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”

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 (laws!)
  - provide a function to “inject” things
  - define an interpreter that eliminates the AST (homomorphism)

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

Freeing The Monad

Monad Operations

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

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

```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

```
1   def runFree[F[_], M[_]: Monad, A](
2       nat: FunctionK[F, M])(free: Free[F, A]): M[A] =
3     free match {
4       case Pure(x)     => Monad[M].pure(x)
5       case Inject(fa) => nat(fa)
6       case FlatMap(fa, f) =>
7         Monad[M].flatMap(runFree(nat)(fa))(x =>
8           runFree(nat)(f(x)))
9     }
```

What about the laws?

```
1  // The associativity law
2  fa.flatMap(f).flatMap(g) ===
3    fa.flatMap(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!
- use them to make it faster
- tradeoff: during construction vs during interpretation

Faster Free Monads

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

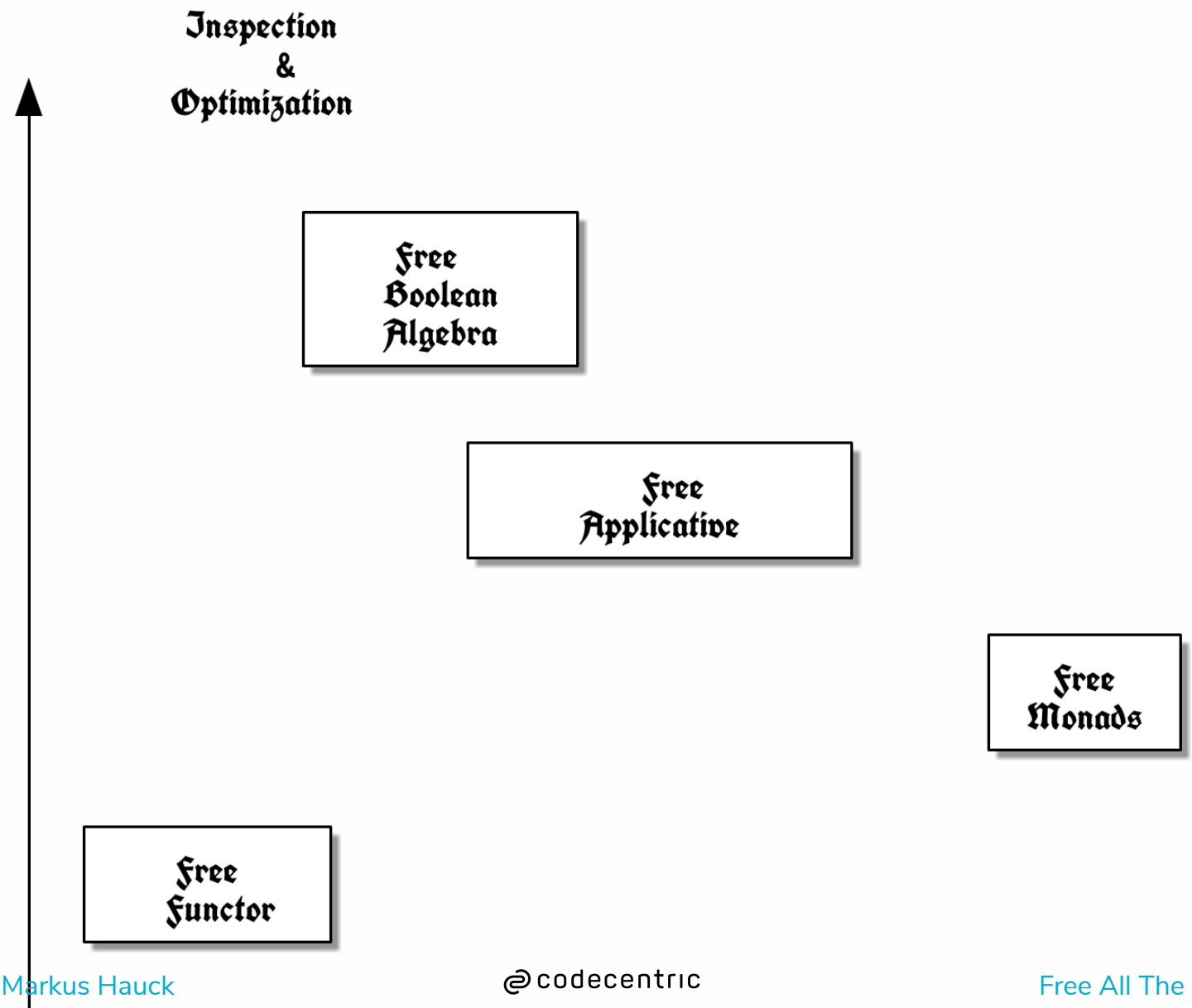
Faster Free Monads

```
1   implicit def freeMonadOpt[F[_]: Functor, A]
2     : Monad[Free[F, ?]] =
3     new Monad[Free[F, ?]] {
4       def pure[A](x: A): Free[F, A] = Pure(x)
5
6       def flatMap[A, B](fa: Free[F, A])(
7           f: A => Free[F, B]): Free[F, B] = fa match {
8         case Pure(x)     => f(x)
9         case Inject(fa) => FlatMap(Inject(fa), f)
10        case FlatMap(ga, g) =>
11          FlatMap(ga, (a: Any) => FlatMap(g(a), f))
12      }
13    }
```

Use Cases

- DSL with monadic expressiveness
- branching, loops, basically everything

Tradeoffs


Inspection
&
Optimization
Free
Boolean
Algebra
Free
Applicative
Free
Monads
Free
Functor

Freeing The Monad

- that’s it for the Monad
- what else?

Freeing The Applicative

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

Freeing The Applicative

- we follow the same pattern
- look at typeclass operations
- create datastructure
- “interpreter”

The Applicative Class

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

Freeing The Applicative

- again the same pattern: we model it as an ADT

```
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]
```

```
1
```

- of course we also need the interpreter

Less Power?!

- why would we consider Applicative if it's less powerful?
- less is more: we can inspect the AST

Freeing The Functor

- we are well equipped by now

Freeing The Functor

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

Freeing The Functor

- clean code alarm: only one subclass
- can we get rid of it?

Disclaimer

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

Free Boolean Algebra

- Wikipedia: boolean algebra + set of generators
- let's go

Boolean Algebras

- seen: common fp type classes
- apply our knowledge to another example: boolean algebras

Your conclusion here