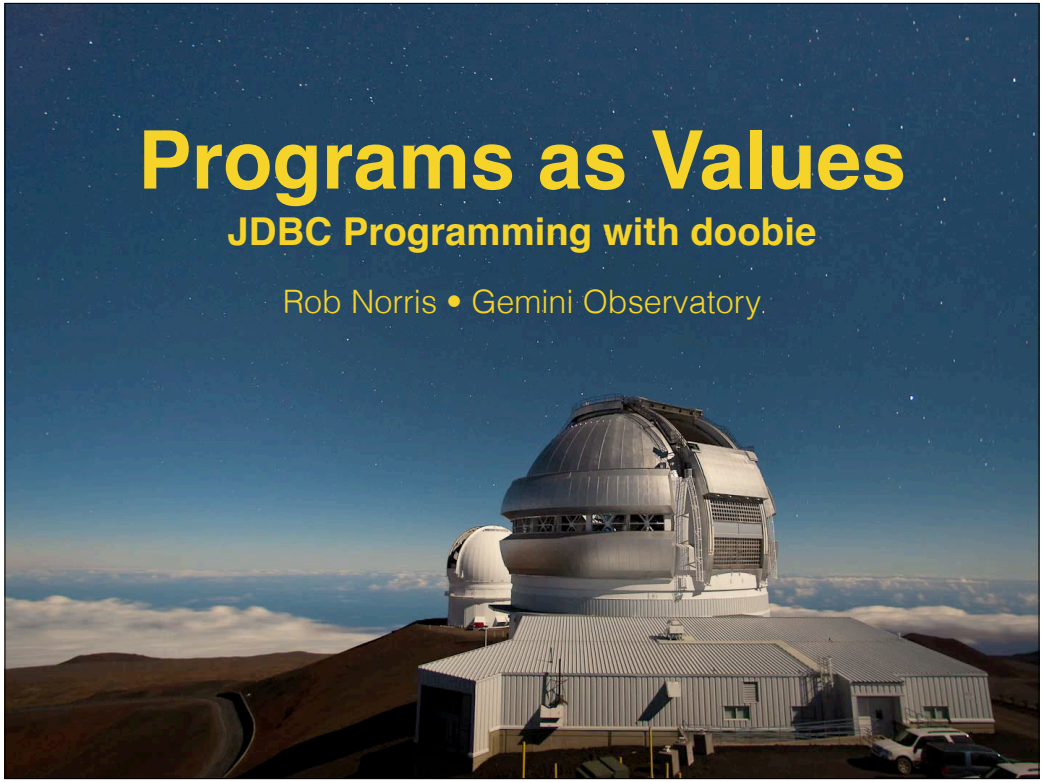


- My name's Rob
- I work for the Gemini Observatory, writing scala software that helps people use this big telescope in Hawaii and another one in Chile.
- This is my only cool slide.

What's this about?

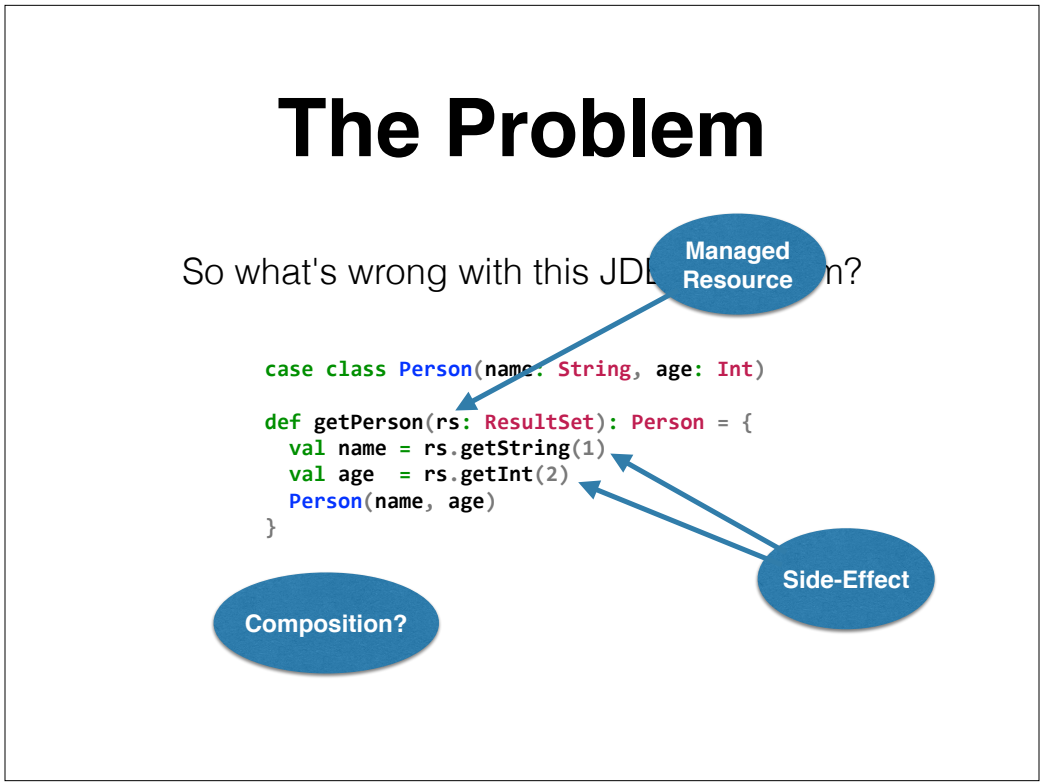
This is a talk about **doobie**, a pure functional database access layer for Scala.

- 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.

I've been working on it for a couple years in my spare time, it's had a few releases, people are using it … so I'd like to talk about how it works and what I think is interesting about it.

So the takeaways …

- not really JDBC's fault … it has an impossible task
- if you saw David Hoyt's talk from yesterday you already know this, but you'll get a chance here to hit it again from a different direction.
- The big thing I want to encourage is try to shift away from operational thinking and toward composition as a way to construct programs. This is how we deal with complexity. Operational thinking doesn't scale.



- The resultset is a lifetime-managed object; if you leak it by assigning to a var or letting it hop onto a future, you're breaking the contract.

- The method calls on **rs** are side-effecting; the result depends on the internal state of the resultset. **and** they can throw exceptions!

- It's not obviously composable ... I don't see anything here that looks like a composable abstraction; it's just side-effecting imperative code.

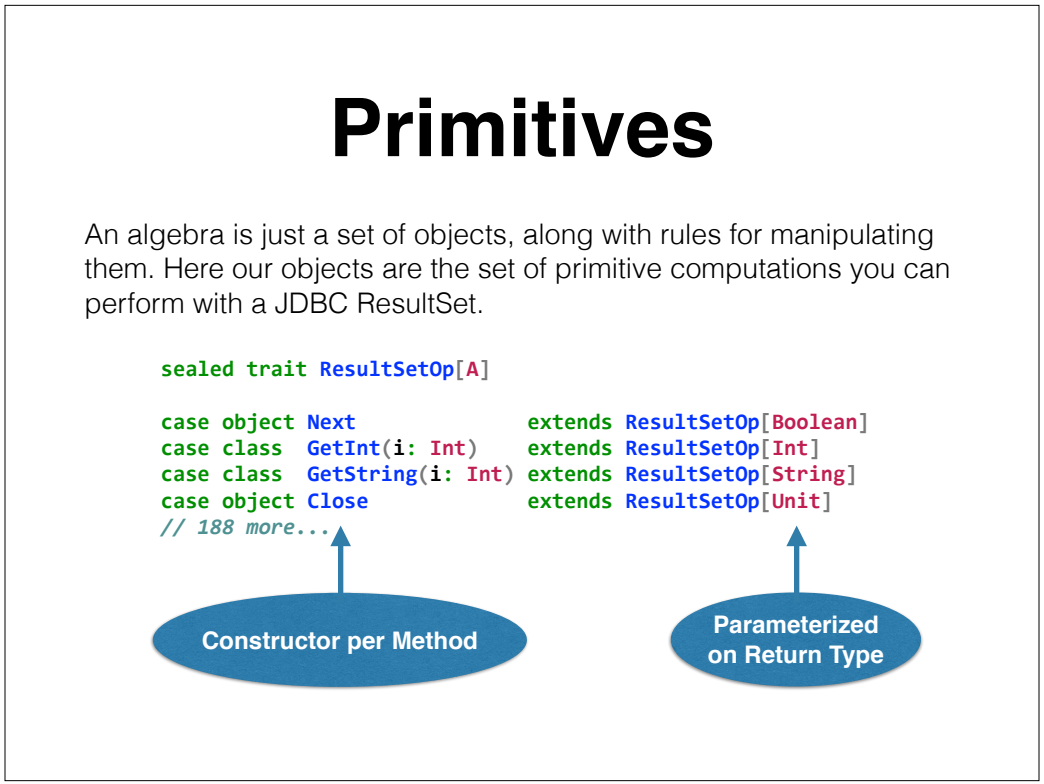
So *my claim is that* the way we deal with this is by turning this program into a data structure. And you're just going to have to trust me for a few slides, that the benefits of this are worth it.

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.

... so what does this look like?

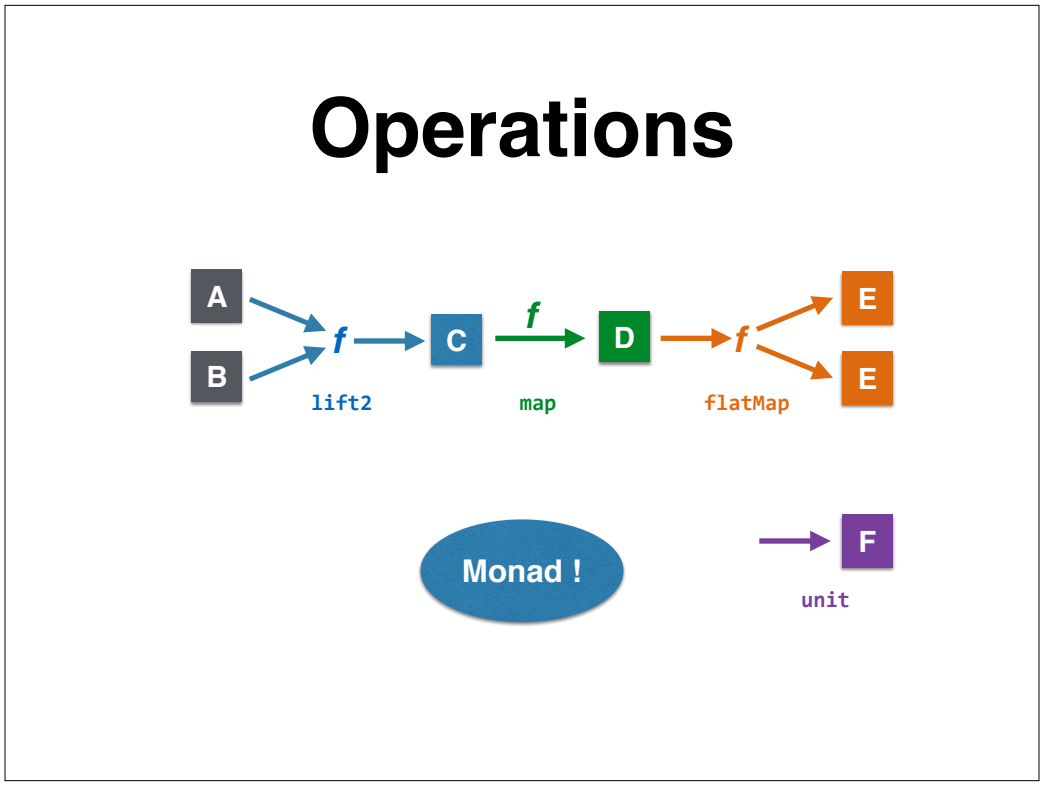


First thing we do is define a data type for our operations.

And here we're going to use as our example, operations you can perform on a ResultSet.

-
-

So we can think of these values as little programs. They **describe** things we want to do. And we could for instance a list of operations that want to perform, and we could write a really simple interpreter that takes the list and a result set and crunches through and executes instructions but you can't really use this to compute useful results. You can get a List[Any] back but that's not useful.



So I'm going to describe some operations for composition that I claim are useful to have.

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)
```

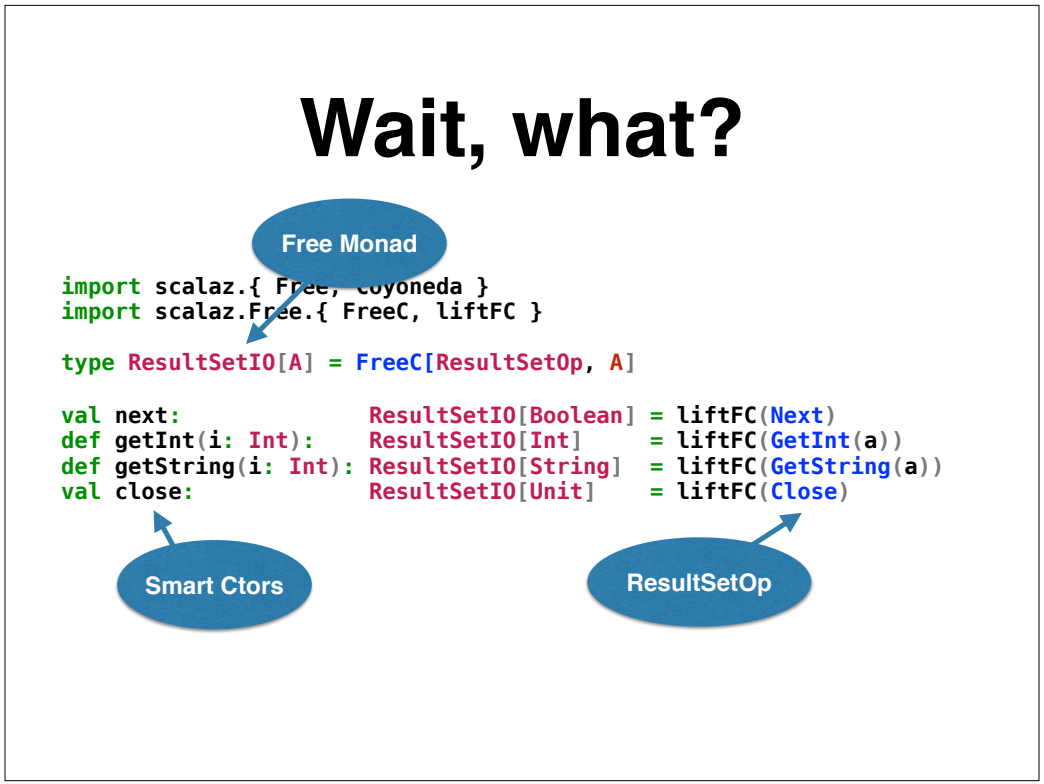
But we don't.

Spare a Monad?

Fancy words. Please be seated.

- `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.
- Abbreviated as `FreeC[S[_],?]`
- And if `S` = `ResultSetOp` we now have a monad.

- data type called Free



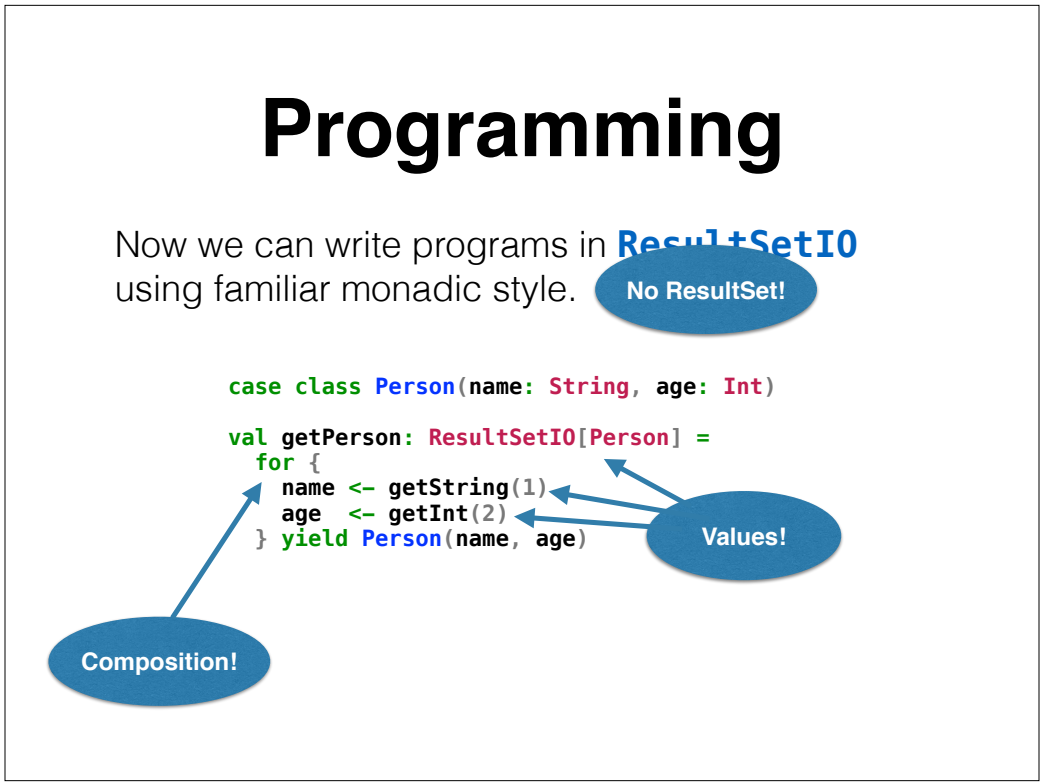
So we have this *new type* and we'll call it ResultSetIO

• for each ResultSetOp • we have a smart constructor • that constructs an instance of this new ResultSetIO type by using this liftFC function that scalaz gives us.

- and ResultSetIO is our free monad

ResultSetOp is not a monad. It's not even a functor, it's just a simple data type.

But ResultSetIO **is** a monad. And you see there's a correspondence between the ctors of ResultSetIO and ResultSetOp.

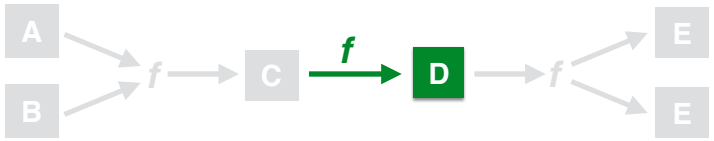


so this program ...

and comparing with our original program, we see some improvements.

- this is just a value … it doesn't "do" anything.
- no resultset … you can't leak a reference because you don't *have* a reference. It appears nowhere in any of these types.
- and now we're composing two little programs and making a bigger one. And we could keep doing this indefinitely and you'll always have a ResultSetIO[Something]. The surface complexity remains constant.

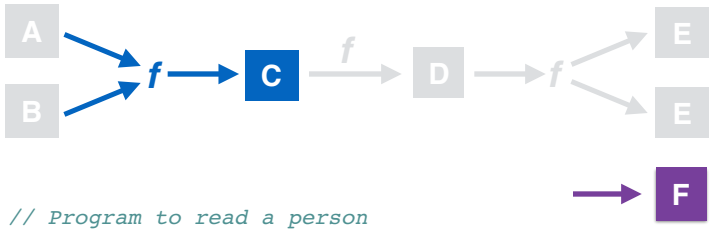
Functor Operations




```
// 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

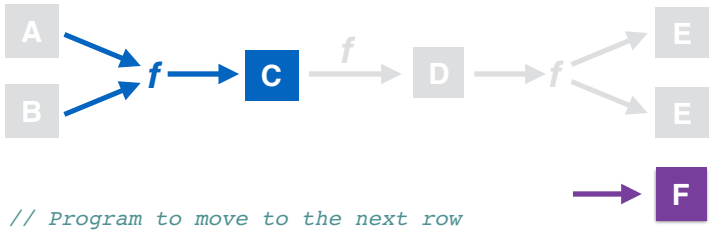



```
// Program to read a person
val getPerson: ResultSetIO[Person] =
  (getString(1) |@| getInt(2)) { (s, n) =>
    Person(s, n)
  }
```

So because our getString and getInt don't depend on one another we can use this applicative composition which is weaker than flatMap.

So what else can we do?

Applicative Operations

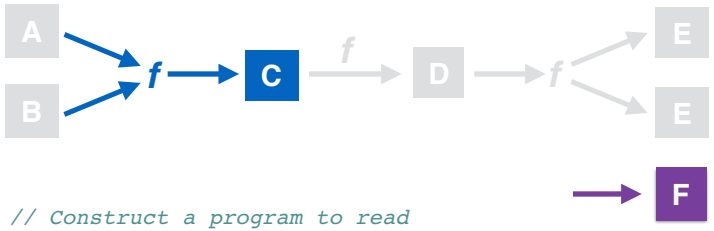



```
// Program to move to the next row
// and then read a person
val getNextPerson: ResultSetIO[Person] =
  next *> getPerson
```

Say we have this program to move to the next row we can compose it with getPerson just by discard the boolean value it returns (we'll be optimistic and assume there's another row).

Ok so what else?

Applicative Operations




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

now we can take getNextPerson and replicate it some number of times to return a *List* of people.

So how does that work?

Applicative Operations

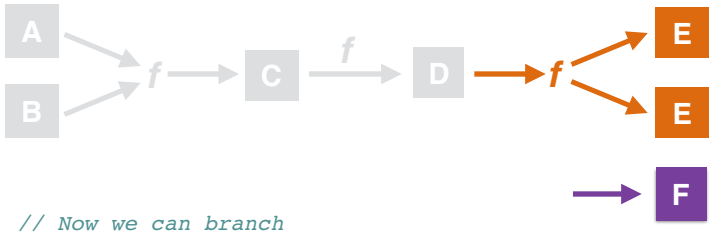
```
// Implementation of replicateM
def getPeople(n: Int): ResultSetIO[List[Person]] =
  getNextPerson.replicateM(n)

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

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

- first we make a list of programs

- then we call *sequence* which swaps the type constructors and gives us a program that produces a list

Ok now think about what we have been doing. We're just following the types. We're using these compositional operations and just working with these programs like any other kind of data, which I think is really cool.

Monad Operations

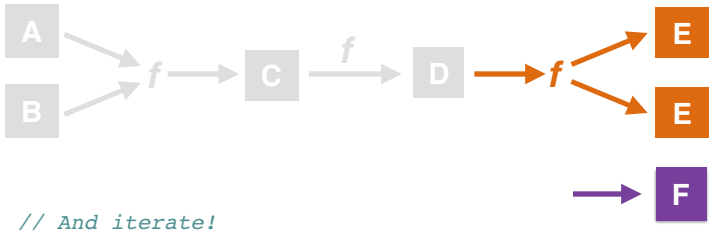



```
// Now we can branch
val getPersonOpt: ResultSetIO[Option[Person]] =
  next.flatMap {
    case true  => getPerson.map(_.some)
    case false => none.point[ResultSetIO]
  }
```

So now we can *branch* (we don't have to be optimistic anymore). We can take the result of Next and if it's true then the next step is you get a person and lift it into Option, otherwise it's just a program that returns none.

And we can iterate too.

Monad Operations




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

Seriously

So this is a totally legit way to read a resultset into a vector of some structured type. And I think it's a really beautiful

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?

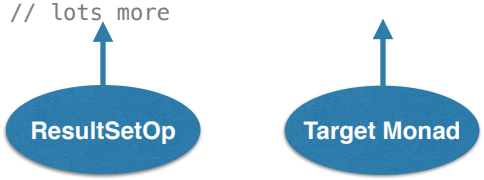
Interpreting

- 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
      }
  }
```

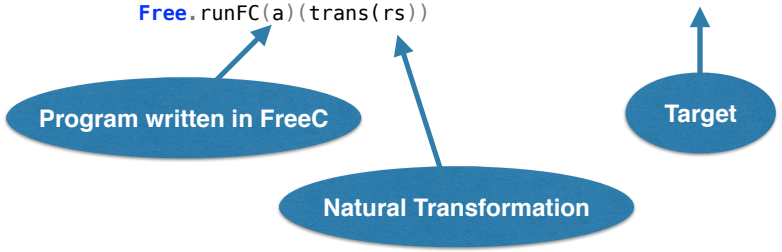


Given a resultset we can construct a ~> from ResultSetOp to IO • it just maps each constructor • to a corresponding IO primitive that operates on our resultset.

Ok, and now what?

Running

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




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

Well now we can write this method

- given a program written in FreeC
- and a ~> which we defined on the previous slide

And here's our program that reads the resultset into a Vector, and now you can actually run it and get your result.

Now notice that we never had to explain how map or flatMap work. We just borrowed some machinery that let us

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 (!)

•

• So if you need access to vendor-specific functionality, it's all there, with a consistent composable functional API all the way down to the metal.

You may have been thinking that this is kind of boilerplatey?

• and it is. The free algebras and interpreters are machine generated. And that accounts for more than 70% of the code in the core.

Ok. So this stuff composes nicely, but it's still really low level. So let's look at some things that we've built up on top of this layer.

Exception Handling

```
val ma = ConnectionIO[A]

ma.attempt // ConnectionIO[Throwable \/ A]
fail(wtf)  // ConnectionIO[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)
...
```

so we have these two primitive operations: attempt, which takes a program that computes an A and gives a program that computes a Throwable \/ A, and then there's a constructor that can cause a failure, which is useful if you want to examine a failure and then let it continue to propagate.

and from these we can construct ...

Mapping via Typeclass

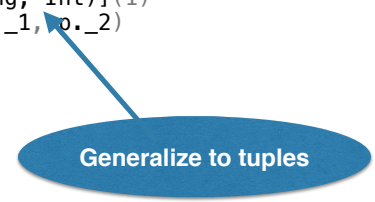
```scala
case class Person(name: String, age: Int)

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

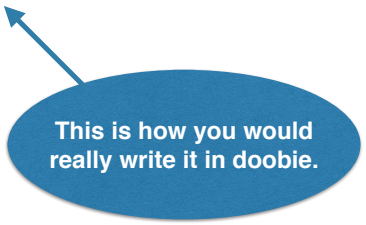
```scala
case class Person(name: String, age: Int)

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

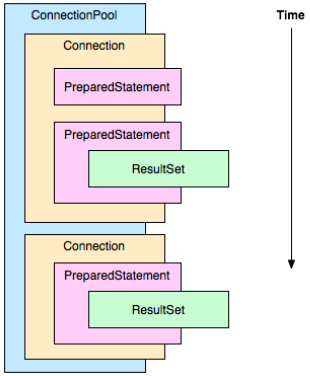
```
case class Person(name: String, age: Int)

get[Person]
```



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



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.

... so let's look at that.

So the structure of a database program looks something like this. You have a connection pool that probably lasts for the lifetime of your program. You have **interactions** with the database on a connection, in which you prepare and execute statements, which might produce resultsets that you have to process. And in doobie the programs that you write at each step have different types.

- But doobie allows you to click them together to make bigger programs that handle these large interactions.
- And for some common types of interactions there's a nice high-level API.

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

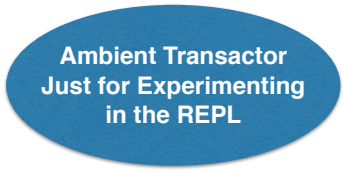
Transactor[Task]

well here's a REPL session

•

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)
```



-

... another nice thing you can do in YOLO mode is typecheck your queries.

YOLO MODE

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

  select code, name, gnp from country where popul

  ✓ 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!

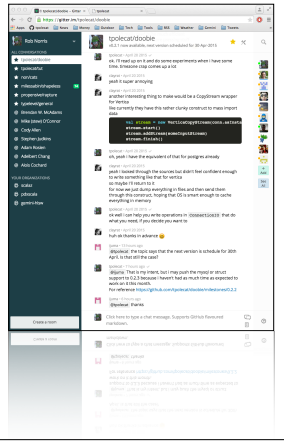
Much More

- 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, …
- … but works with **any JDBC driver** so people are using it with H2, MySQL, MS-SQL, Hive, etc.

Thanks!

https://github.com/tpolecat/doobie

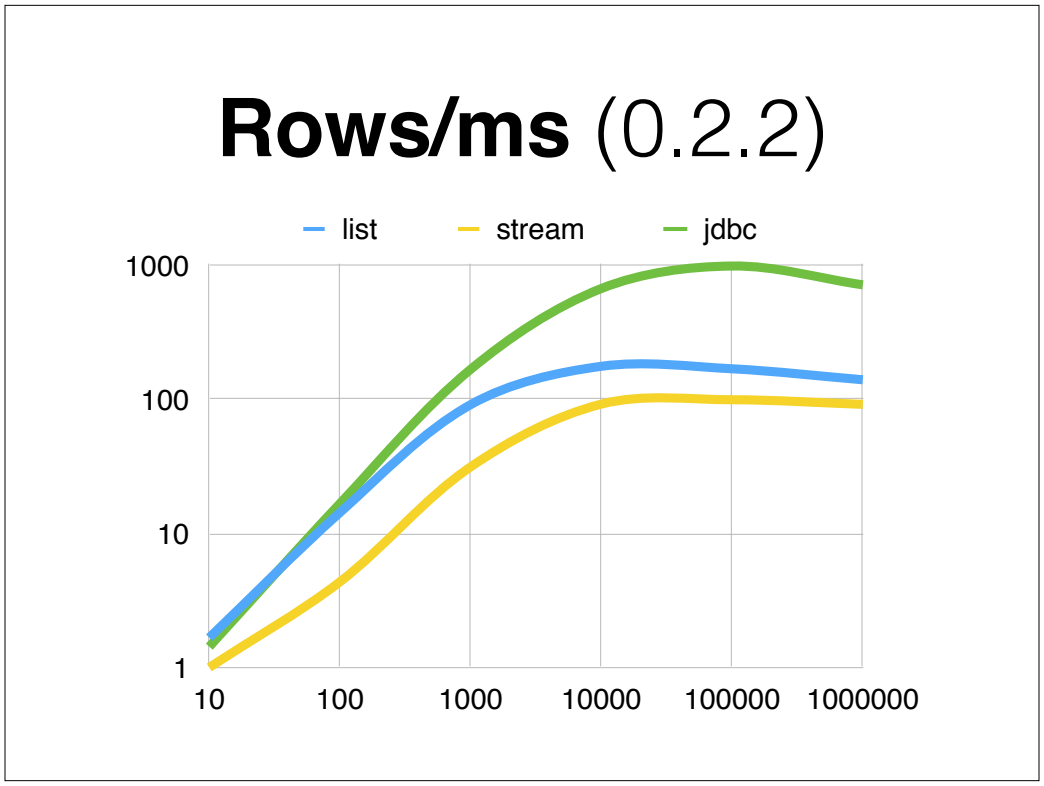
gitter



book of doobie

Perf

There's a cost for using Free because you have this interpretation you're doing … for pure CPU-bound computations it's a constant factor of like 40x. But as soon as you start doing IO it goes away because you're spending all your time waiting on IO. One place where you can see it is when you're reading a big resultset because you're spinning in a tight loop. So I have done some work on that.

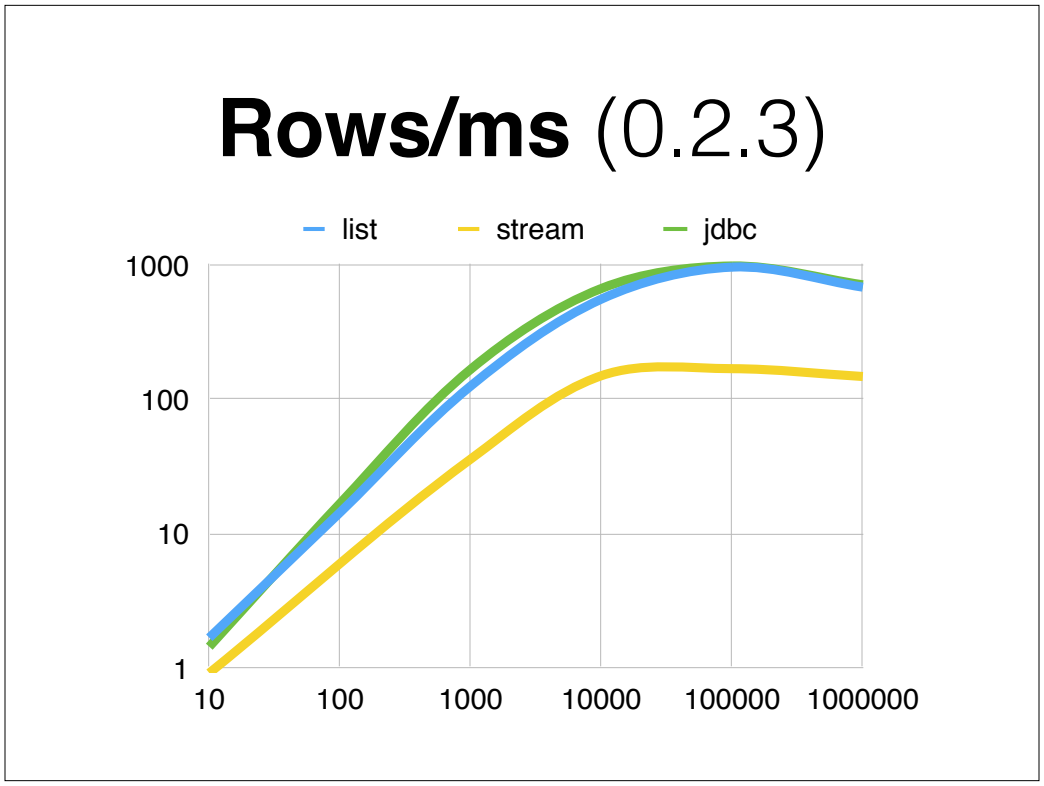


So here's a graph for doobie 0.2.2 which is the current version.

- green line is a java-style imperative jdbc program
- blue line is reading into a list
- yellow line is reading into a list through a stream, which is kind of a pathological case

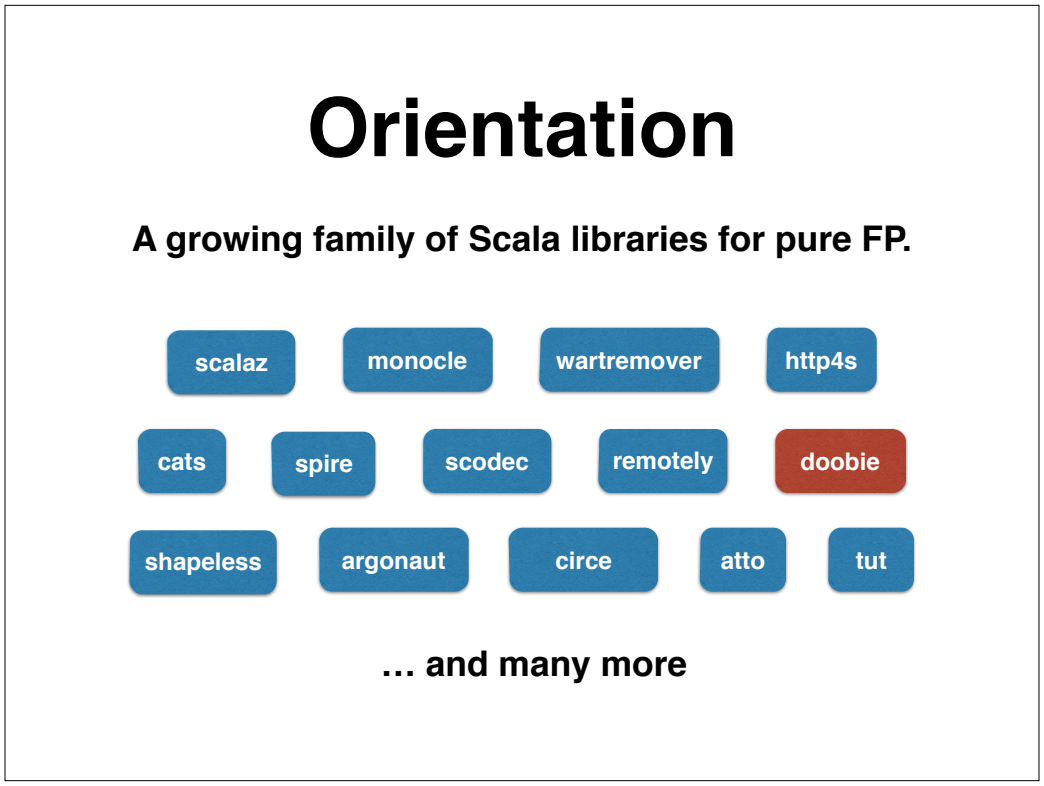
What this is measuring is row reads per ms, it's logarithmic on both scales, so the distance between the lines is *worse* than it looks. If you're reading a million rows jdbc is about 5x faster than the list accumulator and about 10x faster than reading through a stream.

So I did some work and here's what 0.2.3 looks like



The list accumulator is basically in step with raw jdbc. It's really the same line, they just kind of cross back and forth.

The stream accumulator is still a lot slower but I'm still working on it.



… these are libraries that provide pure functional apis for all kinds of things, and really push the type system hard, to help maximize the chances that your program will actually work.