

Type equivalence under laws

Why flatMap is "equivalent" to flatten and map

Sergei Winitzki

Academy by the Bay

2021-05-08

Equivalent formulations of typeclasses I

Monads can be defined via `pure`, `map`, and `flatten`, *or* via `pure` and `flatMap`. It is often said that these methods are “equivalent”:

- 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.

Applicative functors may be defined via `ap` and `pure` *or* via `map2` and `pure`

- P. Chuisano and R. Bjarnason, “*Functional programming in Scala*”



EXERCISE 12.2

Hard: 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]  
}
```

Define in terms of
`map2` and `unit`.

Define in terms
of `apply` and
`unit`.

This must be right, but questions remain...

- What does it mean that `x` is “equivalent in expressiveness” to `y`?
- How can it be that `map2[A, B, C]` is “equivalent” to `ap[A, B]`?

Equivalent formulations of typeclasses II. More examples

We know that `flatMap` is *equal* to the composition of `flatten` and `map`

Also, `flatten` can be expressed via `flatMap`

```
def flatten[A](ffa: F[F[A]]): F[A] = ffa.flatMap(identity)
def flatMap[A, B](p: F[A])(f: A => F[B]): F[B] = p.map(f).flatten

flatten = flatMap(id)    ,       $p \triangleright \text{flatMap}(f) = p \triangleright f^{\uparrow F} \triangleright \text{flatten}$ 
```

The `pure` method can be expressed via `wu` and vice versa:

```
def wu: F[Unit] = pure(())
def pure[A](a: A): F[A] = wu.map(_ => a)

wu = pure(1)    ,       $\text{pure}(a) = wu \triangleright (\_ \rightarrow a)^{\uparrow F}$ 
```

The `filter` method can be expressed via `deflate` and vice versa:

```
def deflate[A](foa: F[Option[A]]): F[A] =
  foa.filter(_.nonEmpty).map(_.get)
def filter[A](fa: F[A])(p: A => Boolean): F[A] =
  deflate(fa.map { a => if (p(a)) Some(a) else None } )
```

Notation: $x \triangleright f$ means `f(x)` or in Scala 2.13, `x.pipe(f)`

$f^{\uparrow F}$ means `_.map(f)` for the functor `F`

Confusing issue 1: “equivalence” of values?

- Yes, we can express `flatten` through `flatMap`, but so what?
- Is 5 “equivalent” to 10 in expressive power?

```
def five: Int = ten / 2
def ten: Int = five * 2
```

Confusing issue 2: “equivalence” of functions with different sets of type parameters?

- How can `pure[A]: A => F[A]` and `wu: F[Unit]` be equivalent?
 - ▶ An extra type parameter means there are many more implementations
- Example of a `pure` that is *not* equivalent to `wu`:

```
def badPure[A](x: A): List[A] = x match {  
  case i: Int    => List(i + 123)  
  case _         => List(x)  
}
```

- The corresponding `wu = List(())`
- We cannot restore `badPure` from `wu`: the corresponding `pure` is `List(_)`

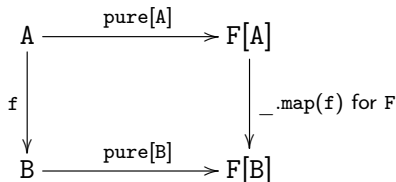
Equivalence under naturality laws I: example

The problem with `badPure` is that it does not work the same for all types

- The code of `badPure` is not **fully parametric**
- To enforce full parametricity, we require a naturality law:

$$\text{pure}(f(x)) == \text{pure}(x) \cdot \text{map}(f)$$

$$x \triangleright f \triangleright \text{pure} = x \triangleright \text{pure} \triangleright f^{\uparrow F}, \quad f:A \rightarrow B \circ \text{pure}^B = \text{pure}^A \circ f^{\uparrow F}$$



Equivalence under naturality laws II: general formulation

The precise meaning of the “equivalent expressive power” of `pure[A]` and `wu`:

- the set of all functions `pure[A] : A => F[A]` satisfying the naturality law is in a one-to-one correspondence with the set of all values `wu : F[Unit]`

Proof:

- Start with `pure` that satisfies the naturality law; define `wu = pure(());` then define `pure2(x) = wu.map(_ => x)`. Show that `pure2 == pure`:

$$\begin{aligned} \text{for an arbitrary } x : \quad \text{pure}_2(x) &= \text{wu} \triangleright (_ \rightarrow x)^{\uparrow F} = 1 \triangleright \text{pure} \triangleright (_ \rightarrow x)^{\uparrow F} \\ \text{use naturality law :} \quad &= 1 \triangleright (_ \rightarrow x) \triangleright \text{pure} = x \triangleright \text{pure} = \text{pure}(x) \end{aligned}$$

- Start with `wu : F[Unit]`; define `pure(x) = wu.map(_ => x)`; then define `wu2 = pure(())`. Show that `wu2 == wu`:

$$\text{wu}_2 = \text{pure}(1) = \text{wu} \triangleright (_ \rightarrow 1)^{\uparrow F} = \text{wu} \triangleright \text{id}^{\uparrow F} = \text{wu} \triangleright \text{id} = \text{wu}$$

The function `pure(x) = wu.map(_ => x)` satisfies the naturality law:

$$\begin{aligned} \text{pure}(x) \triangleright f^{\uparrow F} &= \text{wu} \triangleright (_ \rightarrow x)^{\uparrow F} \triangleright f^{\uparrow F} = \text{wu} \triangleright (_ \rightarrow x \triangleright f)^{\uparrow F} \\ &= \text{wu} \triangleright (_ \rightarrow f(x))^{\uparrow F} = \text{pure}(f(x)) \end{aligned}$$

Equivalence under naturality laws III: general pattern

To prove the equivalence of $p: P[A, B, C]$ and $q: Q[A, B, C]$ under assumption of some naturality laws:

- Implement functions $p2q$ and $q2p$:

```
def p2q[A, B, C]: P[A, B, C] => Q[A, B, C] = ...  
def q2p[A, B, C]: Q[A, B, C] => P[A, B, C] = ...
```

- Show that $q2p(p2q(p)) == p$ and $p2q(q2p(q)) == q$
- Show that $p2q(p)$ satisfies q 's laws, and $q2p(q)$ satisfies p 's laws

The “set of $p: P[A, B, C]$ satisfying a law” is a **refined type**

- The Scala compiler cannot verify laws automatically
- Testing cannot verify laws since type parameters cannot be arbitrary
- Laws must be verified via symbolic reasoning about code

Equivalence under naturality laws IV: further examples

- Equivalence of `flatten[A]` and `flatMap[A, B]` requires a naturality law for `flatMap[A, B]` with respect to `B`

```
p.flatMap(f andThen g) == p.map(f).flatMap(g)
```

$$\text{flatMap}(f \circ g) = f^{\uparrow F} \circ \text{flatMap}(g)$$

- Equivalence of `ap` and `zip` requires a naturality law for each of them

```
def ap[A, B]: F[A => B] => F[A] => F[B] = ...
```

```
ap(r)(p).map(f) == ap(r.map(x => x andThen f))(p)
```

```
def zip[A, B]: (F[A], F[B]) => F[(A, B)] = ...
```

```
zip(p.map(f), q) == zip(p, q).map { case (a, b) => (f(a), b) }
```

- Formulated the “equivalence of expressive power” rigorously
 - ▶ It is a one-to-one correspondence between *refined types*
- In most cases, the equivalence holds only after imposing naturality laws
- Naturality laws constrain code and may eliminate a type parameter
 - ▶ Naturality laws will hold automatically for fully parametric code