8 — Applicative Functors

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2020

Lehrstuhl für Informatik VI, Uni Würzburg

A note on syntax

In the previous lectures, we defined extension methods on our type classes to make them more comfortable to use (we can use them, as if they were methods defined on our objects).

Sometimes, e.g. for proofs, it may be more suitable to call them like a normal function taking all operands as parameters. We can do this with extensions too. For the extension method on the left, the calls on the right are equivalent:

```
extension (a: A)
  def method(b: B) = ???
```

```
a.method(b)

method(a)(b)
// if on a typeclass instance F:
F.method(a)(b)
```

Terminology: effect

Type constructors like `Option`, `List`, `Either`, etc. are often called effects, because they augment values with “extra” capabilities like possible absence (`Option`), multiplicity (`List`), etc.

This is not to be confused with *side* effects (= violation of referential transparency).

For the ones that have a monad instance, we also call them monadic effects.

Limitations of Monads

While monads and functors let us chain operations in the context of their effect using `flatMap` and `map`, there are program flows they can't express.

As an example, we will look at `Either` and its inability to accumulate errors. We will then see an alternative to Either, that can accumulate errors but cannot have a monad instance.

(Not) Accumulating Errors with Either

A common use case is form validation. Say we have a form with three fields. Each field has a method to check it for valid input:

```
def validName(s:String): Either[String, String] = ???
def validBirthdate(s:String): Either[String, LocalDateTime] = ???
def validPhone(s:String): Either[String, Phone] = ???
```

We would like to check all fields and report all errors to the user at once.

```
for
  name  <- validName(field1)
  date  <- validBirthdate(field2)
  phone <- validPhone(field3)
yield CheckedForm(name, date, phone)
```

But if `validName` returns a `Left`, the other two methods are not even called.

Limitations of Monads

Maybe another combinator can help? `map2` and its extensions to higher numbers should evaluate all validations:

```
map3(
  validName(field1),
  validBirthdate(field2),
  validPhone(field3)
)(CheckedForm(_, _, _))
```

Using `map3` all three methods are always called, but if we implement it via `flatMap`, it has the same result. It's not possible to accumulate without breaking monad laws.

mapN on Either

Could we implement map2/3/... for Either, so that it does accumulate errors?

```
extension [E, A](a: Either[E, A])
  def map2[B, C](b: Either[E, B])(f: (A, B) => C) : Either[E, C] = ???
```

mapN on Either

Could we implement map2/3/... for Either, so that it does accumulate errors?

```
extension [E, A](a: Either[E, A])
  def map2[B, C](b: Either[E, B])(f: (A, B) => C) : Either[E, C] = ???
```

The signature requires us to return an Either, whose error type is a supertype of the inputs' error types. We can't combine two Es into one without additional knowledge about them.

Validated to the rescue

It looks like we will need a different type for accumulating errors. We will call it `Validated`:

```
enum Validated[+E, +A]:
  case Valid(a: A)
  // Invalid stores at least one error and possibly more
  case Invalid(head: E, tail: List[E])
```

Note: In Cats, `Validated` looks a bit different, it let's you control what structure is used to accumulate errors. It differs in implementation, but usage is similar.

Validated.map2

We can now have an implementations of map2/3/.../N for Validated, which accumulate errors. An expanded example of our form validation from before:

```
import java.time.*
import java.time.format.*

def validName(name: String): Validated[String, String] =
  if name.nonEmpty then Valid(name)
  else Invalid("Name cannot be empty", Nil)

def validBirthdate(birthdate: String): Validated[String, LocalDateTime] =
  try
    Valid(LocalDateTime.parse(birthdate))
  catch
    case _:DateTimeParseException =>
      Invalid("Birthdate must be in YYYY-MM-DD format", Nil)

def validPhone(phoneNumber: String): Validated[String, String] =
  if phoneNumber.matches("[0-9]{10}") then Valid(phoneNumber)
  else Invalid("Phone number must be 10 digits", Nil)
```

Validated.map2

So returning a single error looks just like with Either. But combining them with map3 will keep all errors:

```
case class CheckedForm(name: String, date: LocalDateTime, phone: String)

def validWebForm(
  name: String, birthdate: String, phone: String
): Validated[String, CheckedForm] =
  map3(
    validName(name),
    validBirthdate(birthdate),
    validPhone(phone)
  )(CheckedForm(_,_,_)) // same call signature as for Either
```

```
// if all fields are invalid, would return:
Invalid("Name cannot be empty",
  List("Birthdate must be in YYYY-MM-DD format",
    "Phone number must be 10 digits"))
```

Can Validated be a Monad?

This achieves what we wanted. But can we also make it a Monad, while keeping the accumulation? We'd need a `flatMap` implementation for a given E (`pure` is simply the constructor for a `Valid`):

```
def flatMap[A,B](fa: Validated[E,A])(f: A => Validated[E,B]): Validated[E,B]
```

Can Validated be a Monad?

This achieves what we wanted. But can we also make it a Monad, while keeping the accumulation? We'd need a `flatMap` implementation for a given E (`pure` is simply the constructor for a `Valid`):

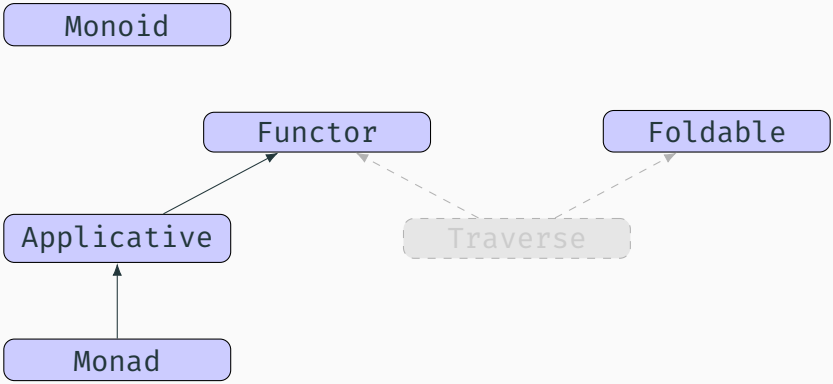
```
def flatMap[A,B](fa: Validated[E,A])(f: A => Validated[E,B]): Validated[E,B]
```

Not possible, because our function requires an A for generating the next error $\Longrightarrow$ can't produce another error, if we already have one.

But `map2` can do more than a Functor, so we want something between Functor and Monad.

Applicative Functors

Typeclass hierarchy



map2 without flatMap

We defined `map2` in terms of `flatMap` in the Monad lecture:

```
def map2[B, C](fb: F[B])(f: (A, B) => C): F[C] =
  fa.flatMap(a => fb.flatMap(b => pure(f(a, b))))
```

But we've justseen that there are types, for which we cannot define a `flatMap` function, that gives `map2` the desired behavior. So we should define a typeclass, that has `map2` but not `flatMap`.

Methods for Applicative

As our hierarchy slide already showed, that typeclass is called `Applicative`. What methods beside `map2` should it provide?

There are some methods we defined for all monads, e.g. `sequence` and `traverse` on the exercise sheet. One possible solution doesnt need `flatMap`, but only `map2` and `pure`:

```
def sequence[A](fas: List[F[A]]): F[List[A]] =
  fas.foldRight[F[List[A]]](pure(Nil))((a, b) => a.map2(b)(_::_))
```

So requiring `pure` allows us to provide those for `Applicative` too.

map2 and ap

Several (but not all) of the monad combinators can be implemented in terms of `map2`, without using `flatMap` directly. And we also do not need `flatMap` to implement `map2`.

We'll look at another combinator, named `ap`, which is equally powerful as `map2` (i.e. we can implement one of both and derive the other from it). It applies a function inside an Applicative `F` to a value also inside an `F`:

```
def ap[A,B](ff: F[A => B])(fa: F[A]): F[B]
```

The Applicative typeclass

As we saw in the hierarchy, an Applicative is also a Functor. So we can create the following trait for our typeclass, with `pure` plus one of `map2` or `ap` as its minimal sets of combinators:

```
trait Applicative[F[_]] extends Functor[F]:
  def pure[A](a: A): F[A]

  def ap[A,B](ff: F[A => B])(fa: F[A]): F[B]
  extension [A](fa: F[A])
    def map2[B,C](fb: F[B])(f: (A, B) =>  C): F[C]

    def map[B](f: A =>  B): F[B]
```

note: map is inherited, added here for clarity

Exercise: Deriving a Functor

Before we implement our applicative for Validated, we will show, that we can derive `map` from `ap` and `map2`

Given `pure` and either `ap` or `map2`

```
def ap[A,B](ff: F[A => B])(fa: F[A]): F[B]
```

```
extension [A](fa: F[A])
  def map2[B,C](fb: F[B])(f: (A, B) =>  C): F[C]
```

implement `map` in terms of them:

```
def map[B](f: A =>  B): F[B]
```

Exercise: Deriving a Functor – Solution

Comparing `ap` with `map`, our signatures look very similar, `ap` just has the function wrapped inside `F`.

Wrapping things in `F` is exactly what `pure` does:

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  ap(pure(f))(fa)
```

Using `map2` looks similar:

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  pure(f).map2(fa)((ff, a) => ff(a))
```

We don't even need to pass a useful second value in `map2`:

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  pure(()).map2(fa)((_, a) => f(a))
```

Exercise: Using `ap` for `map2`

We also said, that `map2` and `ap` are equally powerful. Implement `map2` in terms of `pure` and `ap`:

```
extension [A](fa: F[A])
  def map2[B,C](fb: F[B])(f: (A, B) =>  C): F[C]
```

Hint: You can use `f.curried` to turn a function `f: (A,B) => C` into a curried function `A => (B => C)`. And remember, how we mapped a single `F`:

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  ap(pure(f))(fa)
```

Exercise: Using `ap` for `map2` — Solution

```
extension [A](fa: F[A])
  def map2[B,C](fb: F[B])(f: (A, B) =>  C): F[C] =
    ap(
      ap(pure(f.curried))(fa) // returns F[B => C]
    )(fb) // apply the returned function to F[B], to get F[C]
```

`map3`, `map4`, … can also be implemented with the same pattern by nesting `ap` calls:

```
def map3[A,B,C,D](fa: F[A], fb: F[B], fc: F[C])(f: (A, B, C) =>  D): F[D] =
  ap(ap(ap(pure(f.curried))(fa))(fb))(fc)
```

ap via map2

We can also implement `ap` using `map2`, so they are equally powerful:

```
def ap[A,B](ff: F[A => B])(fa: F[A]): F[B] =
  ff.map2(fa)((f, a) => f(a))
```

Exercise: Implement an Applicative for Validated

```scala
enum Validated[+E, +A]:
  case Valid(a: A)
  // Invalid stores at least one error and possibly more
  case Invalid(head: E, tail: List[E])
```

With all those methods in place, we now only need `pure` and either `ap` or `map2` for an Applicative instance. Implement one for Validated, that accumulates errors. Note that a failure always has at least one error in head, further errors accumulate in tail.

```scala
given [E]: Applicative[[a] =>> Validated[E,a]] with
  def pure[A](a: A) = Valid(a)
  // add map2 or ap here
```

Exercise: Implement an Applicative for Validated — Solution

```
extension [A](fa: Validated[E,A])
  override def map2[B,C](fb: Validated[E,B])(f: (A,B) => C) =
    (fa, fb) match
      case(Valid(a), Valid(b)) => Valid(f(a,b))
      case(Invalid(h1, t1), Invalid(h2,t2)) => Invalid(h1, t1 ++ (h2 :: t2))
      case(Invalid(h,t), _) => Invalid(h,t)
      case(_, Invalid(h,t)) => Invalid(h,t)
```

Implementing `ap` instead of `map2` nearly identical:

```
override def ap[A,B](ff: Validated[E,A => B])(fa: Validated[E,A]) =
  (ff, fa) match
    case(Valid(f), Valid(a)) => Valid(f(a))
    // ... Invalid cases identical
```

Monads are Applicatives

As we can provide `map2` via `flatMap`, we can make `Monad[F]` a subtype of `Applicative[F]`:

```
trait Monad[F[_]] extends Applicative[F]:
  // pure is inherited now, and we can't provide a default

  extension [A](fa: F[A])
    def flatMap[B](f: A => F[B]): F[B]

  // default implementations via flatMap
  def ap[A,B](ff: F[A => B])(fa: F[A]): F[B] = ???
  extension [A](fa: F[A])
    override def map2[B,C](fb: F[B])(f: (A, B) =>  C): F[C] = ???
    override def map[B](f: A =>  B): F[B] = ???
  //...
```

A minimal implementation only needs to override `pure` and `flatMap` (or `compose` or both `flatten` and `map`, see the previous exercise sheet).

Difference between monads and applicative functors

We've seen several minimal sets of combinators for monads (Monad laws can be stated in terms of them):

- `pure` and `flatMap`
- `pure` and `compose`
- `pure`, `map` and `flatten` (see last exercise)

`pure` and `map2`/`ap` are not enough to implement any of the others, can we give a reason for that?

Applicative less powerful

Consider `flatten[A](ffa: F[F[A]]): F[A]`
It removes one layer of `F`.

`Applicative` can only *add* a layer with `pure` and work inside `F` with `map2` / `ap`.

⇒ We cannot implement flatten in terms of `pure` and `map2` / `ap`. The same reasoning can be used for `flatMap` and `compose`.

Extra capabilities of monads

So what can we do with a Monad, that we can't do with an applicative? An example:

```
val F: Applicative[Option] = ???

val depts:    Map[String,String] = ??? // department by employee name
val salaries: Map[String,Double] = ??? // salary by employee name

val o: Option[String] =
  F.map2(depts.get("Alice"))(salaries.get("Alice"))(
    (dept, salary) => s"Alice in $dept makes $salary per year"
  )
```

All lookups returning `Option` are independent here, and are combined afterwards.

Extra capabilities of monads

But what if we have queries returning `Option`, that depend on another `Option`?

```
val F: Applicative[Option] = ???

val idsByName: Map[String, EmployeeID] = ???
val depts:     Map[EmployeeID, String] = ???
val salaries:  Map[EmployeeID, Double] = ???

val o: Option[String] =
  idsByName.get("Bob").flatMap { id =>
    F.map2(depts.get(id))(salaries.get(id))(
      (dept, salary) => s"Bob in $dept makes $salary per year")
  }
```

Queries for salary and department depend on id, which is also queried from a map and so may be missing too.

The `flatMap` in the highlighted line cannot be expressed using methods from `Applicative`.

Difference between monads and applicatives — variants

Some variants of stating the difference between monads and applicatives:

- `Applicative` computations have fixed structure and only *sequence* effects, monadic computations may choose structure dynamically based on previous effects.
- `Applicative` constructs *context-free* computation, `Monad` allows for *context-sensitivity*.
- `Monad` makes effects first class; may be generated at „interpretation“ time instead of ahead of time by the program.

Advantages of applicative functors

Why write code that uses `Applicative` instead of the more powerful `Monad`? Simple, we know types that have an `Applicative` but cannot have a `Monad`, like Validated. But every Monad is also an Applicative. Also, applicative functors compose, which monads (in general) don't (details later).

If code can be written using Applicative only, it could e.g. be changed from fail-fast to error-accumulating, simply by passing another type.

Therefore it is preferable to implement combinators like `sequence` with the lowest amount of combinators possible, to get better reusability through fewer dependencies.

Applicative Laws

Laws for applicative functors

What laws should our functions obey?

Of course we expect applicative functors to obey the identity and composition laws for functors:

```
map(v)(id) == v
map(map(v)(g))(f) == map(v)(f compose g)
```

This implies some laws for applicative functors, if we want to be able to implement `map` via `ap` or `map2` and `pure`.

Laws for `map2`

Laws for Applicatives — map2

The laws for `map2` are:

- left identity
- right identity
- associativity

We start with looking at an implementation of `map` via `map2`.

Laws for Applicatives — Left and right identity

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  map2(pure(()))(fa)((_, a) => f(a))
```

This definition is arbitrary, swapping `pure` and `fa` still obeys functor laws:

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  map2(fa)(pure(()))((a, _) => f(a))
```

`map2` of some `fa` with `pure` and a function that ignores the pure side preserves structure of `fa`. We call these left and right identity laws:

```
map2(pure(()))(fa)((_,a) => a) == fa //left identity
map2(fa)(pure(()))((a,_) => a) == fa //right identity
```

Laws for Applicatives — Associativity

```
def map3[A,B,C,D](fa: F[A], fb: F[B], fc: F[C])(f: (A, B, C) =>  D): F[D] =
```

We implemented `map3` via `ap`, but also possible via combining successively with `map2`. The associativity law for applicative functors tells us, that result of combining `fa` and `fb` first and then `fc` should yield same result as combining `fb` and `fc` first.

We want something similar to the associativity laws we know from monoids and monads:

```
          a |+| (b |+| c) == (a |+| b) |+| c
compose(f, compose(g, h)) == compose(compose(f, g), h)
```

Laws for Applicatives — Associativity

We use the following helper methods to define associativity:

- `product` merges two `F` by tupling.

```
def product[A,B](fa: F[A], fb: F[B]): F[(A,B)] =
  fa.map2(fb)((_,_))
```

- `assoc` just turns nesting of tuples around:

```
def assoc[A,B,C](p: F[(A, (B, C))]): F[((A, B), C)] =
  map(p){ case (a, (b, c)) => ((a, b), c) }
```

With these, the associativity law is:

```
product(product(fa, fb), fc) == assoc(product(fa, product(fb, fc)))
```

Laws for **ap**

Laws for ap

The laws for **ap** are very abstract. We will skip them in the lecture, and they will not be exam relevant. The slides are here if you are interested.

The important lesson that you should remember is:
ap *behaves like normal function application, but inside an applicative functor.*

Laws for Applicatives — Identity

```
map(v)(id) == v
map(map(v)(g))(f) == map(v)(f compose g)
```

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  ap(pure(f))(fa)
```

Looking at our implementation of `map`, we can conclude, that an identity law also is required for `ap`:

```
ap(pure(id))(v) == v
```

Laws for Applicatives — Composition

```
map(v)(id) == v
map(map(v)(g))(f) == map(v)(f compose g)
```

```
extension [A](fa: F[A]) def map[B](f: A =>  B): F[B] =
  ap(pure(f))(fa)
```

The same goes for composition:

```
ap(fbc)(ap(fab)(fa)) == ap(ap(ap(pure(compose))(fbc))(fab))(v)
```

where `compose` is the same function compositon as in the `map` law.

Laws for Applicatives — Homomorphism

The homomorphism law states, that it does not matter, if we apply a function to a value before or after we put them into our applicative context:

```
ap(pure(f))(pure(a)) == pure(f(a))
```

The interchange law states, that function application is the same whether we use `ap` to apply a function to a value in our Applicative or we use `pure` to lift normal function application into applicative context:

```
ap(ff)(pure(x)) == ap(pure((f: A=>B) => f(x)))(ff)
```

`fab` here is a `F[A => B]` and `x` is of type `A`.

With `(f: A => B) => f(x)` we define a function, that takes another function and applies it to `x`.

Composing Applicatives

Exercise: Applicative composition

We mentioned, that applicative functors compose. Let's create a `compose` method for Applicatives:

```
def compose[F[_], G[_]](
  using F: Applicative[F], G: Applicative[G]
): Applicative[[a] =>> F[G[a]]] =
```

Reminder: We've already seen a similar signature for Functor. Here `[a] => F[G[a]]` results in a type constructor that takes a single parameter, denoted by `a` in our type expression. We need a type lambda here instead of writing `F[G[_]]` because of the nesting (we'd be passing `G[_]` to `F[_]`, instead of creating a nested constructor).

Exercise: Applicative product: pure

```scala
def compose[F[_], G[_]](
  using F: Applicative[F], G: Applicative[G]
): Applicative[[a] =>> F[G[a]]] = new Applicative[[a] =>> F[G[a]]]:
    def pure[A](a: A): F[G[A]] = ???
```

Implement `pure`.

Exercise: Applicative product: pure — Solution

We call `pure` on each Applicative, nesting them like our expected result:

```
def pure[A](a: A) = F.pure(G.pure(a))
```

Exercise: Applicative product — Implement map2 or ap

```
override def ap[A,B](fgf: F[G[A => B]])(fga: F[G[A]]): F[G[B]] =
```

```
extension [A](fga: F[G[A]])
  override def map2[B,C](fgb: F[G[B]])(f: (A, B) => C): F[G[C]] =
```

Implement either `map2` or `ap`.

Hint: both start the same way. And remember that you can use both ***map2*** *and* ***ap*** *from* ***F*** *and* ***G***

Exercise: Applicative product — Solution

```
override def ap[A,B](fgf: F[G[A => B]])(fga: F[G[A]]): F[G[B]] =
  F.map2(fgf)(fga)((gf, ga) => G.ap(gf)(ga))
```

```
extension [A](fga: F[G[A]])
  override def map2[B,C](fgb: F[G[B]])(f: (A, B) => C): F[G[C]] =
    F.map2(fga)(fgb)((ga, gb) => G.map2(ga)(gb)(f))
```

More composition

There are more possibilities to compose applicative functors. If `F[_]` and `G[_]` are applicative functors, then `(F[_],G[_])` is too.

```
def product[F[_], G[_]](
  using F: Applicative[F], G: Applicative[G]
): Applicative[[a] =>> (F[a],G[a])] =
```

This will be on an exercise sheet.

Monad composition

Composition is one thing that applicatives can do, but monads can't.

Suppose we wanted to implement `flatMap` for nested monads F and G. The resulting signature would expect a function `f: A => F[G[B]]`.

But `flatMap` on a `F[G[A]]` would expect a function `G[A] => F[X]`. While X could be `G[B]`, we can't use our function `f`, as we can't "unwrap" the `G[A]`.

So we can't implement a general composition for monads. There are ways to combine monads, as long as one is fixed (e.g. we can compose any monad with an Option inside it), which is a topic in the seminar.

Summary

We've seen today:

- how to accumulate multiple errors from fallible computations with Validated
- ...and why Either can't do it, resp. why Validated can't have `flatMap`
- the Applicative typeclass as something between Monads and Functors
  - its operators `ap` and `map2`, which are equally powerful
  - the laws for `map2`
  - that Applicatives can compose by nesting them