Free All The Things

Markus Hauck

codecentric

Free All The Things

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

Goal Of This Talk

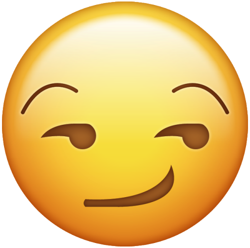
- how many of you wrote a Free X
- how many of you used Free...
  - Monad
  - Applicative
  - Functor
  - other?
- Goal: explain the technique behind “Free X”

The Road Ahead

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 *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
  - create an AST for ops + vars
  - provide a function to “inject” things
  - define an interpreter that eliminates the AST (homomorphism)
  - look at the laws

Why Free

- use Free X as if it was X
- program reified into (data-)structure
- structure can be analyzed/optimized
- one program — many interpretations

Disclaimer Before We Start

- this talk: deep embeddings / initial encoding / data structure representation
- alternative: finally tagless
- not this talk: optimization of free structures

Freeing The Monad

Monad Operations

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

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

Give Me The Laws

```scala
// Left identity
pure(a).flatMap(f) === f(a)

// Right identity
fa.flatMap(pure) === fa

// Associativity
fa.flatMap(f).flatMap(g) ===
  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
  - create an AST for ops + vars
  - provide a function to “inject” things
  - define an interpreter that eliminates the AST (homomorphism)
  - look at the 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

```scala
implicit def freeMonad[F[_], A]: Monad[Free[F, ?]] =
  new Monad[Free[F, ?]] {
    def pure[A](x: A): Free[F, A] = Pure(x)

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

Interpreter

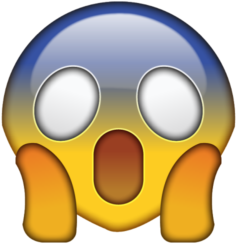
```scala
def runFree[F[_], M[_]: Monad, A](nat: F ~> M)(
    free: Free[F, A]): M[A] = free match {
  case Pure(x)     => Monad[M].pure(x)
  case Inject(fa) => nat(fa)
  case FlatMap(fa, f) =>
    Monad[M].flatMap(runFree(nat)(fa))(x =>
      runFree(nat)(f(x)))
}
```

What about the laws?

```
// The associativity law
FlatMap(FlatMap(fa, f), g) ===
  FlatMap(fa, a => FlatMap(f(a), g))
```

```
val exp1 = FlatMap(FlatMap(fa, f), g)
val exp2 = FlatMap(fa, (a: Int) => FlatMap(f(a), g))

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

```scala
def flatMap[A, B](fa: Free[F, A])(
    f: A => Free[F, B]): Free[F, B] = fa match {
  case Pure(x)     => f(x)
  case Inject(fa) => FlatMap(Inject(fa), f)
  case FlatMap(ga, g) =>
    FlatMap(ga, (a: Any) => FlatMap(g(a), f))
}
```

Use Cases

- 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

- create an AST for ops + vars
- provide a function to “inject” things
- define an interpreter that eliminates the AST (homomorphism)
- look at the laws

The Applicative Class

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

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

AST for FreeApplicative

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

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

final case class Ap[F[_], A, B](
    fab: FreeAp[F, A => B],
    fa: FreeAp[F, A])
    extends FreeAp[F, B]

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

```

Laws

```
// identity
Ap(Pure(identity), v) === v

// composition
Ap(Ap(Ap(Pure(_.compose), u), v), w) ===
  Ap(u, Ap(v, w))

// homomorphism
Ap(Pure(f), Pure(x)) === Pure(f(x))

// interchange
Ap(u, Pure(y)) === Ap(Pure(_(y)), u)
```

Don’t Forget The Laws

```scala
def ap[A, B](fab: FreeAp[F, A => B],
             fa: FreeAp[F, A]): FreeAp[F, B] =
  (fab, fa) match {
    case (Pure(f), Pure(x)) =>
      Pure(f(x)) // homomorphism
    case (u, Pure(y)) =>
      Ap(Pure((f: A => B) => f(y)), u) // interchange
    case (_, _) => Ap(fab, fa)
  }
```

Running FreeApplicatives

```scala
def runFreeAp[F[_], M[_]: Applicative, A](
    nat: F ~> M)(free: FreeAp[F, A]): M[A] =
  free match {
    case Pure(x)     => Applicative[M].pure(x)
    case Inject(fa) => nat(fa)
    case Ap(fab, fa) =>
      Applicative[M]
        .ap(runFreeAp(nat)(fab), runFreeAp(nat)(fa))
  }
```

Freeing The Functor

And Once Again

- create an AST for ops + vars
- provide a function to “inject” things
- define an interpreter that eliminates the AST (homomorphism)
- look at the laws

Freeing The Functor

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

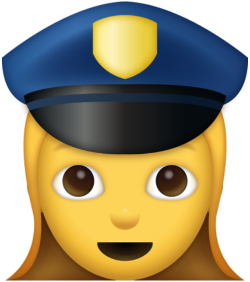
Freeing The Functor

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

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

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

Freeing The Functor

```
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?

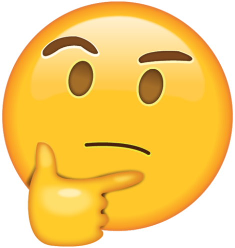
Freeing The Functor

```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
}
```

Now That We Can Free Anything



What should we free?

Disclaimer

- 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)

Free Boolean Algebra

- Wikipedia: boolean algebra + set of generators
- we know what to do, so let's go!

Boolean Algebras

```scala
trait BoolAlgebra[A] {
  def tru: A
  def fls: A

  def not(value: A): A

  def and(lhs: A, rhs: A): A
  def or(lhs: A, rhs: A): A
}
```

Free Boolean Algebra

```scala
sealed abstract class FreeBool[+A]

case object Tru extends FreeBool[Nothing]
case object Fls extends FreeBool[Nothing]

case class Not[A](value: FreeBool[A])
    extends FreeBool[A]
case class And[A](lhs: FreeBool[A], rhs: FreeBool[A])
    extends FreeBool[A]
case class Or[A](lhs: FreeBool[A], rhs: FreeBool[A])
    extends FreeBool[A]
case class Inject[A](value: A) extends FreeBool[A]
```

Free Boolean Algebra

```scala
def runFreeBool[A, B](f: A => B, fb: FreeBool[A])(
    implicit B: BoolAlgebra[B]): B = {
  fb match {
    case Tru        => B.tru
    case Fls        => B.fls
    case Inject(v) => f(v)
    case Not(v)    => B.not(runFreeBool(f, v))
    case Or(lhs, rhs) =>
      B.or(runFreeBool(f, lhs), runFreeBool(f, rhs))
    case And(lhs, rhs) =>
      B.and(runFreeBool(f, lhs), runFreeBool(f, rhs))
  }
}
```

Using Free Bool

- that was easy
- what can we do with our new discovered structure
- DSL: boolean operators
  - true, false
  - and, or
  - xor, implies, nand, nor, nxor

Free Bool Example

Free Bool Optimization

- implement short circuiting (construction vs evaluation)

Partial Evaluation

- what if you don't have all the information?
- partially evaluate predicates
- if evaluates successfully, done
- else, send it on

And What Now?

Free Boolean Algebra

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

Go And Free All The Things!

Can We Minimize Our ASTs?

- remove `Inject` cases from Monad and Applicative