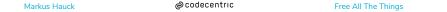
#### Free All The Things

Markus Hauck



### Free All The Things

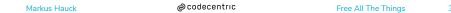
- · well known: free monads
- maybe known: free applicatives
- free monoids
- free <you name it>



#### Goal Of This Talk

- how many of you wrote a Free X
- how many of you used Free...
  - Monad
  - Applicative
  - Functor
  - Boolean Algebra
  - other?
- Goal: explain the technique behind "Free X"
- Be able to apply the "pattern" yourself

Introduction



### What Is Free

A free functor is left adjoint to a forgetful functor What's the problem?



#### What Is Free

- a free X is the minimal thing that satisfies X's laws
- nothing else!



### Why Free

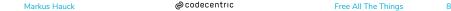
- having a Free X is good for a number of reasons
- use Free X as if it was X
- but the program is reified into some (data-)structure
- this structure can often be analyzed and optimized
- the killer: interpreters of the program can vary

#### Scales of Power

- the structures we will look at, are able to capture computations that have different power abilities
- monad: depend on previous values and branching
- applicative: fixed structure with arbitrary applicative effects in between
- functor: well...
- monoid: limited power, but very flexible and composable
- surprise

### Free Vs Tagless

- we will mostly look at the data structure version of Free X
- the alternative is to use finally tagless representations



what are the operations?

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

- what are the laws?
- (pseudocode)

```
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))
```

- todo: the minimal "thing" that has a Monad instance satisfies the laws
- simple idea: capture as data
- btw: any minimal combination works!

```
trait Monad[F[ ]] {
  def pure[A](x: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A \Rightarrow F[B]): F[B]
```

```
sealed abstract class Free[F[ ], A]
2
   final case class Pure[F[], A](a: A) extends Free[F, A]
3
4
   final case class FlatMap[F[]], A, B](fa: Free[F, A],
5
                                          f: A \Rightarrow Free[F, B]
6
       extends Free[F, B]
```

@ codecentric Markus Hauck Free All The Things

```
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)
```

14

- but what about the laws?!
- clearly we are violating all of them!
- we need one more thing: the interpreter

```
def runFree[F[\_], M[\_]:Monad, A](
       nat: FunctionK[F, M]): Free[F, A] \Rightarrow M[A] = ???
2
```

#### The Laws

together with the interpreter, we have to fulfill the laws

```
1 runFree(nat)(pure(a).flatMap(f)) ===
```

```
2 runFree(nat)(f(a))
```

#### So What?

- the laws tell us what "rewriting" is possible
- here: flatMap has to be associative, that means we can re-associate
- why? Let's look at what happens with normal flatMaps



#### **Use Cases**

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

#### Axis of Power

- this is our base
- a lot of expressiveness in the DSL
- at the cost of the things you can do in the interpreter



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

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



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



### The Applicative Class

```
trait Applicative[F[ ]] {
  def pure [A](x: A): F[A]
  def ap[A, B](fab: F[A \Rightarrow B], fa: F[A]): F[B]
```

1

#### Freeing The Applicative

again the same pattern: we model it as an ADT

```
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]
```

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



we are well equipped by now



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

30

# Your conclusion here