 + \delta t} d\tau P_n(\phi_\tau) \neq 1, \quad (43)$$

while both become non-zero starting with $\delta t = \delta t_0$ immediately below and have zero otherwise. Therefore

$$\int_{\tau_0}^{\tau_0 + \delta t} d\tau P_n(\phi_\tau) \phi_\tau = 1. \quad (44)$$

Motivated by these considerations, we represent the expectation value of $J(\phi_\tau)$ as $\langle J(\phi_\tau) \rangle$ and the mean value of $J(\phi_\tau)$ as $\langle J(\phi_\tau) \rangle_{\text{mean}}$.

$$\langle J(\phi_\tau) \rangle = \exp \left[- \int_{\tau_0}^{\tau_0 + \delta t} W(\phi_\tau, \tau) d\tau \right], \quad (45)$$

where $W(\phi_\tau, \tau)$ is a new unknown function such that $\langle J(\phi_\tau) \rangle = \exp(-W(\phi_\tau, \tau))$ holds through δt terms. $\langle J(\phi_\tau) \rangle_{\text{mean}}$ and $\langle J(\phi_\tau) \rangle$ are related by the same formula

$$\begin{aligned} & \int_{\tau_0}^{\tau_0 + \delta t} d\tau \langle J(\phi_\tau) \rangle \exp \left[- \int_{\tau_0}^{\tau_0 + \delta t} W(\phi_\tau, \tau) d\tau \right] \\ &= \int_{\tau_0}^{\tau_0 + \delta t} d\tau \langle J(\phi_\tau) \rangle \exp \left[-W(\phi_\tau, \tau) \right]. \end{aligned} \quad (46)$$

$$\begin{aligned} &= e^{1 - \delta t^2/2} \int_{\tau_0}^{\tau_0 + \delta t} d\tau \exp \left[-\frac{C}{2} - \frac{\delta t^2}{2} \ln W(\phi_\tau, \tau) \right] \\ &= e^{1 - \delta t^2/2} \int_{\tau_0}^{\tau_0 + \delta t} d\tau \exp \left[\frac{1}{2} \left(1 - \frac{\delta t^2}{2} \frac{1 + \sqrt{1 + 2/W(\phi_\tau, \tau)}}{\sqrt{W(\phi_\tau, \tau)}} \right) \right]. \end{aligned}$$

The evolution of ϕ is a function of time t with field value $\phi = -dt/t$ at the Planck boundary and $\phi = 0$ at the Hubble time. It will be shown that $W(\phi_\tau, \tau) = \langle J(\phi_\tau) \rangle$ is a function of time t and the variable τ and that the random walk of the inflaton field is described by a Langevin equation which can be written in the form

$$\dot{\phi} + M\phi = \eta(t) + \eta(\phi) dt + \sqrt{2D(\phi)} d\eta, \quad (47)$$

The evolution of ϕ is a function of time t with field value $\phi = -dt/t$ at the Planck boundary and $\phi = 0$ at the Hubble time. It will be shown that $W(\phi_\tau, \tau) = \langle J(\phi_\tau) \rangle$ is a function of time t and the variable τ and that the random walk of the inflaton field is described by a Langevin equation which can be written in the form

$$\dot{\phi} + M\phi = \eta(t) + \eta(\phi) dt + \sqrt{2D(\phi)} d\eta, \quad (48)$$

where we suppressed the arguments t , in the last line, it was defined to be Eq. (35), while we let the standard

Atmosphere 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 $X(t)$ of having at least one eternal point in an initial Hubble-like region when the field has value ϕ . As we will now show, this probability satisfies the equation¹

$$L_X - M(t)X(t) - (1-X)(t) = 0, \quad (49)$$

with zero boundary conditions, $X(t) = 0$ at the rehurbing boundary and $X(t) = 1$ at the Planck boundary (if any). There

$$L_X = D(\partial_t \phi + v) \partial_t \phi, \quad (50)$$

is the Yukawa-Floris operator in the local field ϕ variables. Note that Eq. (49) is nonlinear due to the fact that the logarithmic term $(1-X)$ appears.

We remark that $X(t)$ is not necessarily a solution of Eq. (49). However, this solution is unique if the condition for eternal inflation is present, since in that case the probability of having eternal points is certainly constant. As will be demonstrated, the probability $X(t)$ of having at least one eternal point in an initial Hubble-like region is a solution of Eq. (49).

We will begin by deriving that $X(t)$ is a solution of Eq. (49) independently of time parameterization. We recall that the random walk of the inflaton field is described by a Langevin equation which can be written in the form

$$\dot{\phi} + M\phi = \eta(t) + \eta(\phi) dt + \sqrt{2D(\phi)} d\eta, \quad (51)$$

The evolution of ϕ is a function of time t with field value $\phi = -dt/t$ at the Planck boundary and $\phi = 0$ at the Hubble time. It will be shown that $X(t)$ is a function of time t and the variable τ and that the random walk of the inflaton field is described by a Langevin equation which can be written in the form

$$\dot{\phi} + M\phi = \eta(t) + \eta(\phi) dt + \sqrt{2D(\phi)} d\eta, \quad (52)$$

where we suppressed the arguments t , in the last line, it was defined to be Eq. (35), while we let the standard

¹This equation was first derived in [10]. A similar equation was recently derived using a different approach in [10].



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.



A causally extended Schwarzschild-de Sitter spacetime with two causally connected regions meeting at a contact point. The dashed line is the BB horizon.

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
 - ▶ 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. Chiusano and R. Bjarnason, *Functional programming in Scala*

 EXERCISE 12.2

Hand: 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[C] =  
        fa flatMap(a => fb map(b => f(a, b)))  
}
```

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.*
 - ▶ 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

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](#)”

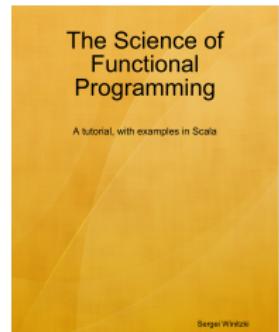
After several years of study, I found *systematic* ways of:

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

Then I wrote 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 help the programmer in designing the code

Examples of reasoning tasks. I

- ➊ 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] = ???
```

- It turns out that `f` *cannot* be implemented, while `g` can be:

```
def g[Z, R, A](e: Either[Z, R => A]): R => Either[Z, A] =  
    r => e.map(f => f(r))           // Scala 2.12
```
- 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; we want to have...
    a <- Future(...) // ... a computation that is run asynchronously,
    b <- Either(...) // a computation whose result may be unavailable,
    c <- Future(...) // a computation that is run asynchronously.
} 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

- ⑤ Different people define a “free monad” via different sets of case classes. Are these definitions equivalent? What is the difference?
- ⑥ How to define a free monad on a `Pointed` functor (i.e., when the functor already has the `pure` method)?
- 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
- Programmers need to develop intuition about how to implement them
- This is a rigorous result (programmers do not need to test it)
 - ▶ The theory of “free” inductive typeclasses and their encodings

What I learned. I. Questions that have rigorous answers

- FP allows programmers to ask and rigorously answer questions like these *while writing code*
- In this aspect of the programmer's work, it is *engineering*

What I learned. II. Software engineers and software artisans

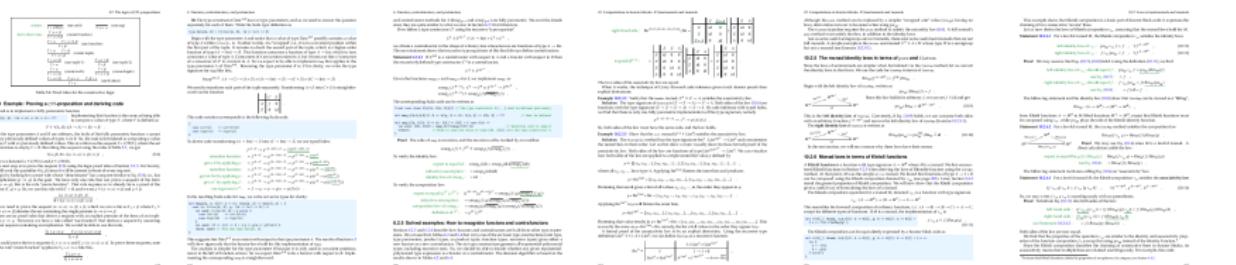
- FP is similar to engineering in a number of ways
 - ▶ Mechanical, electrical, chemical engineering are based on calculus, classical and quantum mechanics, electrodynamics, thermodynamics
 - ▶ FP is based on category theory, type theory, logic proof theory
- Engineers use special terminology
 - ▶ Examples 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”?

What I learned. II. Software engineers and software artisans

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)
```

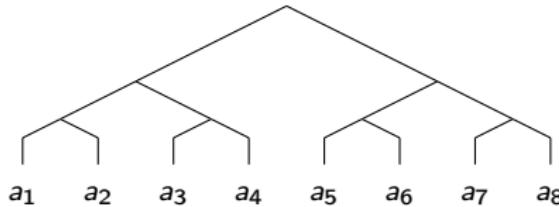
The `map` / `filter` / `reduce` programming style is an FP success story

- 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 `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 proof techniques that save time when proving the laws

What I learned. IV. The logic of types

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 code 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
 - ▶ Then 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
```

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

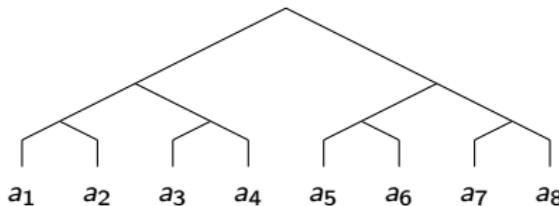
What I learned. V. Miscellaneous surprises

Chapters 1 to 3:

- Nameless functions are used in mathematics too, just hidden

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

- Variable 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) => B) => B = implement
def h[A, B]: (((A => B) => 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 = implement
error: type (((A => B) => B) => A) => 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

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

| | C | D |
|---|----------------------|----------------------|
| A | 0 | $x \rightarrow f(x)$ |
| B | $y \rightarrow g(y)$ | 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`
- Almost all monads are not 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`

What I learned. V. Miscellaneous surprises

Chapters 13 and 14:

- Not all typeclasses have a “free” construction: there is free functor, free contrafunctor, free filterable, free monad, free applicative functor, but not free foldable or free traversable
- “Tagless final” is just a Church encoding of the free monad
- A complete proof of the correctness of the Church encoding is *hard*
 - ▶ My proof uses relational parametricity together with some obscure results from **unpublished talk slides**
 - ★ ... but programmers do not need to study those proofs
- Monad transformers likely exist for all explicitly definable monads, but there is no general method or scheme for defining the transformers
- 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
- 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>