

What I learned about functional programming while writing a book on it

Sergei Winitzki

Scale by the Bay 2021

2021-10-29

Why write a book about functional programming.

My background: theoretical physics

- I used to write academic papers with lots of formulas and diagrams

Figure 10.4: A quadrilateral with two collapsing sides. The quadrilateral is divided into two triangles by a diagonal line. The top triangle has vertices labeled A, B, and C. The bottom triangle has vertices labeled D, E, and F. The sides AB and DE are labeled as collapsing.

Figure 10.5: The closed 4th-order equation between two equally oriented domains. A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI, AJ, AK, AL, AM, AN, AO, AP, AQ, AR, AS, AT, AU, AV, AW, AX, AY, AZ, BA, BB, BC, BD, BE, BF, BG, BH, BI, BJ, BK, BL, BM, BN, BO, BP, BQ, BR, BS, BT, BU, BV, BW, BX, BY, BZ, CA, CB, CC, CD, CE, CF, CG, CH, CI, CJ, CK, CL, CM, CN, CO, CP, CQ, CR, CS, CT, CU, CV, CW, CX, CY, CZ, DA, DB, DC, DD, DE, DF, DG, DH, DI, DJ, DK, DL, DM, DN, DO, DP, DQ, DR, DS, DT, DU, DV, DW, DX, DY, DZ, EA, EB, EC, ED, EE, EF, EG, EH, EI, EJ, EK, EL, EM, EN, EO, EP, EQ, ER, ES, ET, EU, EV, EW, EX, EY, EZ, FA, FB, FC, FD, FE, FF, FG, FH, FI, FJ, FK, FL, FM, FN, FO, FP, FQ, FR, FS, FT, FU, FV, FW, FX, FY, FZ, GA, GB, GC, GD, GE, GF, GG, GH, GI, GJ, GK, GL, GM, GN, GO, GP, GQ, GR, GS, GT, GU, GV, GW, GX, GY, GZ, HA, HB, HC, HD, HE, HF, HG, HH, HI, HJ, HK, HL, HM, HN, HO, HP, HQ, HR, HS, HT, HU, HV, HW, HX, HY, HZ, IA, IB, IC, ID, IE, IF, IG, IH, II, IJ, IK, IL, IM, IN, IO, IP, IQ, IR, IS, IT, IU, IV, IW, IX, IY, IZ, JA, JB, JC, JD, JE, JF, JG, JH, JI, JJ, JK, JL, JM, JN, JO, JP, JQ, JR, JS, JT, JU, JV, JW, JX, JY, JZ, KA, KB, KC, KD, KE, KF, KG, KH, KI, KJ, KK, KL, KM, KN, KO, KP, KQ, KR, KS, KT, KU, KV, KW, KX, KY, KZ, LA, LB, LC, LD, LE, LF, LG, LH, LI, LJ, LK, LL, LM, LN, LO, LP, LQ, LR, LS, LT, LU, LV, LW, LX, LY, LZ, MA, MB, MC, MD, ME, MF, MG, MH, MI, MJ, MK, ML, MM, MN, MO, MP, MQ, MR, MS, MT, MU, MV, MW, MX, MY, MZ, NA, NB, NC, ND, NE, NF, NG, NH, NI, NJ, NK, NL, NM, NO, NP, NQ, NR, NS, NT, NU, NV, NW, NX, NY, NZ, OA, OB, OC, OD, OE, OF, OG, OH, OI, OJ, OK, OL, OM, ON, OO, OP, OQ, OR, OS, OT, OU, OV, OW, OX, OY, OZ, PA, PB, PC, PD, PE, PF, PG, PH, PI, PJ, PK, PL, PM, PN, PO, PP, PQ, PR, PS, PT, PU, PV, PW, PX, PY, PZ, QA, QB, QC, QD, QE, QF, QG, QH, QI, QJ, QK, QL, QM, QN, QO, QP, QQ, QR, QS, QT, QU, QV, QW, QX, QY, QZ, RA, RB, RC, RD, RE, RF, RG, RH, RI, RJ, RK, RL, RM, RN, RO, RP, RQ, RR, RS, RT, RU, RV, RW, RX, RY, RZ, SA, SB, SC, SD, SE, SF, SG, SH, SI, SJ, SK, SL, SM, SN, SO, SP, SQ, SR, SS, ST, SU, SV, SW, SX, SY, SZ, TA, TB, TC, TD, TE, TF, TG, TH, TI, TJ, TK, TL, TM, TN, TO, TP, TQ, TR, TS, TT, TU, TV, TW, TX, TY, TZ, UA, UB, UC, UD, UE, UF, UG, UH, UI, UJ, UK, UL, UM, UN, UO, UP, UQ, UR, US, UT, UU, UV, UW, UX, UY, UZ, VA, VB, VC, VD, VE, VF, VG, VH, VI, VJ, VK, VL, VM, VN, VO, VP, VQ, VR, VS, VT, VU, VW, VX, VY, VZ, WA, WB, WC, WD, WE, WF, WG, WH, WI, WJ, WK, WL, WM, WN, WO, WP, WQ, WR, WS, WT, WU, WV, WW, WX, WY, WZ, XA, XB, XC, XD, XE, XF, XG, XH, XI, XJ, XK, XL, XM, XN, XO, XP, XQ, XR, XS, XT, XU, XV, XW, XX, XY, XZ, YA, YB, YC, YD, YE, YF, YG, YH, YI, YJ, YK, YL, YM, YN, YO, YP, YQ, YR, YS, YT, YU, YV, YW, YX, YY, YZ, ZA, ZB, ZC, ZD, ZE, ZF, ZG, ZH, ZI, ZJ, ZK, ZL, ZM, ZN, ZO, ZP, ZQ, ZR, ZS, ZT, ZU, ZV, ZW, ZX, ZY, ZZ, AA, AB, AC, AD, AE, AF, AG, AH, AI, AJ, AK, AL, AM, AN, AO, AP, AQ, AR, AS, AT, AU, AV, AW, AX, AY, AZ, BA, BB, BC, BD, BE, BF, BG, BH, BI, BJ, BK, BL, BM, BN, BO, BP, BQ, BR, BS, BT, BU, BV, BW, BX, BY, BZ, CA, CB, CC, CD, CE, CF, CG, CH, CI, CJ, CK, CL, CM, CN, CO, CP, CQ, CR, CS, CT, CU, CV, CW, CX, CY, CZ, DA, DB, DC, DD, DE, DF, DG, DH, DI, DJ, DK, DL, DM, DN, DO, DP, DQ, DR, DS, DT, DU, DV, DW, DX, DY, DZ, EA, EB, EC, ED, EE, EF, EG, EH, EI, EJ, EK, EL, EM, EN, EO, EP, EQ, ER, ES, ET, EU, EV, EW, EX, EY, EZ, FA, FB, FC, FD, FE, FF, FG, FH, FI, FJ, FK, FL, FM, FN, FO, FP, FQ, FR, FS, FT, FU, FV, FW, FX, FY, FZ, GA, GB, GC, GD, GE, GF, GG, GH, GI, GJ, GK, GL, GM, GN, GO, GP, GQ, GR, GS, GT, GU, GV, GW, GX, GY, GZ, HA, HB, HC, HD, HE, HF, HG, HH, HI, HJ, HK, HL, HM, HN, HO, HP, HQ, HR, HS, HT, HU, HV, HW, HX, HY, HZ, IA, IB, IC, ID, IE, IF, IG, IH, II, IJ, IK, IL, IM, IN, IO, IP, IQ, IR, IS, IT, IU, IV, IW, IX, IY, IZ, JA, JB, JC, JD, JE, JF, JG, JH, JI, JJ, JK, JL, JM, JN, JO, JP, JQ, JR, JS, JT, JU, JV, JW, JX, JY, JZ, KA, KB, KC, KD, KE, KF, KG, KH, KI, KJ, KK, KL, KM, KN, KO, KP, KQ, KR, KS, KT, KU, KV, KW, KX, KY, KZ, LA, LB, LC, LD, LE, LF, LG, LH, LI, LJ, LK, LL, LM, LN, LO, LP, LQ, LR, LS, LT, LU, LV, LW, LX, LY, LZ, MA, MB, MC, MD, ME, MF, MG, MH, MI, MJ, MK, ML, MM, MN, MO, MP, MQ, MR, MS, MT, MU, MV, MW, MX, MY, MZ, NA, NB, NC, ND, NE, NF, NG, NH, NI, NJ, NK, NL, NM, NO, NP, NQ, NR, NS, NT, NU, NV, NW, NX, NY, NZ, OA, OB, OC, OD, OE, OF, OG, OH, OI, OJ, OK, OL, OM, ON, OO, OP, OQ, OR, OS, OT, OU, OV, OW, OX, OY, OZ, PA, PB, PC, PD, PE, PF, PG, PH, PI, PJ, PK, PL, PM, PN, PO, PP, PQ, PR, PS, PT, PU, PV, PW, PX, PY, PZ, QA, QB, QC, QD, QE, QF, QG, QH, QI, QJ, QK, QL, QM, QN, QO, QP, QQ, QR, QS, QT, QU, QV, QW, QX, QY, QZ, RA, RB, RC, RD, RE, RF, RG, RH, RI, RJ, RK, RL, RM, RN, RO, RP, RQ, RR, RS, RT, RU, RV, RW, RX, RY, RZ, SA, SB, SC, SD, SE, SF, SG, SH, SI, SJ, SK, SL, SM, SN, SO, SP, SQ, SR, SS, ST, SU, SV, SW, SX, SY, SZ, TA, TB, TC, TD, TE, TF, TG, TH, TI, TJ, TK, TL, TM, TN, TO, TP, TQ, TR, TS, TT, TU, TV, TW, TX, TY, TZ, UA, UB, UC, UD, UE, UF, UG, UH, UI, UJ, UK, UL, UM, UN, UO, UP, UQ, UR, US, UT, UU, UV, UW, UX, UY, UZ, VA, VB, VC, VD, VE, VF, VG, VH, VI, VJ, VK, VL, VM, VN, VO, VP, VQ, VR, VS, VT, VU, VW, VX, VY, VZ, WA, WB, WC, WD, WE, WF, WG, WH, WI, WJ, WK, WL, WM, WN, WO, WP, WQ, WR, WS, WT, WU, WV, WW, WX, WY, WZ, XA, XB, XC, XD, XE, XF, XG, XH, XI, XJ, XK, XL, XM, XN, XO, XP, XQ, XR, XS, XT, XU, XV, XW, XX, XY, XZ, YA, YB, YC, YD, YE, YF, YG, YH, YI, YJ, YK, YL, YM, YN, YO, YP, YQ, YR, YS, YT, YU, YV, YW, YX, YY, YZ, ZA, ZB, ZC, ZD, ZE, ZF, ZG, ZH, ZI, ZJ, ZK, ZL, ZM, ZN, ZO, ZP, ZQ, ZR, ZS, ZT, ZU, ZV, ZW, ZX, ZY, ZZ.

At the end of the calculation, we will only need to evaluate $\langle \hat{I} \rangle = \langle \hat{I} \rangle_{\text{eff}}$. An already mentioned above, $\langle \hat{I} \rangle_{\text{eff}}$ is the fraction of trajectories that never visit the NCC during all iterations of the algorithm at $t = 0$. We note that for ϕ_0 in the flatness-dominated regime, the probability $\langle \hat{I} \rangle_{\text{eff}}$ rapidly increases with growing ϕ_0 because there is a significant probability of visiting the NCC at every flatness time step as there is. On the other hand, the probability of visiting the NCC in the no-flatness region is exponentially small, and hence $\langle \hat{I} \rangle_{\text{eff}}$ is nearly constant and almost equal to 1 for ϕ_0 in that regime. Therefore, we expect that $\langle \hat{I} \rangle_{\text{eff}}$ has exponentially strong dependence on ϕ_0 . Moreover, $P_1(\phi_0, t) = \langle \hat{I} \rangle_{\text{eff}}$ also can be shown by considering the presence of external 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 $\hat{I}(t)$ of having at least one external point in an initial flatness-size region where the field has value ϕ_0 . As we will show there, this probability satisfies the equation:

$$\frac{d\hat{I}(t)}{dt} + \hat{I}(t) = \langle \hat{I} \rangle_{\text{eff}} (1 - \hat{I}(t)) \quad (4.121)$$

with zero boundary conditions, $\hat{I}(0) = 0$ at the reheating border and $\hat{I}(\phi_0 = 0) = 0$ at the Friedmann (if any) flow.

$$\hat{I}_0 = \langle \hat{I} \rangle_{\text{eff}} \phi_0 + \hat{I}_0 \phi_0 \quad (4.122)$$

is the Fokker-Planck operator used in the reheating FP equation. Note that Eq. (4.122) is nonlinear due to the presence of the logarithmic term, this equation is of non-linear diffusion type.

We remark that $\hat{I}(t)$ is a trivial solution of Eq. (4.121). However, this solution is exponentially relevant if external inflation is present, since it measures the probability of having external points in certain regions. As will be demonstrated below, the presence of external inflation is equivalent to the existence of a non-trivial solution $\hat{I}(t) \neq 0$ 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 $\hat{I}(t) = 1 - \hat{I}(t)$ and to use the proper time t as the time variable. We recall that the random walk of the inflation field ϕ described by a Langevin equation which can be written as a diffusion equation (4.2):

$$\dot{\phi}(t) = \dot{\phi}(t) + \eta(t) \quad (4.123)$$

The evolution of ϕ in a horizon-size domain with field value ϕ_0 at $t = 0$ time $t = 0$ is given by an independent random walk of ϕ at each of the "diagonal" horizon-size subdomains that were formed out of the original domain. After

- Repented and turned to software engineering in 2010
- I have been studying FP since 2008 (OCam, Haskell, Scala)

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

Sergei Winitzki (SBTB)

What I learned about FP

2021-10-29

2 / 23

Why write 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
 - ▶ *“use this framework, 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 a talk about these concepts
 - ▶ An actual Scala error message:

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

To do FP, should I learn all of this? How do I learn about this?

Why write a book about functional programming. III

Main questions:

- Which theoretical knowledge will actually help write Scala code?
- Where can one learn this FP theory, with definitions and examples?
 - ▶ Where do the monad laws come from? How to verify them?
 - ▶ When is a data structure a functor (or monad, or applicative)?

Reading various materials has given me more questions than answers

- Heuristic explanations without derivations and proofs
 - ▶ Most FP books show code without proofs or rigorous definitions
 - ★ *The Book of Monads* does not prove the laws for any monads
 - ▶ A few books (*Haskell Wikibooks*, *Introduction to functional programming using Haskell*, and *Functional programming in Scala*) include some simple proofs
- Abstract, “academic” theory with no applications
 - ▶ “Monad is just a monoid in the category of endofunctors”

Why write a book about functional programming. IV

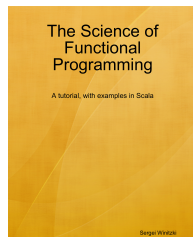
I started writing a **new book** to answer all my FP questions

- by motivating and deriving all results from scratch
- organizing systematically the practice-relevant parts of FP theory

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

Status of the book: 12.5 out of 14 chapters are ready



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

Examples of reasoning tasks. I

- 1 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 code
```

- It turns out that `f` *cannot* be implemented
 - ▶ Not because we are insufficiently clever, but because... math!
- Programmers need to develop intuition about why this is so
- These results are rigorous
 - ▶ The Curry-Howard isomorphism and the LJT algorithm
 - ▶ The code for `g[Z, R, A]` can be generated automatically

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 type 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?
 - ★ For example, `Either[A, (A, A, A)]` is a lawful monad
- This is a rigorous result (programmers do not need to test it)
 - ▶ The theory of monads and their laws
 - ▶ The theory of type constructions of monads

What I learned. II. Functional programming is engineering

- FP is similar to engineering in some ways
 - ▶ Mechanical, electrical, chemical engineering are based on calculus, classical and quantum mechanics, electrodynamics, thermodynamics
 - ★ These sciences give engineers rigorous answers to certain questions relevant to engineering design
 - ▶ FP is based on category theory, type theory, logic proof theory
 - ★ These theories give programmers rigorous answers to certain questions relevant to writing code
 - ▶ Programming in non-FP paradigms is similar to *artisanship*
- 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
 - ▶ What is a delta function? What is a lambda function?
 - ▶ What is the Gibbs free energy? What is the free monad?

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

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

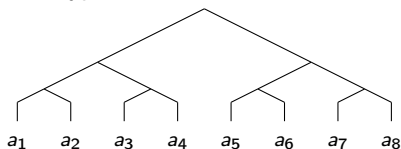
The MFR programming style is an FP success story

- 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 MFR methods: `map`, `filter`, `flatMap`, `zip`, `fold`

- What data types other than `Seq[A]` can support these methods?
 - ▶ Algebraic data types, such as `(A, Either[String, Option[A]])`
 - ▶ 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 a study of MFR

- 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 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 monad then so is `Either[A, P[A]]`
 - ▶ If `P[A]` and `Q[A]` are monads then so is `(P[A], Q[A])`
 - ▶ If `P[A]` is a contravariant functor then `P[A] => A` is a monad
 - ▶ If `P[A]` and `Q[A]` are applicative then so is `Either[P[A], (A, Q[A])]`
 - ★ I found many more type constructions of this kind
 - ▶ 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

- The compiler needs to check all types at compile time
- The language needs to support certain type constructions

Most of FP use cases are based on only six type constructions:

- Unit type — `Unit`
- Type parameters — `f[A](x)`
- 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`), $(A \text{ and } B)$, $(A \text{ or } B)$, $(\text{if } A \text{ then } B)$
- 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, we may just return `Try[Result]` or `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

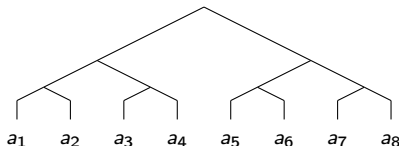
What I learned. V. Miscellaneous surprises

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> |

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

- Functions of type $\text{ADT} \Rightarrow \text{ADT}$ 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
- Subtypes / supertypes are not always the same as supersets / subsets

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
- 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?*
- It is hard to prove the correctness of the Church encoding
 - ▶ 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

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
- Some monad transformers are incomplete, not fully usable for combining monadic effects ([Continuation](#), [Codensity](#))
- *Monad transformers are just pointed endofunctors in the category of monads, what is the problem?*
- Monad transformers have 18 laws

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 500 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>