

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:

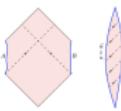


Figure 10. Left: The domain of Minkowski spacetime between two causally separated regions A, B meeting at a constant point $\tau = \tau_0$. Right: The same spacetime diagram showing that since the two causions cannot reach each other, right, a causal separation $\tau - \tau_0 > 0$ is possible. A light cone (shaded pink) covers the entire spacetime, while the light cone (shaded blue) covers the two causions (black line).



Figure 11. A spacetime diagram with two overlapping causal regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 12. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.

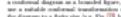


Figure 13. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 14. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 15. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 16. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.

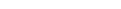


Figure 17. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 18. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 19. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 20. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 21. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 22. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 23. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 24. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 25. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.



Figure 26. A spacetime diagram with two causally overlapping regions A and B meeting at a constant point $\tau = \tau_0$.

integrals and shift the integration variable τ by $\delta\tau$

$$\begin{aligned} & \int_{\mathbb{R}} d\tau e^{i\tau\omega} P_\omega(\phi_0, \tau) \\ &= \int_{\mathbb{R}} d\tau e^{i\tau\omega} \int_{\mathbb{R}} d\tau' e^{i\tau'\omega} \int_{\mathbb{R}} d\tau'' e^{i\tau''\omega} P_\omega(\phi_0 + \delta\phi, \tau - \delta\tau) \\ &= \int_{\mathbb{R}} d\tau e^{i\tau\omega} \int_{\mathbb{R}} d\tau' e^{i\tau'\omega} \int_{\mathbb{R}} d\tau'' e^{i\tau''\omega} P_\omega(\phi_0 + \delta\phi, \tau - \delta\tau) \\ &= \int_{\mathbb{R}} d\tau e^{i\tau\omega} e^{i\delta\phi} P_\omega(\phi_0 + \delta\phi, \tau). \end{aligned} \quad (27)$$

Note that the way we split up $P_\omega(\phi_0, \tau)$ is slightly different from the definition of $J(\phi_0, \tau)$. The integration proceeds from $\tau = 0$ to $\tau = t$ because that is easier to allow the subtraction of $\delta\tau$ in the argument of P_ω . The following,

$$\int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) = \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \delta\tau), \quad (28)$$

is negligible as long as $\delta\tau$ is at least a few times smaller than τ . This is the reason why we can ignore the $\delta\tau$ term in $J(\phi_0, \tau)$ if τ is small enough. The result is that $J(\phi_0, \tau)$ is equal to the (exponentially small) probability of jumping from ϕ_0 to $\phi_0 + \delta\phi$ in time τ , i.e. one Hubble time $\delta\tau$. We can now substitute $J(\phi_0, \tau)$ for $P_\omega(\phi_0, \tau)$ in Eq. (27) to get $J(\phi_0, \tau)$ and finally obtain the equation

$$J(\phi_0, \tau) = \int_{\mathbb{R}} d\tau e^{i\tau\omega} e^{i\delta\phi} J(\phi_0 + \delta\phi, \tau - \delta\tau). \quad (29)$$

This is the first equation describing the function $J(\phi_0, \tau)$ as it is expected to be in the theory.

It is not possible to approximate Eq. (29) by a different equation, because the integration in Eq. (29) goes from $\tau = 0$ to $\tau = t$. To make progress, we will apply the trick of changing the variable τ . Instead of integrating over the variable τ , it is often more convenient, especially in one-dimensional integrals, to integrate over a related variable, such as $x = \tau - \tau_0$, where τ_0 is a fixed reference point. We then have

$$\tilde{\tau} = \int_{\tau_0}^{\tau} dt J(\phi(t))dt, \quad (30)$$

where $J(\phi(t))$ is a known function, thus the coefficients D_i , v_i and A_i are to be calculated as follows,

$$i = \frac{d}{dt}, \quad \tilde{\tau} = \tilde{D} - \frac{D}{2}, \quad \tilde{R} = R - \text{Add}, \quad (31)$$

while the deformation coefficient $K(\phi)$ is defined as

Integration, passing to the \tilde{x} -dimensional time

$$N = \int_{\mathbb{R}} d\tilde{\tau} M \tilde{R}, \quad (32)$$

where we suppressed the arguments ϕ , in the last line, it was defined to be Eq. (25), while Add is the standard

source function

$$\text{off} x = \frac{2}{\sqrt{2\pi}} \int_{\mathbb{R}} e^{-x^2/2} dt. \quad (33)$$

At the end of the calculation, we will only need to evaluate $J(\phi = 0, \phi_0)$. As already mentioned above, $J(\phi_0, \tau)$ is the probability of not violating the NEC owing all requirements (6) starting at $\phi = \phi_0$. The probability of not violating the NEC is the probability $J(\phi_0, \tau)$ rapidly decreases with growing τ because there is a significant probability of violating the NEC in the first Hubble time. On the other hand, the probability of violating the NEC is very small. In fact, the probability $P_\omega(\phi_0, \tau)$ is nearly constant and almost equal to 1 for τ in that regime. Therefore, we expect that $J(\phi_0, \tau)$ is also nearly constant and close to 1 for τ in that regime. This is what we find. Note that $P_\omega(\phi_0, \tau) = \delta(\tau)$ is given by

which holds because causality starting with $\phi = \phi_0$ immediately makes $\tau > 0$ and have zero derivatives. Therefore

$$\begin{aligned} \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) &= 1, \\ \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) e^{-i\tau\omega} &= 1. \end{aligned} \quad (34)$$

Motivated by these considerations, we represent the expectation value $\langle \text{off} x \rangle$ as $\langle \text{off} x \rangle = \int_{\mathbb{R}} d\tilde{\tau} \text{off}(x, \tilde{\tau}) J(\phi_0, \tilde{\tau})$ by the same code

$$\langle \text{off} x \rangle = \exp \left[- \int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \right]. \quad (35)$$

where $W(\phi_0, \tilde{\tau})$ is a one taken care such that $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) = 1$ and $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \text{off}(x, \tilde{\tau}) = \text{off} x$ through $J(\phi_0, \tilde{\tau})$ and request $J(\phi_0, \tilde{\tau})$ by the same code

$$\begin{aligned} & 1 - \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[- \int_{\mathbb{R}} d\tilde{\tau}' W(\phi_0, \tilde{\tau}') \right] \\ &= \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right], \quad (36) \end{aligned}$$

as $\int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right] = 1$. Then we can calculate $\langle \text{off} x \rangle$ by the same code as in Eq. (35).

is the Yukawa-Pfeiffer operator in the boundary field equations. Note that $\langle \text{off} x \rangle$ is not the solution of Eq. (32). However, this solution is not unique. It is not even clear if external inflation is present, since in that case the probability of having eternal points is certainly constant. As will be demonstrated below, the solution of the field equations is not unique. There are two solutions of a non-trivial solution $X(t) \neq 0$ of Eq. (32). We will begin by deriving that $X(t) = X(0)$ is a solution showing its independence of time parameterization.

It is well known that the solution of the field equations is not unique. The solution of $\phi = \phi(t)$ is a function with field value $\phi = \phi_0$ at the Planck boundary and $\phi = 0$ at the horizon boundary. The field $X(t) = 1 - X(0)$ and to use the proper time t as the time variable. We recall that the random walk of the inflaton field is described by a Langevin equation which can be written in the form of a Fokker-Planck equation,

$$\phi(t + \delta t) = \phi(t) + \eta(t) + \sqrt{2 D} \phi(t) \delta t. \quad (37)$$

The evolution of $\phi(t)$ is a brownian motion with field value $\phi = \phi_0$ at the Planck boundary and $\phi = 0$ at the horizon boundary. The field $X(t)$ is a white noise with zero mean and constant variance $2 D$.

This equation was first derived in [10]. A similar equation was mostly derived using a diffusion approximation in [10].

At the end of the calculation, we will only need to evaluate $J(\phi = 0, \phi_0)$. As already mentioned above, $J(\phi_0, \tau)$ is the probability of not violating the NEC owing all requirements (6) starting at $\phi = \phi_0$. The probability of not violating the NEC is the probability $J(\phi_0, \tau)$ rapidly decreases with growing τ because there is a significant probability of violating the NEC in the first Hubble time. On the other hand, the probability of violating the NEC is very small. In fact, the probability $P_\omega(\phi_0, \tau)$ is nearly constant and close to 1 for τ in that regime. Therefore, we expect that $J(\phi_0, \tau)$ is also nearly constant and close to 1 for τ in that regime. This is what we find. Note that $P_\omega(\phi_0, \tau) = \delta(\tau)$ is given by

which holds because causality starting with $\phi = \phi_0$ immediately makes $\tau > 0$ and have zero derivatives. Therefore

$$\begin{aligned} \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) &= 1, \\ \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) e^{-i\tau\omega} &= 1. \end{aligned} \quad (38)$$

Motivated by these considerations, we represent the expectation value $\langle \text{off} x \rangle$ as $\langle \text{off} x \rangle = \int_{\mathbb{R}} d\tilde{\tau} \text{off}(x, \tilde{\tau}) J(\phi_0, \tilde{\tau})$ by the same code

$$\langle \text{off} x \rangle = \exp \left[- \int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \right]. \quad (39)$$

where $W(\phi_0, \tilde{\tau})$ is a one taken care such that $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) = 1$ and $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \text{off}(x, \tilde{\tau}) = \text{off} x$ through $J(\phi_0, \tilde{\tau})$ and request $J(\phi_0, \tilde{\tau})$ by the same code

$$\begin{aligned} & 1 - \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[- \int_{\mathbb{R}} d\tilde{\tau}' W(\phi_0, \tilde{\tau}') \right] \\ &= \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right], \quad (40) \end{aligned}$$

as $\int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right] = 1$. Then we can calculate $\langle \text{off} x \rangle$ by the same code as in Eq. (35).

is the Yukawa-Pfeiffer operator in the boundary field equations. Note that $\langle \text{off} x \rangle$ is not the solution of Eq. (32). However, this solution is not unique. It is not even clear if external inflation is present, since in that case the probability of having eternal points is certainly constant. As will be demonstrated below, the solution of the field equations is not unique. There are two solutions of a non-trivial solution $X(t) \neq 0$ of Eq. (32). We will begin by deriving that $X(t) = X(0)$ is a solution showing its independence of time parameterization.

It is well known that the solution of the field equations is not unique. The field $X(t) = 1 - X(0)$ and to use the proper time t as the time variable. We recall that the random walk of the inflaton field is described by a Langevin equation which can be written in the form of a Fokker-Planck equation,

$$\phi(t + \delta t) = \phi(t) + \eta(t) + \sqrt{2 D} \phi(t) \delta t. \quad (41)$$

The evolution of $\phi(t)$ is a brownian motion with field value $\phi = \phi_0$ at the Planck boundary and $\phi = 0$ at the horizon boundary. The field $X(t)$ is a white noise with zero mean and constant variance $2 D$.

This equation was first derived in [10]. A similar equation was mostly derived using a diffusion approximation in [10].

At the end of the calculation, we will only need to evaluate $J(\phi = 0, \phi_0)$. As already mentioned above, $J(\phi_0, \tau)$ is the probability of not violating the NEC owing all requirements (6) starting at $\phi = \phi_0$.

The probability of not violating the NEC is the probability $J(\phi_0, \tau)$ rapidly decreases with growing τ because there is a significant probability of violating the NEC in the first Hubble time. On the other hand, the probability of violating the NEC is very small. In fact, the probability $P_\omega(\phi_0, \tau)$ is nearly constant and close to 1 for τ in that regime. Therefore, we expect that $J(\phi_0, \tau)$ is also nearly constant and close to 1 for τ in that regime. This is what we find. Note that $P_\omega(\phi_0, \tau) = \delta(\tau)$ is given by

which holds because causality starting with $\phi = \phi_0$ immediately makes $\tau > 0$ and have zero derivatives. Therefore

$$\begin{aligned} \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) &= 1, \\ \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) e^{-i\tau\omega} &= 1. \end{aligned} \quad (43)$$

Motivated by these considerations, we represent the expectation value $\langle \text{off} x \rangle$ as $\langle \text{off} x \rangle = \int_{\mathbb{R}} d\tilde{\tau} \text{off}(x, \tilde{\tau}) J(\phi_0, \tilde{\tau})$ by the same code

$$\langle \text{off} x \rangle = \exp \left[- \int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \right]. \quad (44)$$

where $W(\phi_0, \tilde{\tau})$ is a one taken care such that $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) = 1$ and $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \text{off}(x, \tilde{\tau}) = \text{off} x$ through $J(\phi_0, \tilde{\tau})$ and request $J(\phi_0, \tilde{\tau})$ by the same code

$$\begin{aligned} & 1 - \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[- \int_{\mathbb{R}} d\tilde{\tau}' W(\phi_0, \tilde{\tau}') \right] \\ &= \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right], \quad (45) \end{aligned}$$

as $\int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right] = 1$. Then we can calculate $\langle \text{off} x \rangle$ by the same code as in Eq. (35).

is the Yukawa-Pfeiffer operator in the boundary field equations. Note that $\langle \text{off} x \rangle$ is not the solution of Eq. (32). However, this solution is not unique. It is not even clear if external inflation is present, since in that case the probability of having eternal points is certainly constant. As will be demonstrated below, the solution of the field equations is not unique. There are two solutions of a non-trivial solution $X(t) \neq 0$ of Eq. (32). We will begin by deriving that $X(t) = X(0)$ is a solution showing its independence of time parameterization.

It is well known that the solution of the field equations is not unique. The field $X(t) = 1 - X(0)$ and to use the proper time t as the time variable. We recall that the random walk of the inflaton field is described by a Langevin equation which can be written in the form of a Fokker-Planck equation,

$$\phi(t + \delta t) = \phi(t) + \eta(t) + \sqrt{2 D} \phi(t) \delta t. \quad (46)$$

The evolution of $\phi(t)$ is a brownian motion with field value $\phi = \phi_0$ at the Planck boundary and $\phi = 0$ at the horizon boundary. The field $X(t)$ is a white noise with zero mean and constant variance $2 D$.

This equation was first derived in [10]. A similar equation was mostly derived using a diffusion approximation in [10].

At the end of the calculation, we will only need to evaluate $J(\phi = 0, \phi_0)$. As already mentioned above, $J(\phi_0, \tau)$ is the probability of not violating the NEC owing all requirements (6) starting at $\phi = \phi_0$.

The probability of not violating the NEC is the probability $J(\phi_0, \tau)$ rapidly decreases with growing τ because there is a significant probability of violating the NEC in the first Hubble time. On the other hand, the probability of violating the NEC is very small. In fact, the probability $P_\omega(\phi_0, \tau)$ is nearly constant and close to 1 for τ in that regime. Therefore, we expect that $J(\phi_0, \tau)$ is also nearly constant and close to 1 for τ in that regime. This is what we find. Note that $P_\omega(\phi_0, \tau) = \delta(\tau)$ is given by

which holds because causality starting with $\phi = \phi_0$ immediately makes $\tau > 0$ and have zero derivatives. Therefore

$$\begin{aligned} \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) &= 1, \\ \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) e^{-i\tau\omega} &= 1. \end{aligned} \quad (47)$$

Motivated by these considerations, we represent the expectation value $\langle \text{off} x \rangle$ as $\langle \text{off} x \rangle = \int_{\mathbb{R}} d\tilde{\tau} \text{off}(x, \tilde{\tau}) J(\phi_0, \tilde{\tau})$ by the same code

$$\langle \text{off} x \rangle = \exp \left[- \int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \right]. \quad (48)$$

where $W(\phi_0, \tilde{\tau})$ is a one taken care such that $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) = 1$ and $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \text{off}(x, \tilde{\tau}) = \text{off} x$ through $J(\phi_0, \tilde{\tau})$ and request $J(\phi_0, \tilde{\tau})$ by the same code

$$\begin{aligned} & 1 - \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[- \int_{\mathbb{R}} d\tilde{\tau}' W(\phi_0, \tilde{\tau}') \right] \\ &= \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right], \quad (49) \end{aligned}$$

as $\int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right] = 1$. Then we can calculate $\langle \text{off} x \rangle$ by the same code as in Eq. (35).

is the Yukawa-Pfeiffer operator in the boundary field equations. Note that $\langle \text{off} x \rangle$ is not the solution of Eq. (32). However, this solution is not unique. It is not even clear if external inflation is present, since in that case the probability of having eternal points is certainly constant. As will be demonstrated below, the solution of the field equations is not unique. There are two solutions of a non-trivial solution $X(t) \neq 0$ of Eq. (32). We will begin by deriving that $X(t) = X(0)$ is a solution showing its independence of time parameterization.

It is well known that the solution of the field equations is not unique. The field $X(t) = 1 - X(0)$ and to use the proper time t as the time variable. We recall that the random walk of the inflaton field is described by a Langevin equation which can be written in the form of a Fokker-Planck equation,

$$\phi(t + \delta t) = \phi(t) + \eta(t) + \sqrt{2 D} \phi(t) \delta t. \quad (50)$$

The evolution of $\phi(t)$ is a brownian motion with field value $\phi = \phi_0$ at the Planck boundary and $\phi = 0$ at the horizon boundary. The field $X(t)$ is a white noise with zero mean and constant variance $2 D$.

This equation was first derived in [10]. A similar equation was mostly derived using a diffusion approximation in [10].

At the end of the calculation, we will only need to evaluate $J(\phi = 0, \phi_0)$. As already mentioned above, $J(\phi_0, \tau)$ is the probability of not violating the NEC owing all requirements (6) starting at $\phi = \phi_0$.

The probability of not violating the NEC is the probability $J(\phi_0, \tau)$ rapidly decreases with growing τ because there is a significant probability of violating the NEC in the first Hubble time. On the other hand, the probability of violating the NEC is very small. In fact, the probability $P_\omega(\phi_0, \tau)$ is nearly constant and close to 1 for τ in that regime. Therefore, we expect that $J(\phi_0, \tau)$ is also nearly constant and close to 1 for τ in that regime. This is what we find. Note that $P_\omega(\phi_0, \tau) = \delta(\tau)$ is given by

which holds because causality starting with $\phi = \phi_0$ immediately makes $\tau > 0$ and have zero derivatives. Therefore

$$\begin{aligned} \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) &= 1, \\ \int_{\mathbb{R}} d\tau P_\omega(\phi_0, \tau) e^{-i\tau\omega} &= 1. \end{aligned} \quad (51)$$

Motivated by these considerations, we represent the expectation value $\langle \text{off} x \rangle$ as $\langle \text{off} x \rangle = \int_{\mathbb{R}} d\tilde{\tau} \text{off}(x, \tilde{\tau}) J(\phi_0, \tilde{\tau})$ by the same code

$$\langle \text{off} x \rangle = \exp \left[- \int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \right]. \quad (52)$$

where $W(\phi_0, \tilde{\tau})$ is a one taken care such that $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) = 1$ and $\int_{\mathbb{R}} d\tilde{\tau} W(\phi_0, \tilde{\tau}) \text{off}(x, \tilde{\tau}) = \text{off} x$ through $J(\phi_0, \tilde{\tau})$ and request $J(\phi_0, \tilde{\tau})$ by the same code

$$\begin{aligned} & 1 - \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[- \int_{\mathbb{R}} d\tilde{\tau}' W(\phi_0, \tilde{\tau}') \right] \\ &= \int_{\mathbb{R}} d\tilde{\tau} e^{i\tilde{\tau}\omega} \exp \left[-W(\phi_0, \tilde{\tau}) \right], \quad (53) \end{aligned}$$

as $\int_{\mathbb{R}} d\$

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. 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.*

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](#)”

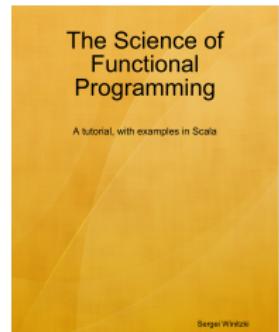
After several years of study, I found *systematic* ways of:

- 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

- 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] = ???
```

- ➋ 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
```

- ➌ It turns out that `f` *cannot* be implemented
- ➍ Programmers need to develop intuition about why this is so
- ➎ 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”?

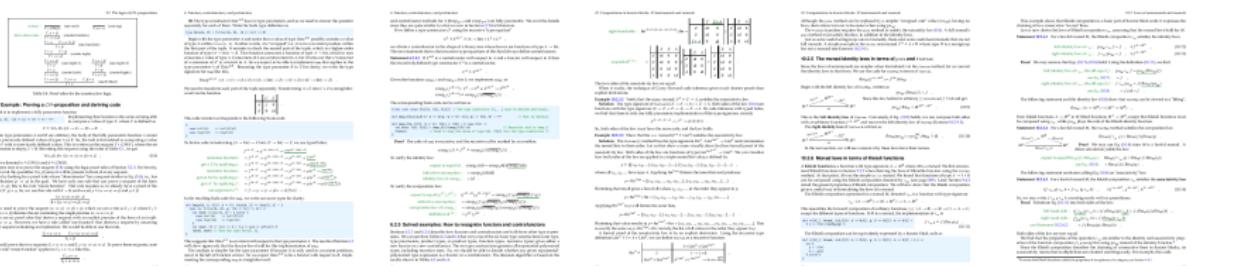
What I learned. II. Software engineers and software artisans

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)
```

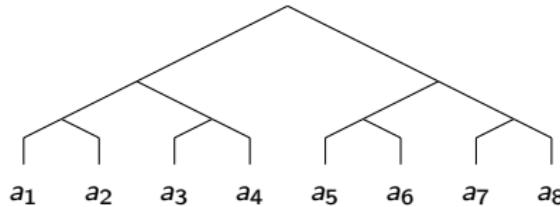
The `map` / `filter` / `reduce` programming style is an FP success story

- 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

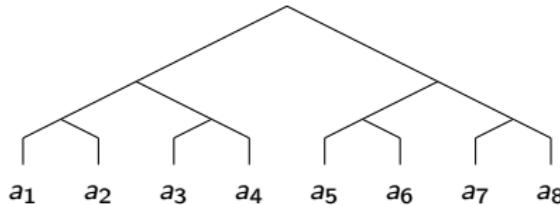
What I learned. V. Miscellaneous surprises

Chapters 1 to 3:

- Nameless functions are used in mathematics too, just hidden

$\sum_{n=1}^{100} n^2$	<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>

- 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) => 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
 - ▶ 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	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`
 - ▶ 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>