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**.
- Programs are **composable** values.

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**.
- Programs are **composable** values.
- Lather, rinse, **repeat**. This is a great strategy for making terrible things enjoyable.

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)
}
```

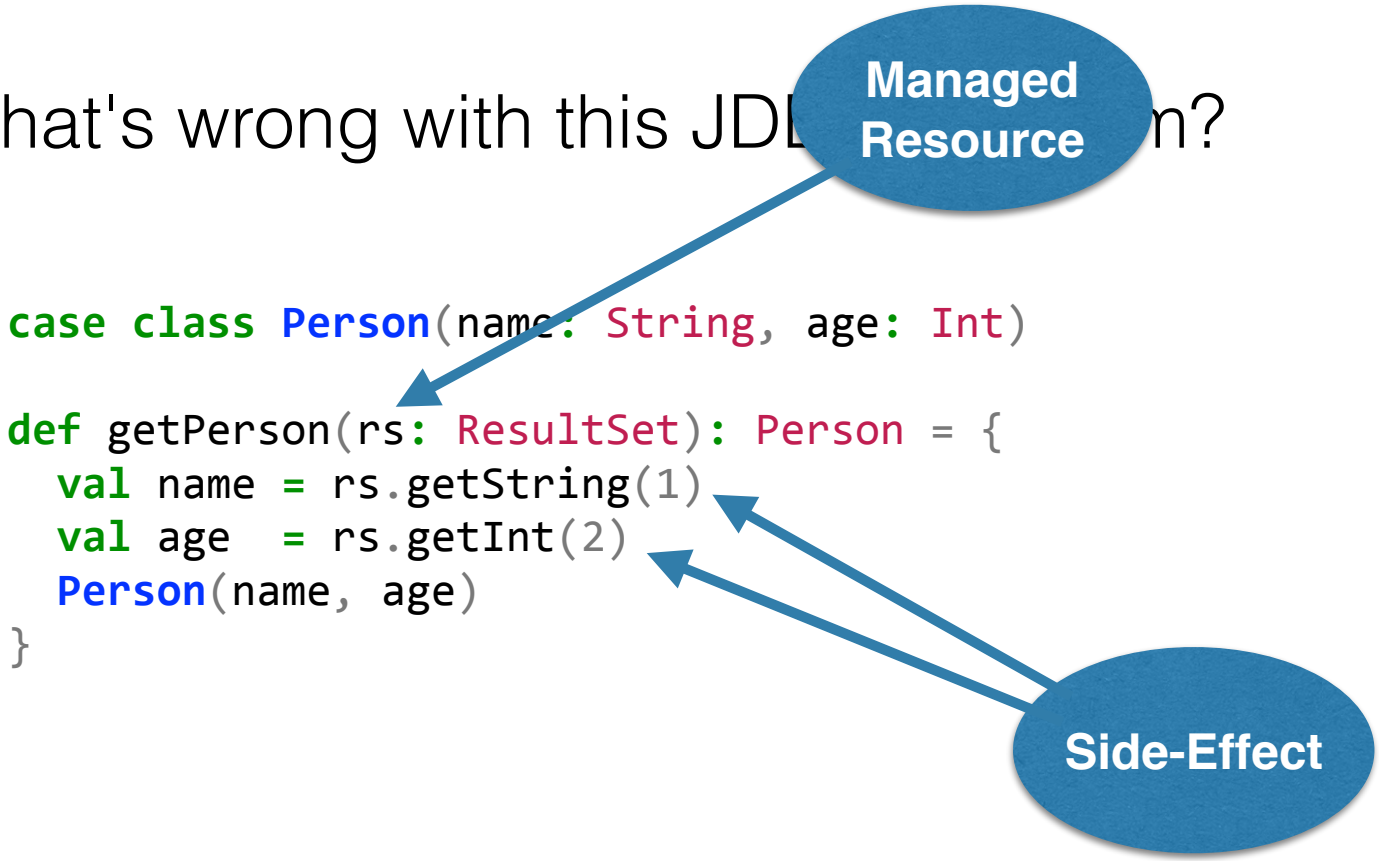
The Problem

So what's wrong with this JDBC code?


```
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)
}
```

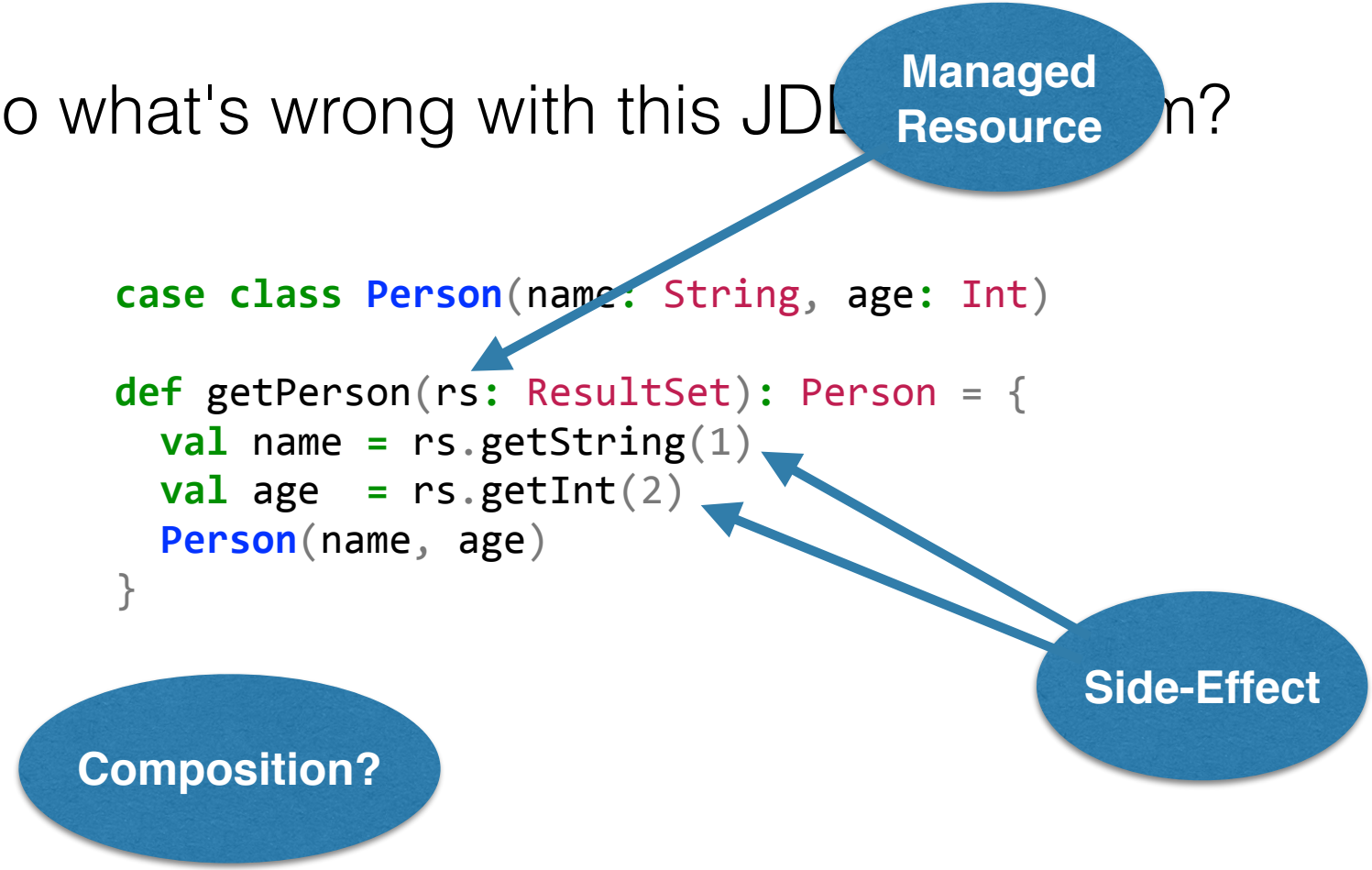
The Problem

So what's wrong with this JDBC program?



The Problem

So what's wrong with this JDBC program?



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.

Operations

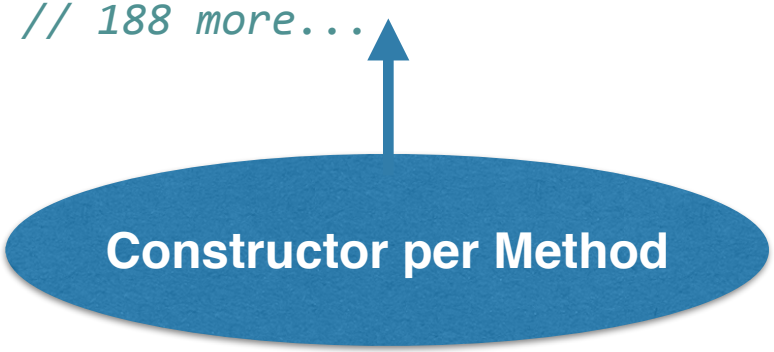
An algebra is just a set of objects, along with rules for manipulating them. Here our objects are the set of operations you can perform with a JDBC ResultSet.

```
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...
```

Operations

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

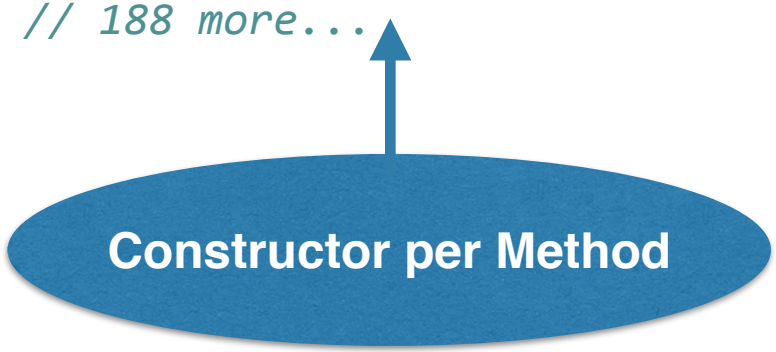
```
sealed trait ResultSetOp[A]

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...
```

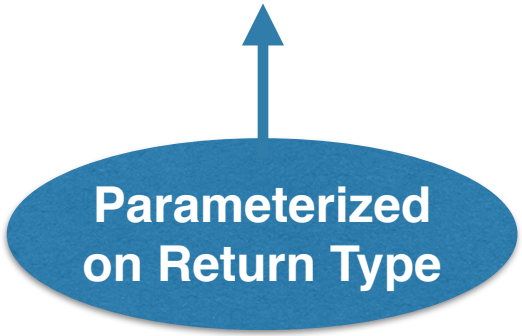


Operations

An algebra is just a set of objects, along with rules for manipulating them. Here our objects are the set of operations 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...
```





Combining

Combining

- `ResultSetOp[A]` means that we will produce an `A`

Combining

- `ResultSetOp[A]` means that we will produce an `A`
- We need to be able to look at this `A` and decide what to do next ... some `ResultSetOp[B]`.

Combining

- `ResultSetOp[A]` means that we will produce an `A`
- We need to be able to look at this `A` and decide what to do next ... some `ResultSetOp[B]`.
- So the general combining rule looks like this:

Combining

- `ResultSetOp[A]` means that we will produce an `A`
- We need to be able to look at this `A` and decide what to do next ... some `ResultSetOp[B]`.
- So the general combining rule looks like this:

```
ResultSetOp[A] => (A => ResultSetOp[B]) => ResultSetOp[B]
```

Combining

- `ResultSetOp[A]` means that we will produce an `A`
- We need to be able to look at this `A` and decide what to do next ... some `ResultSetOp[B]`.
- So the general combining rule looks like this:

```
ResultSetOp[A] => (A => ResultSetOp[B]) => ResultSetOp[B]
```

- We also need a way to yield arbitrary results:

Combining

- `ResultSetOp[A]` means that we will produce an `A`
- We need to be able to look at this `A` and decide what to do next ... some `ResultSetOp[B]`.
- So the general combining rule looks like this:

```
ResultSetOp[A] => (A => ResultSetOp[B]) => ResultSetOp[B]
```

- We also need a way to yield arbitrary results:

```
A => ResultSetOp[A]
```

Combining

- `ResultSetOp[A]` means that we will produce an `A`
- We need to be able to look at this `A` and decide what to do next ... some `ResultSetOp[B]`.
- So the general combining rule looks like this:

```
ResultSetOp[A] => (A => ResultSetOp[B]) => ResultSetOp[B]
```

- We also need a way to yield arbitrary results:

```
A => ResultSetOp[A]
```

- Look familiar?

Combining

build out a rationale for flatMap and unit

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

Spare a Monad?

Fancy words. Please be seated.

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

Spare a Monad?

Fancy words. Please be seated.

- `Free[F[_],?]` is a **monad** for any **functor** `F`.
- `Coyoneda[F[_],?]` is a **functor** for any `F` at all.
- By substitution, `Free[Coyoneda[F[_],?],?]` is a **monad** for any `F` at all.
- Set `F` = `ResultSetOp` and watch what happens.

Wait, what?

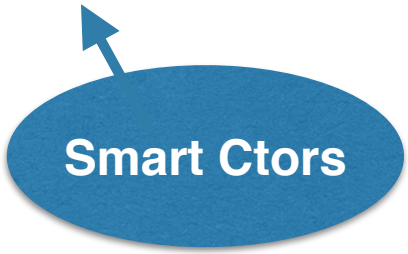
```
import scalaz.{ Free, Coyoneda }
import scalaz.Free.liftFC

type ResultSetIO[A] = Free[Coyoneda[ResultSetOp, ?], A]
```

Wait, what?

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

type ResultSetIO[A] = Free[Coyoneda[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)
```

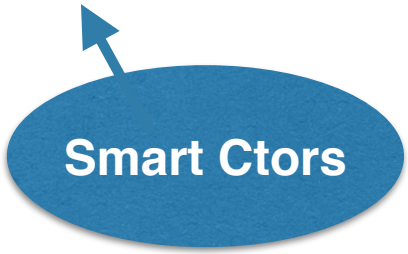
Wait, what?

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

type ResultSetIO[A] = Free[Coyoneda[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)
```



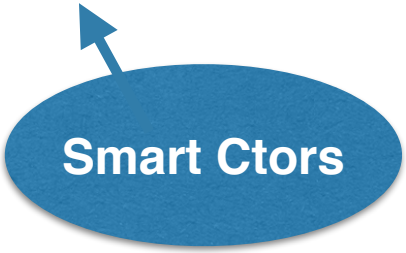
Wait, what?

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

type ResultSetIO[A] = Free[Coyoneda[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.liftFC

type ResultSetIO[A] = Free[Coyoneda[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)
```



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.


```
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.


```
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!

Composition!

Functor Operations

```scala
val fa: ResultSetIO[A] = ...

fa.map(a => b)       // ResultSetIO[B]

fa.fpair             // ResultSetIO[(A, A)]

fa.strengthL(42)     // ResultSetIO[(Int, A)]
fa.strengthR(42)     // ResultSetIO[(A, Int)]

fa.fproduct(a => b) // ResultSetIO[(A, B)]

fa.as("foo")         // ResultSetIO[String]
fa >| "foo"          // same thing

fa.void              // ResultSetIO[Unit]
```

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(_))
```

Applicative Composition

```
val fa: ResultSetIO[A] = ...
val fb: ResultSetIO[B] = ...

fa *> fb     // ResultSetIO[B]
fa <* fb     // ResultSetIO[A]

fa.tuple(fb) // ResultSetIO[(A, B)]

(fa |@| fb) { (a, b) => c } // ResultSetIO[C]

fa.replicateM(42)  // ResultSetIO[List[A]]
```

Applicative Composition

```scala
// 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

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

Traversable Functors

```scala
val fa: ResultSetIO[A] = ...

List(fa, fa, fa).sequenceU // ResultSetIO[List[A]]

def foo(n: Int): ResultSetIO[A] = ...

List(1,2,3).traverseU(foo) // ResultSetIO[List[A]]
```

Traversable Functors

```
// Assume we can read a person with a given ID
def readPersonById(id: Long): ConnectionIO[Person] =
  ...

// We can then read a list of people
def readPeopleByIds(ids: List[Long]): ConnectionIO[List[Person]] =
  ids.traverseU(readPerson)
```

Monadic Composition

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

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

Monadic Composition

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

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

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

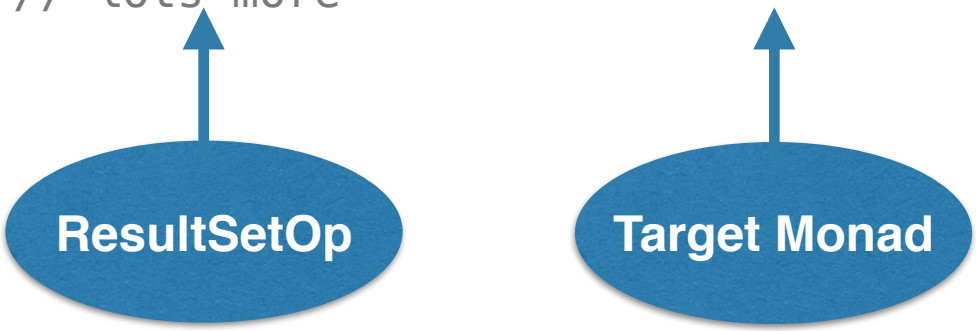
Here we interpret into **`scalaz.effect.IO`**

```
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
      }
  }
```



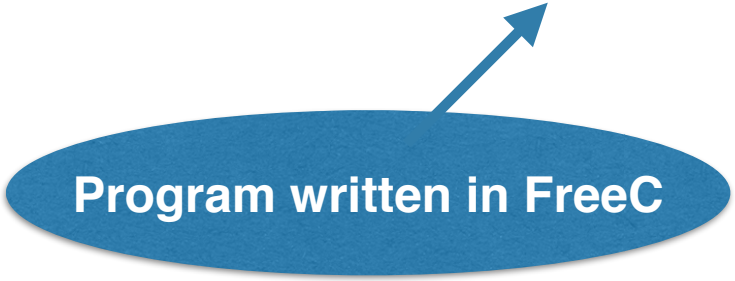
Running

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

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

Running

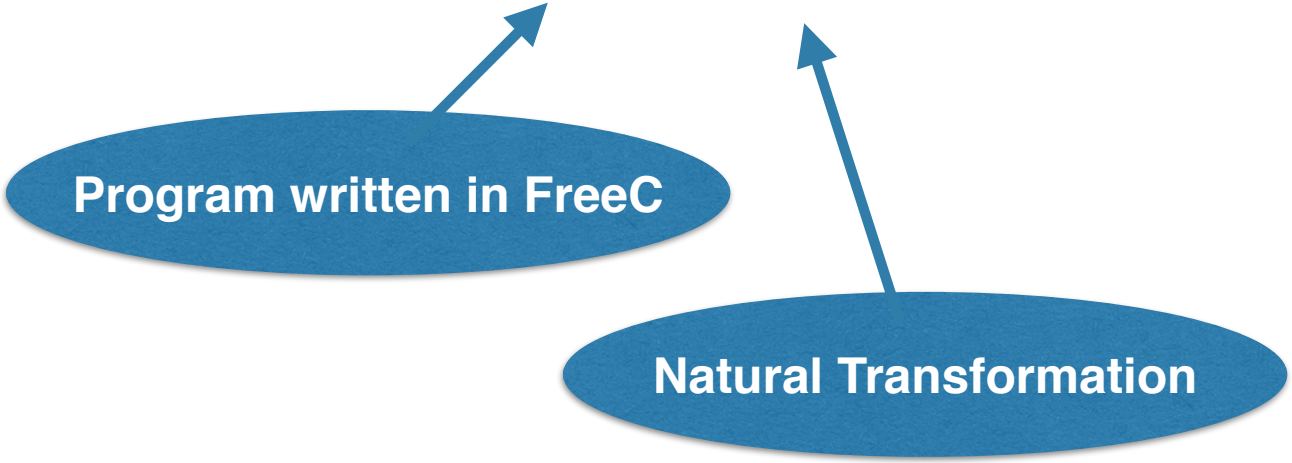
```scala
def toIO[A](a: ResultSetIO[A], rs: ResultSet): IO[A] =
  Free.runFC(a)(trans(rs))
```




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

Running

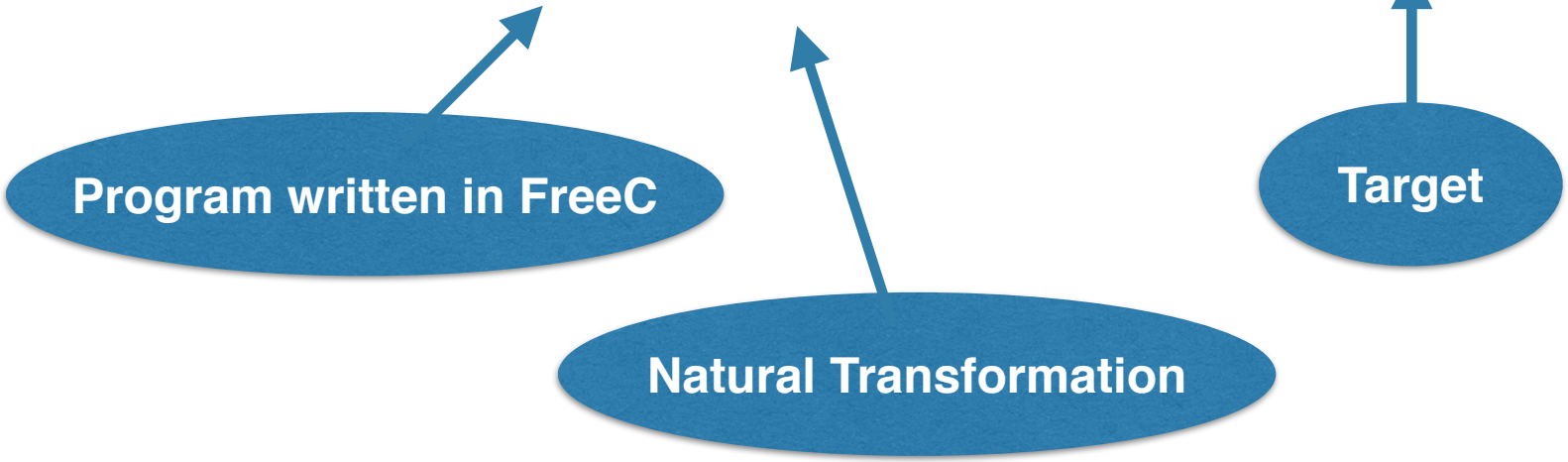
```
def toIO[A](a: ResultSetIO[A], rs: ResultSet): IO[A] =
  Free.runFC(a)(trans(rs))
```




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

Running

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




```
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]                    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.

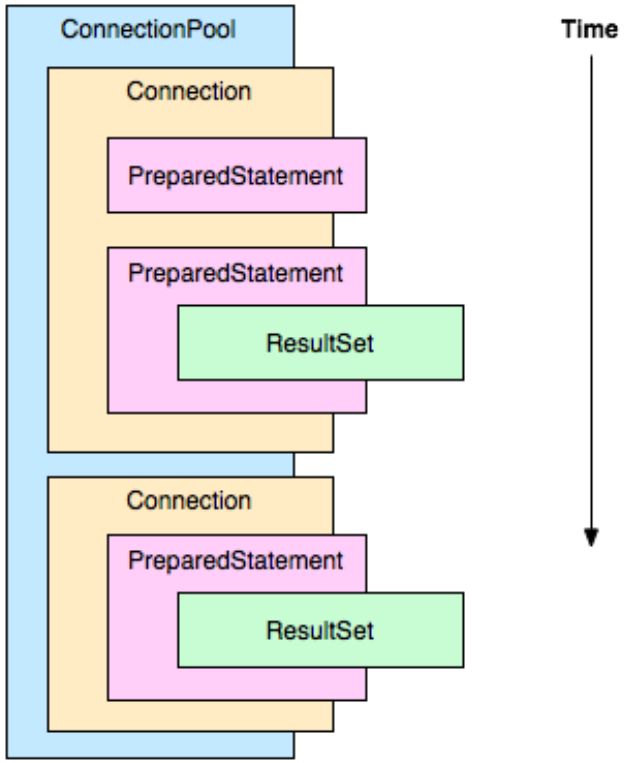
```
BlobIO[A]                    CallableStatementIO[A]
ClobIO[A]                    ConnectionIO[A]
DatabaseMetaDataIO[A]        DriverIO[A]
DriverManagerIO[A]           NClobIO[A]
PreparedStatementIO[A]       RefIO[A]
ResultSetIO[A]               SQLDataIO[A]
SQLInputIO[A]                SQLOutputIO[A]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

- Pure functional support for all primitive operations.
- 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.ensuring(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)
...
```

Embedding



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

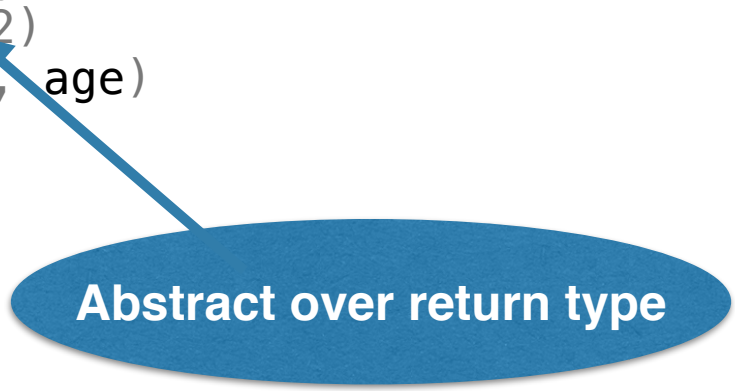
```
// A high-level ConnectionIO constructor
// Prepare a statement and handle it with the provided program, always closing safely.
def prepareStatement(sql: String)(k: PreparedStatementIO[A]): ConnectionIO[A] =
  C.prepareStatement(sql).flatMap(s => C.liftPreparedStatement(s, k ensuring PS.close))
```

Mapping via Typeclass

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

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

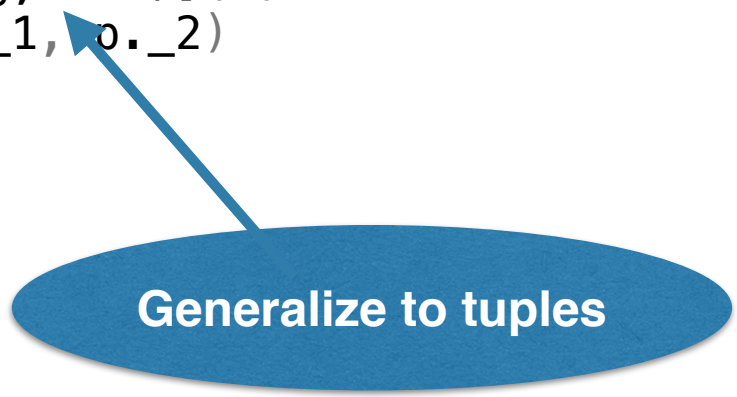
Mapping via Typeclass

```scala
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

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

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



Mapping via Typeclass

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

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

Mapping via Typeclass

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

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

Mapping via Typeclass

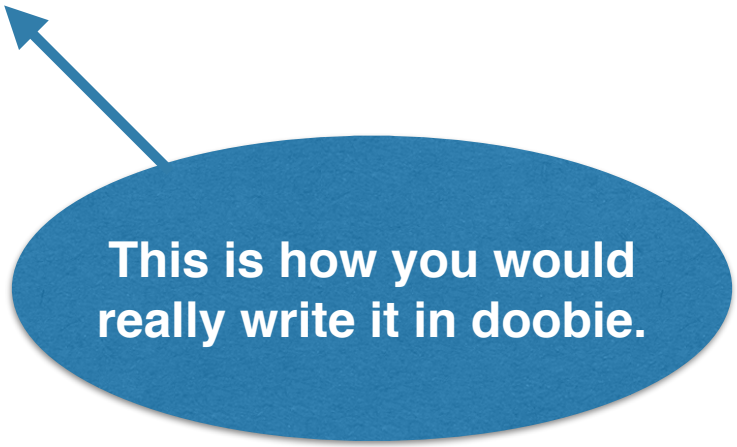
```scala
case class Person(name: String, age: Int)

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]
```



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]]
```

SQL Literals

```
scala> (sql"select code, name, population, gnp from country"
     |   .query[Country] // Query0[Country]
     |   .process        // Process[ConnectionIO, Country]
     |   .take(5)        // Process[ConnectionIO, Country]
     |   .list           // ConnectionIO[List[Country]]
     |   .transact(xa)   // Task[List[Country]]
     |   .run            // List[Country]
     |   .foreach(println))
  Country(AFG,Afghanistan,22720000,Some(5976.0))
  Country(NLD,Netherlands,15864000,Some(371362.0))
  Country(ANT,Netherlands Antilles,217000,Some(1941.0))
  Country(ALB,Albania,3401200,Some(3205.0))
  Country(DZA,Algeria,31471000,Some(49982.0))
```

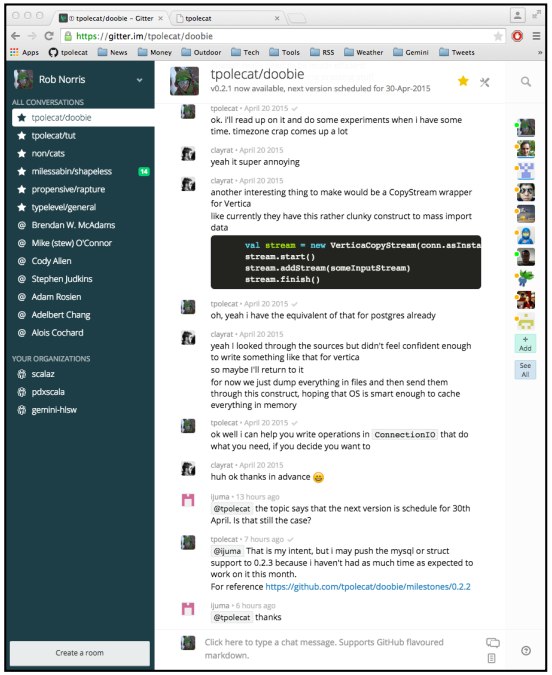
Much More

- **Connection Pooling** with HikariCP, easily extended for other pooling implementations.
- **Syntax aplenty**, to make fancy types easier to work with.
- **YOLO Mode** for fast REPL experimentation without guilt.
- **PostgreSQL Support**: Geometric Types, Arrays, PostGIS types, LISTEN/NOTIFY, CopyIn/Out, Large Objects, …
- **Typechecked SQL Literals** in the REPL and via a Specs2 trait.

Moar Info

https://github.com/tpolecat/doobie

gitter



book of doobie