Free All The Things

Markus Hauck

Free All The Things

- well known: free monads
- maybe known: free applicatives
- free monoids
- free <you name it>

Goal Of This Talk

- how many of you wrote a Free X
- how many of you used Free...
  - Monad
  - Applicative
  - Functor
  - other?
- Goal: explain the technique behind “Free X” + example

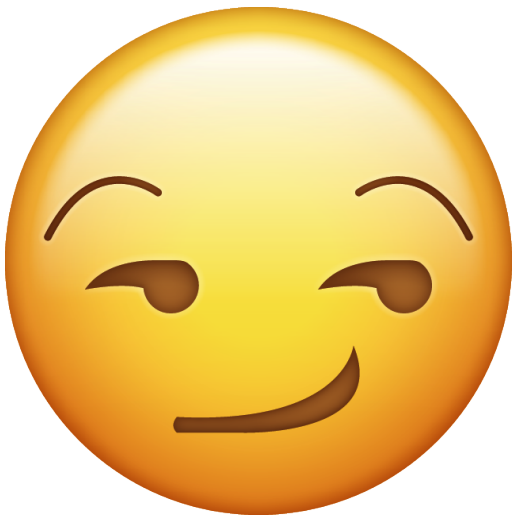
The Road Ahead

- Demonstrate a recipe to “free” things
- Using: Free Monads, Applicatives, Functors
- New thing: Free Boolean Algebra + Example

What's The Problem

A free functor is left adjoint to a forgetful functor

what's the problem?



What Is Free

A free “thing” **FreeA** on a type(class) *A* is a *A* and a function

```
def inject(x: A): FreeA
```

such that for any other “thing” *B* and a function

```
val f: A => B
```

there exists a unique homomorphism `g` such that

```
g.compose(inject) === f
```

What Is Free

- still sounds complicated?
- there is a recipe
  - AST
  - `inject`
  - interpreter
  - check laws

Why Free

- nice API using typeclass
- use Free X as if it was X
- program reified into datastructure
- structure can be analyzed/optimized
- one program — many interpretations

Disclaimer Before We Start

- deep embeddings / initial encoding / data structure representation
- not: finally tagless, optimization

Freeing The Monad

The Monad Typeclass

```
1    trait Monad[F[_]] {
2      def pure[A](x: A): F[A]
3
4      def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
5    }
```

Give Me The Laws

```
1  // Left identity
2  pure(a).flatMap(f) === f(a)
3
4  // Right identity
5  fa.flatMap(pure) === fa
6
7  // Associativity
8  fa.flatMap(f).flatMap(g) ===
9    fa.flatMap(a => f(a).flatMap(g))
```

Applying The Recipe

```scala
trait Monad[F[_]] {
  def pure[A](x: A): F[A]

  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

- now comes our recipe
  - AST
  - `inject`
  - interpreter
  - check laws

Freeing The Monad

```scala
sealed abstract class Free[F[_], A]

final case class Pure[F[_], A](a: A)
    extends Free[F, A]

final case class FlatMap[F[_], A, B](
    fa: Free[F, A],
    f: A => Free[F, B])
    extends Free[F, B]

final case class Inject[F[_], A](fa: F[A])
    extends Free[F, A]
```

Freeing The Monad

```
1   implicit def freeMonad[F[_], A]: Monad[Free[F, ?]] =
2     new Monad[Free[F, ?]] {
3       def pure[A](x: A): Free[F, A] = Pure(x)
4
5       def flatMap[A, B](fa: Free[F, A])(
6           f: A => Free[F, B]): Free[F, B] =
7         FlatMap(fa, f)
8     }
```

Interpreter

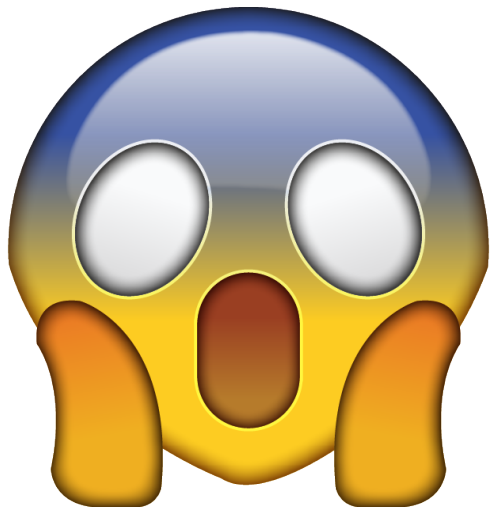
```
1    def runFree[F[_], M[_]: Monad, A](nat: F ~> M)(
2        free: Free[F, A]): M[A] = free match {
3      case Pure(x)     => Monad[M].pure(x)
4      case Inject(fa) => nat(fa)
5      case FlatMap(fa, f) =>
6        Monad[M].flatMap(runFree(nat)(fa))(x =>
7          runFree(nat)(f(x)))
8    }
```

What about the laws?

```
1  // The associativity law
2  fa.flatMap(f).flatMap(g) ===
3    fa.flatMap(fa, a => f(a).flatMap(g))
```

```
1  val exp1 = FlatMap(FlatMap(fa, f), g)
2  val exp2 = FlatMap(fa, (a: Int) => FlatMap(f(a), g))
3
4  exp1 != exp2
```

What about the laws?



The Laws

- actually, we don't satisfy them
- programmer: after interpretation it's no longer visible
- mathematician: that's not the free monad!
- tradeoff: during construction vs during interpretation

The Right Free Monad

- common transformation: associate `flatMap`'s to the right
- avoids having to rebuild the tree repeatedly during construction
- how: during construction time

Transforming Free Monads

```
1    def flatMap[A, B](fa: Free[F, A])(
2        f: A => Free[F, B]): Free[F, B] = fa match {
3      case Pure(x)     => f(x) // Left identity
4      case Inject(fa) => FlatMap(Inject(fa), f)
5      case FlatMap(ga, g) => // Associativity
6        FlatMap(ga, (a: Any) => FlatMap(g(a), f))
7    }
```

Transforming Free Monads

```
1    def flatMap[A, B](fa: Free[F, A])(
2        f: A => Free[F, B]): Free[F, B] = fa match {
3      case Pure(x)     => f(x) // Left identity
4      case Inject(fa) => FlatMap(Inject(fa), f)
5      case FlatMap(ga, g) => // Associativity
6        FlatMap(ga, (a: Any) => FlatMap(g(a), f))
7    }
```

Transforming Free Monads

```
1    def flatMap[A, B](fa: Free[F, A])(
2        f: A => Free[F, B]): Free[F, B] = fa match {
3      case Pure(x)     => f(x) // Left identity
4      case Inject(fa) => FlatMap(Inject(fa), f)
5      case FlatMap(ga, g) => // Associativity
6        FlatMap(ga, (a: Any) => FlatMap(g(a), f))
7    }
```

We Freed Monads

- DSL with monadic expressiveness
- context sensitive, branching, loops, fancy control flow
- familiarity with monadic style for DSL
- big drawback: interpreter has limited possibilities

Freeing The Applicative

Freeing The Applicative

- free monads are great, but also limited
- we can’t analyze the programs
- how about a smaller abstraction?

Recall

- we follow the same pattern
- AST
- `inject`
- interpreter
- check laws

The Applicative Typeclass

```
1    trait Applicative[F[_]] {
2      def pure[A](x: A): F[A]
3
4      def ap[A, B](fab: F[A => B], fa: F[A]): F[B]
5    }
```

AST for FreeApplicative

```
1   sealed abstract class FreeAp[F[_], A]
2
3   final case class Pure[F[_], A](a: A)
4       extends FreeAp[F, A]
5
6   final case class Ap[F[_], A, B](
7       fab: FreeAp[F, A => B],
8       fa: FreeAp[F, A])
9       extends FreeAp[F, B]
10
11  final case class Inject[F[_], A](fa: F[A])
12      extends FreeAp[F, A]
```

```
1
```

Laws

```
1  // identity
2  Ap(Pure(identity), v) === v
3
4  // composition
5  Ap(Ap(Ap(Pure(_.compose), u), v), w) ===
6    Ap(u, Ap(v, w))
7
8  // homomorphism
9  Ap(Pure(f), Pure(x)) === Pure(f(x))
10
11 // interchange
12 Ap(u, Pure(y)) === Ap(Pure(_(y)), u)
```

Don’t Forget The Laws

```
1  def ap[A, B](fab: FreeAp[F, A => B],
2               fa: FreeAp[F, A]): FreeAp[F, B] =
3    (fab, fa) match {
4      case (Pure(f), Pure(x)) =>
5        Pure(f(x)) // homomorphism
6      case (u, Pure(y)) =>
7        Ap(Pure((f: A => B) => f(y)), u) // interchange
8      case (_, _) => Ap(fab, fa)
9    }
```

Don’t Forget The Laws

```
1  def ap[A, B](fab: FreeAp[F, A => B],
2               fa: FreeAp[F, A]): FreeAp[F, B] =
3    (fab, fa) match {
4      case (Pure(f), Pure(x)) =>
5        Pure(f(x)) // homomorphism
6      case (u, Pure(y)) =>
7        Ap(Pure((f: A => B) => f(y)), u) // interchange
8      case (_, _) => Ap(fab, fa)
9    }
```

Running FreeApplicatives

```
1    def runFreeAp[F[_], M[_]: Applicative, A](
2        nat: F ~> M)(free: FreeAp[F, A]): M[A] =
3      free match {
4        case Pure(x)    => Applicative[M].pure(x)
5        case Inject(fa) => nat(fa)
6        case Ap(fab, fa) =>
7          Applicative[M]
8            .ap(runFreeAp(nat)(fab), runFreeAp(nat)(fa))
9      }
```

We Freed Applicatives

- DSL with applicative expressiveness
- context insensitive
- pure computation over effectful arguments
- more freedom during interpretation

Freeing The Functor

And Once Again

- AST
- `inject`
- interpreter
- check laws

The Functor Typeclass

```
1    trait Functor[F[_]] {
2      def map[A, B](fa: F[A])(f: A => B): F[B]
3    }
```

Freeing The Functor

```
1  sealed abstract class FreeFunctor[F[_], A]
2
3  case class Fmap[F[_], X, A](fa: F[X])(f: X => A)
4      extends FreeFunctor[F, A]
5
6  case class Inject[F[_], A](fa: F[A])
7      extends FreeFunctor[F, A]
```

Freeing The Functor

```
1  sealed abstract class FreeFunctor[F[_], A]
2
3  case class Fmap[F[_], X, A](fa: F[X])(f: X => A)
4      extends FreeFunctor[F, A]
5
6  case class Inject[F[_], A](fa: F[A])
7      extends FreeFunctor[F, A]
```

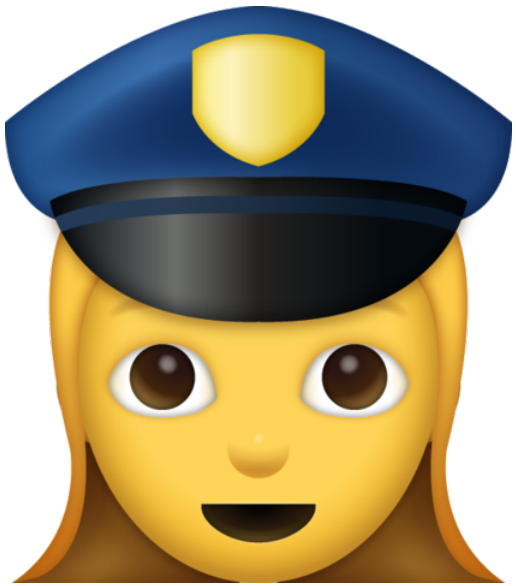
Freeing The Functor

```scala
sealed abstract class FreeFunctor[F[_], A]

case class Fmap[F[_], X, A](fa: F[X])(f: X => A)
    extends FreeFunctor[F, A]

def inject[F[_], A](value: F[A]) =
  Fmap(value)(identity)
```

Clean Code Police



- only one subclass?

Freeing The Functor

```scala
sealed abstract class Fmap[F[_], A] {
  type X
  def fa: F[X]
  def f: X => A
}

def inject[F[_], A](v: F[A]) = new Fmap[F, A] {
  type X = A
  def fa = v
  def f = identity
}
```

Freeing The Functor

```scala
sealed abstract class Coyoneda[F[_], A] {
  type X
  def fa: F[X]
  def f: X => A
}

def inject[F[_], A](v: F[A]) = new Coyoneda[F, A] {
  type X = A
  def fa = v
  def f = identity
}
```

Free Functor Instance

```scala
implicit def coyoFun[F[_]]: Functor[Coyoneda[F, ?]] =
  new Functor[Coyoneda[F, ?]] {
    def map[A, B](coyo: Coyoneda[F, A])(
        g: A => B): Coyoneda[F, B] =
      new Coyoneda[F, B] {
        type X = coyo.X
        def fa = coyo.fa
        def f = g.compose(coyo.f)
      }
  }
```

Free Functor Instance

```
1   implicit def coyoFun[F[_]]: Functor[Coyoneda[F, ?]] =
2     new Functor[Coyoneda[F, ?]] {
3       def map[A, B](coyo: Coyoneda[F, A])(
4           g: A => B): Coyoneda[F, B] =
5         new Coyoneda[F, B] {
6           type X = coyo.X
7           def fa = coyo.fa
8           def f = g.compose(coyo.f)
9         }
10    }
```

Free Functor Interpreter

```scala
def runCoyo[F[_]: Functor, A](
    coyo: Coyoneda[F, A]): F[A] =
  Functor[F].map(coyo.fa)(coyo.f)
```

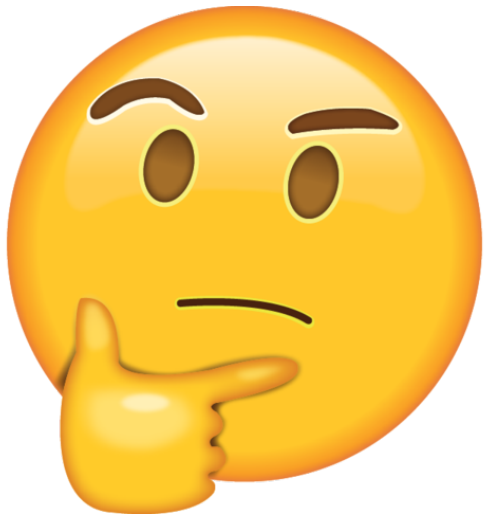
We Freed Functors

- DSL with hmm functorial expressiveness?
- map fusion! (functor law)
- boring interpreter, though

We Freed Functors

- DSL with hmm functorial expressiveness?
- map fusion! (functor law)
- boring interpreter, though
- still fun!

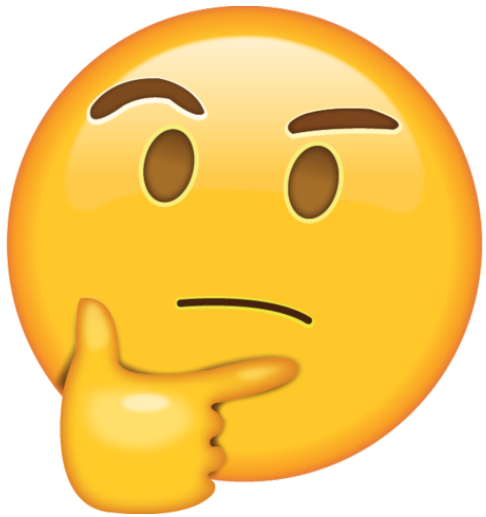
Now That We Can Free Anything



What should we free?

Monads Applicatives Functors

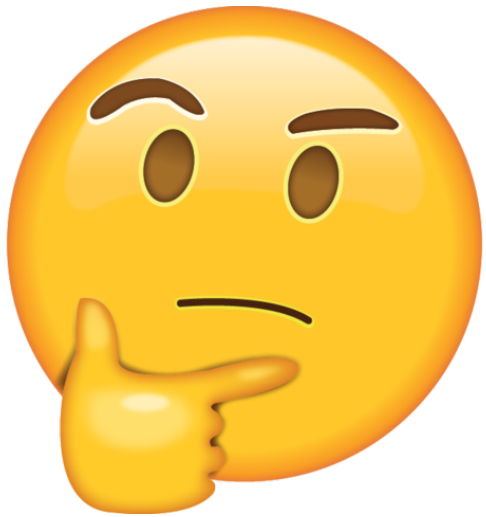
Now That We Can Free Anything



What should we free?

Monads Applicatives Functors Monoids

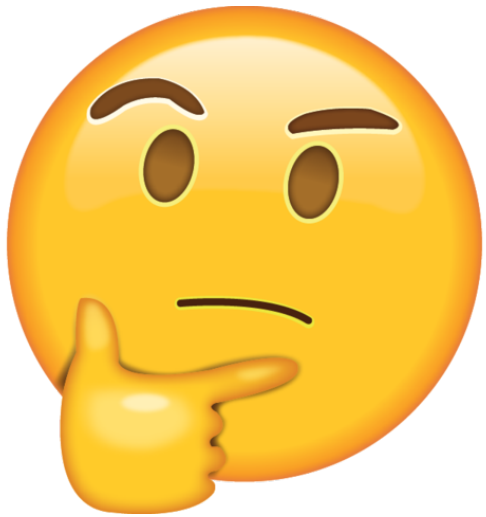
Now That We Can Free Anything



What should we free?

Monads Applicatives Functors Monoids Semigroups

Now That We Can Free Anything



What should we free?

Monads Applicatives Functors Monoids Semigroups Groups

Credit Where It's Due

- Once upon a time: https://engineering.wingify.com/posts/Free-objects/
- use **free boolean algebra** to define DSL for event predicates
- credits to Chris Stucchio (@stucchio)

Let’s Free A Boolean Algebra

- DSL: and, or, not, true, false
- we know what to do, so let’s go!

- AST
- `inject`
- interpreter
- check laws

Boolean Algebras

```
1    trait BoolAlgebra[A] {
2      def tru: A
3      def fls: A
4
5      def not(value: A): A
6
7      def and(lhs: A, rhs: A): A
8      def or(lhs: A, rhs: A): A
9    }
```

Free Boolean Algebra

```
1   sealed abstract class FreeBool[+A]
2
3   case object Tru extends FreeBool[Nothing]
4   case object Fls extends FreeBool[Nothing]
5
6   case class Not[A](value: FreeBool[A])
7       extends FreeBool[A]
8   case class And[A](lhs: FreeBool[A], rhs: FreeBool[A])
9       extends FreeBool[A]
10  case class Or[A](lhs: FreeBool[A], rhs: FreeBool[A])
11      extends FreeBool[A]
12  case class Inject[A](value: A) extends FreeBool[A]
```

Free Boolean Algebra

```
1   def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
2       implicit B: BoolAlgebra[B]): B = {
3     fb match {
4       case Tru         => B.tru
5       case Fls         => B.fls
6       case Inject(v) => f(v)
7       case Not(v)      => B.not(runFreeBool(v)(f))
8       case Or(lhs, rhs) =>
9         B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
10      case And(lhs, rhs) =>
11        B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
12    }
13  }
```

Free Boolean Algebra

```
1   def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
2       implicit B: BoolAlgebra[B]): B = {
3     fb match {
4       case Tru        => B.tru
5       case Fls        => B.fls
6       case Inject(v) => f(v)
7       case Not(v)     => B.not(runFreeBool(v)(f))
8       case Or(lhs, rhs) =>
9         B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
10      case And(lhs, rhs) =>
11        B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
12    }
13  }
```

Free Boolean Algebra

```
1   def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
2       implicit B: BoolAlgebra[B]): B = {
3     fb match {
4       case Tru         => B.tru
5       case Fls         => B.fls
6       case Inject(v) => f(v)
7       case Not(v)      => B.not(runFreeBool(v)(f))
8       case Or(lhs, rhs) =>
9         B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
10      case And(lhs, rhs) =>
11        B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
12    }
13  }
```

Free Boolean Algebra

```
1   def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
2       implicit B: BoolAlgebra[B]): B = {
3     fb match {
4       case Tru         => B.tru
5       case Fls         => B.fls
6       case Inject(v) => f(v)
7       case Not(v)      => B.not(runFreeBool(v)(f))
8       case Or(lhs, rhs) =>
9         B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
10      case And(lhs, rhs) =>
11        B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
12    }
13  }
```

Free Boolean Algebra

```
1   def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
2       implicit B: BoolAlgebra[B]): B = {
3     fb match {
4       case Tru         => B.tru
5       case Fls         => B.fls
6       case Inject(v) => f(v)
7       case Not(v)      => B.not(runFreeBool(v)(f))
8       case Or(lhs, rhs) =>
9         B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
10      case And(lhs, rhs) =>
11        B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
12    }
13  }
```

Using Free Bool

- that was simple (though boilerplate-y)
- what can we do with our new discovered structure
- reminder: boolean operators
  - true, false
  - and, or
  - xor, implies, nand, nor, nxor

Free Bool Example: Search

Partial Evaluation

- assume: implemented short circuiting
- what if you don’t have all the information?
- partially evaluate predicates
- if evaluates successfully, done
- else, send it on

We Freed Boolean Algebras

- good example of underused free structure
- partial evaluation
- serialize the AST (JSON, Protobuf, Avro, ...)
- exercise: minimize AST representation

Resources

- Free Boolean Algebra by Chris Stucchio
  `https://engineering.wingify.com/posts/Free-objects/`
- Source Code:

Go And Free All The Things!

Introduction

Free Monad

Free Applicative

Free Functor

Free Boolean Algebra

Conclusion

Bonus

- remove `Inject` cases from Monad and Applicative
- apply recipe to `Monoid` and get `List` (hint: laws)
- free Magmas
- define free X using alternative minimal set of ops of the typeclass