What I learned about functional programming

while writing a book on it

Sergei Winitzki

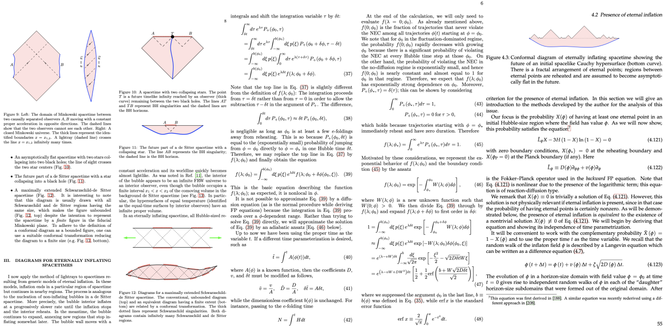
Scale by the Bay 2021

2021-10-29

Why write a book about functional programming. I

My background: theoretical physics

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



- Repented and turned to software engineering in 2010

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

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

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?

What I did *not* want to see:

- 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
- Academic theory for the sake of theory, with no applications
  - "*Monad is just a monoid in the category of endofunctors*"
  - *The Book of Monads* chapter 18 (adjunctions)
  - Lawvere theories as an alternative to monads

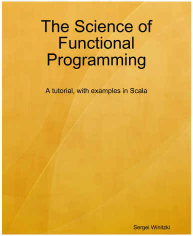
Why write a book about functional programming. IV

Reading various materials has given me more questions than answers
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?
- 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

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. 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* in some ways
- 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 $2 \le p \le q$) 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)
```

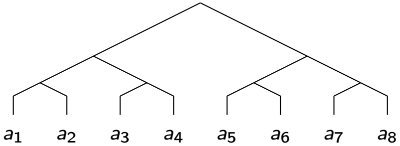
The MFR programming style is an FP success story

- 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?
  - 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 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, 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

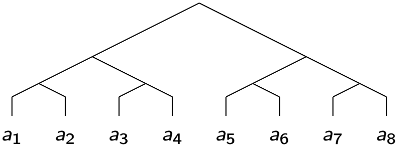
What I learned. V. Miscellaneous surprises

Chapters 1 to 3:

- Nameless functions are used in mathematics too, just hidden

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

- Many algorithms require non-tail-recursive code (`map` for a tree)
- Perfect-shaped trees *can* be defined via recursive ADTs



What I learned. V. Miscellaneous surprises

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

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

```
import io.chymyst.ch._

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

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

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

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

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

What I learned. V. Miscellaneous surprises

Chapters 6 to 8:

- 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))
}
```

$$\begin{array}{|c||cc|} & C & D \\ \hline A & \mathbb{0} & x \to f(x) \\ B & y \to g(y) & \mathbb{0} \end{array}$$

- 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

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
- 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