

Free monads from scratch

A way to deal with effectful programs

Guillaume Bogard - guillaumebogard.dev

Bonjour ! 🖐️

My name is Guillaume Bogard. I'm a functional programmer, working mostly in Scala.

I love roller-coasters, and Age of Empires.

You can follow me on Twitter @bogardguillaume and on guillaumebogard.dev

Agenda

1. Effects and the substitution model of evaluation
2. Free monads, interpreters, and their benefits
3. Implementing Free monads in Scala and Haskell
4. Free monads *in the real world*

**What is functional programming
about?**

What is the *root benefit* of functional programming?

Functional programming is expression-oriented.

**Evaluating a program is just
evaluating expressions
recursively until there are no
more sub-expressions**

**Expressions are equivalent if they
have the same normal form***

But there's a catch, more on that in a minute

How substitution works

Large expressions are computed by evaluating every sub-expression

```
plusOne a = a + 1
```

```
x = plusOne $ plusOne $ plusOne 16
```

```
x = plusOne $ plusOne 16 + 1
```

```
x = plusOne $ plusOne 17
```

```
x = plusOne 17 + 1
```

```
x = plusOne 18
```

```
x = 18 + 1
```

```
x = 19
```


How substitution works

At any point, I can substitute an expression with its normal form without changing the program.

```
x = plusOne $ plusOne $ plusOne 16
```

is equivalent to

```
x = plusOne $ 18
```

How substitution works

Expressions can be moved around freely without changing the program. In other words, the order of evaluation is irrelevant.

```
a = Musician "John"  
b = Musician "Paul"  
c = Musician "Ringo"  
d = Musician "George"
```

```
song = play [a, b, c, d]
```

How substitution works

Expressions can be moved around freely without changing the program. In other words, the order of evaluation is irrelevant.

```
b = Musician "Paul"  
a = Musician "John"  
d = Musician "George"  
c = Musician "Ringo"
```

```
song = play [a, b, c, d]
```

How substitution works

Expressions can be moved around freely without changing the program. In other words, the order of evaluation is irrelevant.

```
c = Musician "Ringo"  
b = Musician "Paul"  
d = Musician "George"  
a = Musician "John"
```

```
song = play [a, b, c, d]
```

Why substitution matters

The ability to build programs out of freely-manipulable expressions is at the ♥ of what makes FP so appealing

- **Dramatically reduced cognitive load**

Order of execution is irrelevant. Expressions can be reasoned about in complete isolation.

The meaning of a program is mostly inferable from signatures alone.

- **Easy unit tests**

I give you this, you should give me that

- **Unmatched adaptability**

What we call *refactoring* is manipulating expressions without altering the program.

Sprinkle some performance boosts on of that:

- **Fearless concurrency**

Remember, Time isn't a thing anymore

- **Programs could be inspected and optimized ahead of evaluation**

After all, programs are *just data* right ?

In short, substitution is at the ♥ of everything we love about FP

But, remember, there's a catch

Expressions are equivalent if they have the same normal form*

Assuming the program is *referentially transparent

When substitution breaks

We you start putting *side-effects* everywhere, you lose the ability to manipulate expressions freely

```
def greet(name: String): String = { blastAlderaan(); s"Hello $name!" }  
val x = greet("Mike")
```

isn't remotely the same as

```
val x = "Hello Mike!"
```

Signatures, like `greet(name: String): String`, become useless.

Common practices that break substitution

- Using the standard input / output
- Throwing exceptions
- Performing network calls
- Relying on shared mutable state
- Reading and writing to files

And so much more.

**How do build *anything* useful
without breaking substitution ?**

Turning *actions* into *effects*

Turning imperative side-effects into mere *descriptions*, and deferring their execution until the *end of the world*, allows us to freely substitute these descriptions.

We use pure data structures to describe the intended behavior, without running any externally visible effect until we have to.

The type system will help track the nature of the effects, in addition to the type of values our programs produce.

It's just data!TM

- every Clojure developer ever

Example: replacing Exceptions with Either

Instead of altering the control flow of the program, we use a type to encode the *possibility of a failure*.

Substitution still holds.

```
def getSecretContent(user: User) =  
  if(user.role == "admin") Right(42)  
  else Left(Forbidden)
```

```
val result = getSecretContent(  
  User("emma", role = "customer")  
)
```

```
val result = Left(Forbidden)
```

Example: replacing Exceptions with Either

Assumptions are easily verified

```
getSecretContent(User("emma", role = "customer")) shouldBe Left(Forbidden)
```


Why effects matter

When *side-effects* are turned into declarative structures tracked by the type system, we don't call them side-effect anymore, but *effects*.

Effects matter because they enable solutions to every computing problem, while satisfying the substitution principle

Turning arbitrary instructions into effects

The IO monad can turn any set of instructions into a referentially-transparent value.

Substitution holds until the IO is ran, which usually happens *at the end of the world*.





```
def greetUser(name: String) = IO {  
  println(s"Hello $name!")  
}
```

```
def run = greetUser("Hans") >> exit
```

```
def run = IO {  
  println(s"Hello Hans!")  
} >> exit
```

Why IO isn't always ideal

Remember why we care about substitution in the first place.

-  Can we safely move expressions around and exchange them for their normal forms? Absolutely.
-  Does the type system tell us about the nature of our program's effects? Kind of
-  Can we inspect our program and optimize it before it is evaluated? Not really
-  Is it easy to unit-test our program? Nope

Can we do better ?

**Well, it would be a disappointing talk if we couldn't,
right?**

II

Free monads and their interpreters

What is a Free monad?

A *Free monad* (in computer science, not necessarily in category theory) is a structure of type `Free f a` that gives you a monad for any functor `f`.

In short, it can turn any functor into a monad.

How is it useful ?

A free monad allows building programs using the terms of a *domain-specific language (DSL)*.

Said programs are pure, freely-inspectable data structures that don't produce anything.

They are meant to be *interpreted* into another monad that will actually produce the values we want.

How is it useful ?

Free enables us to decouple the description of any program from its actual evaluation.

The type system will tell us about the specific DSL in which the program is defined.

💡 As a bonus, we can change interpreters without affecting our business logic, making Free a suitable alternative to the *tagless final encoding*.

How Free works

1. Define a DSL F that will describe the capabilities of our program
2. Define *smart constructors* that will lift terms of the DSL into the Free context
3. Build a program using these constructors and $\gg=$
You can inspect and test your program without performing any effect
4. Interpret your program into another monad (usually IO) using a natural transformation from F to IO

A Free monad for subscriptions management

Haskell samples use the `free` package, Scala samples use Cats

We start by defining our DSL

```
sealed trait UserStoreDsl[T]

case class GetUser(id: UserId)
  extends UserStoreDsl[User]

case class GetSubscription(userId: UserId)
  extends UserStoreDsl[Option[Subscription]]

case class DeleteSubscription(id: SubId)
  extends UserStoreDsl[Unit]

case class Subscribe(user: User)
  extends UserStoreDsl[Subscription]
```

```
data UserStoreDsl next
  = GetUser UserId (User -> next)
  | GetSubscription
    UserId
    (Maybe Subscription -> next)
  | DeleteSubscription SubId next
  | Subscribe User (Subscription -> next)
  deriving (Functor)

type UserStore = Free UserStoreDsl
```

Then the smart constructors

```
def getUser(id: UserId): UserStore[User] =  
  Free.liftF(GetUser(id))
```

```
def getSubscription(id: UserId)  
  : UserStore[Option[Subscription]] =  
  Free.liftF(GetSubscription(id))
```

```
def deleteSubscription(id: SubId)  
  : UserStore[Unit] =  
  Free.liftF>DeleteSubscription(id))
```

```
def subscribe(user: User)  
  : UserStore[Subscription] =  
  Free.liftF(Subscribe(user))
```

```
getUser :: UserId -> UserStore User  
getUser uid = liftF (GetUser uid id)
```

```
getSubscription ::  
  UserId -> UserStore (Maybe Subscription)  
getSubscription uid =  
  liftF (GetSubscription uid id)
```

```
deleteSubscription :: SubId -> UserStore ()  
deleteSubscription subId =  
  liftF (DeleteSubscription subId ())
```

```
subscribe :: User -> UserStore Subscription  
subscribe u = liftF (Subscribe u id)
```

Then we use our DSL to build a program

```
def updateSubscription(userId: UserId)
  : UserStore[Unit] =
  for {
    user <- getUser(userId)
    oldSub <- getSubscription(userId)
    _ <- oldSub match {
      case Some(sub) =>
        deleteSubscription(sub.id)
      case None      =>
        ().pure[UserStore]
    }
    _ <- subscribe(user)
  } yield ()
```

```
updateSubscription
  :: UserId -> UserStore ()

updateSubscription userId = do
  user <- getUser userId
  oldSub <- getSubscription userId
  case oldSub of
    Just (Subscription id) ->
      deleteSubscription id
    Nothing -> pure ()
  subscribe user
  return ()
```

We then define an interpreter for our DSL

```
val userStoreCompiler = new (UserStoreDsl ~> IO) {  
  def apply[A](fa: UserStoreDsl[A]) = fa match {  
    case GetUser(id) =>  
      // Fetch user from database  
      User(id).pure[IO].map(_.asInstanceOf[A])  
  
    case GetSubscription(UserId("123")) =>  
      Subscription(SubId("1"))  
        .some.pure[IO].map(_.asInstanceOf[A])  
  
    case GetSubscription(_) =>  
      Option.empty  
        .pure[IO].map(_.asInstanceOf[A])  
  
    case DeleteSubscription(_) =>  
      IO.unit.map(_.asInstanceOf[A])  
  
    case Subscribe(_) =>  
      Subscription(SubId("new-sub"))  
        .pure[IO].map(_.asInstanceOf[A])  
  }  
}
```

```
type f ~> g = forall x. f x -> g x
```

```
userStoreInterpreter :: UserStoreDsl ~> IO  
userStoreInterpreter (GetUser id next) = do  
  -- Fetch user from database  
  let user = User (UserId "123")  
  pure (next user)
```

```
userStoreInterpreter  
  (GetSubscription (UserId "123") next) = do  
    let sub = Subscription (SubId "1")  
    pure (next (Just sub))
```

```
userStoreInterpreter (GetSubscription _ next) =  
  pure (next Nothing)
```

```
userStoreInterpreter (DeleteSubscription _ next) =  
  pure next
```

```
userStoreInterpreter (Subscribe _ next) = do  
  let sub = Subscription (SubId "new-sub")  
  pure (next sub)
```

Finally, we can interpret our program *into the IO monad*.
We've built a PL inside our PL!

```
val program: IO[Unit] =  
  updateSubscription(UserId("123"))  
    .foldMap(userStoreCompiler)
```

```
result :: IO ()  
result = foldFree userStoreInterpreter  
  (updateSubscription (UserId "123"))
```


Now the type of [the program] tells us exactly what kind of effects it uses: a much healthier situation than a single monolithic IO monad. For example, our types guarantee that executing a term in the Term Teletype monad will not overwrite any files on our hard disk. The types of our terms actually have something to say about their behavior!

Wouter Swierstra, *Data Types à la carte*



Implementing Free monads

Based on Wouter Swierstra's paper
Data Types à la carte

Abstract

This paper describes a technique for assembling both data types and functions from isolated individual components. We also explore how the same technology can be used to combine free monads and, as a result, structure Haskell's monolithic IO monad.

1 Introduction

Implementing an evaluator for simple arithmetic expressions in Haskell is entirely straightforward.

```
data Expr = Val Int | Add Expr Expr
eval :: Expr -> Int
eval (Val x)    = x
eval (Add x y)  = eval x + eval y
```

Once we have chosen our data type, we are free to define new functions over expressions. For instance, we might want to render an expression as a string:

```
render :: Expr -> String
render (Val x)    = show x
render (Add x y)  = "(" ++ render x ++ " + " ++ render y ++ ")"
```

If we want to add new operators to our expression language, such as multiplication, we are on a bit of a sticky wicket. While we could extend our data type for expressions, this will require additional cases for the functions we have defined so far. Phil Wadler (1998) has dubbed this in the *Expression Problem*:

The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

As the above example illustrates, Haskell can cope quite nicely with new function definitions; adding new constructors, however, forces us to modify existing code.

In this paper, we will examine one way to address the Expression Problem in Haskell. Using the techniques we present, you can define data types, functions, and even certain monads in a modular fashion.

What is *Data types à la carte*?

The paper describes a technique for extending data types and functions without affecting existing code, solving P. Wadler's *Expression problem*.

It later shows how the same technique can be used to constrain programs into a specific set of effects, rather than the monolithic IO.

What is the expression problem?

Let `Expr` be a sum type, describing the expressions of a very simple calculator that can only add integers

```
data Expr = Val Int
          | Add Expr Expr
```

```
sealed trait Expr

case class Val(value: Int)
  extends Expr
case class Add(a: Expr, b: Expr)
  extends Expr
```

What is the expression problem?

To extend the abilities of the calculator to add multiplication, one could extend Expr

```
data Expr = Val Int
          | Add Expr Expr
          | Multiply Expr Expr
```

⚠ However, this would force us to adjust and recompile any existing code that uses that uses this data type.

In short, the expression problem is about extending data types and functions in a *backward-compatible* way

Swierstra's solution: a polymorphic Expr type.

We could have a polymorphic data type Expr, where the type parameter, f constraints the type of expressions happening in the subtree.

```
data Expr f = In (f (Expr f))    case class Expr[F[_]](in: F[Expr[F]])
```

Then we can define data types for expressions consisting of integers, or additions of sub-expressions.

Val doesn't actually use its type parameter because it accepts no sub-expression.

```
data Val e = Val Int
type ValExpr = Expr Val
```

```
data Add e = Add e e
type AddExpr = Expr Add
```

```
case class Val[T](value: Int)
type ValExpr = Expr[Val]
```

```
case class Add[T](a: T, b: T)
type AddExpr = Expr[Add]
```

Combining data types is achieved using their coproducts.
The coproduct works like `Either` for type constructors.

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

The Scala version is quite more verbose:

```
sealed trait Coproduct[F[_], G[_], A]
```

```
case class L[A, F[_], G[_]](in: F[A]) extends Coproduct[F, G, A]
```

```
case class R[A, F[_], G[_]](in: G[A]) extends Coproduct[F, G, A]
```


Using the coproduct, we can build programs made both Val and Add expressions:

```
type ValOrAddExpr = Expr (Val :+: Add)

program :: ValOrAddExpr
program =
  In
    ( Inr
      ( Add
        (In (Inl (Val 10)))
        ( In
          ( Inr
            ( Add
              (In (Inl (Val 2)))
              (In (Inl (Val 5)))
            )
          )
        )
      )
    )
```

```
type ValOrAdd[T] = Coproduct[Val, Add, T]
type ValOrAddExpr = Expr[ValOrAdd]

val program: ValOrAddExpr = Expr(
  R(
    Add(
      Expr(L(Val(10))),
      Expr(
        R(
          Add(
            Expr(L(Val(2))),
            Expr(L(Val(5)))
          )
        )
      )
    )
  )
```

Before we can evaluate the program, we must observe that `Val` and `Add` are both functors

```
instance Functor Val where
  fmap f (Val x) = Val x

instance Functor Add where
  fmap f (Add a b) = Add (f a) (f b)
```

```
implicit val valFunctor: Functor[Val] =
  new Functor[Val] {
    def map[A, B](fa: Val[A])(f: A => B) =
      Val(fa.value)
  }

implicit val addFunctor: Functor[Add] =
  new Functor[Add] {
    def map[A, B](fa: Add[A])(f: A => B) =
      Add(f(fa.a), f(fa.b))
  }
```

And also that the coproduct of two functors is itself a functor.

```
instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (Inl term) = Inl (fmap f term)
  fmap f (Inr term) = Inr (fmap f term)
```

```
implicit def coproductFunctor[F[_], G[_]](
  implicit functorF: Functor[F],
  functorG: Functor[G]
): Functor[Coproduct[F, G, *]] = new Functor[Coproduct[F, G, *]] {
  def map[A, B](fa: Coproduct[F, G, A])(f: A => B): Coproduct[F, G, B] =
    fa match {
      case L(expr) => L[B, F, G](functorF.fmap(expr)(f))
      case R(expr) => R[B, F, G](functorG.fmap(expr)(f))
    }
}
```

This means that `Val :+: Add` is a functor

Our data types being functors is the precondition for evaluation.

Given `Expr f` and a function `F a -> a`, we can define a fold that will evaluate the expressions recursively until we get `a`.

```
foldExpr :: Functor f => (f a -> a) -> Expr f -> a
foldExpr f (In term) = f (fmap (foldExpr f) term)
```

```
def foldExpr[F[_]: Functor, T](eval: F[T] => T)(expr: Expr[F]): T =
  eval(expr.in.map(foldExpr(eval)))
```

All what's left to do now is specify how a single step of recursion should be evaluated:

```
evalVal :: Val t -> Int
evalVal (Val number) = number
```

```
evalAdd :: Add Int -> Int
evalAdd (Add a b) = a + b
```

```
evalValOrAdd :: (Val :+: Add) Int -> Int
evalValOrAdd (Inl t) = evalVal t
evalValOrAdd (Inr t) = evalAdd t
```

```
def evalVal[T](term: Val[T]): Int =
  term.value
def evalAdd(term: Add[Int]): Int =
  term.a + term.b
def evalValOrAdd(term: ValOrAdd[Int]) =
  term match {
    case L(term) => evalVal(term)
    case R(term) => evalAdd(term)
  }
```

Note: In his paper, Swierstra defines a type class called `Eval` to derive instances for coproducts automatically. I have elided it in favor of a more concise example.

When we put everything together, we can evaluate programs of type `Val` `:+:` Add integers.

```
result :: Int
```

```
result = foldExpr evalValOrAdd program
```

```
val result: Int = foldExpr(evalValOrAdd)(program)
```

The type of our program tells us exactly what operations are supported.

Generalizing to Free monads

This way of folding over functors can be generalized to free monads to build more complex programs.

Free monads are monads of the form

```
data Free f a = Pure a
              | Impure (f (Free f a))
```

consisting of pure values and impure effects, constructed using a functor `f`.

The techniques we've shown before can be generalized to Free monads, to build complex programs out of simple data types.

When `f` is a functor, `Free f` is a monad, as shown by these instances:

```
instance Functor f => Functor (Free f) where
  fmap f (Pure x) = Pure (f x)
  fmap f (Impure x) = Impure (fmap (fmap f) x)
```

```
instance Functor f => Applicative (Free f) where
  pure x = Pure x
  (<*>) = ap
```

```
instance Functor f => Monad (Free f) where
  (Pure x) >>= f = f x
  (Impure x) >>= f = Impure (fmap (>>= f) x)
```

In short, a `Free` monad gives us a monad out of any functor.

```
implicit def freeFunctor[F[_]: Functor] =
  new Functor[Free[F, *]] {
    def map[A, B](fa: Free[F, A])(f: A => B) = fa match {
      case Pure(x) => Pure(f(x))
      case Impure(x) => Impure(x.map(_.map(f)))
    }
  }
```

```
implicit def freeMonad[F[_]: Functor] =
  new Monad[Free[F, *]] {

    def map[A, B](fa: Free[F, A])(f: A => B) =
      freeFunctor[F].fmap(fa)(f)
```

```
    def flatMap[A, B](fa: Free[F, A])
      (f: A => Free[F, B]) = fa match {
      case Pure(x) => f(x)
      case Impure(x) => Impure(x.map(_.flatMap(f)))
    }
  }
```

```
    def pure[A](x: A): Free[F, A] = Pure(x)
  }
```

Just like earlier, we can define a simple DSL, along with some *smart constructors*:

```
data UserStoreDsl next
  = GetUser UserId (User -> next)
  | Subscribe User next
  deriving (Functor)

type UserStore = Free UserStoreDsl

getUser :: UserId -> UserStore User
getUser id = Impure (GetUser id Pure)

subscribe :: User -> UserStore ()
subscribe user =
  Impure (Subscribe user (Pure ()))
```

```
sealed trait UserStoreDsl[Next]
case class GetUser[Next](
  id: UserId,
  next: User => Next
) extends UserStoreDsl[Next]

case class Subscribe[Next](user: User, next: Next)
  extends UserStoreDsl[Next]

type UserStore[A] = Free[UserStoreDsl, A]

def getUser(id: UserId): UserStore[User] =
  Impure(GetUser(id, Pure(_)))
def subscribe(user: User): UserStore[Unit] =
  Impure(Subscribe(user, Pure(())))
```

We then use this DSL to build a very simple program.

Because `Free UserStoreDsl` is a monad, we can express dependant computations very easily 🙌🙌

```
freeProgram :: UserStore User
freeProgram = do
  user <- getUser (UserId "123")
  subscribe user
  return user
```

```
val freeProgram: UserStore[User] =
  for {
    user <- getUser(UserId("123"))
    _ <- subscribe(user)
  } yield use
```

Let's evaluate the program! If we had a function $f : a \rightarrow IO\ a$, specifying how a single instruction is evaluated, we could use it to fold over the entire program.

We will encapsulate this functions into a type class:

```
class Functor f => Exec f where  
  exec :: f a -> IO a  
  
trait Exec[F[_]] {  
  def exec[A](fa: F[A]): IO[A]  
}
```

Let's define an Exec instance for our DSL:

```
instance Exec UserStoreDsl where
  exec (GetUser id next) = do
    let user = User id
    return (next user)
  exec (Subscribe user next) = do
    putStrLn
      ("User" <> show user <> " has subscribed")
    return next
```

```
implicit val execUserStore = new Exec[UserStoreDsl] {
  def exec[A](fa: UserStoreDsl[A]): IO[A] = fa match {

    case GetUser(id, next) =>
      IO(User(id)).map(next)

    case Subscribe(user, next) =>
      IO {
        println(s"User $user has subscribed!")
      } as next
  }
}
```

We're almost done! Let's define the fold, and use it to evaluate our program! 🎉

```
execAlgebra :: Exec f => Free f a -> IO a
execAlgebra (Pure x) = return x
execAlgebra (Impure x) =
  exec x >>= execAlgebra
```

-- Finally we run the program

```
freeResult :: IO User
freeResult = execAlgebra freeProgram
```

```
def execAlgebra[F[_]: Functor, A](
  fa: Free[F, A]
)(implicit exec: Exec[F]): IO[A] = fa match {
  case Pure(x)    => IO.pure(x)
  case Impure(x) =>
    exec.exec(x).flatMap(execAlgebra[F, A](x))
}
```

```
val freeResult: IO[User] =
  execAlgebra(freeProgram)
```

IV

Free monads in the real world

Real-world practice #1: mocking

Free monads allow mocking parts of our program selectively.
Mocking is as easy as writing an interpreter.

```
val userStoreCompiler  
  : UserStoreDsl ~> IO = ???
```

```
val mockCompiler  
  : UserStoreDsl ~> IO = {  
  case GetUser("123") => mockUser.pure[IO]  
  case other          => userStoreCompiler(other)  
}
```

```
getUser(UserId("123")).foldMap(mockCompiler).unsafeRunSync() shouldBe mockUser
```


Real-world practice #2: swapping interpreters

The same program can be interpreted using different interpreters, e.g. to provide various storage backends for a data-access layer.

```
sealed trait KeyValueStoreDsl[A]  
case class Put(key: String, value: String) extends KeyValueStore[Unit]  
case class Get(key: String) extends KeyValueStore[Option[String]]  
  
val inMemoryCompiler: KeyValueStoreDsl ~> State[Map[String, String], *] = ???  
val redisCompiler: KeyValueStoreDsl ~> IO = ???  
val irminCompiler: KeyValueStoreDsl ~> IO = ???
```

Real-world practice #3: monadic interface for libraries

Monads reduce the needed amount of documentation and give many operations for free.

Doobie uses free monads to model database transactions on which users can use for-comprehensions, `>>`, `<<`, as etc.

```
val transaction: ConnectionIO[Int] = for {  
  nbUsers <- sql"delete * from users".update.run  
  nbMsgs <- sql"delete * from messages".update.run  
} yield nbUsers + nbMsgs
```

Multiple databases are supported through *transactors* which are ... Free monad interpreters :)

Real-word practice #4: combining DSLs

**Every program should specifically select the operations it needs,
instead of having access to every possible operation.**

Combining DSLs

To combine DSLs, we must recall the coproduct of two functors is itself a functor.

We can use `Free` to get a monad instance for the coproduct of two DSLs, and use both DSLs in the same program.

```
data EmailDsl next
  = SendConfirmationEmail User next
  deriving (Functor)

type UserStoreOrEmail =
  Free (UserStoreDsl :+: EmailDsl)

combinedProgram :: UserId -> UserStoreOrEmail ()
combinedProgram id = do
  user <- Impure (Inl (GetUser id Pure))
  Impure (Inl (Subscribe user (Pure ())))
  Impure (Inr (SendConfirmationEmail user (Pure ())))
  return ()
```

What I haven't covered: ergonomics

Manually *lifting* functors into coproducts is tedious and implies a lot of boilerplate.

Fortunately, the problem is addressed in Swierstra's paper in the chapter *Automating injections*.

One can define a type class, $:<:$, that, given a type and coproduct *knows* how to inject it in the coproduct.

This type class is implemented in *Cats* under the name `Inject`

Acknowledgments 🙏

This talk wouldn't have been possible without the work of these amazing people:

- [Cats](#) contributors for their work on the cats-free module, among everything else.
- [James Haydon](#) with his article [Free monads for cheap interpreters](#)
- [Edward A. Kmett](#) and contributors for the [free](#) Haskell package
- [Adam Rosien](#) with his article, [What is an effect](#) which provides an excellent introduction to effects, and a nice motivation for Free monads
- [Wouter Swierstra](#) with his paper, [Data types à la carte](#) which provides the reference implementation for Free monads, on which this presentation's code is mostly based.

Get in touch 📧

E-mail: hey@guillaumebogard.dev - Homepage: guillaumebogard.dev - Twitter: [@bogardguillaume](https://twitter.com/bogardguillaume)