Programs as Values

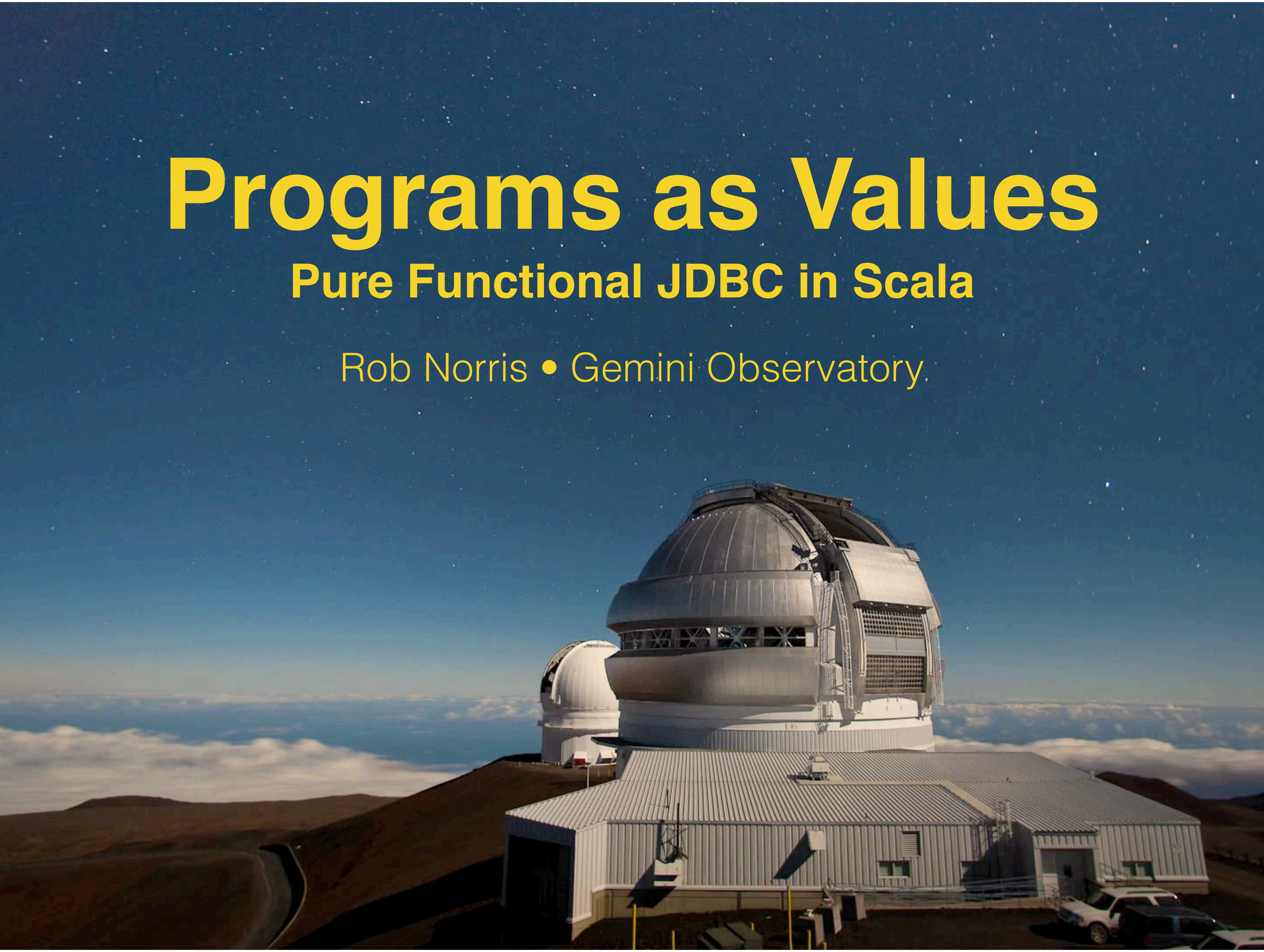
Pure Functional JDBC in Scala

Rob Norris • Gemini Observatory

Programs as Values

Pure Functional JDBC in Scala

Rob Norris • Gemini Observatory



What's this about?

This is a talk about the **<u>implementation</u>** 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**. Many other APIs are also 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**. Many other APIs are also terrible.

- Transforming a legacy API to a pure-functional one can be entirely **mechanical** in some cases.

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**. Many other APIs are also terrible.
- Transforming a legacy API to a pure-functional one can be entirely **mechanical** in some cases.
- General technique to gain equational reasoning and good compositional properties, basically for **free**, without loss of functionality.

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 JD n?

Managed Resource

```scala
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 JD n?

Managed Resource

```scala
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)
}
```

Side-Effect

The Problem

So what's wrong with this JDB… n?

Managed Resource

```scala
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)
}
```

Side-Effect

Composition?

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.

```scala
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.

```scala
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...
```

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.

```scala
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...
```

Constructor per Method

Parameterized on Return Type

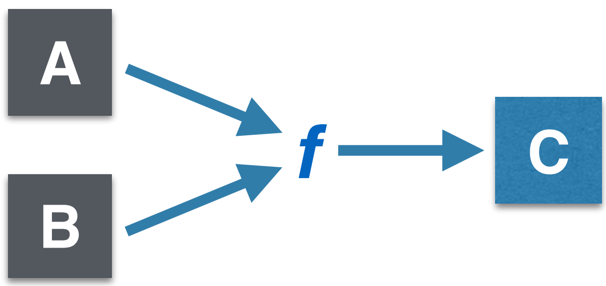
Operations

Operations

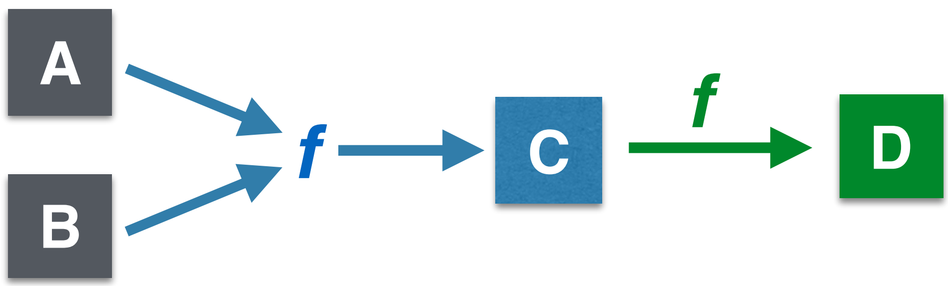
A

B

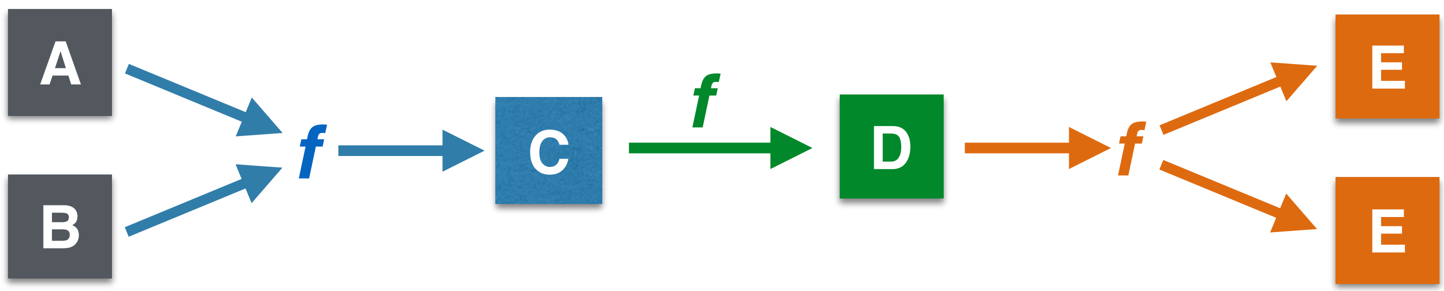
Operations



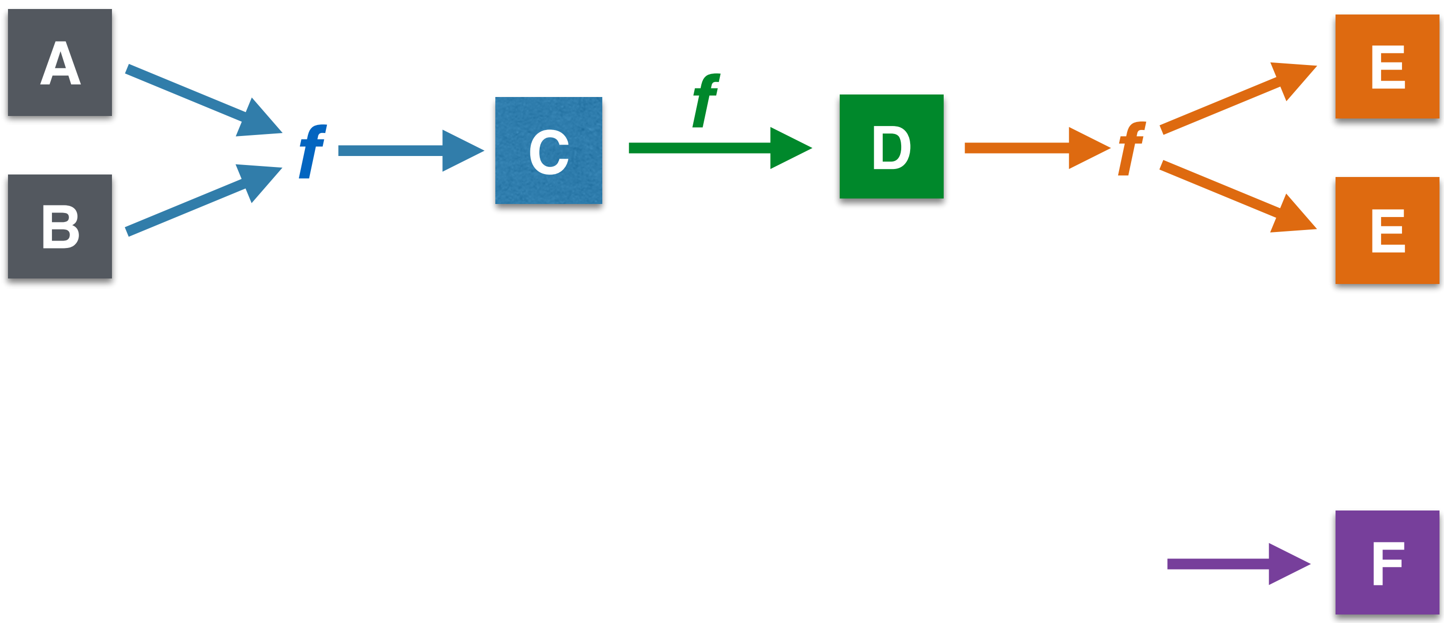
Operations



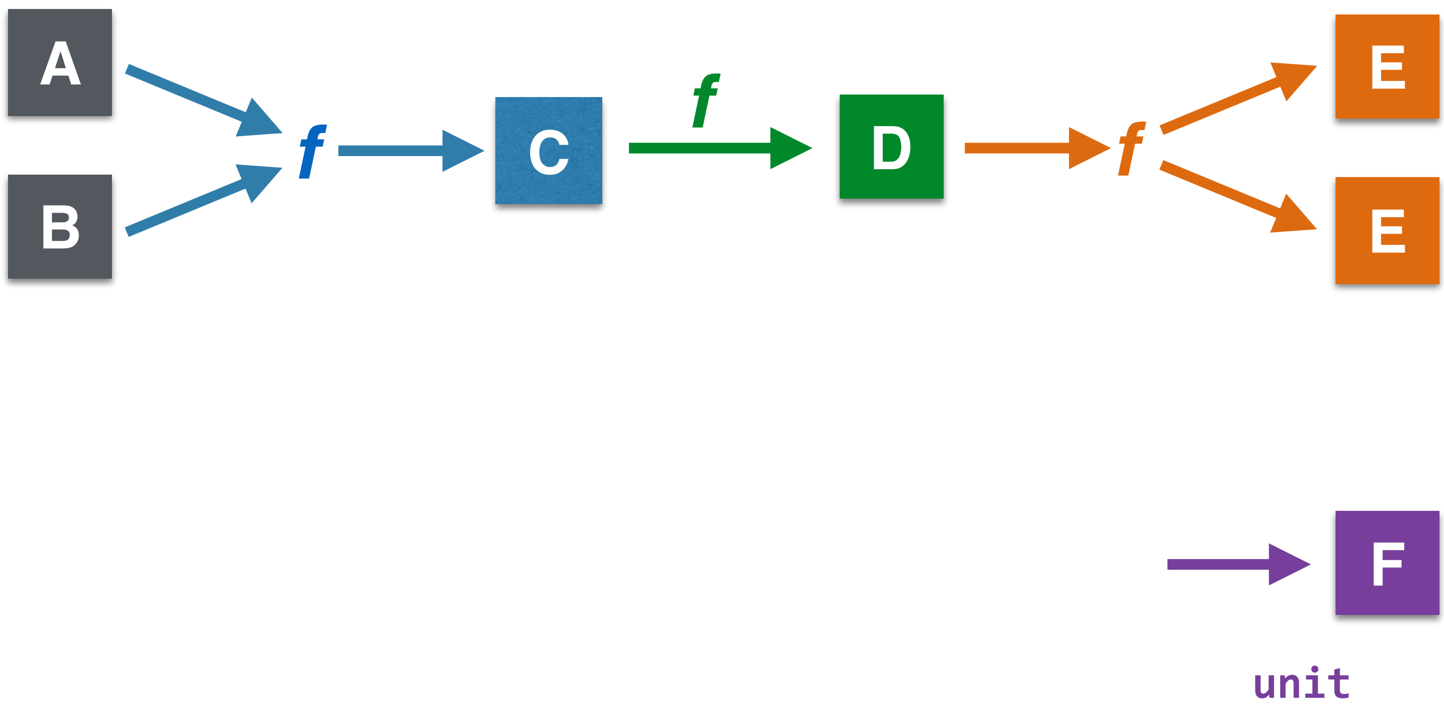
Operations



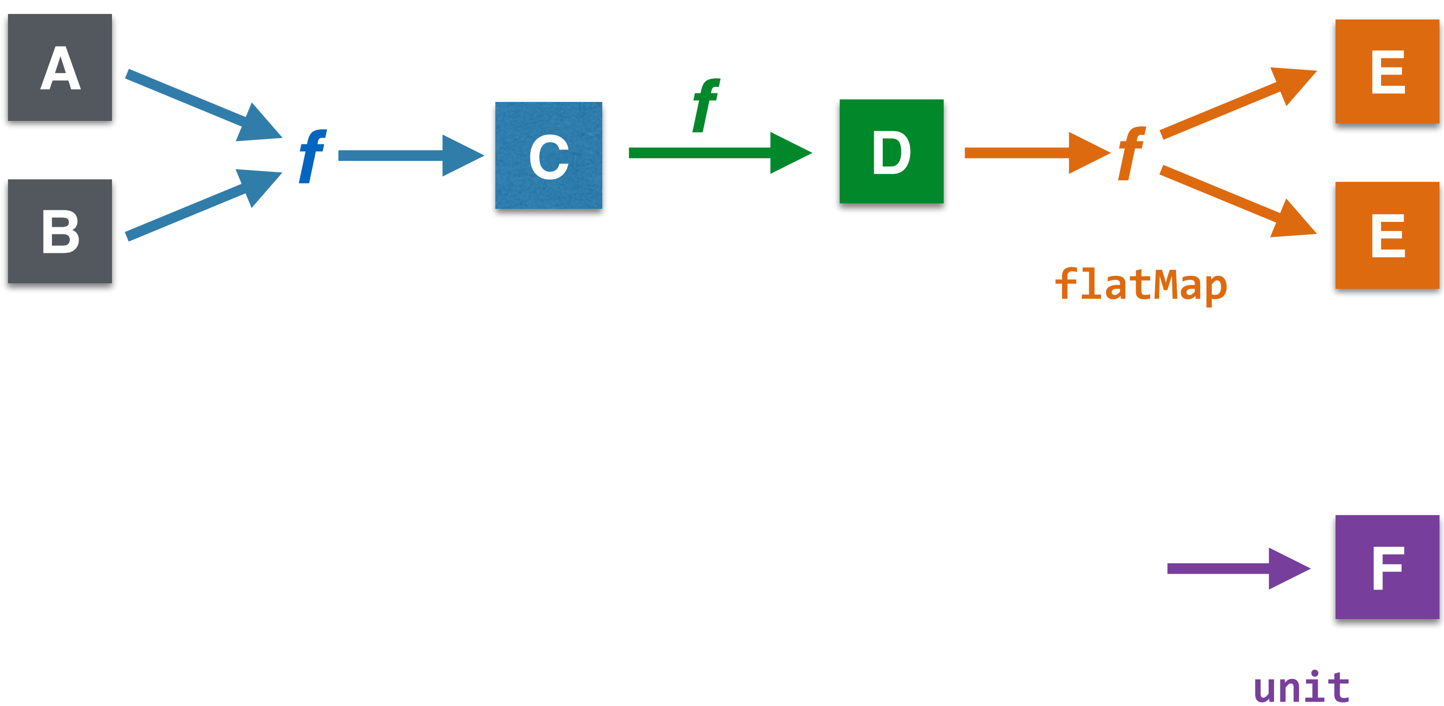
Operations



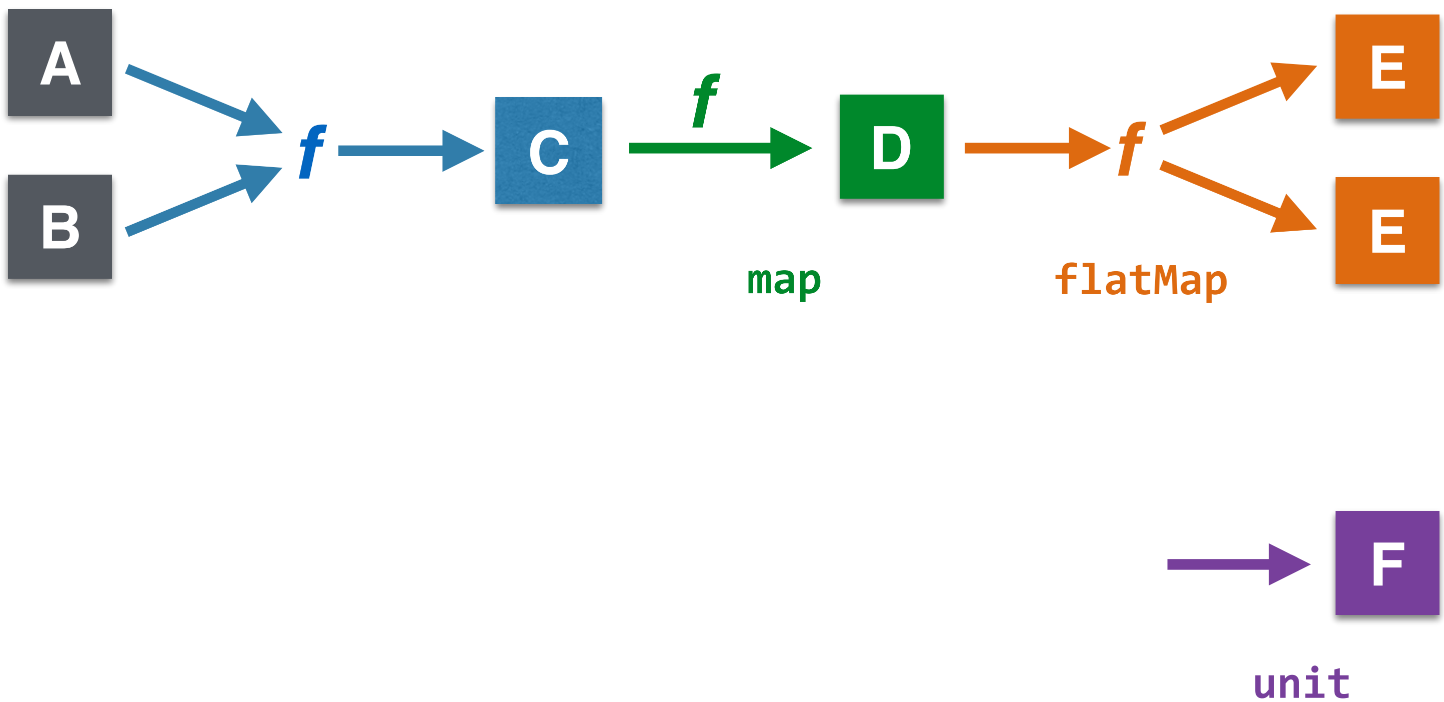
Operations



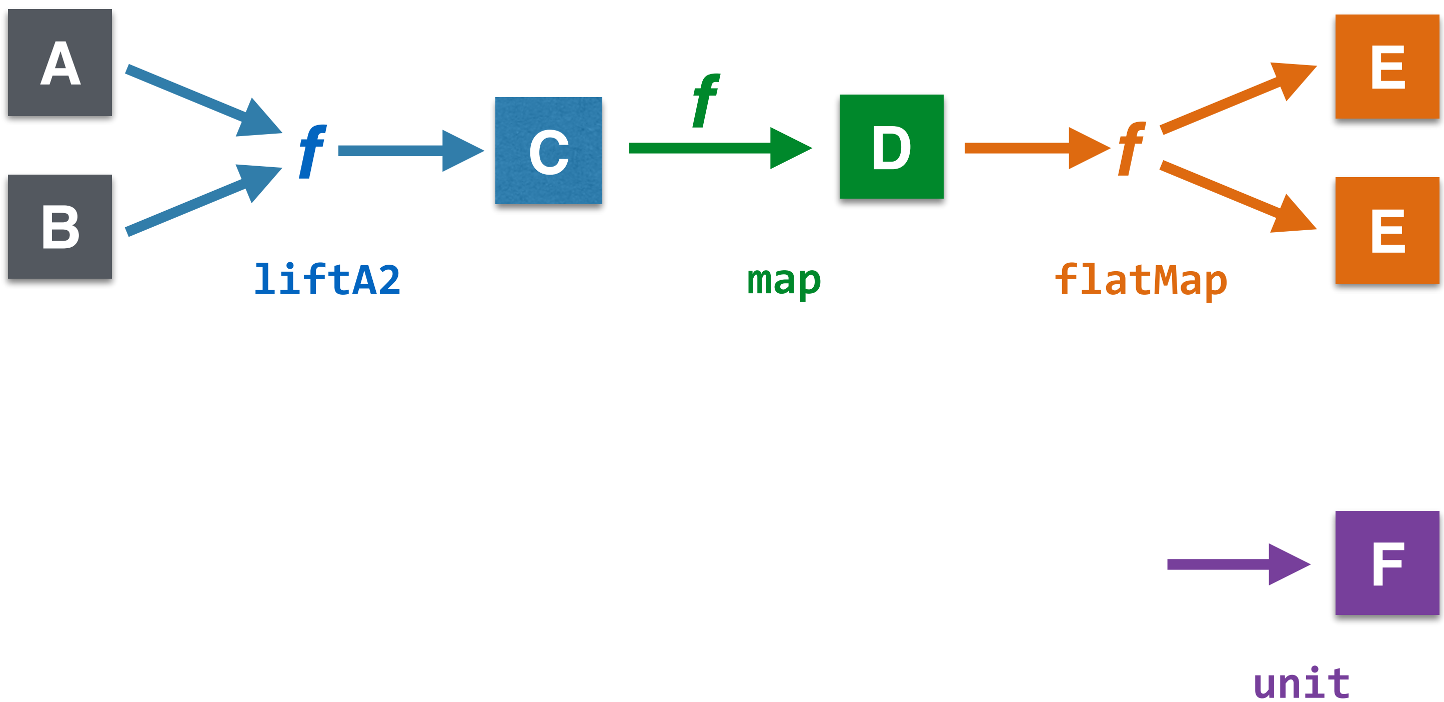
Operations



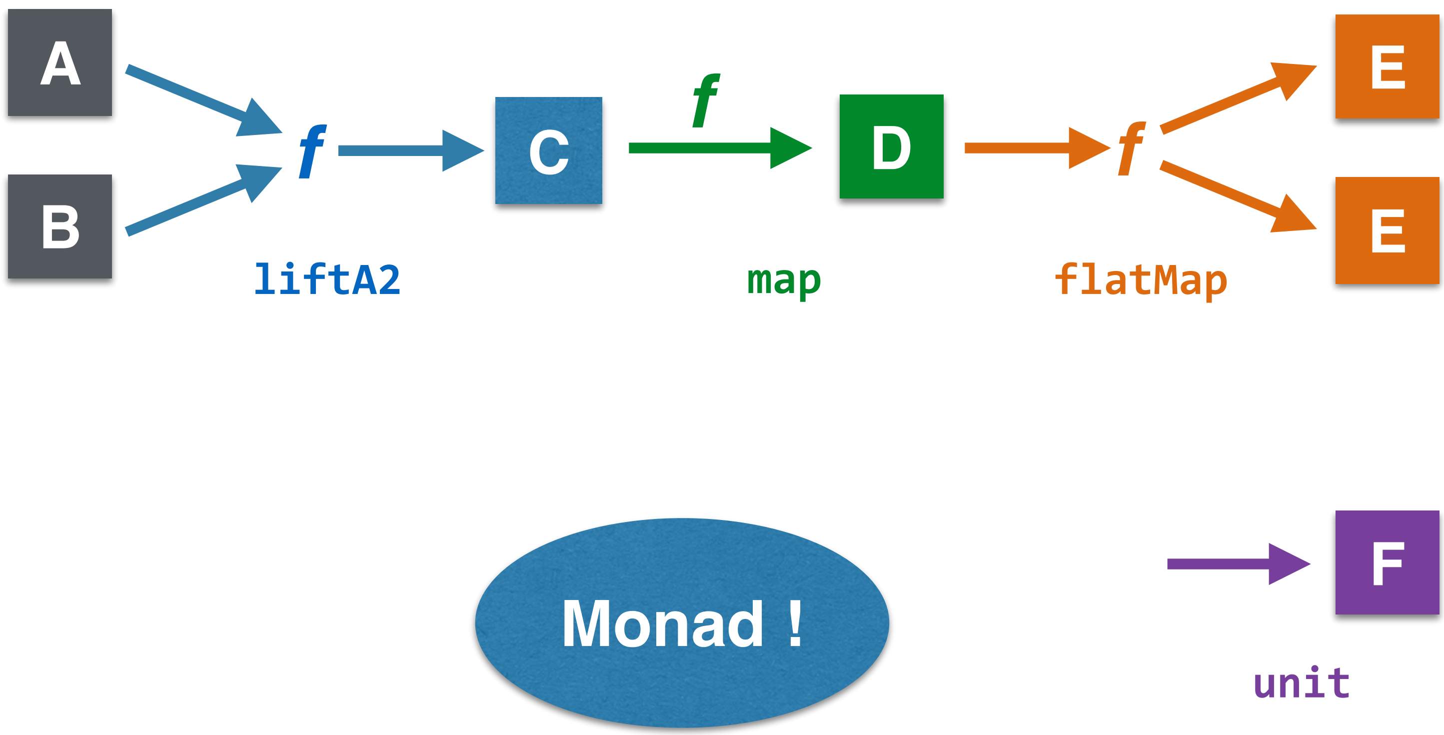
Operations



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?

Spare a Monad?

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

Spare a Monad?

- `Free[F[_],?]` is a **monad** for any **functor** `F`.
- `Coyoneda[S[_],?]` is a **functor** for any `S` at all.

Spare a Monad?

- `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.

Spare a Monad?

- `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?

- `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?

Free Monad

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

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

Wait, what?

Free Monad

```scala
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?

Free Monad

```scala
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)
```

Smart Ctors

Wait, what?

Free Monad

```
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)
```

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!

```scala
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!

```scala
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.

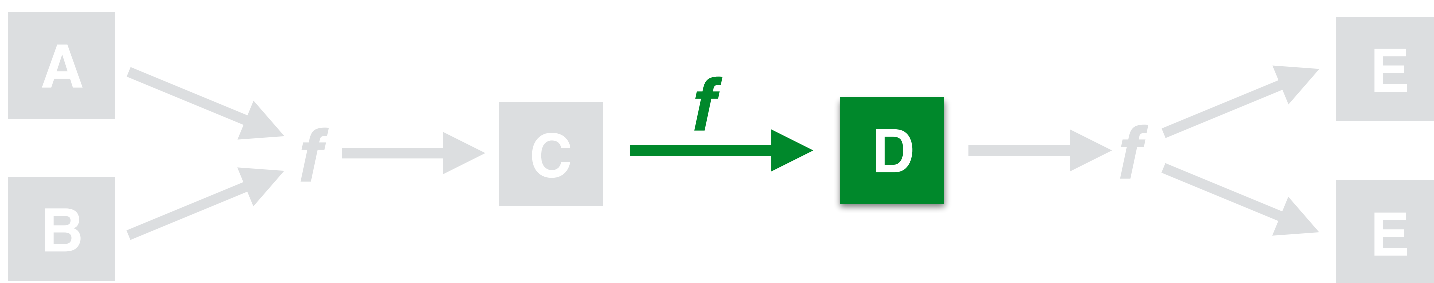
No ResultSet!

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

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

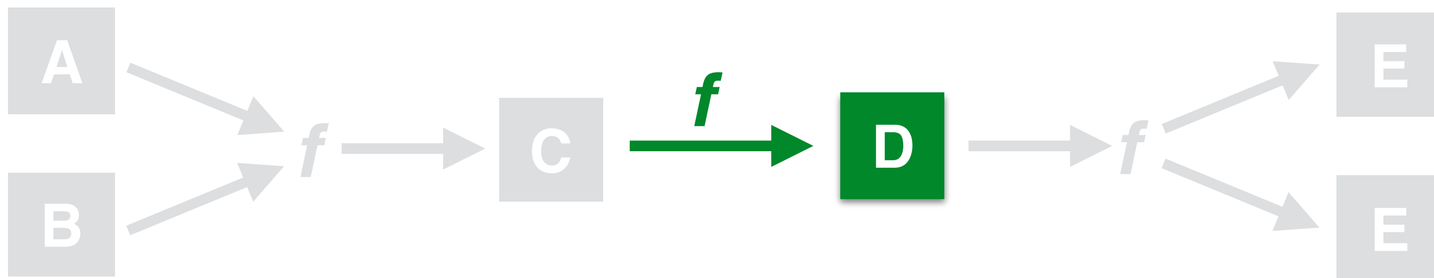
Values!

Composition!

Functor Operations

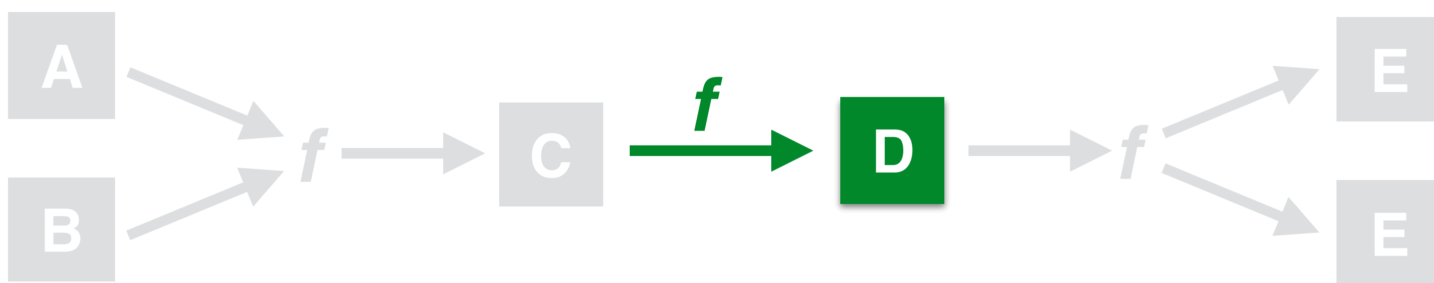


Functor Operations




```
// 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(_))
```

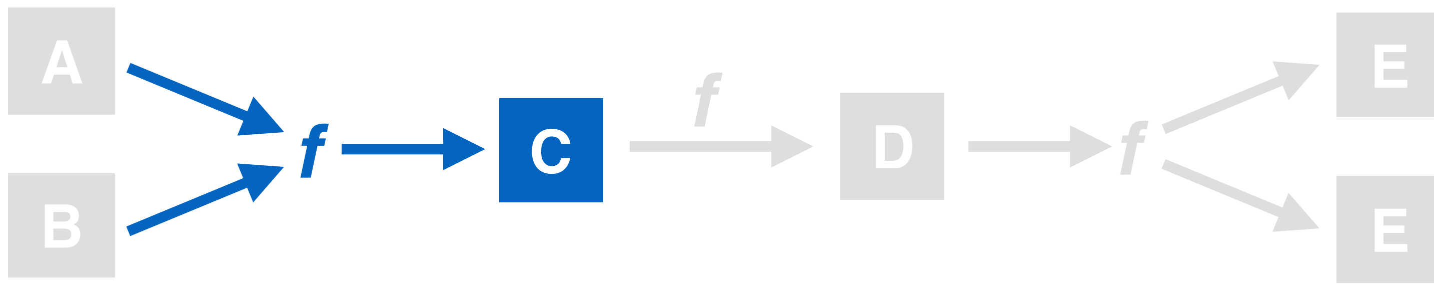
Functor Operations



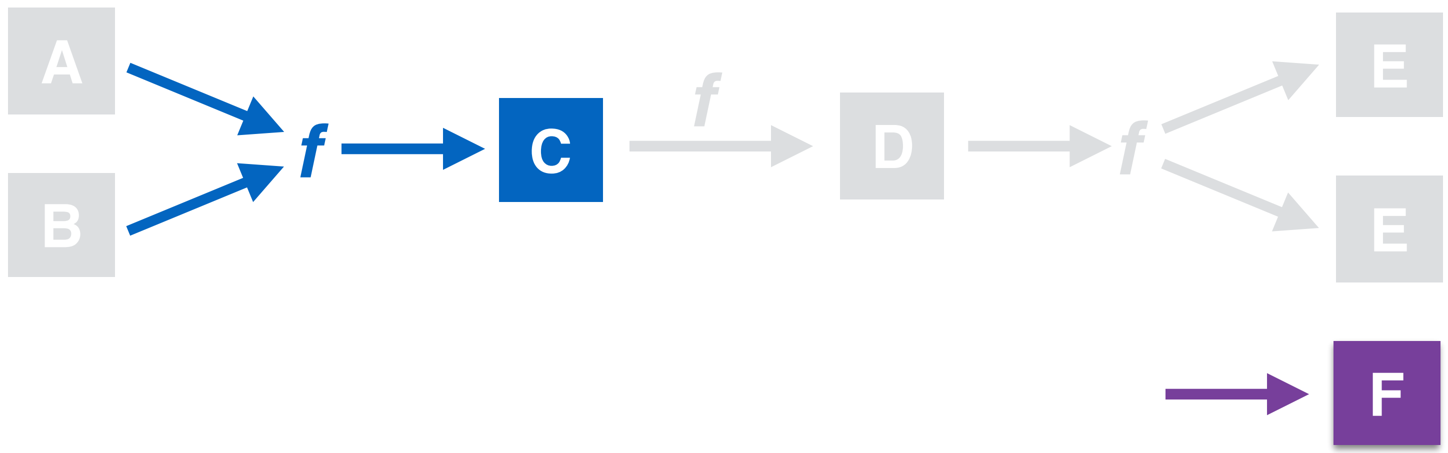

```scala
// Construct a program to read a Date at column n
def getDate(n: Int): ResultSetIO[java.util.Date] =
  getLong(n).map(new java.util.Date(_))
```

```
fpair
strengthL
strengthR
fproduct
as
void
```

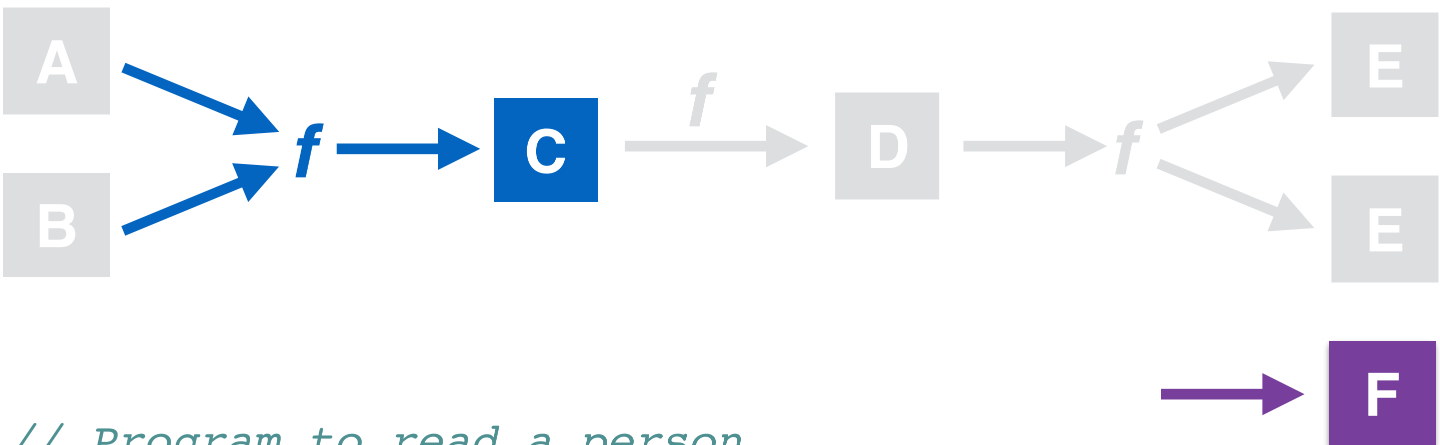
Applicative Operations



Applicative Operations

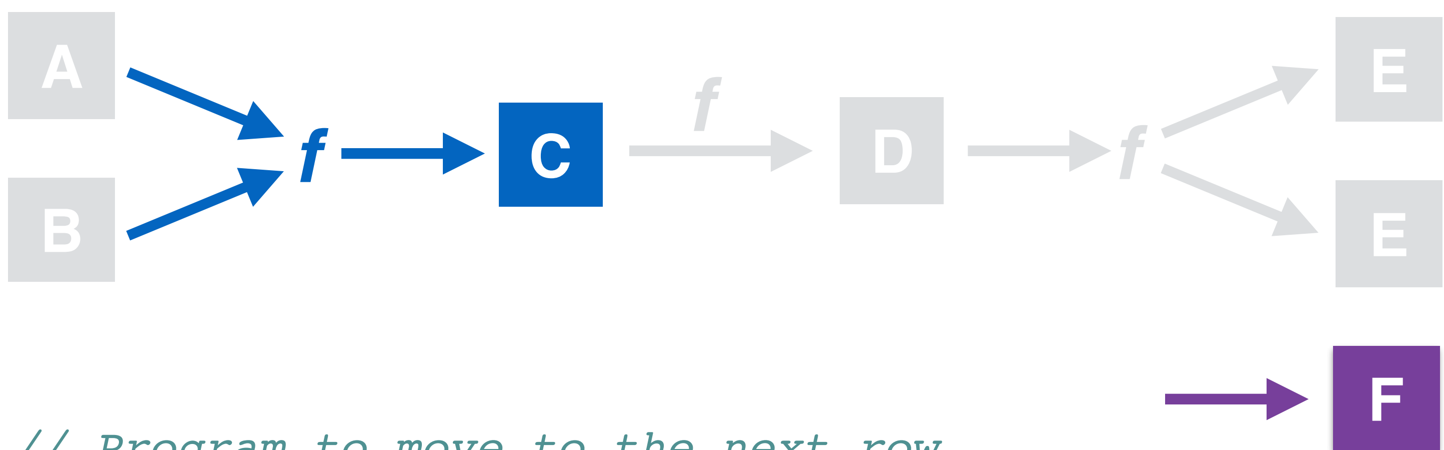


Applicative Operations



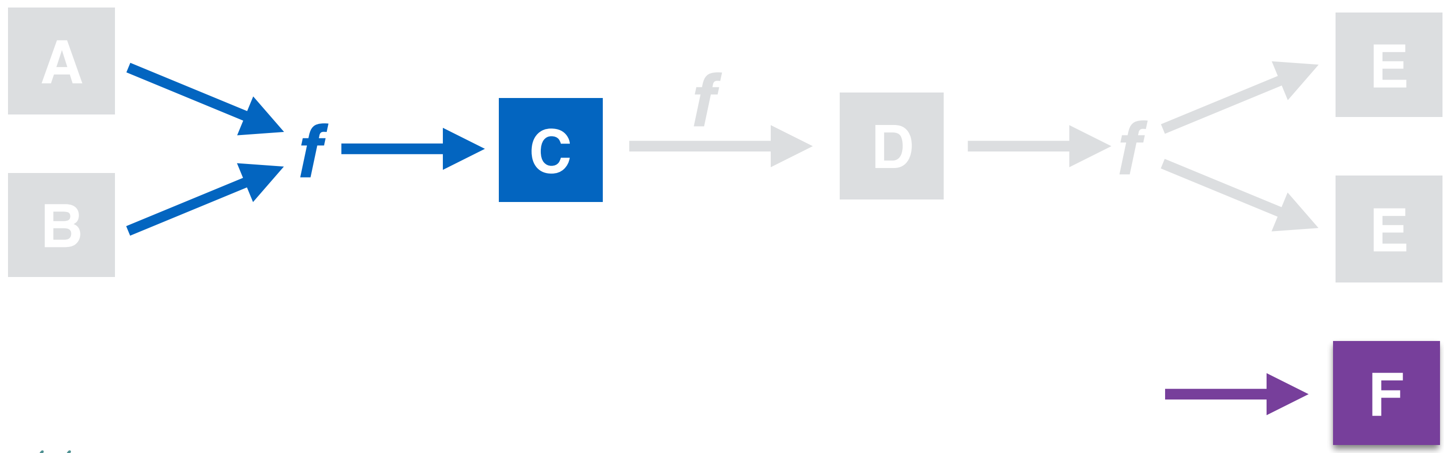

```
// Program to read a person
val getPerson: ResultSetIO[Person] =
  (getString(1) |@| getInt(2)) { (s, n) =>
    Person(s, n)
  }
```

Applicative Operations



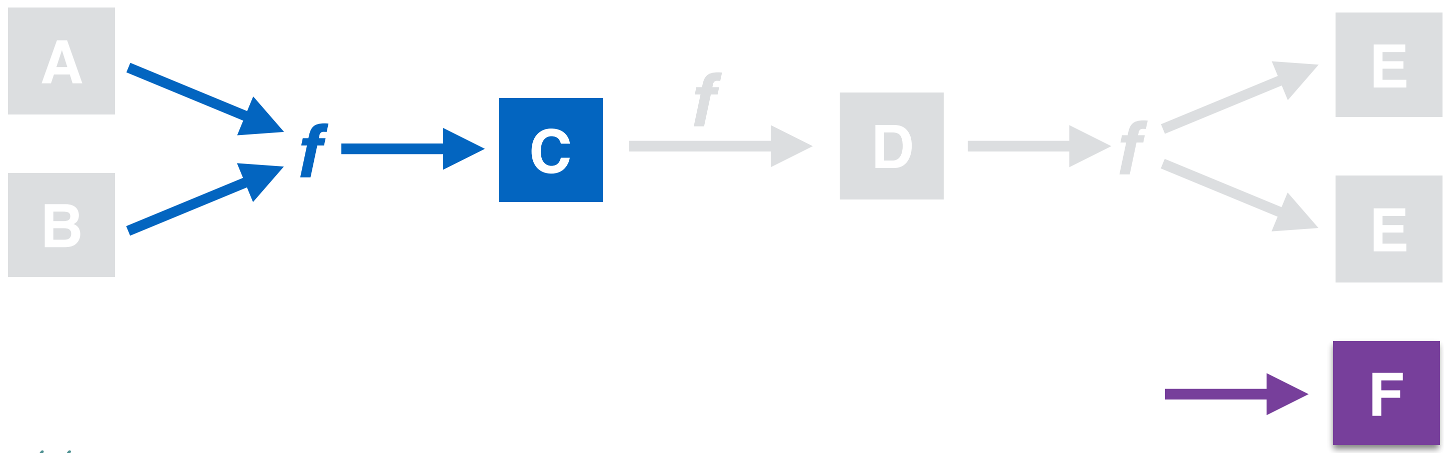

```
// Program to move to the next row
// and then read a person
val getNextPerson: ResultSetIO[Person] =
  next *> getPerson
```

Applicative Operations




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

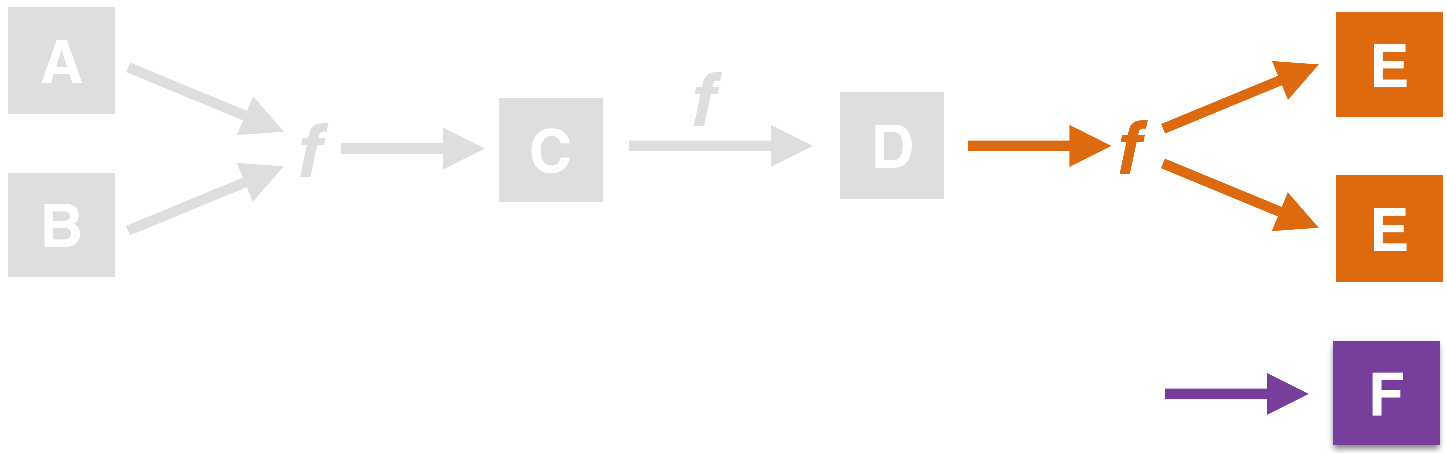
Applicative Operations



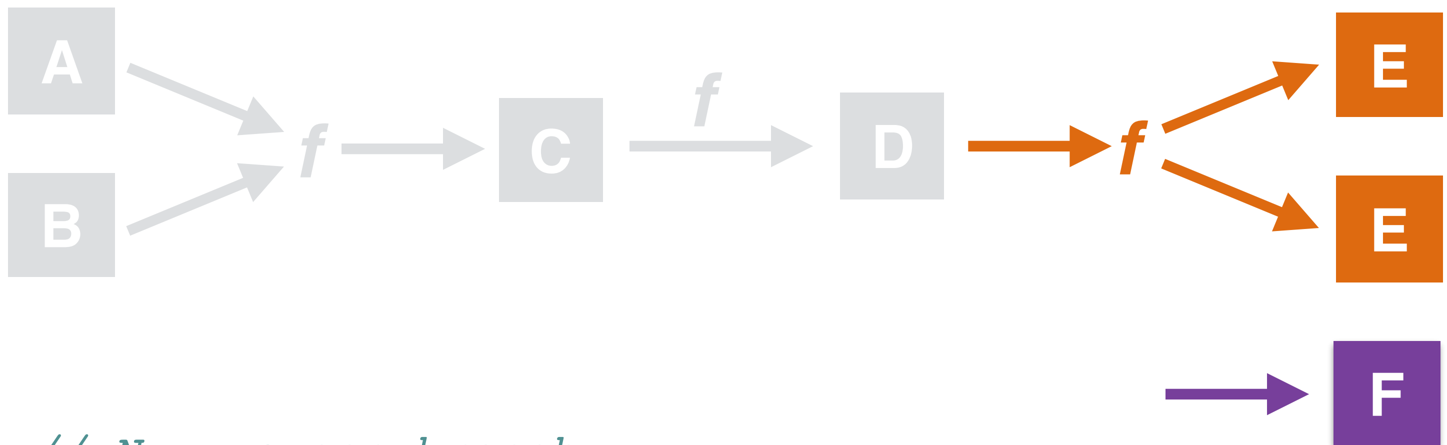

```
// Iterate!
def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)
```

Traverse[List]

Monad Operations

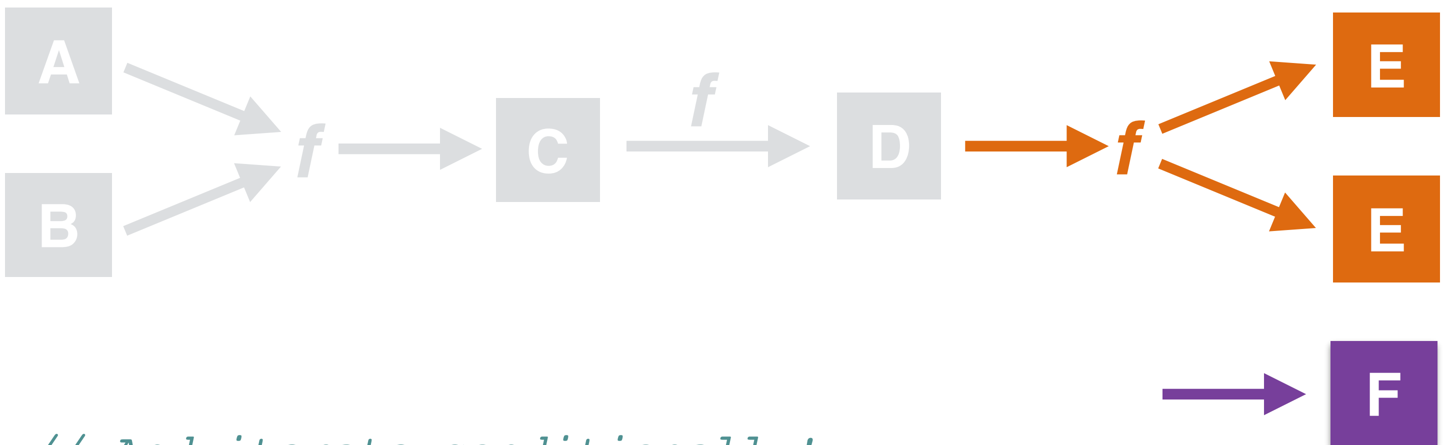


Monad Operations



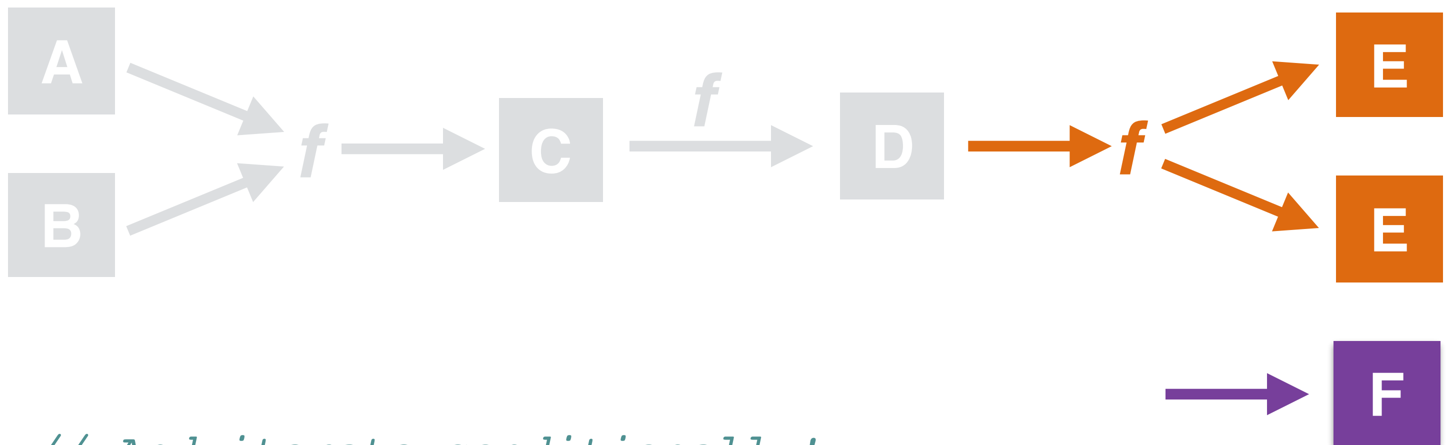

```scala
// Now we can branch
val getPersonOpt: ResultSetIO[Option[Person]] =
  next.flatMap {
    case true  => getPerson.map(_.some)
    case false => none.point[ResultSetIO]
  }
```

Monad Operations




```
// And iterate conditionally!
val getAllPeople: ResultSetIO[Vector[Person]] =
  getPerson.whileM[Vector](next)
```

Monad Operations




```
// And iterate conditionally!
val getAllPeople: ResultSetIO[Vector[Person]] =
  getPerson.whileM[Vector](next)
```

Okaaay…

Interpreting

Interpreting

- To "run" our program we **interpret** our algebra 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** our algebra 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** our algebra 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`.

- So we need a universally quantified function value; in scalaz we write it as `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`**

```scala
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

Interpreting

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

```scala
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

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

Program written in FreeC

Target

Natural Transformation

Running

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

Program written in FreeC

Natural Transformation

Target

```scala
val prog = getPerson.whileM[List](next)
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]                    CallableStatementIO[A]
ClobIO[A]                    ConnectionIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                 CallableStatementIO[A]
ClobIO[A]                 ConnectionIO[A]
DatabaseMetaDataIO[A]     DriverIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                  CallableStatementIO[A]
ClobIO[A]                  ConnectionIO[A]
DatabaseMetaDataIO[A]      DriverIO[A]
DriverManagerIO[A]         NClobIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                   CallableStatementIO[A]
ClobIO[A]                   ConnectionIO[A]
DatabaseMetaDataIO[A]       DriverIO[A]
DriverManagerIO[A]          NClobIO[A]
PreparedStatementIO[A]      RefIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                 CallableStatementIO[A]
ClobIO[A]                 ConnectionIO[A]
DatabaseMetaDataIO[A]     DriverIO[A]
DriverManagerIO[A]        NClobIO[A]
PreparedStatementIO[A]    RefIO[A]
ResultSetIO[A]            SQLDataIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                  CallableStatementIO[A]
ClobIO[A]                  ConnectionIO[A]
DatabaseMetaDataIO[A]      DriverIO[A]
DriverManagerIO[A]         NClobIO[A]
PreparedStatementIO[A]     RefIO[A]
ResultSetIO[A]             SQLDataIO[A]
SQLInputIO[A]              SQLOutputIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                    CallableStatementIO[A]
ClobIO[A]                    ConnectionIO[A]
DatabaseMetaDataIO[A]        DriverIO[A]
DriverManagerIO[A]           NClobIO[A]
PreparedStatementIO[A]       RefIO[A]
ResultSetIO[A]               SQLDataIO[A]
SQLInputIO[A]                SQLOutputIO[A]
StatementIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                 CallableStatementIO[A]
ClobIO[A]                 ConnectionIO[A]
DatabaseMetaDataIO[A]     DriverIO[A]
DriverManagerIO[A]        NClobIO[A]
PreparedStatementIO[A]    RefIO[A]
ResultSetIO[A]            SQLDataIO[A]
SQLInputIO[A]             SQLOutputIO[A]
StatementIO[A]
```

Fine. What's doobie?

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

```
BlobIO[A]                 CallableStatementIO[A]
ClobIO[A]                 ConnectionIO[A]
DatabaseMetaDataIO[A]     DriverIO[A]
DriverManagerIO[A]        NClobIO[A]
PreparedStatementIO[A]    RefIO[A]
ResultSetIO[A]            SQLDataIO[A]
SQLInputIO[A]             SQLOutputIO[A]
StatementIO[A]
```

- Pure functional support for all primitive operations.

Fine. What's doobie?

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

```
BlobIO[A]                    CallableStatementIO[A]
ClobIO[A]                    ConnectionIO[A]
DatabaseMetaDataIO[A]        DriverIO[A]
DriverManagerIO[A]           NClobIO[A]
PreparedStatementIO[A]       RefIO[A]
ResultSetIO[A]               SQLDataIO[A]
SQLInputIO[A]                SQLOutputIO[A]
StatementIO[A]
```

- Pure functional support for all primitive operations.

- Low-level API is **Machine-generated** ✨

Exception Handling

```
val ma = ConnectionIO[A]

ma.attempt // ConnectionIO[Throwable \/ A]

// General
ma.attemptSome(handler)
ma.except(handler)
ma.exceptSome(handler)
ma.onException(action)
ma.ensuring(sequel)

// SQLException
ma.attemptSql
ma.attemptSqlState
ma.attemptSomeSqlState(handler)
ma.exceptSql(handler)
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

```scala
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

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

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

Generalize to Products

Mapping via Typeclass

```scala
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

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

get[Person]
```

Mapping via Typeclass

```scala
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]] =
  get[Person].whileM[List](next)
```

Streaming

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

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

Streaming

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

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

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

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
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]
```

Typed

Yields Countries

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,849
Country(IND,India,447114
Country(JPN,Japan,3787042.
```

Transactor[Task]

Statement Checking

```
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]
```

Statement Checking

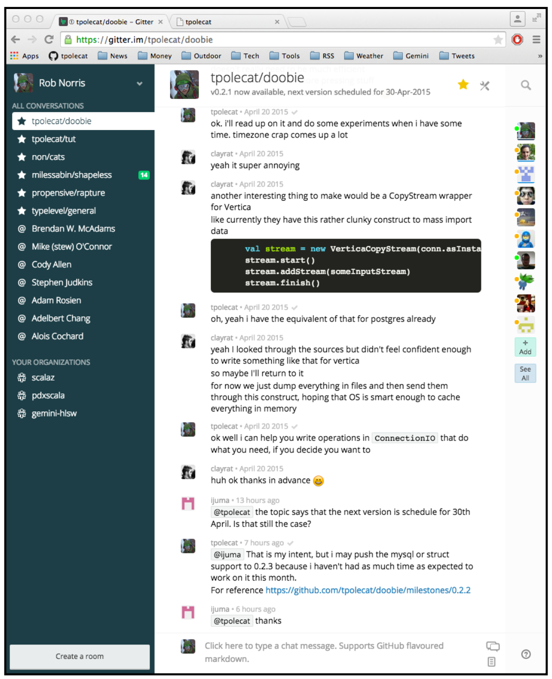
```
scala> biggerThan(0).check.run

  select code, name, gnp from country where popu

  ✓ 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...

https://github.com/tpolecat/doobie

gitter



book of doobie