Reasoning about types and code

What we functional programmers need to know

Sergei Winitzki

Academy by the Bay

2019-08-17

A new draft book: *The Science of Functional Programming*

With examples in Scala. https://github.com/winitzki/sofp

Practitioners of functional programming need to know how to...

- ...reason about types:
  - design the required custom data types for the given application
  - derive an equivalent simpler type when possible
  - use type constructions to create data types with required properties
- ...reason about code:
  - verify that given implementations satisfy the required laws (e.g. monad)
  - derive lawful custom implementations of important typeclasses
  - verify that certain functions are computationally equivalent
  - derive an equivalent simpler code when possible

This requires a *very limited* amount of mathematics (polynomials, monoids)
The book will explain (with examples and exercises):

- known techniques of reasoning about types and type constructors
- known techniques for symbolic calculations with code
- deriving and verifying laws symbolically (as equations for functions)
- real-life motivations for (and applications of) these techniques

Examples of reasoning tasks

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

```
val result = for {
  a <- Future(...) // A computation that takes time and may fail.
  b <- Either(...) // A computation whose result may be unavailable.
  c <- Future(...) // A computation that takes time and may fail.
} yield ??? // Continue computations when results are available.
```

   Should `result` have type `Either[Z,Future[A]]` or `Future[Either[Z,A]]`?

3. Can we implement `flatMap` for the type constructor `Option[(A, A, A)]`?
4. What type describes a chain of `Future[A]` operations that, on any failure, will automatically execute specified cleanups in reverse order?
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 generated by a `Pointed` functor (i.e. when the functor already has the `pure` method)? Will that type have better performance than the standard free monad generated by a functor?

Short notation for types

- A concise mathematical notation for types and type constructors:

Scala syntax	Type notation
type parameter `A`	A
tuple type `(A, B)`, or `case class C(a: A, b: B)`	$A \times B$
disjunctive type `Either[A,B]` or `trait/case classes`	$A + B$
function type `A => B`	$A \Rightarrow B$
`Unit` or a "named" unit type	$\mathbb{1}$
primitive type (`Int`, `String`, etc.)	Int, String, ...
`Nothing` (the void type)	$\mathbb{0}$
type constructor, `type P[A] = Option[(A,A,A)]`	$P^A \triangleq \mathbb{1} + A \times A \times A$
type constructor, `P[_]`	$P^\bullet$
parameterized function, `def f[A]: A => (A, A)`	$f^A : A \Rightarrow A \times A$

This notation proved convenient for reasoning about equivalence of types:

$$(A + B) \times C \cong A \times C + B \times C \quad , \qquad A \Rightarrow B \Rightarrow C \cong A \times B \Rightarrow C \quad ,$$

$$\mathbb{0} + A \cong A \quad , \qquad A + B \Rightarrow C \cong (A \Rightarrow C) \times (B \Rightarrow C)$$

Short notation for code

- A concise mathematical notation for code:

Scala syntax	Type notation
variable, or function argument `x: A`	$x^{:A}$
tuple value `(a, b)`	$a \times b$ or $a^{:A} \times b^{:B}$
value of disjunctive type `Left[A, B](x)`	$x^{:A} + \mathbb{0}^{:B}$
nameless function `{x: A => expr }`	$x^{:A} \Rightarrow \text{expr}$
`Unit` value, `()`	1
`def f[A, B]: A => B => A = x => _ => x`	$f^{:A\Rightarrow B\Rightarrow A} \triangleq x^{:A} \Rightarrow \_^{:B} \Rightarrow x$
`identity[A]` (the identity function)	id^{A} or $\text{id}^{:A\Rightarrow A}$
forward composition, `f andThen g`	$f \fatsemi g$ or $f^{:A\Rightarrow B} \fatsemi g^{:B\Rightarrow C}$
argument piping, `x.pipe(f)` (Scala 2.13)	$x \triangleright f$ where $x^{:A}$ and $f^{:A\Rightarrow B}$
lifted function, `p.map(f)` where `p: P[A]`	$p \triangleright f^{\uparrow P}$ where $p^{:P^{A}}$

This notation proved convenient for reasoning about equational laws:

$$f^{\uparrow P} \fatsemi g^{\uparrow P} \fatsemi h^{\uparrow P} = (f \fatsemi g \fatsemi h)^{\uparrow P} \quad , \qquad x \triangleright f \triangleright g = x \triangleright f \fatsemi g$$

Example I. Free monad generated by a type constructor

A free monad generated by a type constructor `F[_]` is...

1. From a blog post by Gabriel Gonzalez (2012), translated into Scala:

```
sealed trait Free1[F[_], T]
case class Pure[F[_], T](t: T)                  extends Free1[F, T]
case class Flatten[F[_], T](f: F[Free1[F, T]]) extends Free1[F, T]
```

2. From a talk given by Rúnar Bjarnason (2014):

```
sealed trait Free2[F[_], T]
case class Pure[F[_], T](t: T) extends Free2[F, T]
case class Bind[F[_], T, A](f: F[A], g: A => Free2[F, T])
     extends Free2[F, T]
```

3. From a talk given by Kelley Robinson (2016):

```
sealed trait Free3[F[_], T]
case class Pure[F[_], T](t: T)         extends Free3[F, T]
case class Suspend[F[_], T](f: F[T])   extends Free3[F, T]
case class FlatMap[F[_], T, A](f: Free3[F, A], g: A => Free3[F, T])
     extends Free3[F, T]
```

Are these definitions different? How to reason about them?

Example I. Writing the definitions in the type notation

A free monad generated by a type constructor `F[_]` is...

1. From G. Gonzalez: $\text{Free}_1^{F,T} \triangleq T + F^{\text{Free}_1^{F,T}}$

```
sealed trait Free1[F[_], T]
case class Pure[F[_], T](t: T)                   extends Free1[F, T]
case class Flatten[F[_], T](f: F[Free1[F, T]]) extends Free1[F, T]
```

2. From R. Bjarnason: $\text{Free}_2^{F,T} \triangleq T + \exists A.\, F^A \times (A \Rightarrow \text{Free}_2^{F,T})$

```
sealed trait Free2[F[_], T]
case class Pure[F[_], T](t: T) extends Free2[F, T]
case class Bind[F[_], T, A](f: F[A], g: A => Free2[F, T])
     extends Free2[F, T]
```

3. From K. Robinson: $\text{Free}_3^{F,T} \triangleq T + F^T + \exists A.\, \text{Free}_3^{F,A} \times (A \Rightarrow \text{Free}_3^{F,T})$

```
sealed trait Free3[F[_], T]
case class Pure[F[_], T](t: T)           extends Free3[F, T]
case class Suspend[F[_], T](f: F[T])    extends Free3[F, T]
case class FlatMap[F[_], T, A](f: Free3[F, A], g: A => Free3[F, T])
     extends Free3[F, T]
```

Example I. Three definitions of the free monad

What is the difference between definitions 1 and 2?

- Definition $\text{Free}_1^{F,T} \triangleq T + F^{\text{Free}_1^{F,T}}$ assumes that $F^\bullet$ is a functor
- Definition $\text{Free}_2^{F,T} \triangleq T + \exists A.\, F^A \times (A \Rightarrow \text{Free}_2^{F,T})$ works for any $F^\bullet$
  - if $F^\bullet$ is a functor, the Yoneda identity gives $F^B \cong \exists A.\, F^A \times (A \Rightarrow B)$
  - then $T + \exists A.\, F^A \times (A \Rightarrow \text{Free}_2^{F,T}) \cong T + F^{\text{Free}_2^{F,T}}$
- So, definitions 1 and 2 are equivalent when $F^\bullet$ is a functor

What is the difference between definitions 2 and 3?

- Can we replace $\text{Free}_3^{F,A}$ by F^A in definition 3? Yes.

$$
\begin{aligned}
& T + F^T + \exists A.\, \underline{\text{Free}_3^{F,A}} \times (A \Rightarrow \text{Free}_3^{F,T}) \\
& = T + \underline{F^T} + \exists A.\, \left(A + F^A + \exists B.\, \text{Free}_3^{F,B} \times (B \Rightarrow \text{Free}_3^{F,A})\right) \times (A \Rightarrow \text{Free}_3^{F,T}) \\
& = T + \underline{\exists A.\, F^A \times (A \Rightarrow T)} + \underline{\exists A.\, A \times (A \Rightarrow \text{Free}_3^{F,T})} + \exists A.\, F^A \times (A \Rightarrow \text{Free}_3^{F,T}) \\
& \qquad + \exists A.\, \exists B.\, \underline{\text{Free}_3^{F,B} \times (B \Rightarrow \text{Free}_3^{F,A}) \times (A \Rightarrow \text{Free}_3^{F,T})} \quad .
\end{aligned}
$$

By the monad's associativity law, we reduce $\text{Free}_3^{F,B} \times (B \Rightarrow \text{Free}_3^{F,A}) \times (A \Rightarrow \text{Free}_3^{F,T})$ to $\text{Free}_3^{F,B} \times (B \Rightarrow \text{Free}_3^{F,A} \times (A \Rightarrow \text{Free}_3^{F,T}))$. By inductive assumption, this is $\text{Free}_3^{F,B} \times (B \Rightarrow \text{Free}_3^{F,T})$ and can be replaced by $F^B \times (B \Rightarrow \text{Free}_3^{F,A})$.

Example II. The selection monad transformer

- The selection monad is defined as `Sel[A] = (A => R) => A`
  - This is related to the "search monad", `(A => Boolean) => Option[A]`
    - but *not* related to the "search monad" in Haskell's monad-dijkstra
- The paper "Monad Transformers for Backtracking Search" https://arxiv.org/pdf/1406.2058.pdf defines the selection monad transformer `SelT[A] = (A => M[R]) => M[A]`. The author writes,

  The proof that the selection monad satisfies the monad laws was found using an ad-hoc computer program written by Martin Escardó [...] The author has verified by hand that the selection monad transformer preserves the unit laws, however the proof for the associativity law appears to be unmanageable. The author is currently working on a formally verified proof that the selection monad transformer preserves the monad laws.

- The proof was too hard because (most likely) the author tried to write symbolic calculations in Haskell syntax, which is not well adapted to proofs. (This would have been worse if using Scala syntax.)
  - The proof is 5 lines of calculations in the short code notation
  - Doing a formally verified proof is *much harder* than proving by hand!
    - (and you *already* need to know how to prove it by hand)

Example II. Proof of the associativity law

The selection monad transformer's type is $T^{M,A} \triangleq (A \Rightarrow M^R) \Rightarrow M^A$
Define Kleisli functions $f : A \Rightarrow T^{M,B} \triangleq A \Rightarrow (B \Rightarrow M^R) \Rightarrow M^B$
Use a trick: flip arguments, $f : (B \Rightarrow M^R) \Rightarrow A \Rightarrow M^B$ and $g : (C \Rightarrow M^R) \Rightarrow B \Rightarrow M^C$
Define Kleisli composition, $f \diamond_T g : (C \Rightarrow M^R) \Rightarrow A \Rightarrow M^C$ using the known $\diamond_M$ as

$$f \diamond_T g \triangleq k^{:C \Rightarrow M^R} \Rightarrow f(g(k) \diamond_M k) \diamond_M g(k)$$

The operation $\diamond_M$ already satisfies the associativity law, $(p \diamond_M q) \diamond_M r = p \diamond_M (q \diamond_M r)$
Prepare the result of $g \diamond_T h$:

$$(g \diamond_T h)(k) = \underline{g(h(k) \diamond_M k) \diamond_M h(k)}$$

Now check associativity law for the flipped Kleisli composition $\diamond_T$:

$$\begin{aligned}
&(f \diamond_T g) \diamond_T h = k \Rightarrow (\underline{f \diamond_T g})(h(k) \diamond_M k) \diamond_M h(k) \\
&= k \Rightarrow f(g(\underline{h(k) \diamond_M k}) \diamond_M \underline{h(k) \diamond_M k}) \diamond_M g(\underline{h(k) \diamond_M k}) \diamond_M h(k) \quad , \\
&f \diamond_T (g \diamond_T h) = k \Rightarrow f((\underline{g \diamond_T h})(k) \diamond_M k) \diamond_M (\underline{g \diamond_T h})(k) \\
&= k \Rightarrow f(\underline{g(h(k) \diamond_M k) \diamond_M h(k)} \diamond_M k) \diamond_M \underline{g(h(k) \diamond_M k) \diamond_M h(k)} \quad .
\end{aligned}$$

So we have proved the associativity law, $(f \diamond_T g) \diamond_T h = f \diamond_T (g \diamond_T h)$

Summary

- Mastering the type/code reasoning takes about 6 months of practice
  - and you will be able to use FP much more effectively in actual coding
  - the special notation helped me a lot
- Lots of explanations, examples, and exercises in the upcoming book
- Current progress: chapters 1–6 ready after a second proofreading
- `https://github.com/winitzki/sofp`