The last frontier and beyond

Think Outside The Box™

of what's

What is FP good for?

FP is good for modeling data

FP is good for modeling effects

FP apps are like nice little boxes

But still...

Data “escapes” the nice and tidy realm of our FP programs

In every project

We need to write some specific code for:

- Serializing data

In every project

We need to write some specific code for:

- Serializing data
    - In JSON

In every project

We need to write some specific code for:

- Serializing data
    - In JSON, Avro

In every project

We need to write some specific code for:

- Serializing data
    - In JSON, Avro, Protobuf

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro , Protobuf, Thrift

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro, Protobuf, Thrift, JSON, JSON, JSON

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro , Protobuf, Thrift, JSON, JSON, JSON
- Validating user input

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro, Protobuf, Thrift, JSON, JSON, JSON
- Validating user input
- Reading configurations

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro , Protobuf, Thrift, JSON, JSON, JSON
- Validating user input
- Reading configurations
- Accessing data stored in data bases

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro, Protobuf, Thrift, JSON, JSON, JSON
- Validating user input
- Reading configurations
- Accessing data stored in data bases
- Generating random data

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro, Protobuf, Thrift, JSON, JSON, JSON
- Validating user input
- Reading configurations
- Accessing data stored in data bases
- Generating random data
- Pretty printing

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro , Protobuf, Thrift, JSON, JSON, JSON
- Validating user input
- Reading configurations
- Accessing data stored in data bases
- Generating random data
- Pretty printing
- Comparing values

In every project

We need to write some specific code for:

- Serializing data
  - In JSON, Avro, Protobuf, Thrift, JSON, JSON, JSON
- Validating user input
- Reading configurations
- Accessing data stored in data bases
- Generating random data
- Pretty printing
- Comparing values

Across projects

Writing such code

- Is repetitive
- Mechanical
- Brings almost no business value

Can we do better?

💡 Let’s derive this boilerplate at compile-time 💡

Compile-time derivation

Compile-time derivation

- Many solutions:
  - Scala macros, scalameta
  - shapeless
  - magnolia
  - scalaz-deriving

Compile-time derivation

- Many solutions:
  - Scala macros, scalameta
  - shapeless
  - magnolia
  - scalaz-deriving
- And specialized libs:
  - scodec,
  - circe
  - avro4s,
  - scalacheck-shapeless, etc.

Compile-time derivation

- Many solutions:
  - Scala macros, scalameta
  - shapeless
  - magnolia
  - scalaz-deriving
- And specialized libs:
  - scodec,
  - circe
  - avro4s,
  - scalacheck-shapeless, etc.

Many different apis

Compile-time derivation

- Many solutions:
    - Scala macros, scalameta
    - shapeless
    - magnolia
    - scalaz-deriving
- And specialized libs:
    - scodec,
    - circe
    - avro4s,
    - scalacheck-shapeless, etc.

Not easily customized

Many different apis

Compile-time derivation

- Many solutions:
  - Scala macros, scalameta
  - shapeless
  - magnolia
  - scalaz-deriving
- And specialized libs:
  - scodec,
  - circe
  - avro4s,
  - scalacheck-shapeless, etc.

Slows down compilation

Not easily customized

Many different apis

Compile-time derivation

- Many solutions:
  - Scala macros, scalameta
  - shapeless
  - magnolia
  - scalaz-deriving
- And specialized libs:
  - scodec,
  - circe
  - avro4s,
  - scalacheck-shapeless, etc.

Unfriendly error messages

Slows down compilation

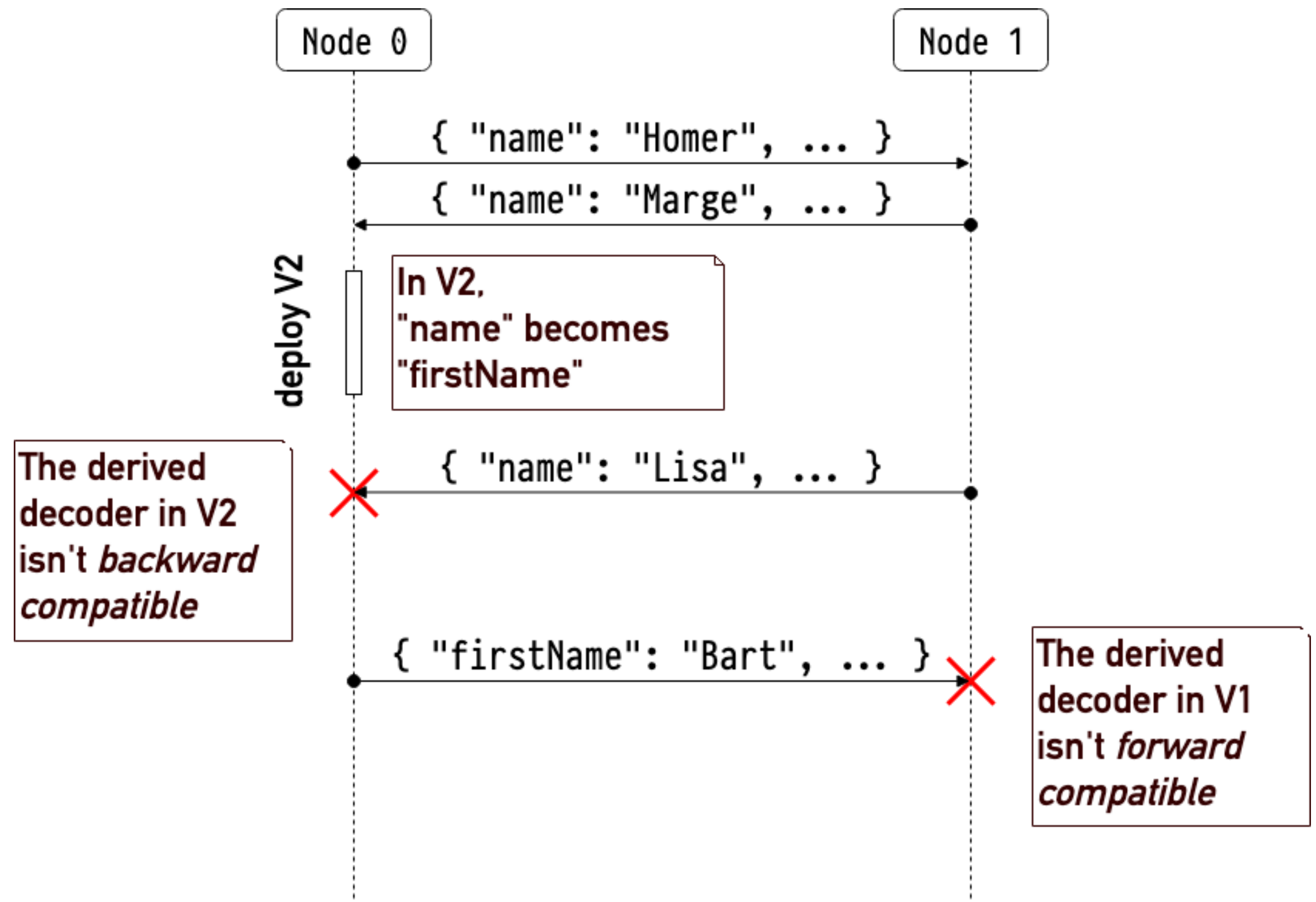
Not easily customized

Many different apis

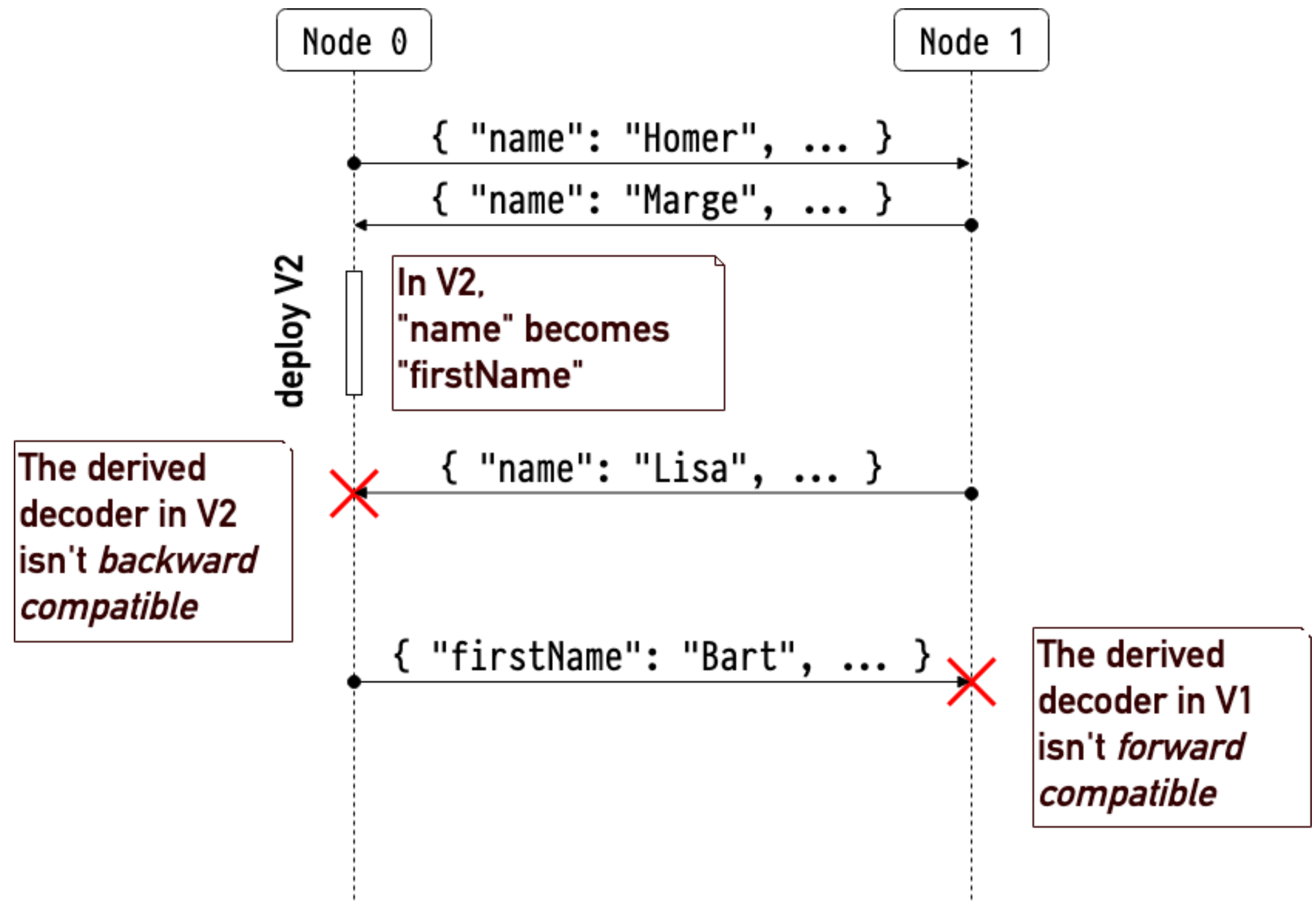
But it gets worse…

Compile-time derivation makes the model difficult to evolve

The evolution problem



The evolution problem



Let´s solve the evolution problem!

To solve the evolution problem

We first need:

- A uniform way to abstract over the structure of data
- A runtime reification of this abstraction
- A method to derive “operations” from this reification

Such
abstraction
exists, it’s called
schema

Let´s define schemas

The “A” in ADT

Every Algebraic Data Type can be represented using only:

- Unit
- Sum (Either)
- Product (Tuple2)

```
type Bit  = Either[Unit, Unit]
type Byte = (Bit, (Bit, (Bit, (Bit, (Bit, (Bit, (Bit, Bit)))))))

type Option[A] = Either[Unit, A]

// intuitively:              Either[Unit, (A, List[A])]
type List[A]   = Fix[λ[α => Either[Unit, (A, α)]]]
```

The “A” in ADT (cont’d)

The same principle applies to our beloved sealed traits and case classes

```scala
sealed trait User
case class Admin(credentials: String)                                    extends User
case class Customer(firstName: String, lastName: String, age: Int) extends User

type Admin_    = String
type Customer_ = (String, (String, Int))
type User_     = Either[Admin_, Customer_]
```

That’s all we need to abstract over any possible data structure

But that wouldn’t be very convenient

In real-world applications

We also need some “convenience” constructors for schemas:

- Primitive types
- Sequences
- Records
- Unions

Isomorphisms

Given a Schema[A] and an Iso[A, B], we can build a Schema[B]

```scala
val bit: Schema[Bit] = unit :+: unit

val bit2Boolean = Iso[Either[Unit, Unit], Boolean]
  { bit  => bit.fold(true, false)}
  { bool => if(bool) Left(()) else Right(())}

val boolean: Schema[Boolean] = iso(bit, bit2Boolean)
```

It's actually slightly more complicated

Yay! Higher-Kinded Recursion Schemes

- Like regular recursion-schemes, but the carrier of algebras is of kind `* -> *`
- Functions are replaced by natural transformation
- Actually not that big of a deal, but makes one feel smart

```
sealed trait SchemaF[S[_], A]

case class Sum[S[_], A, B](left: S[A], right: S[B])  extends SchemaF[S, A \/ B]
case class Prod[S[_], A, B](left: S[A], right: S[B]) extends SchemaF[S, (A, B)]
// etc...
```

OK... now what?

Where’ve we got so far?

Remember, we want:

- A uniform way to abstract over the structure of data ✓
- A runtime reification of this abstraction ✓
- A method to derive “operations” from this reification ?

What is an “operation”?

An operation on `A` is something equivalent to a function that:

- Takes an `A` as argument
- Returns an `A`
- Takes an `A` and returns an `A`

In summary, simply `F[A]`.

What is “deriving”?

Deriving an operation `F` from a schema is coming up with a function:

`Schema[A] => F[A]` for any `A`

Such polymorphic function is called natural transformation and is written:

`Schema ~> F`

So “deriving `F`” means “building a `Schema ~> F`”

And how do we do that?

Intuitively, a schema is a tree.

So we fold that tree into an `F[_]`.

Starting from the leaves (primitive types) we walk back up the tree, combining smaller `F[_]` into bigger ones.

For example, when we reach a `Prod` node we combine the `F[A]` and `F[B]` into an `F[(A, B)]`.

This is typically done by a (higher-kinded) catamorphism of an algebra over a schema

Solving the evolution problem

The evolution problem: recap

- Only one version of each type in the code base
- Backward compatibility (new nodes read old data)
- Forward compatibility (old nodes read new data)

The evolution problem: recap

- Only one version of each type in the code base
- Backward compatibility (new nodes read old data)
- Forward compatibility (old nodes read new data)

It's "just" a matter of coming up with alternative readers.

The evolution strategy

1. Define a set of backward/forward compatible migration steps

2. Define other schemas in terms of the current one

3. Use that to produce an uprading/downgrading schema

4. Derive a reader from it

Step 1: Migration steps

Just an ADT describing b/f compatible migration steps

```scala
sealed trait MigrationStep
case class AddField[A](name: String, schema: Schema[A], default: A) extends MigrationStep
case class RenameField(oldName: String, newName: String)            extends MigrationStep
// etc.
```

Step 2: define migrations

We use this ADT to define older schemas in terms of the current one

```scala
// The current version can be manually defined or derived at compile-time
val personV2: Schema[Person] = ???

val personV1: Schema[Person] =
  Schema
    .upgradingVia(AddField("age", prim(ScalaInt), 0))
    .to(personV2)

val personV0: Schema[Person] =
  Schema
    .upgradingVia(RenameField("name", "username"))
    .to(personV1)
```

Step 3: Upgrading/downgrading schemas

Let's suppose the current version of Person looks like:

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

The personV1 upgrading schema from the previous slide could be manually written as:

```scala
val personV1 = iso(
  personV2,
  Iso[(String, String), Person]
    (pair => Person(0, pair._1, pair._2))
    (pers => (pers.username, pers.email))
)
```

Step 4: Deriving readers

Upgrading/downgrading schemas are… just schemas!

We can derive operations from them like we do with other schemas:

```
val personV1Reads: Reads[Person] = personV1.to[play.api.libs.json.Reads]
```

🎉 Problem solved!!! 🎊

Err... well,
that's actually
not that easy

Problem #1: not enough type safety

- The « recursion-scheme-y » encoding hides the internal structure
- But we need to make sure that a given migration « makes sense »
- Do our introduced `Iso`s align with the rest of the schema?

Solution #1: Introduce a phantom type

- Tag the schema constructors with a type representing their internal structure
- Use that structure to verify stuff at compile time
- A migration becomes a function: `SchemaZ[R1, A] => SchemaZ[R2, A]`

```
sealed trait Tagged[R]

type SchemaZ[Repr, A] = Schema[A] with Tagged[Repr]
```

Problem #2: Scalac doesn't help (how surprising is that, huh?)

- Migrations are in fact dependent functions `SchemaZ[R1, A] => SchemaZ[R2, A]` where `R2` depends on `R1`.
- In the general case, scalar fails to infer `R2`.
- (It even ends up saying stuff like `one was not equal to one`, **charming**)

Solution #2: Just give up…

- … On solving the general case
- Everything works « at the shallowest depth »
  - You can add/remove/rename fields of a record (resp. branches of an union)
  - But you cannot change their inner schema
- So let's just force the user to
  - define their schemas at top-level and
  - compose schemas using functions

Problem #3: This isn't practical, at all

- Leads to too finely grained definitions
- When migrating a schema, you need to redefine all the schemas that depend on it
- You end up redefining everything for each version
- That's precisely what we wanted to avoid in the first place
- `<insert a Grumpy Cat (RIP) picture here>`
- `<make it two>`
- `<or three>`

Solution #3: Type-level Schema Registry

- Define a `Version` as an heterogeneous list (acting as a stack) of functions (that construct schemas)
- Each such constructor can depend on the results of what's defined « below »
- Perform some implicit wizardry to « weave » these functions together
- Voilà!

The end result

```scala
 val current = Current
    .schema(
      record(
        "name" -*>: prim(JsonString) :*: "active" -*>: prim(JsonBool),
        Iso[(String, Boolean), User](User.apply)(u => (u.name, u.active))
      )
    ).schema((u: Schema[User]) => ... )// Some Person schema depending on User

val version0 = current.migrate[User].change(_.addField("name", "John Doe »))

val personV0 = version0.lookup[Person] // will contain a migrated User
```

Schemas give us a lot of things for “free”!

Random data generators

```
val personGen = personSchema.to[Gen]
```

Eq

```
val personEq = personSchema.to[Eq]
```

Ordering

```
val personOrd = personSchema.to[Ordering]
```

Show

```scala
val personShow = personSchema.to[Show]
```

Forms and UIs

```
val personForm = personSchema.to[Form]
```

Avro

```scala
type Avro[A]    = GenericContainer => A
val personAvro = personSchema.to[Avro]
```

SQL queries and migrations

Generic data pipelines

Coming soon… SchemaZ!

These ideas are in active development: https://github.com/spartanz/schemaz

So far we have:

- Schema representation ✓
- Derivation mechanism ✓
- Migration/evolution 🚧

Ask me anything `@ValentinKasas`

Your contribution is very welcome!

Special thanks

John A De Goes

@jdegoes

Dominic Egger

@GrafBlutwurst

I'm @ValentinKasas

Solution architect @ 47 Degrees

We're Hiring!

https://47deg.com