Free All The Things

Markus Hauck @markus1189

Free All The Things

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

This Talk

- explain the technique behind "Free X"
- apply the technique to different examples
- takeaway: it's easier than you thought
- Source Code: `https://github.com/markus1189/free-all-the-things/tree/upnorth`

Let's Get Started

Ready?

What's The Problem

A free functor is left adjoint to a forgetful functor

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 complicated!!!11
- a Free X is the **minimal** structure that **satisfies the laws** of the typeclass X (mine)
- rejoice: there is a pretty mechanical recipe
  - Define AST
  - Add `Inject`
  - Write Interpreter
  - Check laws

Why Free

- nice API using typeclass
- use Free X as if it was X
- lift your DSL ops using `inject` (or `lift`)
- 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, using Free Monads
- for some typeclasses, there are different sets of minimal operations (Monad: three!)
- fun thing: technique works for every one of them, homework

Freeing The Monad

The Monad Typeclass

- we use the `pure` and `flatMap` version

```
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

```
// 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]
}
```

- remember our recipe
  - Define AST
  - Add `Inject`
  - Write 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

```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
fa.flatMap(f).flatMap(g) ===
  fa.flatMap(fa, a => f(a).flatMap(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

Transforming Free Monads: Old Instance

```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)
  }
```

Transforming Free Monads: Optimized flatMap

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

Transforming Free Monads: Optimized flatMap

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

Transforming Free Monads: Done

- what do we learn?
- laws are not boring, they allow refactorings and optimization
- just by adhering to the law, we made our Free Monad more stack safe

```
FlatMap(FlatMap(FlatMap(fa, f), g), ...)
FlatMap(fa, x => FlatMap(f(x), y => FlatMap(g(y), z => ...)))
```

We Freed Monads

- DSL with monadic expressiveness
- context sensitive, branching, loops, fancy control flow
- familiarity with monadic style for DSL

Freeing The Functor

And Once Again

- Define AST
- Add `Inject`
- Write Interpreter
- Check laws

The Functor Typeclass

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

The Functor Laws

```
// identity law
fa.map(identity) === fa

// composition
fa.map(f).map(g) === fa.map(f.compose(g))
```

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

- cool! that was easy
- can we get rid of something?

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

```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

Clean Code Police

- only one subclass?

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)
```

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 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)
- interesting: combine with other structures
- boring interpreter, though
- still fun!

Freeing The Monoid

The Monoid Typeclass

```scala
trait Monoid[A] {
  def empty: A
  def combine(x: A, y: A): A
}
```

The Free Monoid — First Try

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

case object Empty extends FreeMonoid[Nothing]

case class Inject[A](x: A) extends FreeMonoid[A]

case class Combine[A](x: FreeMonoid[A],
                      y: FreeMonoid[A])
    extends FreeMonoid[A]
```

The Laws

```
// left identity
empty |+| x === x

// right identity
x |+| empty === x

// associativity
1 |+| (2 |+| 3) === (1 |+| 2) |+| 3
```

The Laws and Free Monoid

- let's try to enforce those laws in our structure
- goal: correct by construction
- arbitrary decision: associate left vs **right**

Fixing Associativity

```scala
sealed trait NotCombine[+A]

sealed abstract class FreeMonoid[+A]

case object Empty
    extends FreeMonoid[Nothing]
    with NotCombine[Nothing]

case class Inject[A](x: A)
    extends FreeMonoid[A]
    with NotCombine[A]

case class Combine[A](x: NotCombine[A],
                      y: FreeMonoid[A])
    extends FreeMonoid[A]
```

The Problem With Neutral Elements

- get rid completely? not possible
- limit ourselves to a single element
- restrict `Combine` to have only real values on the left side
- goal: minimal canonical structure

Minimizing Structure — Extract Inject

```scala
case class Inject[A](x: A)

sealed abstract class FreeMonoid[+A]

case object Empty extends FreeMonoid[Nothing]

case class Combine[A](x: Inject[A], y: FreeMonoid[A])
    extends FreeMonoid[A]
```

Minimizing Structure — Remove Inject

```
sealed abstract class FreeMonoid[+A]

case object Empty extends FreeMonoid[Nothing]

case class Combine[A](x: A, y: FreeMonoid[A])
    extends FreeMonoid[A]
```

The Monoid Instance

```scala
implicit def monoid[A]: Monoid[FreeMonoid[A]] =
  new Monoid[FreeMonoid[A]] {
    override def empty = Empty
    override def combine(
        x: FreeMonoid[A],
        y: FreeMonoid[A]): FreeMonoid[A] = x match {
      case Empty         => y
      case Combine(h, t) => Combine(h, combine(t, y))
    }
  }
```

Minimizing Structure — Looks Familiar?

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

case object Empty extends FreeMonoid[Nothing]

case class Combine[A](x: A, y: FreeMonoid[A])
    extends FreeMonoid[A]
```

Minimizing Structure — List

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

case object Nil extends List[Nothing]

case class Cons[A](head: A, tail: List[A])
    extends List[A]
```

We Freed Monoids

- DSL for combining things
- works beautifully with folds
- interpretation can be in parallel (associativity)

Now That We Can Free Anything

What should we free?

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

- Define AST
- Add `Inject`
- Write Interpreter
- Check laws

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](fb: FreeBool[A])(f: A => B)(
    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(v)(f))
    case Or(lhs, rhs) =>
      B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
    case And(lhs, rhs) =>
      B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
  }
}
```

Free Boolean Algebra

```scala
def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
    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(v)(f))
    case Or(lhs, rhs) =>
      B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
    case And(lhs, rhs) =>
      B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
  }
}
```

Free Boolean Algebra

```scala
def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
    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(v)(f))
    case Or(lhs, rhs) =>
      B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
    case And(lhs, rhs) =>
      B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
  }
}
```

Free Boolean Algebra

```scala
def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
    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(v)(f))
    case Or(lhs, rhs) =>
      B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
    case And(lhs, rhs) =>
      B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
  }
}
```

Free Boolean Algebra

```scala
def runFreeBool[A, B](fb: FreeBool[A])(f: A => B)(
    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(v)(f))
    case Or(lhs, rhs) =>
      B.or(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
    case And(lhs, rhs) =>
      B.and(runFreeBool(lhs)(f), runFreeBool(rhs)(f))
  }
}
```

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

```scala
sealed trait Search
case class Term(t: String) extends Search
case class After(date: Date) extends Search
case class InText(t: String) extends Search
case class InUrl(url: String) extends Search

// and the usual smart ctors
```

Free Bool Example: Search

- after sneaking in syntactic sugar behind the scenes:

```
val search = term("FP") &
  after("20180101") &
  !(term("Java") | inText("spring")) &
  inUrl("upnorth")
```

The Site Type

```scala
case class Site(terms: List[String],
                url: String,
                indexedAt: Date,
                text: String)
```

- all predicates are run against a collection of these

Free Bool Example: Search

```
def evalSearch(pred: FreeBool[Search])(
    site: Site): Boolean = {
  def nat(s: Search): Boolean = s match {
    case Term(t)           => site.terms.contains(t)
    case After(d)          => site.indexedAt > d
    case InText(t: String) => site.text.contains(t)
    case InUrl(w)          => site.url.contains(w)
  }

  runFreeBool(pred)(nat)
}

val result = Sites.all().filter(evalSearch(search))
```

Free Bool Example: Search

```scala
def evalSearch(pred: FreeBool[Search])(
    site: Site): Boolean = {
  def nat(s: Search): Boolean = s match {
    case Term(t)              => site.terms.contains(t)
    case After(d)             => site.indexedAt > d
    case InText(t: String) => site.text.contains(t)
    case InUrl(w)             => site.url.contains(w)
  }

  runFreeBool(pred)(nat)
}

val result = Sites.all().filter(evalSearch(search))
```

Free Bool Example: Search

```scala
def evalSearch(pred: FreeBool[Search])(
    site: Site): Boolean = {
  def nat(s: Search): Boolean = s match {
    case Term(t)            => site.terms.contains(t)
    case After(d)           => site.indexedAt > d
    case InText(t: String) => site.text.contains(t)
    case InUrl(w)           => site.url.contains(w)
  }

  runFreeBool(pred)(nat)
}

val result = Sites.all().filter(evalSearch(search))
```

Free Bool Example: Search

```scala
def evalSearch(pred: FreeBool[Search])(
    site: Site): Boolean = {
  def nat(s: Search): Boolean = s match {
    case Term(t)            => site.terms.contains(t)
    case After(d)           => site.indexedAt > d
    case InText(t: String)  => site.text.contains(t)
    case InUrl(w)           => site.url.contains(w)
  }

  runFreeBool(pred)(nat)
}

val result = Sites.all().filter(evalSearch(search))
```

Demo Time

But Wait There’s More

- short circuiting and other optimization
- what if you don’t have all the information?
  - partially evaluate predicates
  - if evaluates successfully, done
  - else, send it on
- core language vs extension
  - Chris also demonstrates extension
  - translate a rich language to base instructions
  - with all the advantages

Optimizing Boolean Algebras

```scala
def optimize[A](fa: FreeBool[A]): FreeBool[A] =
  fa match {
    case Tru           => Tru
    case Fls           => Fls
    case Inject(v)     => Inject(v)
    case Not(Not(v))   => v
    case Not(v)        => Not(v)
    case Or(Tru, _)    => Tru
    case Or(_, Tru)    => Tru
    case Or(x, y)      => Or(x, y)
    case And(Fls, _)   => Fls
    case And(_, Fls)   => Fls
    case And(x, y)     => And(x, y)
  }
```

Partial Evaluation

- idea: you might have only partial information
- evaluate as much as possible
- optimal: we can already reduce without needing more information
- otherwise: send it on (JSON, Protobuf, ...)

Partial Evaluation

```scala
def partialEvaluator[A, B](p: FreeBool[A])(
    f: A => Option[B])(implicit B: BoolAlgebra[B])
  : Either[FreeBool[A], B] = p match {
  case Tru        => Right(B.tru)
  case Inject(v) => f(v).toRight(p)
  case Or(lhs, rhs) =>
    val (l, r) = (partialEvaluator(lhs)(f),
                  partialEvaluator(rhs)(f))
    // check if fully evaluated
    ???
  case _ => ???
}
```

Partial Evaluation

```scala
// fulltext not available
case class SiteMetadata(terms: List[String],
                        url: String,
                        indexedAt: Date)
def partially(meta: SiteMetadata)(
    p: Search): Option[Boolean] = p match {
  case Term(t)         => Some(meta.terms.contains(t))
  case After(d)        => Some(meta.indexedAt > d)
  case InUrl(w)        => Some(meta.url.contains(w))
  case InText(t: String) => None
}
```

Demo Time

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: `https://github.com/markus1189/free-all-the-things/tree/upnorth`

Go And Free All The Things!

Markus Hauck (@markus1189)

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

- Define AST
- Add `Inject`
- Write Interpreter
- Check laws

The Applicative Typeclass

```scala
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

```
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))
  }
```

We Freed Applicatives

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

Going Deeper

- try to encode one of the normal forms for boolean algebras
- try to remove `Inject` cases from Monad and Applicative
- free Magmas
- define free X using alternative minimal set of ops of the typeclass