

# Programs as Values

## Pure Functional Database Access in Scala

Rob Norris • Gemini Observatory



# Programs as Values

## Pure Functional Database Access in Scala

Rob Norris • Gemini Observatory



# What's this about?

This is a talk about the implementation of **doobie**, a principled library for database access in Scala.

# What's this about?

This is a talk about the implementation of **doobie**, a principled library for database access in Scala.

- JDBC programming is **terrible**.

# What's this about?

This is a talk about the implementation of **doobie**, a principled library for database access in Scala.

- JDBC programming is **terrible**.
- Free monads [over free functors] are **awesome**.

# What's this about?

This is a talk about the implementation of **doobie**, a principled library for database access in Scala.

- JDBC programming is **terrible**.
- Free monads [over free functors] are **awesome**.
- Think of programs as **composable** values, rather than imperative processes.

# What's this about?

This is a talk about the implementation of **doobie**, a principled library for database access in Scala.

- JDBC programming is **terrible**.
- Free monads [over free functors] are **awesome**.
- Think of programs as **composable** values, rather than imperative processes.
- Lather, rinse, **repeat**. This is a great strategy for making terrible APIs tolerable.

# The Problem

So what's wrong with this JDBC program?

```
case class Person(name: String, age: Int)

def getPerson(rs: ResultSet): Person = {
    val name = rs.getString(1)
    val age  = rs.getInt(2)
    Person(name, age)
}
```

# The Problem

So what's wrong with this JDBC code?

Managed  
Resource

```
case class Person(name: String, age: Int)

def getPerson(rs: ResultSet): Person = {
    val name = rs.getString(1)
    val age  = rs.getInt(2)
    Person(name, age)
}
```

# The Problem

So what's wrong with this JDBC code?

```
case class Person(name: String, age: Int)

def getPerson(rs: ResultSet): Person = {
    val name = rs.getString(1)
    val age  = rs.getInt(2)
    Person(name, age)
}
```

Managed  
Resource

Side-Effect

# The Problem

So what's wrong with this JDBC code?

```
case class Person(name: String, age: Int)

def getPerson(rs: ResultSet): Person = {
    val name = rs.getString(1)
    val age  = rs.getInt(2)
    Person(name, age)
}
```

Managed  
Resource

Composition?

Side-Effect

# The Strategy

Here is our game plan:

# The Strategy

Here is our game plan:

- Let's talk about primitive operations as **values** ...  
these are the smallest meaningful **programs**.

# The Strategy

Here is our game plan:

- Let's talk about primitive operations as **values** ... these are the smallest meaningful **programs**.
- Let's define rules for **combining** little programs to make bigger programs.

# The Strategy

Here is our game plan:

- Let's talk about primitive operations as **values** ... these are the smallest meaningful **programs**.
- Let's define rules for **combining** little programs to make bigger programs.
- Let's define an **interpreter** that consumes these programs and performs actual work.

# Primitives

An algebra is just a set of objects, along with rules for manipulating them. Here our objects are the set of primitive computations you can perform with a JDBC ResultSet.

```
sealed trait ResultSetOp[A]

case object Next          extends ResultSetOp[Boolean]
case class GetInt(i: Int) extends ResultSetOp[Int]
case class GetString(i: Int) extends ResultSetOp[String]
case object Close         extends ResultSetOp[Unit]
// 188 more...
```

# Primitives

An algebra is just a set of objects, along with rules for manipulating them. Here our objects are the set of primitive computations you can perform with a JDBC ResultSet.

```
sealed trait ResultSetOp[A]  
  
case object Next  
case class GetInt(i: Int)  
case class GetString(i: Int)  
case object Close  
// 188 more...
```



case object Next	extends ResultSetOp[Boolean]
case class GetInt(i: Int)	extends ResultSetOp[Int]
case class GetString(i: Int)	extends ResultSetOp[String]
case object Close	extends ResultSetOp[Unit]

Constructor per Method

# Primitives

An algebra is just a set of objects, along with rules for manipulating them. Here our objects are the set of primitive computations you can perform with a JDBC ResultSet.

```
sealed trait ResultSetOp[A]
```

```
case object Next
case class GetInt(i: Int)
case class GetString(i: Int)
case object Close
// 188 more...
```

Constructor per Method

```
extends ResultSetOp[Boolean]
extends ResultSetOp[Int]
extends ResultSetOp[String]
extends ResultSetOp[Unit]
```

Parameterized  
on Return Type

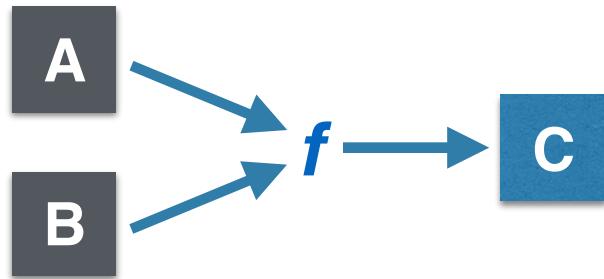
# **Operations**

# Operations

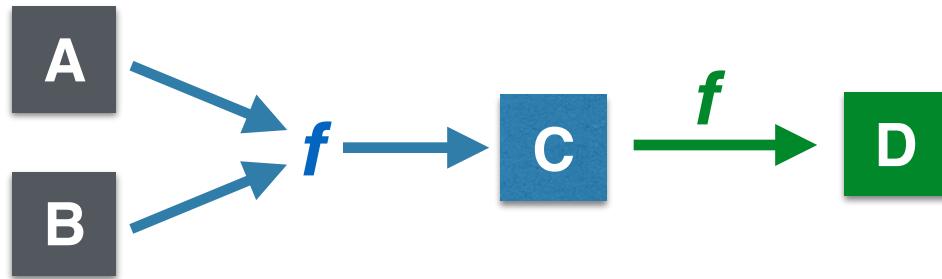
A

B

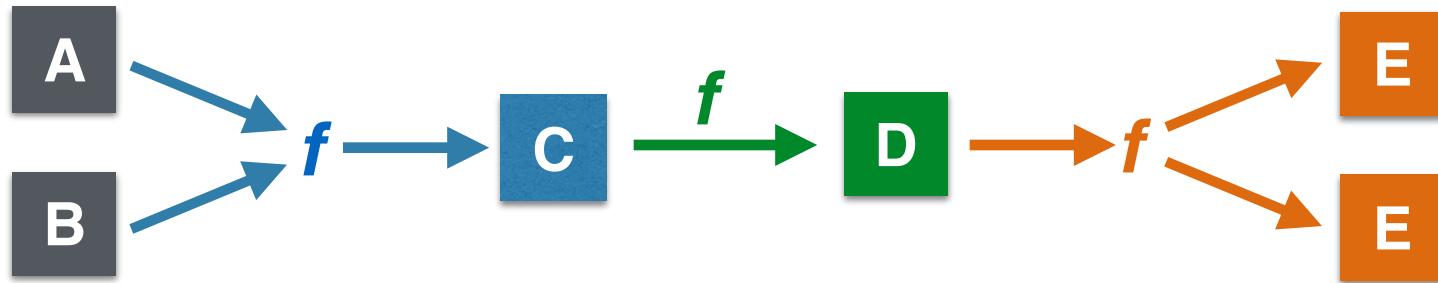
# Operations



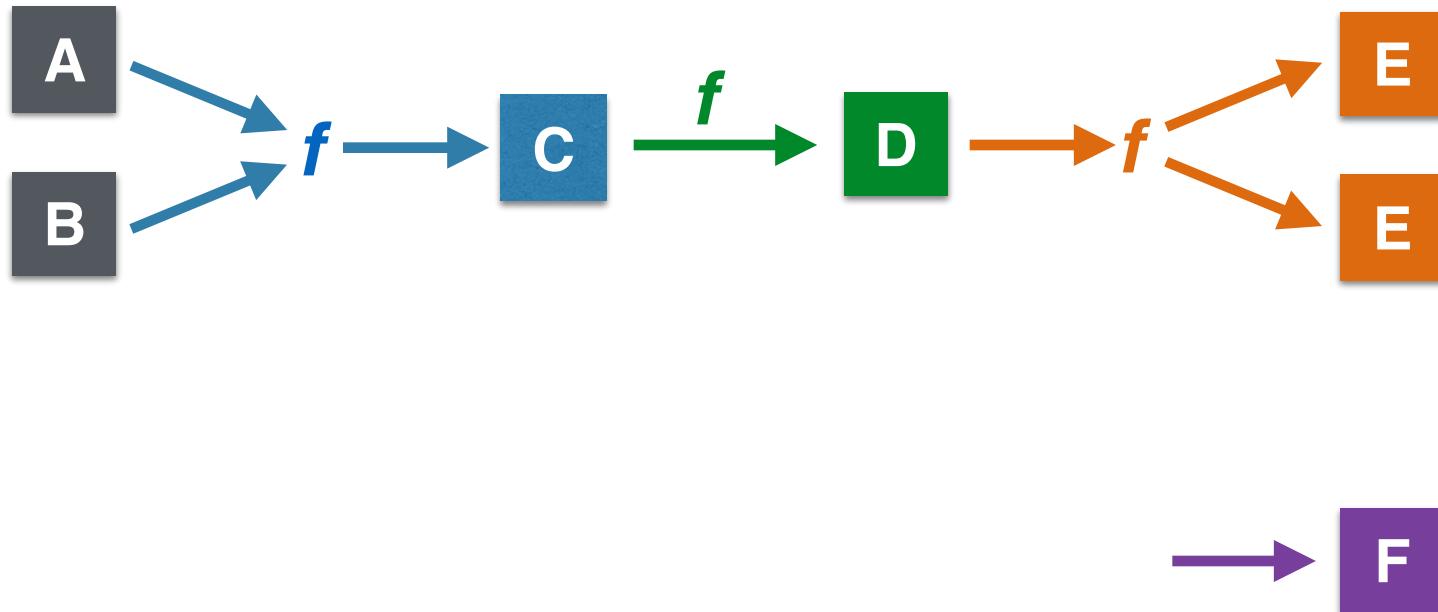
# Operations



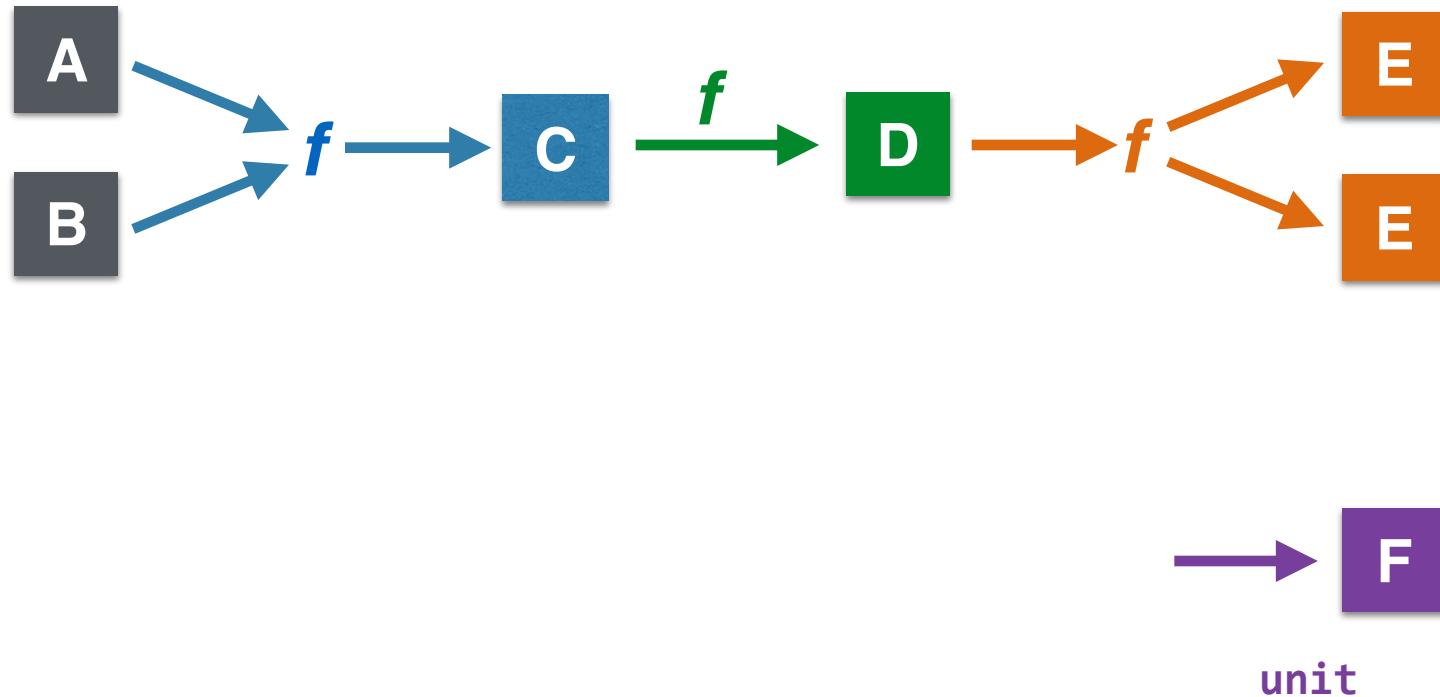
# Operations



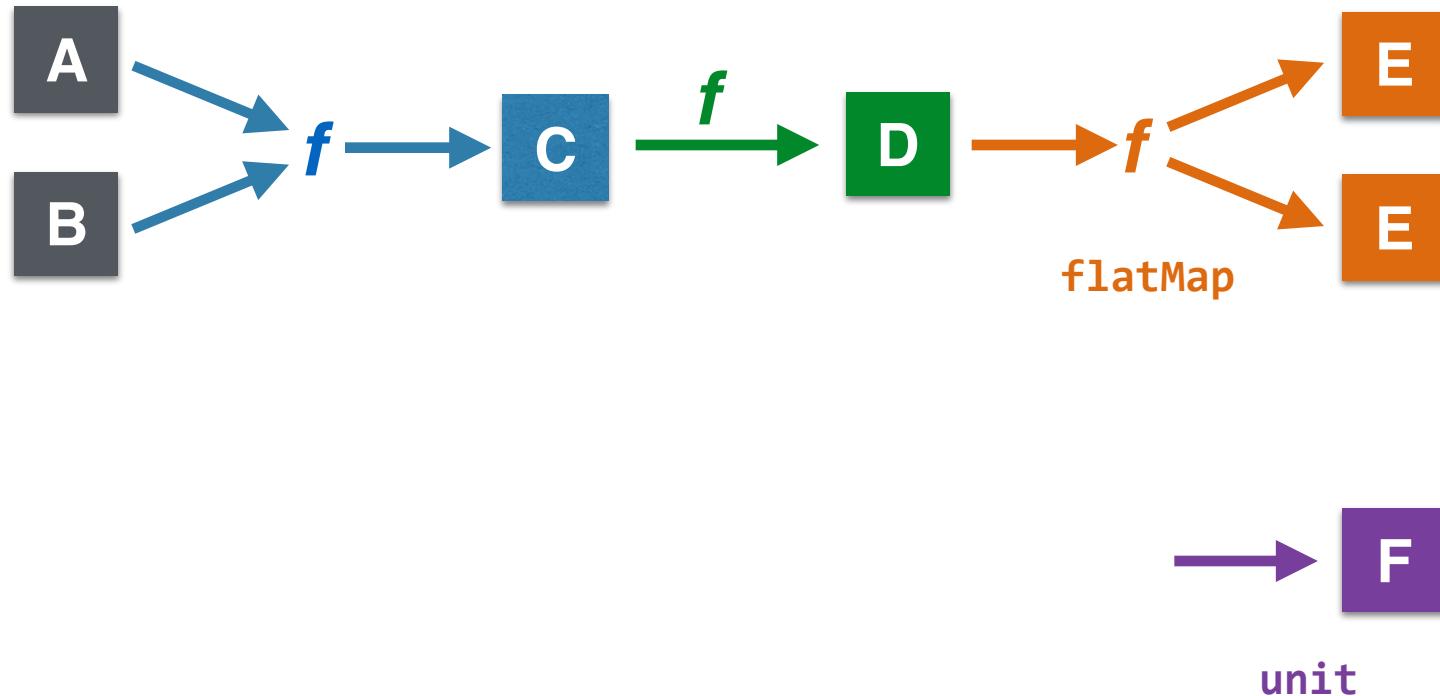
# Operations



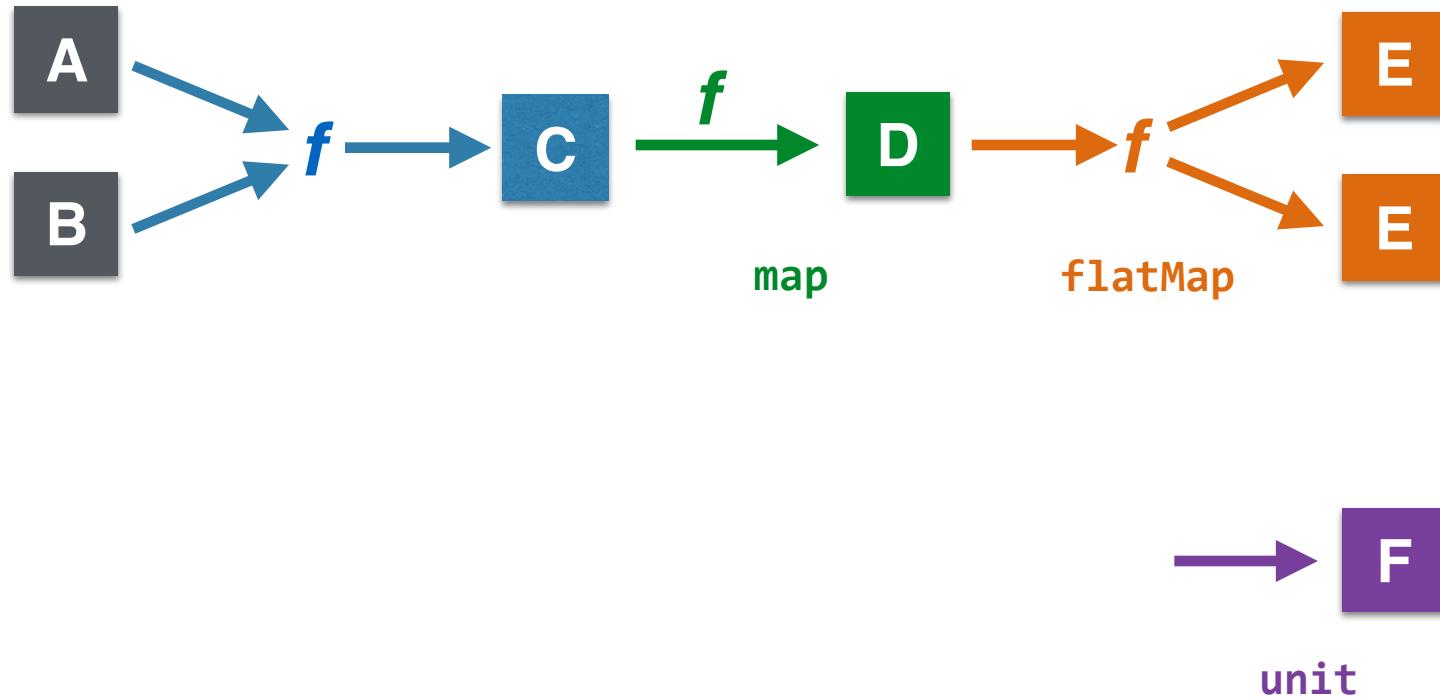
# Operations



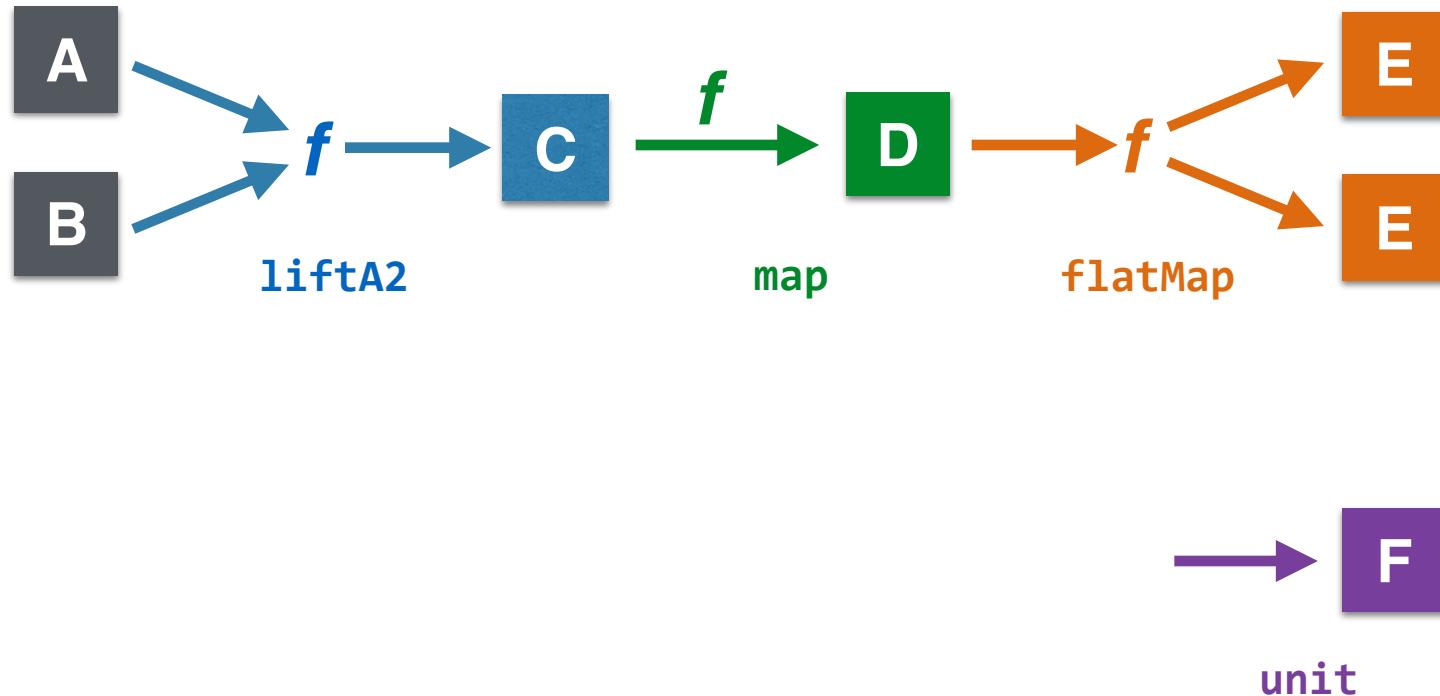
# Operations



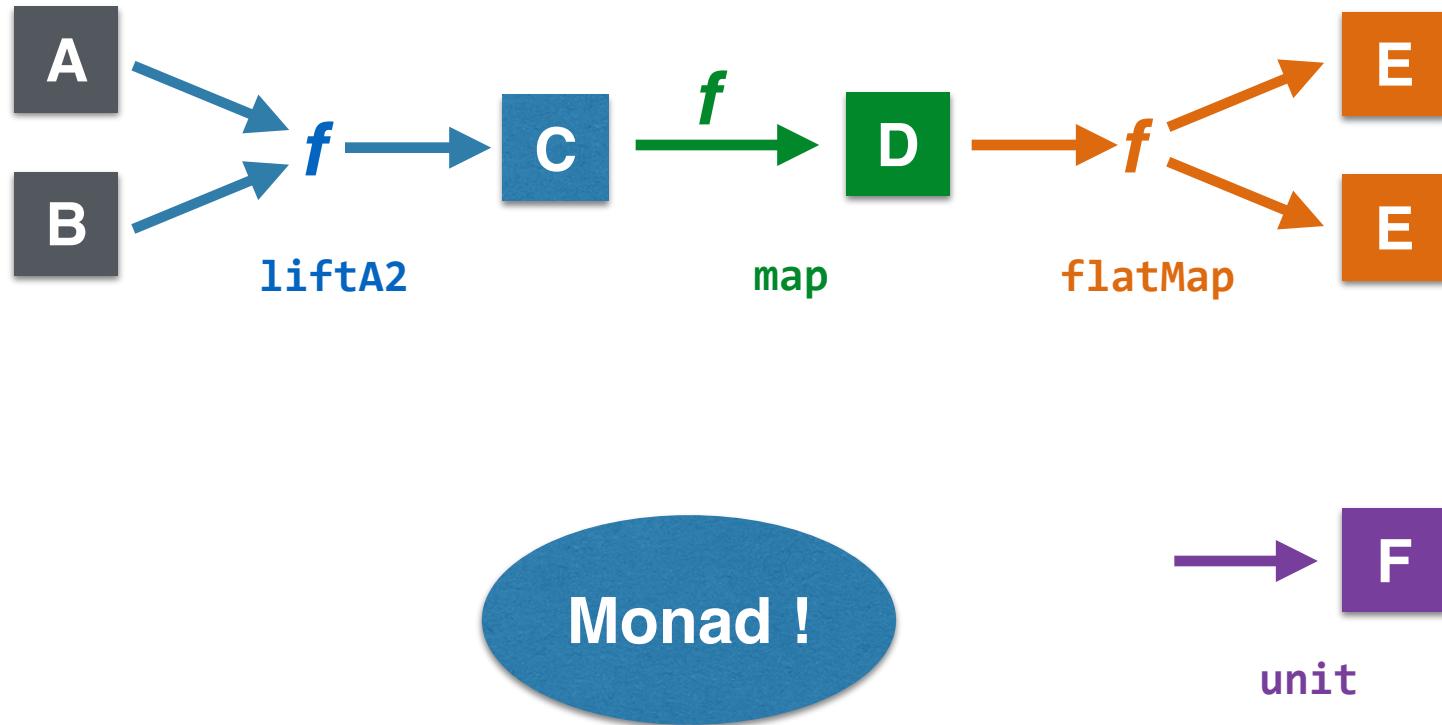
# Operations



# Operations



# Operations



# If only...

If we had **Monad[ResultSetOp]** we could do this:

```
case class Person(name: String, age: Int)

val getPerson: ResultSetOp[Person] =
  for {
    name <- GetString(1)
    age  <- GetInt(2)
  } yield Person(name, age)
```

# If only...

If we had **Monad[ResultSetOp]** we could do this:

```
case class Person(name: String, age: Int)

val getPerson: ResultSetOp[Person] =
  for {
    name <- GetString(1)
    age  <- GetInt(2)
  } yield Person(name, age)
```

But we don't.

# Spare a Monad?

Fancy words. Please be seated.

# Spare a Monad?

Fancy words. Please be seated.

- **Free** [**F**[\_], ?] is a **monad** for any **functor F**.

# Spare a Monad?

Fancy words. Please be seated.

- **Free** [ $\mathbf{F}[\_]$ , ?] is a **monad** for any **functor**  $\mathbf{F}$ .
- **Coyoneda** [ $\mathbf{S}[\_]$ , ?] is a **functor** for any  $\mathbf{S}$  at all.

# Spare a Monad?

Fancy words. Please be seated.

- **Free** [ $\mathbf{F}[\underline{\_}]$ , ?] is a **monad** for any **functor**  $\mathbf{F}$ .
- **Coyoneda** [ $\mathbf{S}[\underline{\_}]$ , ?] is a **functor** for any  $\mathbf{S}$  at all.
- By substitution, **Free** [**Coyoneda** [ $\mathbf{S}[\underline{\_}]$ , ?], ?] is a **monad** for any  $\mathbf{S}$  at all.

# Spare a Monad?

Fancy words. Please be seated.

- **Free** [**F**[\_], ?] is a **monad** for any **functor** **F**.
- **Coyoneda** [**S**[\_], ?] is a **functor** for any **S** at all.
- By substitution, **Free** [**Coyoneda** [**S**[\_], ?], ?] is a **monad** for any **S** at all.
- scalaz abbreviates this type as **FreeC** [**S**, ?]

# Spare a Monad?

Fancy words. Please be seated.

- **Free**[**F**[\_, ?] is a **monad** for any **functor** **F**.
- **Coyoneda**[**S**[\_, ?] is a **functor** for any **S** at all.
- By substitution, **Free**[**Coyoneda**[**S**[\_, ?], ?] is a **monad** for any **S** at all.
- scalaz abbreviates this type as **FreeC**[**S**, ?]
- Set **S** = **ResultSetOp** and watch what happens.

# Wait, what?

```
import scalaz.{ Free, Coyoneda }
import scalaz.Free.{ FreeC, liftFC }

type ResultSetIO[A] = FreeC[ResultSetOp, A]
```

# Wait, what?

```
import scalaz.{ Free, Coyoneda }
import scalaz.Free.{ FreeC, liftFC }

type ResultSetIO[A] = FreeC[ResultSetOp, A]

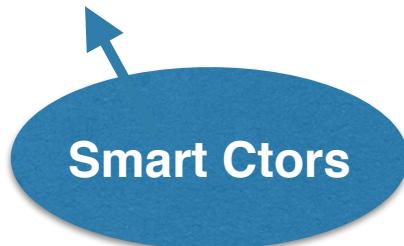
val next:          ResultSetIO[Boolean] = liftFC(Next)
def getInt(i: Int): ResultSetIO[Int]   = liftFC(GetInt(a))
def getString(i: Int): ResultSetIO[String] = liftFC(GetString(a))
val close:         ResultSetIO[Unit]   = liftFC(Close)
```

# Wait, what?

```
import scalaz.{ Free, Coyoneda }
import scalaz.Free.{ FreeC, liftFC }
```

```
type ResultSetIO[A] = FreeC[ResultSetOp, A]
```

```
val next:          ResultSetIO[Boolean] = liftFC(Next)
def getInt(i: Int): ResultSetIO[Int]    = liftFC(GetInt(a))
def getString(i: Int): ResultSetIO[String] = liftFC(GetString(a))
val close:         ResultSetIO[Unit]     = liftFC(Close)
```



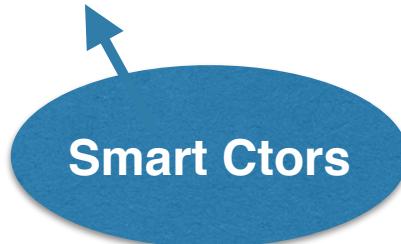
# Wait, what?

```
import scalaz.{ Free, Coyoneda }
import scalaz.Free.{ FreeC, liftFC }
```

```
type ResultSetIO[A] = FreeC[ResultSetOp, A]
```

```
val next:
def getInt(i: Int):
def getString(i: Int):
val close:
```

ResultSetIO[Boolean]	= liftFC(Next)
ResultSetIO[Int]	= liftFC(GetInt(a))
ResultSetIO[String]	= liftFC(GetString(a))
ResultSetIO[Unit]	= liftFC(Close)



# Wait, what?

Free Monad

```
import scalaz.{ Free, coyoneda }
import scalaz.Free.{ FreeC, liftFC }
```

```
type ResultSetIO[A] = FreeC[ResultSetOp, A]
```

```
val next:
def getInt(i: Int):
def getString(i: Int):
val close:
```

ResultSetIO[Boolean]	= liftFC(Next)
ResultSetIO[Int]	= liftFC(GetInt(a))
ResultSetIO[String]	= liftFC(GetString(a))
ResultSetIO[Unit]	= liftFC(Close)

Smart Ctors

ResultSetOp

# Programming

Now we can write programs in **ResultSetIO** using familiar monadic style.

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    name <- getString(1)
    age  <- getInt(2)
  } yield Person(name, age)
```

# Programming

Now we can write programs in **ResultSetIO**  
using familiar monadic style.

No ResultSet!

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    name <- getString(1)
    age  <- getInt(2)
  } yield Person(name, age)
```

# Programming

Now we can write programs in **ResultSetIO**  
using familiar monadic style.

No ResultSet!

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    name <- getString(1)
    age  <- getInt(2)
  } yield Person(name, age)
```

Values!

# Programming

Now we can write programs in **ResultSetIO**  
using familiar monadic style.

No ResultSet!

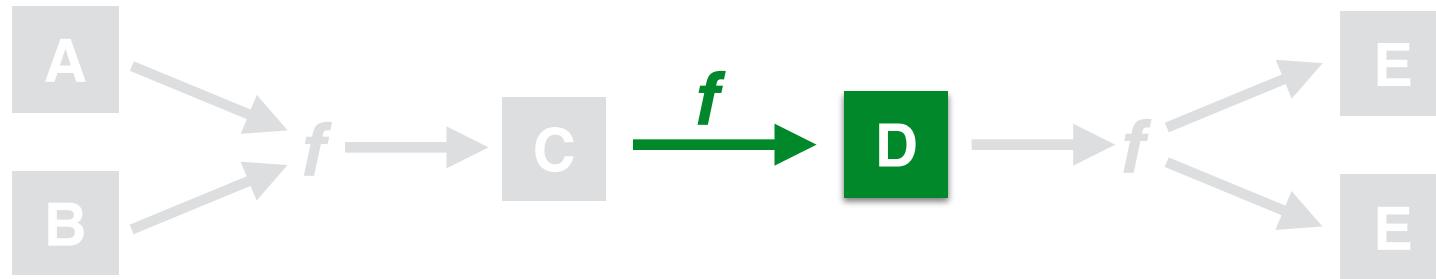
```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    name <- getString(1)
    age  <- getInt(2)
  } yield Person(name, age)
```

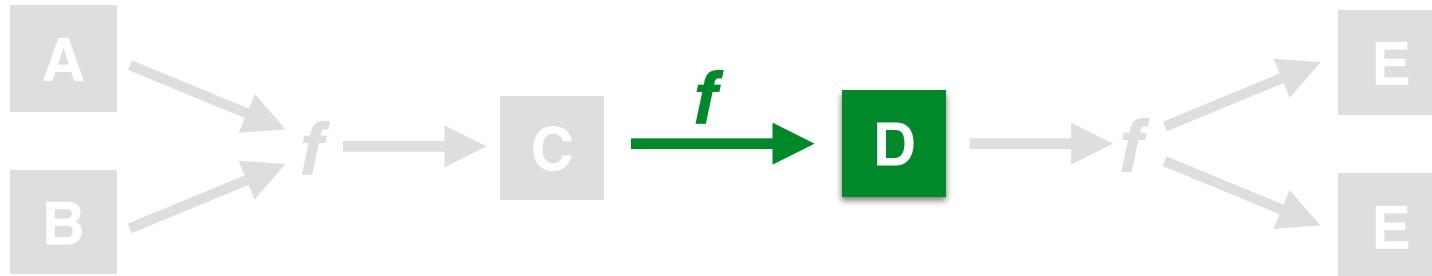
Values!

Composition!

# Functor Operations

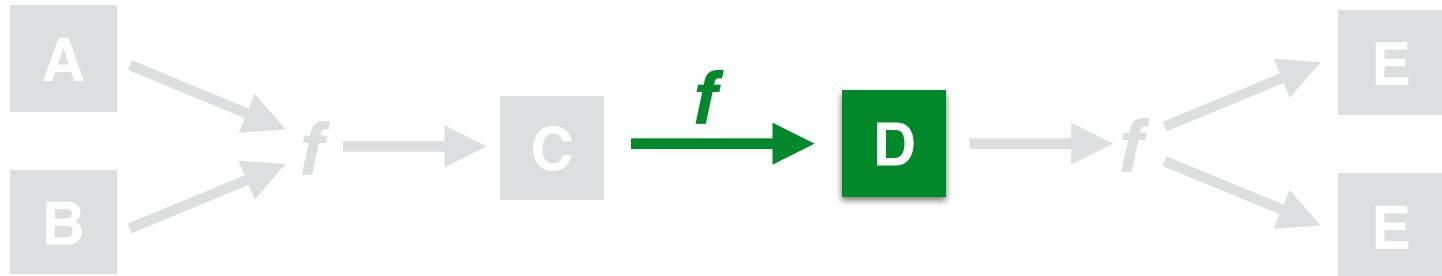


# Functor Operations



```
// Construct a program to read a Date at column n
def getDate(n: Int): ResultSetIO[java.util.Date] =
  getLong(n).map(new java.util.Date(_))
```

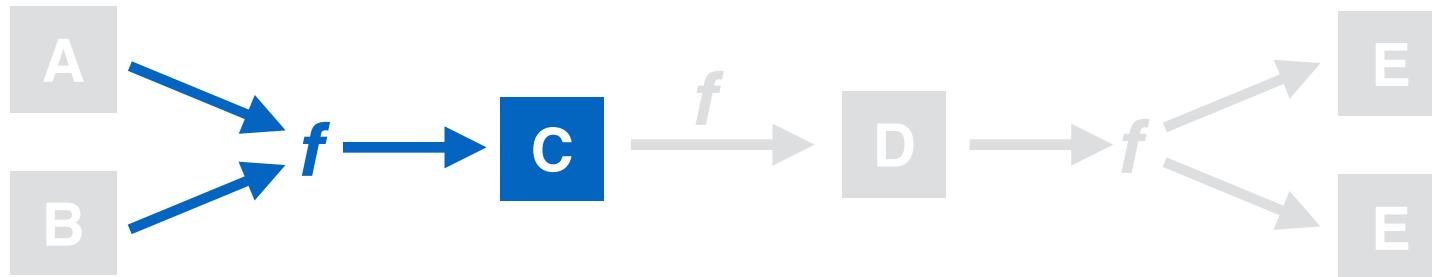
# Functor Operations



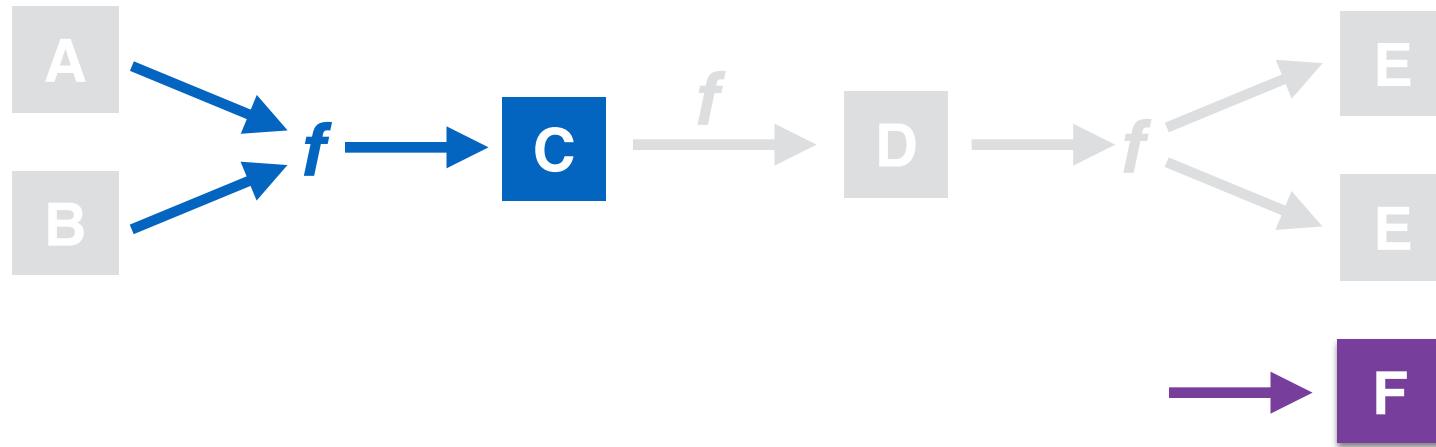
```
// Construct a program to read a Date at column n
def getDate(n: Int): ResultSetIO[java.util.Date] =
  getLong(n).map(new java.util.Date(_))
```

**fpair**  
**strengthL**  
**strengthR**  
**fproduct**  
**as**  
**void**

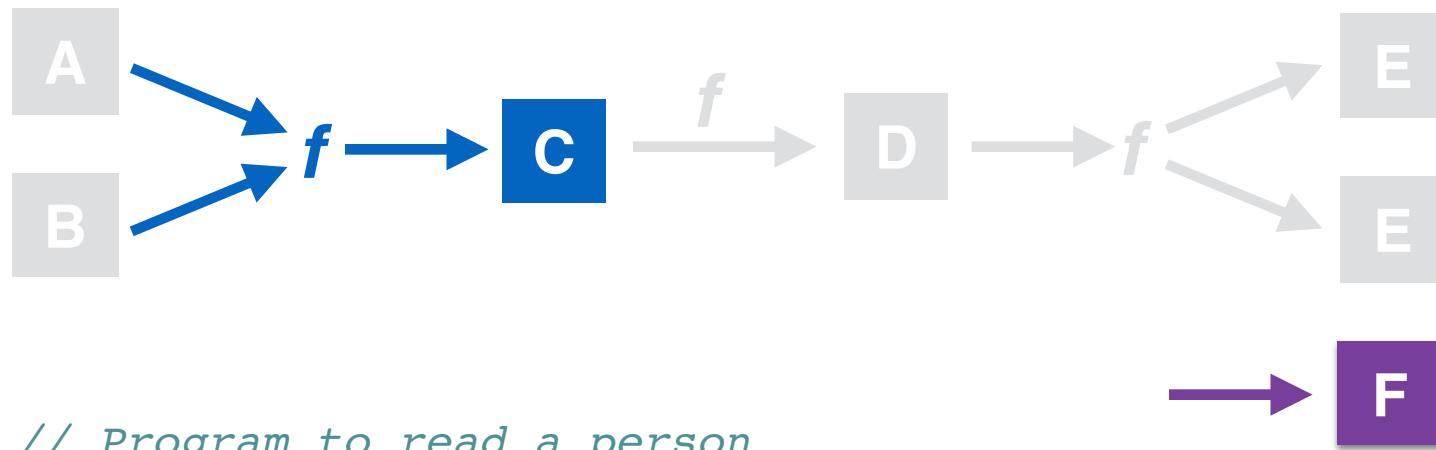
# Applicative Operations



# Applicative Operations

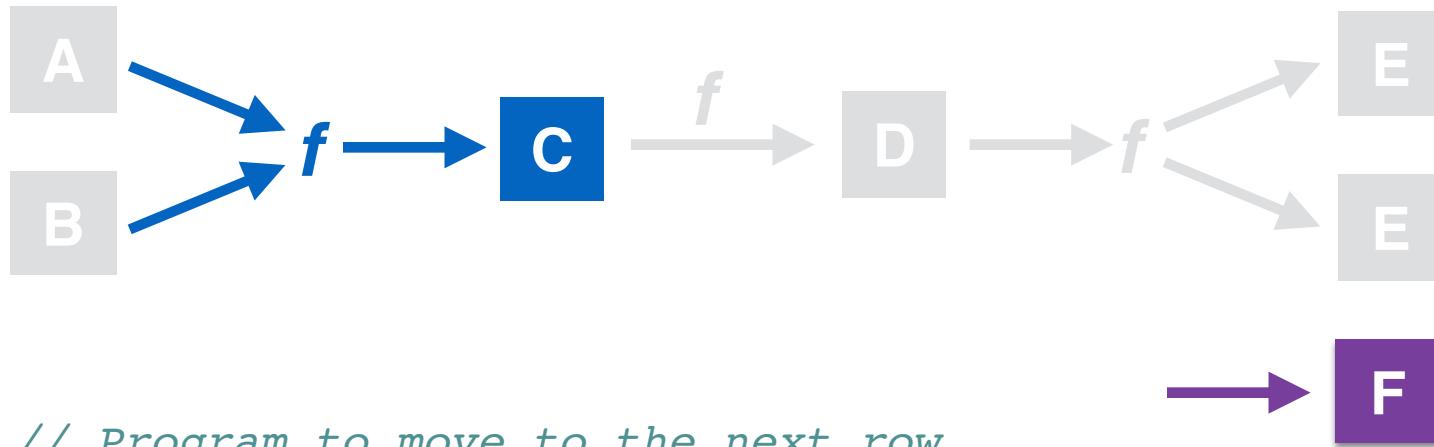


# Applicative Operations



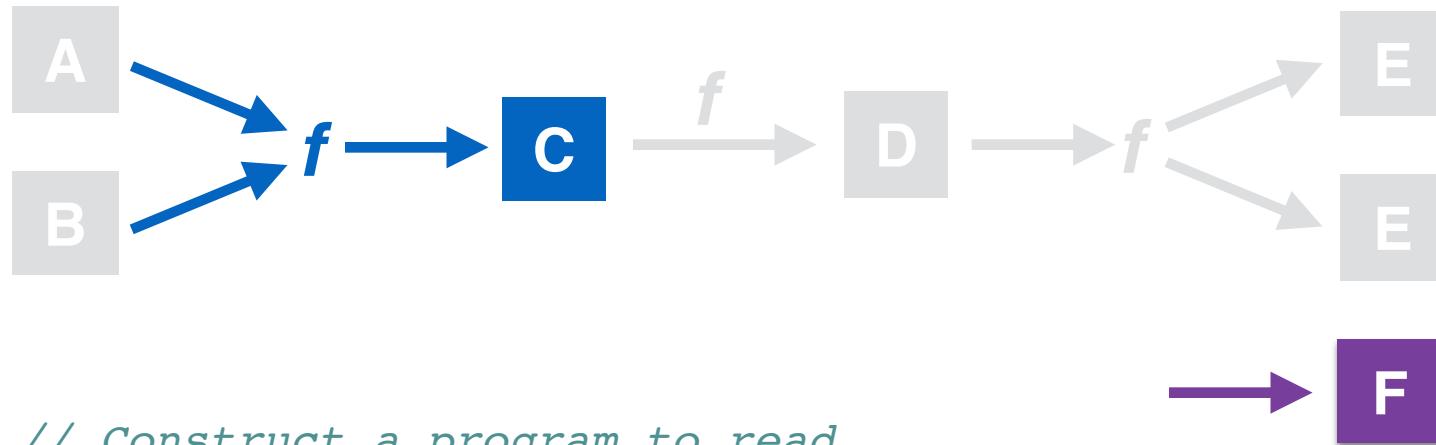
```
// Program to read a person
val getPerson: ResultSetIO[Person] =
  (getString(1) |@| getInt(2)) { (s, n) =>
    Person(s, n)
}
```

# Applicative Operations



```
// Program to move to the next row  
// and then read a person  
val getNextPerson: ResultSetIO[Person] =  
  next *> getPerson
```

# Applicative Operations



```
// Construct a program to read  
// a list of people  
def getPeople(n: Int): ResultSetIO[List[Person]] =  
  getNextPerson.replicateM(n)
```

# Applicative Operations

```
// Implementation of replicateM
def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)
```

# Applicative Operations

```
// Implementation of replicateM
def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)

// List[ResultSetIO[Person]]
List.fill(n)(getNextPerson)
```

# Applicative Operations

```
// Implementation of replicateM
def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)

// List[ResultSetIO[Person]]
List.fill(n)(getNextPerson)

// ResultSetIO[List[Person]]
List.fill(n)(getNextPerson).sequence
```

# Applicative Operations

```
// Implementation of replicateM
def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)

// List[ResultSetIO[Person]]
List.fill(n)(getNextPerson)

// ResultSetIO[List[Person]]
List.fill(n)(getNextPerson).sequence
```



# Applicative Operations

```
// Implementation of replicateM
def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)
```

```
// List[ResultSetIO[Person]]
List.fill(n)(getNextPerson)
```

```
// ResultSetIO[List[Person]]
List.fill(n)(getNextPerson) .sequence
```

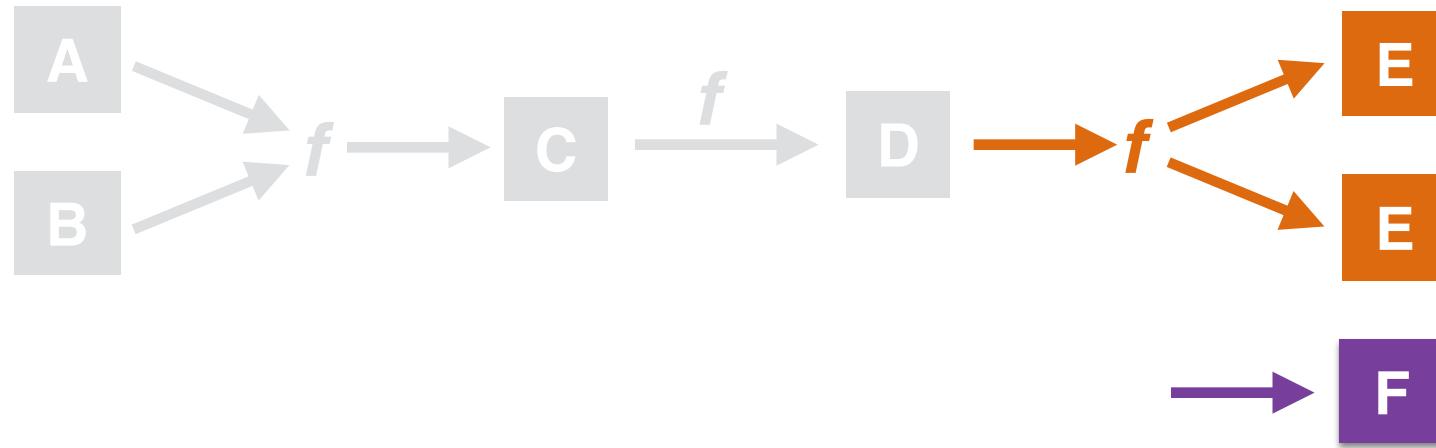


Traverse[List]

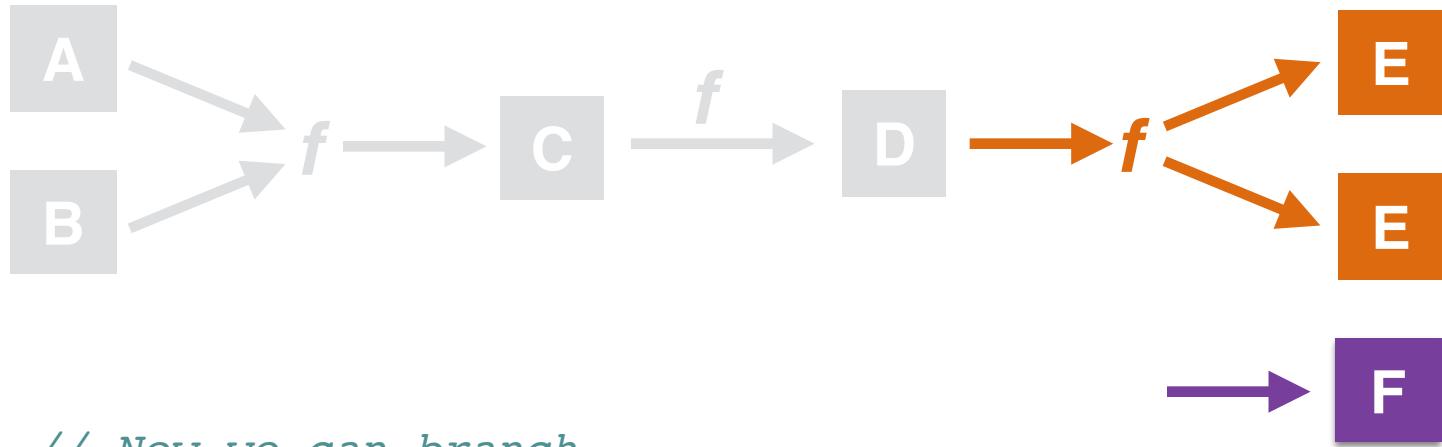


Awesome

# Monad Operations



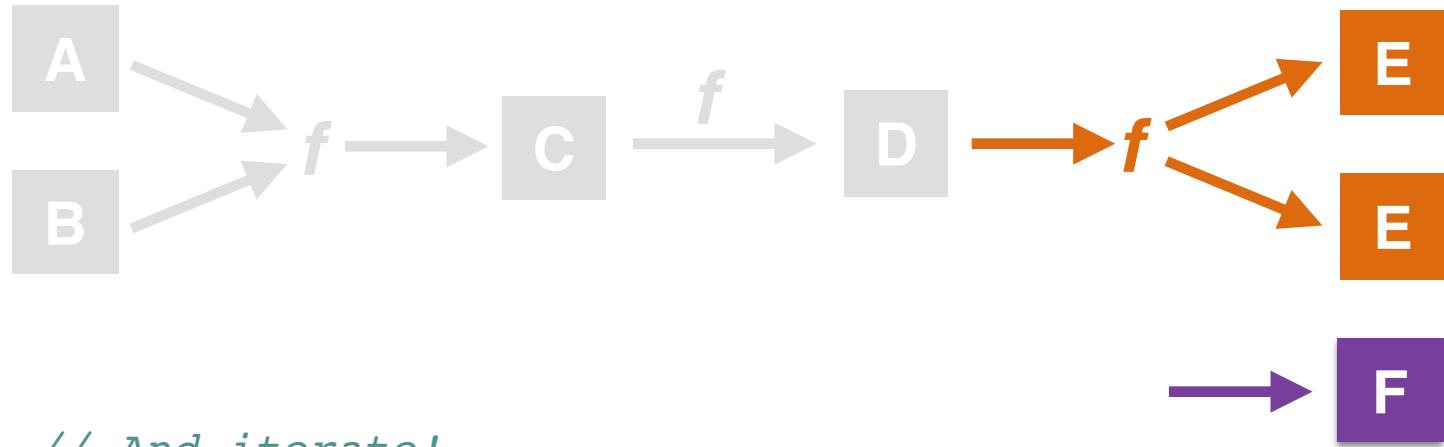
# Monad Operations



// Now we can branch

```
val getPersonOpt: ResultSetIO[Option[Person]] =  
  next.flatMap {  
    case true  => getPerson.map(_.some)  
    case false => none.point[ResultSetIO]  
  }
```

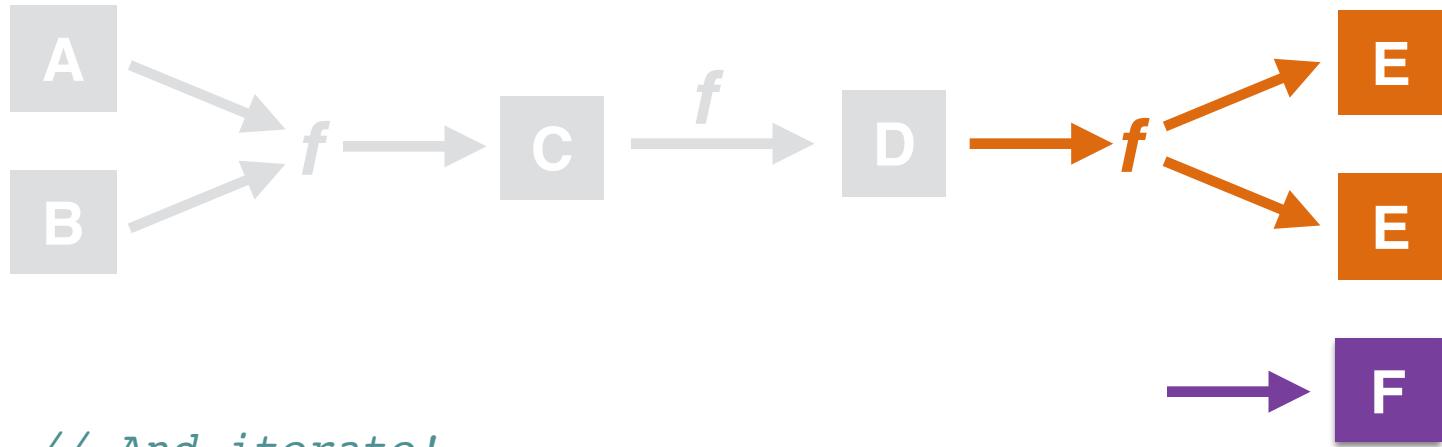
# Monad Operations



// And iterate!

```
val getAllPeople: ResultSetIO[Vector[Person]] =  
  next.wholeM[Vector](getPerson)
```

# Monad Operations



// And iterate!

```
val getAllPeople: ResultSetIO[Vector[Person]] =  
  next.whileM[Vector](getPerson)
```

Seriously

**Okaaay...**

# Okaaay...

- We can write little programs with this made-up data type and they are pure values and have nice compositional properties.

# Okaaay...

- We can write little programs with this made-up data type and they are pure values and have nice compositional properties.
- But how do we, um ... run them?

# Interpreting

# Interpreting

- To "run" our program we **interpret** it into some *target monad* of our choice. We're returning our loaner in exchange for a "real" monad.

# Interpreting

- To "run" our program we **interpret** it into some *target monad* of our choice. We're returning our loaner in exchange for a "real" monad.
- To do this, we need to provide a mapping from **ResultSetOp [A]** (our original data type) to **M[A]** for any **A**.

# Interpreting

- To "run" our program we **interpret** it into some *target monad* of our choice. We're returning our loaner in exchange for a "real" monad.
- To do this, we need to provide a mapping from **ResultSetOp** [**A**] (our original data type) to **M[A]** for any **A**.
- This is called a **natural transformation** and is written **ResultSetOp ~> M**.

# Interpreting

Here we interpret into **scalaz.effect.IO**

```
def trans(rs: ResultSet) =  
  new (ResultSetOp ~> IO) {  
    def apply[A](fa: ResultSetOp[A]): IO[A] =  
      fa match {  
        case Next          => IO(rs.next)  
        case GetInt(i)    => IO(rs.getInt(i))  
        case GetString(i) => IO(rs.getString(i))  
        case Close         => IO(rs.close)  
        // lots more  
      }  
  }
```

# Interpreting

Here we interpret into **scalaz.effect.IO**

```
def trans(rs: ResultSet) =  
  new (ResultSetOp ~> IO) {  
    def apply[A](fa: ResultSetOp[A]): IO[A] =  
      fa match {  
        case Next          => IO(rs.next)  
        case GetInt(i)     => IO(rs.getInt(i))  
        case GetString(i) => IO(rs.getString(i))  
        case Close         => IO(rs.close)  
        // lots more  
      }  
  }
```



# Interpreting

Here we interpret into **scalaz.effect.IO**

```
def trans(rs: ResultSet) =  
  new (ResultSetOp ~> IO) {  
    def apply[A](fa: ResultSetOp[A]): IO[A] =  
      fa match {  
        case Next          => IO(rs.next)  
        case GetInt(i)     => IO(rs.getInt(i))  
        case GetString(i) => IO(rs.getString(i))  
        case Close         => IO(rs.close)  
        // lots more  
      }  
  }
```

ResultSetOp

Target Monad



# Running

```
def toIO[A](a: ResultSetIO[A], rs: ResultSet): IO[A] =  
  Free.runFC(a)(trans(rs))
```

# Running

```
def toIO[A](a: ResultSetIO[A], rs: ResultSet): IO[A] =  
  Free.runFC(a)(trans(rs))
```

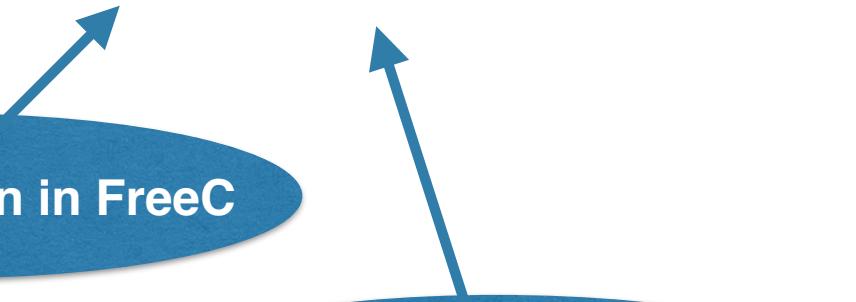
Program written in FreeC

# Running

```
def toIO[A](a: ResultSetIO[A], rs: ResultSet): IO[A] =  
  Free.runFC(a)(trans(rs))
```

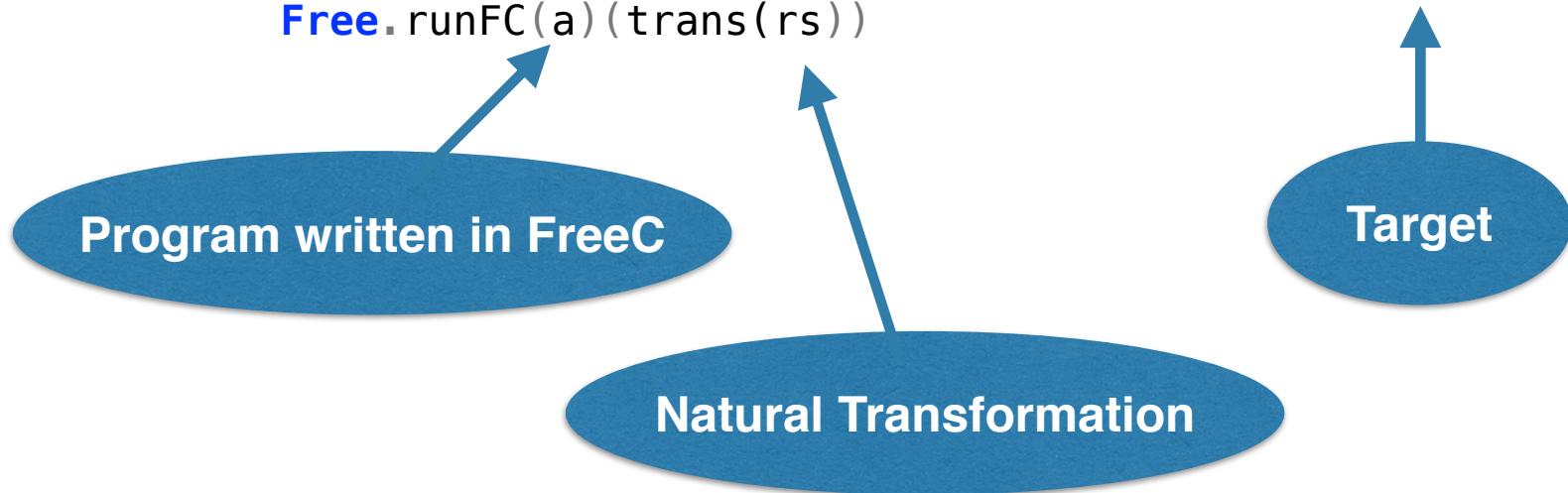
Program written in FreeC

Natural Transformation



# Running

```
def toIO[A](a: ResultSetIO[A], rs: ResultSet): IO[A] =  
  Free.runFC(a)(trans(rs))
```



# Running

```
def toIO[A](a: ResultSetIO[A], rs: ResultSet): IO[A] =  
  Free.runFC(a)(trans(rs))
```

Program written in FreeC

Natural Transformation

Target

```
val prog = next.whileM[List](getPerson)  
toIO(prog, rs).unsafePerformIO // List[Person]
```

**Fine. What's doobie?**

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

**BlobIO**[[A](#)]

**CallableStatementIO**[[A](#)]

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`DriverManagerIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

`NClobIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`DriverManagerIO[A]`

`PreparedStatementIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

`NClobIO[A]`

`RefIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`DriverManagerIO[A]`

`PreparedStatementIO[A]`

`ResultSetIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

`NClobIO[A]`

`RefIO[A]`

`SQLDataIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`DriverManagerIO[A]`

`PreparedStatementIO[A]`

`ResultSetIO[A]`

`SQLInputIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

`NClobIO[A]`

`RefIO[A]`

`SQLDataIO[A]`

`SQLOutputIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`DriverManagerIO[A]`

`PreparedStatementIO[A]`

`ResultSetIO[A]`

`SQLInputIO[A]`

`StatementIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

`NClobIO[A]`

`RefIO[A]`

`SQLDataIO[A]`

`SQLOutputIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`DriverManagerIO[A]`

`PreparedStatementIO[A]`

`ResultSetIO[A]`

`SQLInputIO[A]`

`StatementIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

`NClobIO[A]`

`RefIO[A]`

`SQLDataIO[A]`

`SQLOutputIO[A]`

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

`BlobIO[A]`

`ClobIO[A]`

`DatabaseMetaDataIO[A]`

`DriverManagerIO[A]`

`PreparedStatementIO[A]`

`ResultSetIO[A]`

`SQLInputIO[A]`

`StatementIO[A]`

`CallableStatementIO[A]`

`ConnectionIO[A]`

`DriverIO[A]`

`NClobIO[A]`

`RefIO[A]`

`SQLDataIO[A]`

`SQLOutputIO[A]`

- Pure functional support for all primitive operations.

# Fine. What's doobie?

- Low-level API is basically exactly this, for **all** of JDBC.

<code>BlobIO[A]</code>	<code>CallableStatementIO[A]</code>
<code>ClobIO[A]</code>	<code>ConnectionIO[A]</code>
<code>DatabaseMetaDataIO[A]</code>	<code>DriverIO[A]</code>
<code>DriverManagerIO[A]</code>	<code>NClobIO[A]</code>
<code>PreparedStatementIO[A]</code>	<code>RefIO[A]</code>
<code>ResultSetIO[A]</code>	<code>SQLDataIO[A]</code>
<code>SQLInputIO[A]</code>	<code>SQLOutputIO[A]</code>
<code>StatementIO[A]</code>	

- Pure functional support for all primitive operations.
- Machine-generated (!)

# Exception Handling

```
val ma = ConnectionIO[A]

ma.attempt // ConnectionIO[Throwable ∕ A]

// General                                // SQLException
ma.attemptSome(handler)                  ma.attemptSql
ma.except(handler)                      ma.attemptSqlState
ma.exceptSome(handler)                 ma.attemptSomeSqlState(handler)
ma.onException(action)                  ma.exceptSql(handler)
ma.ensure(sequel)                       ma.exceptSqlState(handler)
                                         ma.exceptSomeSqlState(handler)

// PostgreSQL (hundreds more)
ma.onWarning(handler)
ma.onDynamicResultSetsReturned(handler)
ma.onImplicitZeroBitPadding(handler)
ma.onNullValueEliminatedInSetFunction(handler)
ma.onPrivilegeNotGranted(handler)
...
```

# Mapping via Typeclass

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    name <- getString(1)
    age  <- getInt(2)
  } yield Person(name, age)
```

# Mapping via Typeclass

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    name <- get[String](1)
    age  <- get[Int](2)
  } yield Person(name, age)
```

Abstract over return type

# Mapping via Typeclass

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    p <- get[(String, Int)](1)
  } yield Person(p._1, p._2)
```

Generalize to tuples

# Mapping via Typeclass

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  for {
    p <- get[Person](1)
  } yield p
```



Generalize to Products

# Mapping via Typeclass

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  get[Person](1)
```

# Mapping via Typeclass

```
case class Person(name: String, age: Int)

val getPerson: ResultSetIO[Person] =
  get[Person]
```

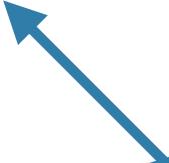
# Mapping via Typeclass

```
case class Person(name: String, age: Int)  
get[Person]
```

# Mapping via Typeclass

```
case class Person(name: String, age: Int)
```

```
get[Person]
```



This is how you would  
really write it in doobie.

# **Streaming**

# Streaming

```
// One way to read into a List
val readAll: ResultSetIO[List[Person]] =
  next.whileM[List](get[Person])
```

# Streaming

```
// One way to read into a List
val readAll: ResultSetIO[List[Person]] =  
  next.whileM[List] (get[Person])  
  
// Another way
val people: Process[ResultSetIO, Person] =  
  process[Person]
```

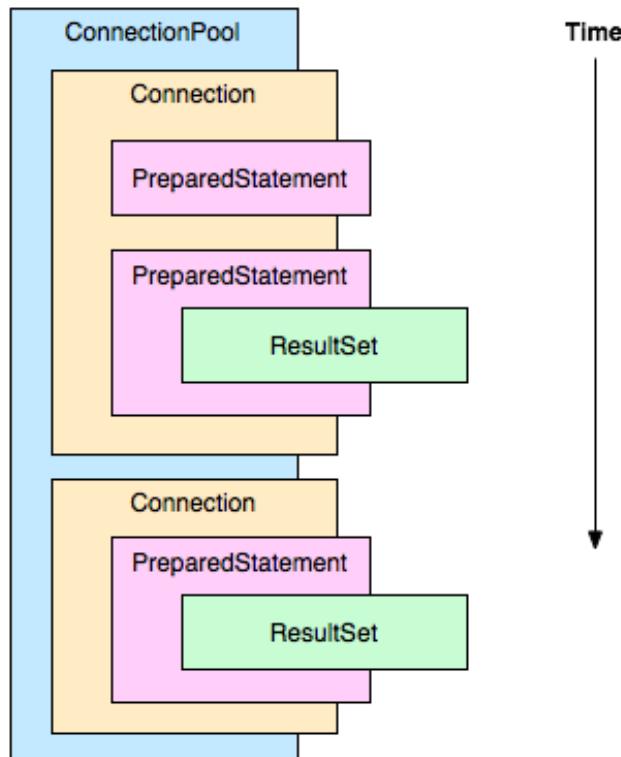
# Streaming

```
// One way to read into a List
val readAll: ResultSetIO[List[Person]] =
  next.whileM[List](get[Person])

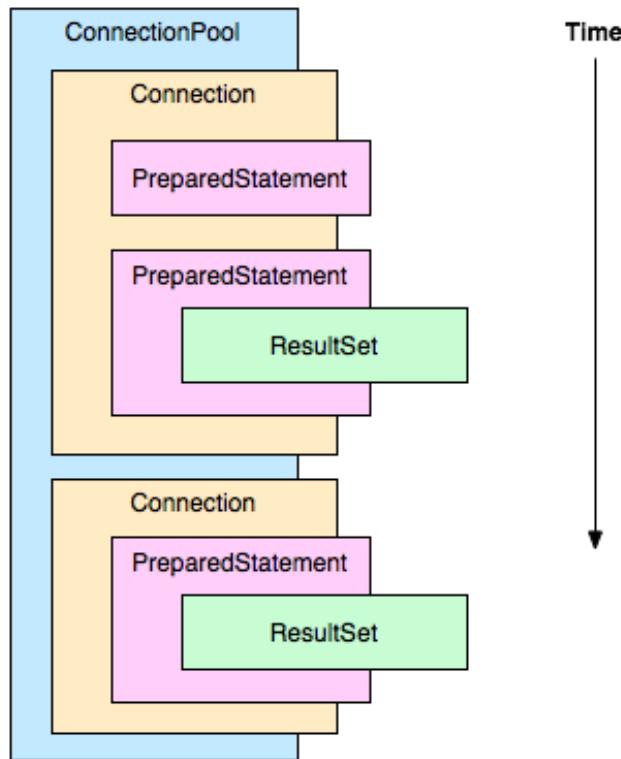
// Another way
val people: Process[ResultSetIO, Person] =
  process[Person]

people
  .filter(_.name.length > 5)
  .take(20)
  .moreStuff
  .list           // ResultSetIO[List[Person]]
```

# High-Level API

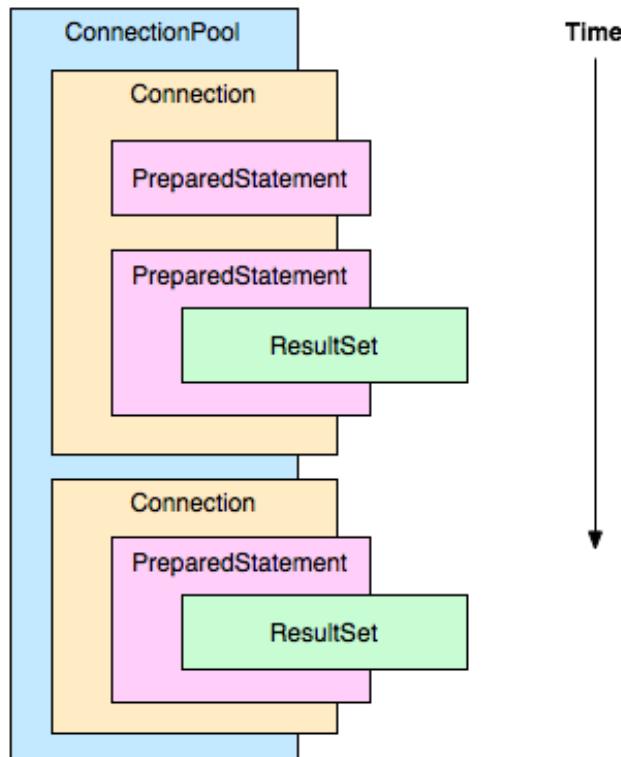


# High-Level API



**doobie** programs can be nested, matching the natural structure of database interactions.

# High-Level API



**doobie** programs can be nested, matching the natural structure of database interactions.

Some of the common patterns are provided in a high-level API that abstracts the lifting.

# High-Level API

```
case class Country(name: String, code: String, pop: BigDecimal)

def biggerThan(pop: Int) =
  sql"""
    select code, name, gnp from country
    where population > $pop
  """.query[Country]
```

# High-Level API

```
case class Country(name: String, code: String, pop: BigDecimal)
```

```
def biggerThan(pop: Int) =  
  sql"""  
    select code, name, gnp from country  
    where population > $pop  
  """.query[Country]
```

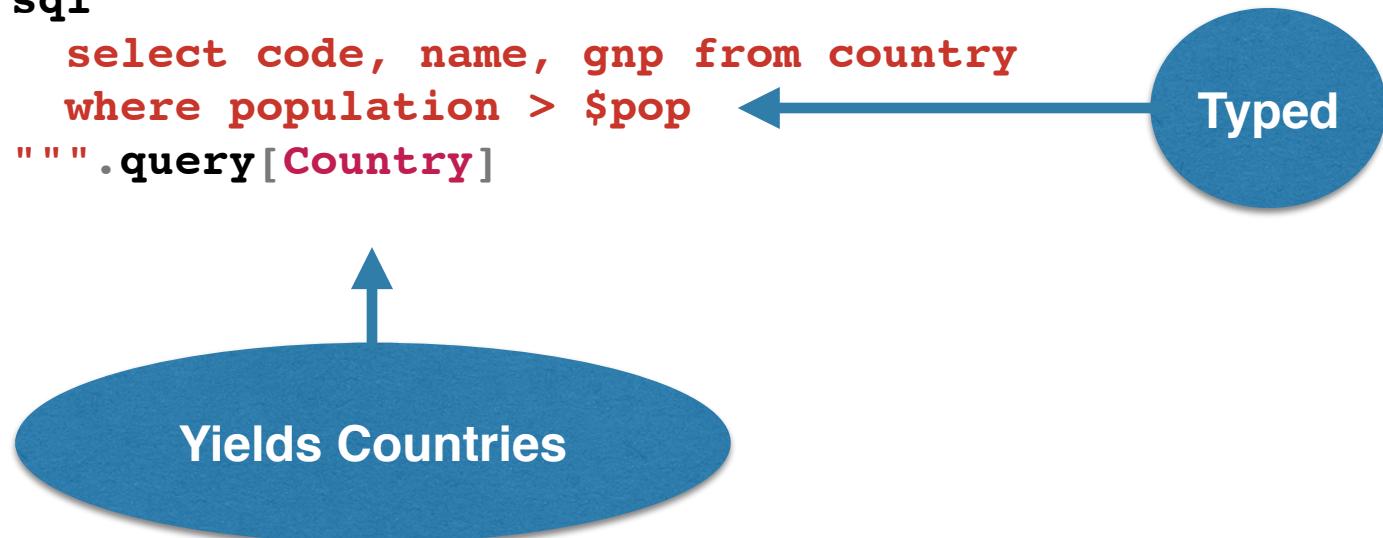


Typed

# High-Level API

```
case class Country(name: String, code: String, pop: BigDecimal)
```

```
def biggerThan(pop: Int) =  
  sql"""  
    select code, name, gnp from country  
    where population > $pop  
  """.query[Country]
```



# High-Level API

```
scala> biggerThan(100000000)
|   .process          // Process[ConnectionIO, Person]
|   .take(5)          // Process[ConnectionIO, Person]
|   .list             // ConnectionIO[List[Person]]
|   .transact(xa)    // Task[List[Person]]
|   .run              // List[Person]
|   .foreach(println)
Country(BGD,Bangladesh,32852.00)
Country(BRA,Brazil,776739.00)
Country(IDN,Indonesia,84982.00)
Country(IND,India,447114.00)
Country(JPN,Japan,3787042.00)
```

# High-Level API

```
scala> biggerThan(100000000)
      .process          // Process[ConnectionIO, Person]
      .take(5)          // Process[ConnectionIO, Person]
      .list             // ConnectionIO[List[Person]]
      .transact(xa)    // Task[List[Person]]
      .run              // List[Person]
      .foreach(println)
Country(BGD,Bangladesh,32852.00)
Country(BRA,Brazil,776739.00)
Country(IDN,Indonesia,84951.00)
Country(IND,India,447114.00)
Country(JPN,Japan,3787042.00)
```



A blue arrow points from the `.run` method in the code snippet to a blue oval containing the text `Transactor[Task]`.

# YOLO Mode

```
scala> biggerThan(100000000)
|   .process      // Process[ConnectionIO, Person]
|   .take(5)       // Process[ConnectionIO, Person]
|   .quick        // Task[Unit]
|   .run          // Unit
Country(BGD,Bangladesh,32852.00)
Country(BRA,Brazil,776739.00)
Country(IDN,Indonesia,84982.00)
Country(IND,India,447114.00)
Country(JPN,Japan,3787042.00)
```

# YOLO Mode

```
scala> biggerThan(100000000)
|   .process      // Process[ConnectionIO, Person]
|   .take(5)       // Process[ConnectionIO, Person]
|   .quick        // Task[Unit]
|   .run          // Unit
Country(BGD,Bangladesh,32852.00)
Country(BRA,Brazil,776739.00)
Country(IDN,Indonesia,84982.00)
Country(IND,India,447114.00)
Country(JPN,Japan,3787042.00)
```

Ambient Transactor  
Just for Experimenting  
in the REPL

# YOLO MODE

```
scala> biggerThan(0).check.run
```

```
select code, name, gnp from country where population > ?
```

- ✓ SQL Compiles and Typechecks
- ✓ P01 Int → INTEGER (int4)
- ✓ C01 code CHAR (bpchar) NOT NULL → String
- ✓ C02 name VARCHAR (varchar) NOT NULL → String
- ✗ C03 gnp NUMERIC (numeric) NULL → BigDecimal
  - Reading a NULL value into BigDecimal will result in a runtime failure. Fix this by making the schema type NOT NULL or by changing the Scala type to Option[BigDecimal]

# YOLO MODE

```
scala> biggerThan(0).check.run
```

```
select code, name, gnp from country where population > 0
```

- ✓ SQL Compiles and Typechecks
- ✓ P01 Int → INTEGER (int4)
- ✓ C01 code CHAR (bpchar) NOT NULL → String
- ✓ C02 name VARCHAR (varchar) NOT NULL → String
- ✗ C03 gnp NUMERIC (numeric) NULL → BigDecimal
  - Reading a NULL value into BigDecimal will result in a runtime failure. Fix this by making the schema type NOT NULL or by changing the Scala type to Option[BigDecimal]

Can also do this in  
your unit tests!

# **Much More**

# Much More

- Extremely simple **custom type mappings** for columns and composite types.

# Much More

- Extremely simple **custom type mappings** for columns and composite types.
- **Connection Pooling** with HikariCP, easily extended for other pooling implementations.

# Much More

- Extremely simple **custom type mappings** for columns and composite types.
- **Connection Pooling** with HikariCP, easily extended for other pooling implementations.
- **Syntax aplenty**, to make fancy types easier to work with.

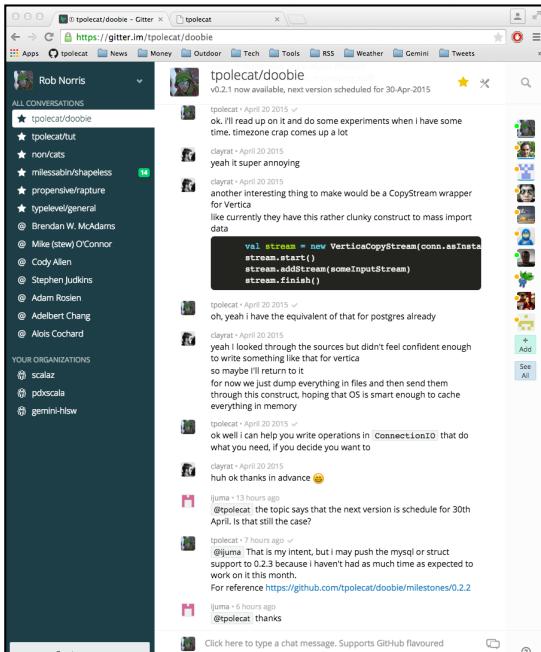
# Much More

- Extremely simple **custom type mappings** for columns and composite types.
- **Connection Pooling** with HikariCP, easily extended for other pooling implementations.
- **Syntax aplenty**, to make fancy types easier to work with.
- **PostgreSQL Support**: Geometric Types, Arrays, PostGIS types, LISTEN/NOTIFY, CopyIn/Out, Large Objects, ...

# Moar Info

<https://github.com/tpolecat/doobie>

## gitter



## book of doobie

A screenshot of the 'book of doobie' website, showing examples of database mapping and code snippets. It includes sections on construction, mapping to JSON, and a note about Argonaut serialization.

According to the error message we need a `Meta[PersonId]` instance. So how do we get one? The simplest way is by basing it on an existing instance, using `xmap`, which is like the invariant functor `imap` but ensures that `null` values are never observed. So we simply provide `String > PersonId` and vice-versa and we're good to go.

```
Implicit val PersonIdMeta: Meta[PersonId] =  
  Meta[String].xmap[PersonId].unsafeFromLegacy(_).toLegacy()
```

Now it compiles as a column value and as a `Composite` that maps to a single column:

```
scala> sql"select * from person where id = $pid"  
res0: doobie.syntax.string.SqlInterpolator#Builder[shapeless.:>[PersonId,shapeless.HNil]] = doobie.syntax.string.SqlInterpolator$Builder@14d6e06  
scala> Composite[PersonId].length  
res1: Int = 1  
scala> sql"select \"$pid\"@doobie:123".query[PersonId].quick.run  
PersonId(podiatry,123)
```

Note that the `Composite` width is now a single column. The rule is: if there exists an instance `Meta[A]` in scope, it will take precedence over any automatic derivation of `Composite[A]`.

**Meta by Construction**

Some modern databases support a `json` column type that can store structured data as a JSON document, along with various SQL extensions to allow querying and selecting arbitrary sub-structures. So an obvious thing we might want to do is provide a mapping from Scala model objects to JSON columns, via some kind of JSON serialization library.

We can construct a `Meta` instance for the `argonaut.Json` type by using the `Meta.ether` constructor, which constructs a direct object mapping via JDBC's `.getObjectContext` and `.setObjectContext`. In the case of PostgreSQL, the JSON values are marshalled via the `PODObject` type, which encapsulates an unsplicing `(String, String)` pair representing the schema type and its string value.

Here we go:

```
Implicit val JsonMeta: Meta[Json] =  
  Meta.ether[Json](\"json\").xmap[Json]({  
    case Parse.parseJson(x) => Left(x)  
    case Left(x) => Right(x.toJson)  
  })  
  .a new PODObject(c \_.setObjectContext(a).setObjectContext(a))) \_.setObjectContext(a).setObjectContext(a))
```

Given this mapping to and from `Json` we can construct a further mapping to any type that has a `CodecJson` instance. The `xmap` constrains us to reference types and requires a `TypeTag` for diagnostics, so the full type constraint is `A >: Null : CodecJson[TypeTag]`. On failure we throw an exception; this indicates a logic or schema problem.

```
def CodecMeta[A >: Null : CodecJson[TypeTag]]: Meta[A] =  
  MetaJson(xmap[A]({  
    case Left(x) => Left(x.getObjectContext)  
    case Right(x) => Right(x.setObjectContext)  
  }))
```

Let's make sure it works. Here is a simple data type with an argonaut serializer, taken straight from the website.