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 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**. 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)
}
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 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 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 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.
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.

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
Operations

\[ f \]

\[ \text{A} \quad \text{B} \quad \text{C} \]
Operations

\[ A \xrightarrow{f} B \xrightarrow{f} C \xrightarrow{f} D \]
Operations
Operations

A \xrightarrow{f} B \xrightarrow{f} C \xrightarrow{f} D \xrightarrow{f} E\quad E\quad F
Operations

\[
\begin{align*}
A & \xrightarrow{f} C \\
B & \xrightarrow{f} C \\
C & \xrightarrow{f} D \\
D & \xrightarrow{f} E \\
D & \xrightarrow{f} E \\
E & \xrightarrow{\text{unit}} F
\end{align*}
\]
Operations

\[
\text{flatMap} \\
\text{unit}
\]
Operations

A \xrightarrow{f} C \xrightarrow{f} D \xrightarrow{f} E

B \xrightarrow{f} C

map

flatMap

E \xrightarrow{f} E

F

unit
Operations

A \xrightarrow{f} C \xrightarrow{f} D \xrightarrow{f} E

B \xrightarrow{liftA2} C

D \xrightarrow{map} E

D \xrightarrow{flatMap} E

F \xrightarrow{unit}
Operations

- liftA2
- map
- flatMap
- unit
If only…

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

```scala
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:

```scala
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?

• \texttt{Free[F[\_],\_]} is a \texttt{monad} for any \texttt{functor F}.
Spare a Monad?

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

• **scalaz** abbreviates this type as **FreeC** \([S, ?]\)
Spare a Monad?

- \textbf{Free} [\textit{F} [\_], ?] \textit{is a monad} for any \textit{functor} \textit{F}.

- \textbf{Coyoneda} [\textit{S} [\_], ?] \textit{is a functor} for any \textit{S} at all.

- By substitution, \textbf{Free} [\textbf{Coyoneda} [\textit{S} [\_], ?], ?] \textit{is a monad} for any \textit{S} at all.

- scalaz abbreviates this type as \textbf{FreeC} [\textit{S}, ?]

- Set \textit{S} = \textbf{ResultSetOp} and watch what happens.
Wait, what?

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

type ResultSetIO[A] = FreeC[ResultSetOp, A]
import scalaz.{ Free, Coyoneda }
import scalaz.Free.{ FreeC, liftFC }

type ResultSetIO[A] = FreeC[ResultSetOp, A]
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)
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)
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)
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)
Now we can write programs in `ResultSetIO` using familiar monadic style.

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

val getPerson: ResultSetIO[Person] =
  for {
    name <- getString(1)
    age  <- getInt(2)
  } yield Person(name, age)
```
Now we can write programs in `ResultSetIO` using familiar monadic style.

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

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

**No ResultSet!**
Now we can write programs in `ResultSetIO` using familiar monadic style.

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

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

**Composition!**

**Values!**

**No ResultSet!**

**ResultSetIO**
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(_))
// 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(_))
Applicative Operations

\[ \text{A} \xrightarrow{f} \text{B} \xrightarrow{f} \text{C} \xrightarrow{f} \text{D} \xrightarrow{f} \text{E} \]
Applicative Operations
Applicative Operations

// Program to read a person
val getPerson: ResultSetIO[Person] =
  (getString(1) |@| getInt(2)) { (s, n) =>
    Person(s, n)
  }
// Program to move to the next row
// and then read a person
val getNextPerson: ResultSetIO[Person] =
  next *> getPerson
// Iterate!

def getPeople(n: Int): ResultSetIO[List[Person]] =
    getNextPerson.replicateM(n)
Applicative Operations

// Iterate!

def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)
Monad Operations

\[ A \xrightarrow{f} C \xrightarrow{f} D \xrightarrow{f} E F \]
Monad Operations

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

A \[f\] C \[f\] D \[f\] E \[f\] F
// 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`

```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
      }
  }
```
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
      }
  }

Here we interpret into `scalaz.effect.IO`
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)
        caseGetInt(i) => IO(rs.getInt(i))
        caseGetString(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))
Running

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

val prog = getPersontList(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
- ClobIO
- CallableStatementIO
- ConnectionIO
Fine. What's doobie?

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

<table>
<thead>
<tr>
<th>BlobIO[A]</th>
<th>CallableStatementIO[A]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClobIO[A]</td>
<td>ConnectionIO[A]</td>
</tr>
<tr>
<td>DatabaseMetaDataIO[A]</td>
<td>DriverIO[A]</td>
</tr>
</tbody>
</table>
Fine. What's doobie?

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

<table>
<thead>
<tr>
<th>BlobIO[A]</th>
<th>CallableStatementIO[A]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClobIO[A]</td>
<td>ConnectionIO[A]</td>
</tr>
<tr>
<td>DatabaseMetaDataIO[A]</td>
<td>DriverIO[A]</td>
</tr>
<tr>
<td>DriverManagerIO[A]</td>
<td>NClobIO[A]</td>
</tr>
</tbody>
</table>
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]
  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.
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]
  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]
  - 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.

- Low-level API is **Machine-generated ✨**
val ma = ConnectionIO[A]

ma.attempt // ConnectionIO[Throwable \ A]

// General
ma.attemptSome(handler)
ma.exceptions[(handler)]
ma.exceptionsSome[(handler)]
ma.onException(action)
ma.ensuring(sequel)

// SQLException
ma.attemptSql
ma.attemptSqlState
ma.exceptionsSql[(handler)]
ma.exceptionsSqlState[(handler)]
ma.exceptionsSomeSqlState[(handler)]

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

Exception Handling
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)
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]] =
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

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

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,84982.00)
Country(IND,India,447114.00)
Country(JPN,Japan,3787042.00)
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 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]

Can also do this in your unit tests!
Much More...

https://github.com/tpolecat/doobie

gitter

book of doobie