

Free All The Things

Markus Hauck @markus1189

@codecentric





# Let's Get Started

Ready?



A free functor is left adjoint to a forgetful functor



# 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
```





# Why Free

- but why should I care at all?
- 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
- finally tagless possible, but transformation/inspection of AST is more complex
- for some typeclasses, there are different sets of minimal operations (Monad: three!)
- talk might include some hand waving, but for your own good
- fun thing: technique works for every one of them, homework



# The Monad Typeclass

- we use the pure and flatMap version

```

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  }

```

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

- the Monad typeclass instance just wraps things up

```

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

- going from our FreeMonad back into regular Monad-Land

```

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

```



# 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
- let's see what we can do here

# Transforming Free Monads: Old Instance

```

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      }

```

# Transforming Free Monads: Optimized flatMap

```

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: Optimized flatMap

```

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: Done

- Quiz: what about `fa.flatMap(pure) === fa`
- 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

```
1 FlatMap(FlatMap(FlatMap(fa, f), g), ...)
2 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

```

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

- cool! that was easy
- next step is to try to minimize the structure
- can we get rid of something?

# 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



# Clean Code Police



- only one subclass?
- what do we currently use the trait for?

# 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)
    
```

# 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 }

```

- wait a moment, does that look familiar?

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



# The Monoid Typeclass

```

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

```

- let's start blindly following our mechanical approach

# 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
- (almost) arbitrary decision: associate left vs **right**
- you will see why we chose to associate to the right soon!

# Fixing Associativity

- so this is okay:

```
1 Combine(Inject(1), Combine(Inject(2), Inject(3)))
```

- but this should not be possible:

```
1 Combine(Combine(Inject(1), Inject(2)), Inject(3))
```

# 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

- still problem: `Combine(Empty, Empty)`
- get rid completely? not possible
- limit ourselves to a single element
- restrict `Combine` to have only `Inject` 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]

```

- but now Inject looks silly...

## 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     }

```

- note: that might start to look familiar

## Minimizing Structure — Looks Familiar?

```

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]
    
```

## 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]
```

# 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)



# 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 }

```



# 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)
- skip trying to encode laws (because I don't know how)
- but as long as target is lawful, morally okay
- 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

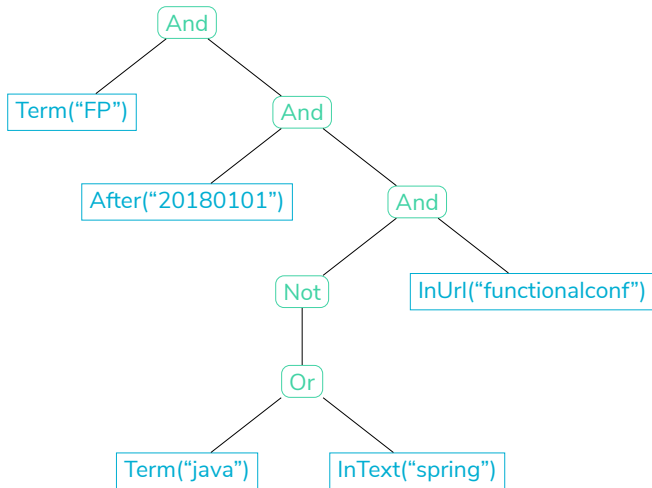
- after sneaking in syntactic sugar behind the scenes:

```

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

```

# Free Bool Example: Search



# The Site Type

```

1  case class Site(terms: List[String],
2                      url: String,
3                      indexedAt: Date,
4                      text: String)

```

- all predicates are run against a collection of these
- we know how to tear down FreeBool, but not yet how to interpret Search values

## 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))

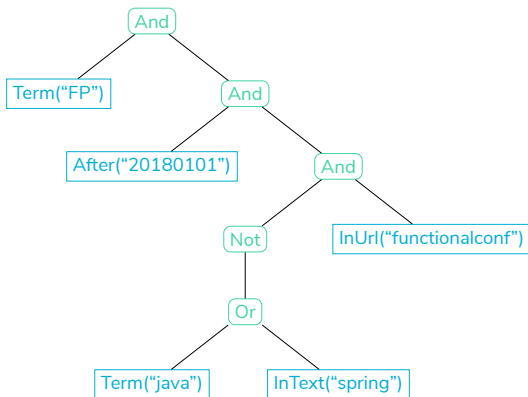
```

# Free Bool Example: Search

```

1  Site(terms = List("FP", "bangalore"),
2      url = "functionalconf.com",
3      indexedAt = "20181212",
4      text = "lots of body")

```

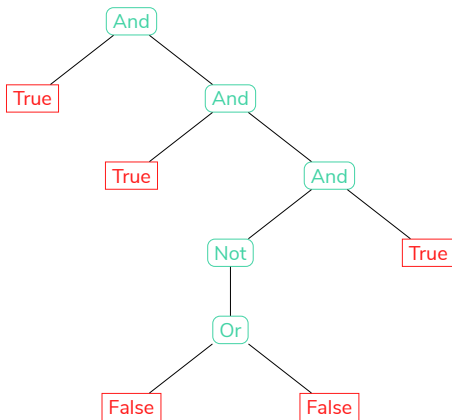


# Free Bool Example: Search

```

1  Site(terms = List("FP", "bangalore"),
2      url = "functionalconf.com",
3      indexedAt = "20181212",
4      text = "lots of body")

```



# 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

```

1  def optimize[A](fa: FreeBool[A]): FreeBool[A] =
2    fa match {
3      case Tru          => Tru
4      case Fls          => Fls
5      case Inject(v)    => Inject(v)
6      case Not(Not(v))  => optimize(v)
7      case Not(v)       => Not(optimize(v))
8      case Or(Tru, _)   => Tru
9      case Or(_, Tru)   => Tru
10     case Or(x, y)      => Or(optimize(x), optimize(y))
11     case And(Fls, _)  => Fls
12     case And(_, Fls)  => Fls
13     case And(x, y)    => And(optimize(x), optimize(y))
14   }

```

# Partial Evaluation

- problem: you might have only partial information
- e.g. full text of indexed website slow to load
- evaluate as much as possible
- optimal: we can already reduce without needing more information
- otherwise: send it on (JSON, Protobuf, ...)

# Partial Evaluation

```

1  def partialEvaluator[A, B](p: FreeBool[A])(
2      f: A => Option[B])(implicit B: BoolAlgebra[B])
3      : Either[FreeBool[A], B] = p match {
4      case Tru          => Right(B.tru)
5      case Inject(v)    => f(v).toRight(p)
6      case Or(lhs, rhs) =>
7          val (l, r) = (partialEvaluator(lhs)(f),
8                          partialEvaluator(rhs)(f))
9          // check if fully evaluated
10         ???
11      case _ => ???
12  }

```



# Partial Evaluation

```

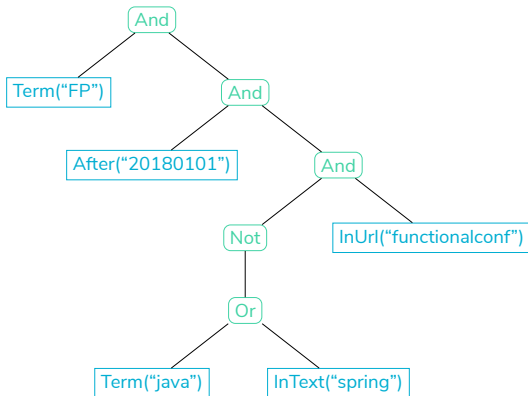
1  case class SiteMetadata(terms: List[String],
2                             url: String,
3                             indexedAt: Date)
4
5  def partially(meta: SiteMetadata)(
6      p: Search): Option[Boolean] = p match {
7      case Term(t)          => Some(meta.terms.contains(t))
8      case After(d)         => Some(meta.indexedAt > d)
9      case InUrl(w)         => Some(meta.url.contains(w))
10     case InText(t: String) => None
11 }

```

# Partial Evaluation

```

1  SiteMetadata(terms = List("FP", "bangalore"),
2      url = "functionalconf.com",
3      indexedAt = "20180929")
    
```

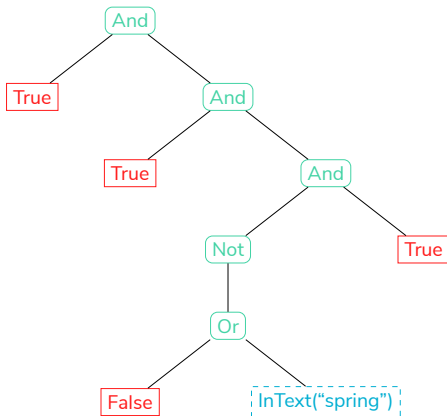


# Partial Evaluation

```

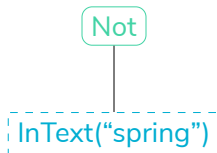
1 SiteMetadata(terms = List("FP", "bangalore"),
2   url = "functionalconf.com",
3   indexedAt = "20180929")

```



# Partial Evaluation

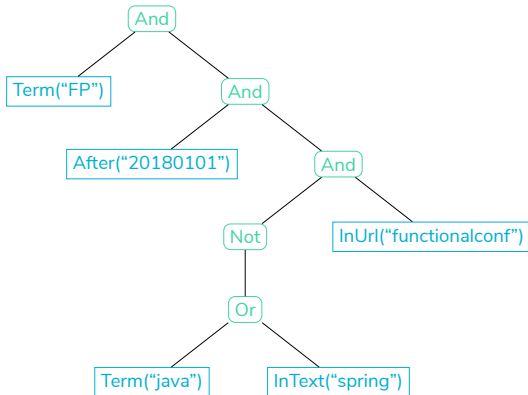
```
1 SiteMetadata(terms = List("FP", "bangalore"),
2   url = "functionalconf.com",
3   indexedAt = "20180929")
```

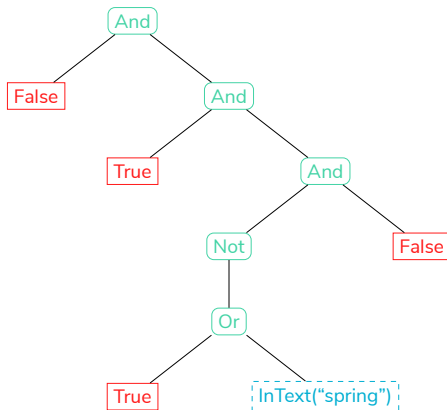


# Partial Evaluation

```

1  SiteMetadata(terms = List("Java", "spring"),
2    url = "spring.io",
3    indexedAt = "20180929")
  
```





# Partial Evaluation

```
1 SiteMetadata(terms = List("Java", "spring"),
2               url = "spring.io",
3               indexedAt = "20180929")
```

False

# We Freed Boolean Algebras

- good example of underused free structure
- partial evaluation
- whole-program optimizations
- serialize the AST (JSON, Protobuf, ...)



# 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/functionalconf>

# Go And Free All The Things!

Markus Hauck (@markus1189)

## Introduction

## Free Monad

## Free Functor

## Free Monoid

# Free Boolean Algebra

## Conclusion

# 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

```
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      }
```

# 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

# 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