

# 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
- Source Code: <https://github.com/markus1189/free-all-the-things/tree/lambdacnf>

# 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

```
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  }
```

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

# Freeing The Monad

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

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



# Clean Code Police



- only one subclass?

# Freeing The Functor

```
1  sealed abstract class Fmap[F[_], A] {  
2    type X  
3    def fa: F[X]  
4    def f: X => A  
5  }  
6  
7  def inject[F[_], A](v: F[A]) = new Fmap[F, A] {  
8    type X = A  
9    def fa = v  
10   def f = identity  
11 }
```

# Freeing The Functor

```
1  sealed abstract class Coyoneda[F[_], A] {  
2    type X  
3    def fa: F[X]  
4    def f: X => A  
5  }  
6  
7  def inject[F[_], A](v: F[A]) = new Coyoneda[F, A] {  
8    type X = A  
9    def fa = v  
10   def f = identity  
11 }
```

# 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 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

```
1  def runCoyo[F[_]: Functor, A](  
2      coyo: Coyoneda[F, A]): F[A] =  
3      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!



# Freeing The Monoid

# The Monoid Typeclass

```
1  trait Monoid[A] {  
2    def empty: A  
3    def combine(x: A, y: A): A  
4  }
```

# The Free Monoid — First Try

```
1  sealed abstract class FreeMonoid[+A]
2
3  case object Empty extends FreeMonoid[Nothing]
4
5  case class Inject[A](x: A) extends FreeMonoid[A]
6
7  case class Combine[A](x: FreeMonoid[A],
8                        y: FreeMonoid[A])
9      extends FreeMonoid[A]
```

# The Laws

```
1  // left identity
2  empty |+| x === x
3
4  // right identity
5  x |+| empty === x
6
7  // associativity
8  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

```
1  sealed trait NotCombine[+A]
2
3  sealed abstract class FreeMonoid[+A]
4
5  case object Empty
6      extends FreeMonoid[Nothing]
7      with NotCombine[Nothing]
8
9  case class Inject[A](x: A)
10     extends FreeMonoid[A]
11     with NotCombine[A]
12
13 case class Combine[A](x: NotCombine[A],
14                       y: FreeMonoid[A])
15     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

```
1  case class Inject[A](x: A)
2
3  sealed abstract class FreeMonoid[+A]
4
5  case object Empty extends FreeMonoid[Nothing]
6
7  case class Combine[A](x: Inject[A], y: FreeMonoid[A])
8      extends FreeMonoid[A]
```



# Minimizing Structure — Remove Inject

```
1  sealed abstract class FreeMonoid[+A]
2
3  case object Empty extends FreeMonoid[Nothing]
4
5  case class Combine[A](x: A, y: FreeMonoid[A])
6      extends FreeMonoid[A]
```

# The Monoid Instance

```
1  implicit def monoid[A]: Monoid[FreeMonoid[A]] =
2    new Monoid[FreeMonoid[A]] {
3      override def empty = Empty
4      override def combine(
5        x: FreeMonoid[A],
6        y: FreeMonoid[A]): FreeMonoid[A] = x match {
7        case Empty      => y
8        case Combine(h, t) => Combine(h, combine(t, y))
9      }
10   }
```

# Minimizing Structure — List

```
1  sealed abstract class List[+A]
2
3  case object Nil extends List[Nothing]
4
5  case class Cons[A](head: A, tail: List[A])
6      extends List[A]
```

# Now That We Can Free Anything



## What should we free?

Monads

# Now That We Can Free Anything



## What should we free?

Monads Applicatives

# 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   }
```

# 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

```
1  sealed trait Search
2  case class Term(t: String) extends Search
3  case class After(date: Date) extends Search
4  case class InText(t: String) extends Search
5  case class InUrl(url: String) extends Search
6
7  // and the usual smart ctors
```

# Free Bool Example: Search

- assuming some implicits we can write:

```
1 val search = term("FP") &  
2   after("20180101") &  
3   !(term("Java") | inText("spring")) &  
4   inUrl("lambdaconf")
```

# Free Bool Example: Search

```
1  def evalSearch(pred: FreeBool[Search])(
2    site: Site): Boolean = {
3    def nat(s: Search): Boolean = s match {
4      case Term(t)          => site.terms.contains(t)
5      case After(d)         => site.indexedAt > d
6      case InText(t: String) => site.text.contains(t)
7      case InUrl(w)         => site.url.contains(w)
8    }
9
10   runFreeBool(pred)(nat)
11 }
12
13 val result = Sites.all().filter(evalSearch(search))
```

# Free Bool Example: Search

```
1  def evalSearch(pred: FreeBool[Search])(  
2    site: Site): Boolean = {  
3    def nat(s: Search): Boolean = s match {  
4      case Term(t)           => site.terms.contains(t)  
5      case After(d)          => site.indexedAt > d  
6      case InText(t: String) => site.text.contains(t)  
7      case InUrl(w)          => site.url.contains(w)  
8    }  
9  
10   runFreeBool(pred)(nat)  
11 }  
12  
13 val result = Sites.all().filter(evalSearch(search))
```

# Free Bool Example: Search

```
1  def evalSearch(pred: FreeBool[Search])(  
2    site: Site): Boolean = {  
3    def nat(s: Search): Boolean = s match {  
4      case Term(t)           => site.terms.contains(t)  
5      case After(d)          => site.indexedAt > d  
6      case InText(t: String) => site.text.contains(t)  
7      case InUrl(w)          => site.url.contains(w)  
8    }  
9  
10   runFreeBool(pred)(nat)  
11 }  
12  
13 val result = Sites.all().filter(evalSearch(search))
```

# Free Bool Example: Search

```
1  def evalSearch(pred: FreeBool[Search])(  
2    site: Site): Boolean = {  
3    def nat(s: Search): Boolean = s match {  
4      case Term(t)           => site.terms.contains(t)  
5      case After(d)          => site.indexedAt > d  
6      case InText(t: String) => site.text.contains(t)  
7      case InUrl(w)          => site.url.contains(w)  
8    }  
9  
10   runFreeBool(pred)(nat)  
11 }  
12  
13 val result = Sites.all().filter(evalSearch(search))
```



# 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

# 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/lambdaconf>

# Go And Free All The Things!

Markus Hauck (@markus1189)

Introduction

Free Monad

Free Applicative

Free Functor

Free Monoid

Free Boolean Algebra

Conclusion

# Bonus

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