Away with the types!

Dave Gurnell, @davegurnell

underscore

Working with
partial type information

Algebraic data type
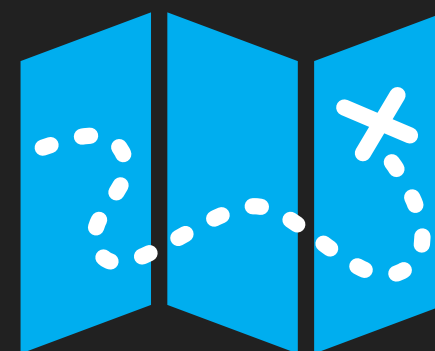
Product types
"ANDs" of other types

Sum types
"ORs" of other types

```scala
sealed trait Survey

final case class WaterQuality(
    temperature: Int,
    ph: Double,
    // ...
) extends Survey

final case class Litter(
    itemCollected: String,
    quantity: Int,
    // ...
) extends Survey
```

Habitat	Vegetation
Amenity	Camera trapping
Water quality	Animal footprint
Litter cleanups	Radiotracking
Invasive flora/fauna	Spotlighting

```scala
sealed trait Survey { ... }
final case class CanalEnvironmentSurvey(...) extends Survey
final case class Ecostatus(...) extends Survey
final case class LutonLeaJuniorRiverWarden(...) extends Survey
final case class LutonLeaRiverlution(...) extends Survey
final case class MorphSurvey(...) extends Survey
final case class Thames21BadgedGroupEvent(...) extends Survey
final case class Thames21GreenWallModule(...) extends Survey
final case class Thames21InvasiveSpecies(...) extends Survey
final case class Thames21Litter(...) extends Survey
final case class Thames21RapidAppraisal(...) extends Survey
final case class Thames21Vegetation(...) extends Survey
final case class Thames21WaterQuality(...) extends Survey
final case class RoyalParksCameraTrapping(...) extends Survey
final case class RoyalParksFootprintTunnel(...) extends Survey
final case class RoyalParksRadiotracking(...) extends Survey
final case class RoyalParksSpotlighting(...) extends Survey
final case class UrbanRiverSurvey(...) extends Survey
final case class WrtWestcountryCsi(...) extends Survey
final case class WrtUpstreamThinking(...) extends Survey
```

```scala
case class UrbanRiverSurvey(
    secondSurveyor: Option[Surveyor],
    surveyDetails: UrsSurveyDetails,
    siteInformation: UrsSiteInformation,
    stretchEngineering: UrsStretchEngineering,
    channelDimensions: UrsChannelDimensions,
    spotChecks: List[UrsSpotCheck],
    sweepUp: UrsSweepUp,
    bankProfileAndProtection: UrsBankProfileAndProtection,
    channelDynamics: UrsChannelDynamics,
    artificialInfluences: UrsArtificialInfluences,
    extentOfPollution: UrsExtentOfPollution,
    habitatFeatures: UrsHabitatFeatures,
    specialFeatures: UrsSpecialFeatures,
    ecologicalCharacteristics: UrsEcologicalCharacteristics
) extends Survey

case class UrsSurveyDetails(
    wfdWaterBodyId: Option[String],
    riverName: Option[String],
    stretchName: Option[String],
    stretchCode: Option[String],
    neasProjectName: Option[String],
    neasProjectCode: Option[String],
    neasSurveyType: Option[UrsNeasSurveyType]
)

case class UrsSiteInformation(
    stretchLength: Option[Int],
    upstreamLocation: Option[Wgs84],
    downstreamLocation: Option[Wgs84],
    distanceFromSource: Option[Int],
    slope: Option[Double],
    solidGeologyCode: Option[String],
    driftGeologyCode: Option[String],
    photographs: UploadSubfolder,
    surveyBank: Option[UrsSurveyBank],
    surveyStart: Option[UrsSurveyStart],
    photoCredit: Option[String],
    photoLicense: Option[String],
    photoReferences: Option[String],
    bedVisible: Boolean,
    adverseConditions: Boolean,
    adverseConditionsSummary: Option[String],
)

case class UrsStretchEngineering(
    planform: Option[UrsPlanform],
    crossProfile: Option[UrsCrossProfile],
    reinforcementLevel: Option[UrsReinforcementLevel],
)

case class UrsChannelDimensions(
    onceOnlyDistanceFromUpstream: Option[Int],
    onceOnlyLocation: Option[Wgs84],
    onceOnlyAtRiffleOrRun: Boolean,
    onceOnlySpotCheck: Option[Int],
    channelBankfullWidth: Option[Double],
    channelWaterWidth: Option[Double],
    channelWaterDepth: Option[Double],
    leftBankTopHeight: Option[Double],
    leftEmbankedHeight: Option[Double],
    rightBankTopHeight: Option[Double],
    rightEmbankedHeight: Option[Double],
)

case class UrsSpotCheck(
    number: Int,
    location: Option[Wgs84],
    leftBankMaterial: Option[UrsBankMaterial],
    rightBankMaterial: Option[UrsBankMaterial],
    leftBankProtection: Option[UrsBankProtection],
    rightBankProtection: Option[UrsBankProtection],
    leftBankMarginalFeature: Option[UrsMarginalFeature],
    rightBankMarginalFeature: Option[UrsMarginalFeature],
    channelSubstrate: Option[UrsChannelSubstrate],
    flowType: Option[UrsFlowType],
    channelFeature: Option[UrsChannelFeature],
    leftBankLandUse: Option[UrsLandUse],
    rightBankLandUse: Option[UrsLandUse],
    leftBankTopStructure: Option[UrsBankStructure],
    rightBankTopStructure: Option[UrsBankStructure],
    leftBankFaceStructure: Option[UrsBankStructure],
    rightBankFaceStructure: Option[UrsBankStructure],
    vegetation: PercentageDistribution[UrsVegetation]
)

object UrsSpotCheck {
    def default: List[UrsSpotCheck] =
        (1 to 10).map(num => UrsSpotCheck(number = num)).toList
}

case class UrsSweepUp(
    sweepUpChannelSubstrate: Option[UrsChannelSubstrate],
    sweepUpChannelVegetation: PercentageDistribution[UrsVegetation],
    chokedWithMacrophytes: YesNoUnknown,
    macrophyteNotes: Option[String],
)

case class UrsBankProfileAndProtection(
    bankProfileAndProtectionAmalgamated: Boolean,
    leftBankNaturalProfile: PercentageDistribution[UrsNaturalBankProfile],
    rightBankNaturalProfile: PercentageDistribution[UrsNaturalBankProfile],
    leftBankArtificialReinforcement: PercentageDistribution[UrsArtificialBankReinforcement],
    rightBankArtificialReinforcement: PercentageDistribution[UrsArtificialBankReinforcement],
    leftBankArtificialProfile: PercentageDistribution[UrsArtificialBankProfile],
    rightBankArtificialProfile: PercentageDistribution[UrsArtificialBankProfile],
    leftBankProtection: PercentageDistribution[UrsBankProtection],
    rightBankProtection: PercentageDistribution[UrsBankProtection]
)

case class UrsChannelDynamics(
    channelDynamics: Map[UrsChannelDynamicsCategory, UrsChannelDynamicsExtent],
    channelDynamicsFeatures: Map[UrsChannelDynamicsFeature, UrsApe]
)

case class UrsArtificialInfluences(
    artificialFeatures: Counts[UrsArtificialFeature],
    artificialFeaturesNotes: Option[String],
    bridgeTypes: Counts[UrsBridgeType],
    nuisanceSpecies: Map[UrsNuisanceSpecies, UrsSpeciesFrequency],
    otherNuisanceSpecies: Option[String],
    recentManagement: Map[UrsManagementFeature, UrsApe],
    recentManagementNotes: Option[String],
    otherNotes: Option[String],
)

case class UrsExtentOfPollution(
    pollutionIndicators: Map[UrsPollutionIndicator, UrsApe],
    pollutionSources: Counts[UrsPollutionSource],
    waterClarity: Option[UrsWaterClarity],
    waterClarityNotes: Option[String],
)

case class UrsHabitatFeatures(
    countedHabitatFeatures: Counts[UrsCountedHabitatFeature],
    percentageHabitatFeatures: PercentageDistribution[UrsPercentageHabitatFeature]
)

case class UrsSpecialFeatures(
    specialFeatures: Map[UrsSpecialFeature, UrsApe],
    treeFeatures: Map[UrsTreeFeature, UrsApe],
    leftBankTreeDistribution: Option[UrsTreeDistribution],
    rightBankTreeDistribution: Option[UrsTreeDistribution],
    photographs: UploadSubfolder
)

case class UrsEcologicalCharacteristics(
    ecologicalCharacteristics: Map[UrsEcologicalCharacteristic, Boolean],
    ecologicalCharacteristicsNotes: Option[String],
    observedProtectedSpecies: Map[UrsProtectedSpecies, Boolean],
    physicalSignsOfProtectedSpecies: Map[UrsProtectedSpecies, Boolean],
    suitableHabitatForProtectedSpecies: Map[UrsProtectedSpecies, Boolean]
)
```
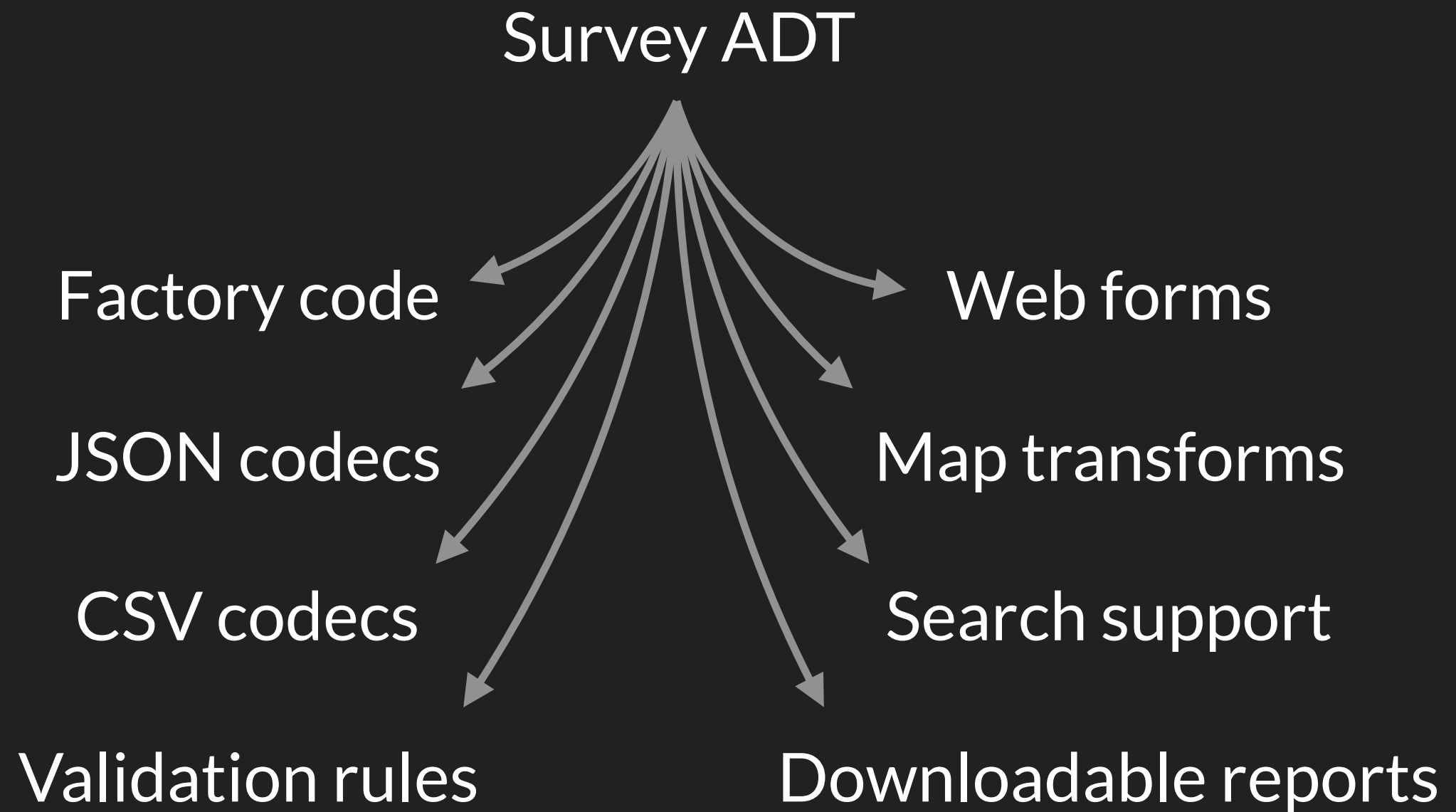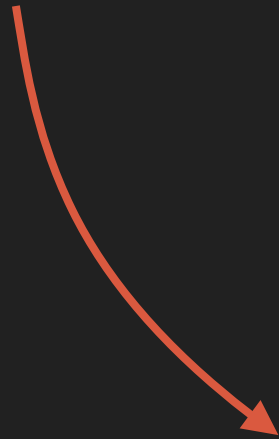
Survey ADT

User-defined surveys
(like *Survey Monkey*)

*Users exist
at run time!*

User-defined surveys
(like *Survey Monkey*)

How do we...

Edit types at runtime?

Store types in a database?

Send types over the wire?

Still manage to write working Scala?

We have to change representations

Agenda

Concrete
vs
Generic

Types
vs
Schemas

Error handling monads

Type classes

Interpreters

Representations

```scala
case class WaterQuality(
    location    : Location,
    timestamp   : ZonedDateTime,
    river       : String,
    temperature : Double,
    ph          : Option[Double],
    turbidity   : Turbidity)
```

```scala
case class Location(
    lat: Double,
    lng: Double)

sealed trait Turbidity
case class QualitativeTurbidity(description: String)
    extends Turbidity
case class QuantitativeTurbidity(value: Int)
    extends Turbidity

case class WaterQuality(
    location     : Location,
    timestamp    : ZonedDateTime,
    river        : String,
    temperature  : Double,
    ph           : Option[Double],
    turbidity    : Turbidity)
```

Concrete ⟹ Generic

"WaterQuality" ⇒ "Data"

```scala
sealed trait Data

case class BooleanData(value: Boolean) extends Data
case class IntData(value: Int) extends Data
case class DoubleData(value: Double) extends Data
case class StringData(value: String) extends Data
case class TimestampData(value: DateTime) extends Data
case object NullData extends Data

case class ListData(values: List[Data]) extends Data

case class ProductData(values: Map[String, Data])
  extends Data
case class SumData(tpe: String, value: Data)
  extends Data
```

```scala
sealed trait Data

case class BooleanData(value: Boolean) extends Data
case class IntData(value: Int) extends Data
case class DoubleData(value: Double) extends Data
case class StringData(value: String) extends Data
case class TimestampData(value: DateTime) extends Data
case object NullData extends Data

case class ListData(values: List[Data]) extends Data


case class ProductData(values: Map[String, Data])
  extends Data
case class SumData(tpe: String, value: Data)
  extends Data
```

Value-level encoding of ADTs

```scala
val location: Location =
  Location(
    lat = 52.0,
    lng = 0.0)

val turbidity: Turbidity =
  QualitativeTurbidity(
    description = "Cloudy")

val survey: WaterQuality =
  WaterQuality(
    location    = location,
    timestamp   = ZonedDateTime.now,
    river       = Some("Thames"),
    temperature = Some(10.0),
    ph          = None,
    turbidity   = turbidity)
```

```scala
val location = ProductData(ListMap(
  "lat" -> DoubleData(52.0),
  "lng" -> DoubleData(0.0)))

val turbidity = SumData(
  "QualitativeTurbidity",
  ProductData(ListMap(
    "description" -> StringData("Cloudy")
  )))

val survey = ProductData(ListMap(
  "location"    -> location,
  "timestamp"   -> TimestampData(ZonedDateTime.now),
  "river"       -> StringData("Thames"),
  "temperature" -> DoubleData(10.0),
  "ph"          -> NullData,
  "turbidity"   -> turbidity))
```

We still have types

But we've lost
a lot of type information

```
val survey: WaterQuality =
  // ...

val temp: Double =
  survey.temperature
```

```scala
val survey: Data =
  // ...

val temperature: Option[Data] =
  survey match {
    case ProductData(fields) =>
      fields.get("temperature")

    case _ =>
      None
  }
```

```scala
val survey: Data =
  // ...

val temperature: Option[Double] =
  survey match {
    case ProductData(fields) =>
      fields.get("temperature") match {
        case DoubleData(temp) =>
          Some(temp)

        case _ =>
          None
      }

    case _ =>
      None
  }
```

```
val survey: WaterQuality =
  // ...

val latitude: Double =
  survey.location.lat
```

WHAT HAS BEEN SEEN...

Cannot be un-seen.

Lots of operations can fail

How do we cope with failure?

We'll use an error handling monad!

We'll use Either!

Accessing fields

```scala
sealed trait Data {

  def get(field: String): Either[String, Data] =
    ???

}
```

```scala
sealed trait Data {

  def get(field: String): Either[String, Data] =
    this match {
      case ProductData(fields) =>
        fields
          .get(field)
          .toRight(s"field not found: $field")

      case SumData(_, data) =>
        data.get(field)

      case _ =>
        Left(s"field not found: $field")
    }

}
```

```scala
val latitude: Either[String, Double] =
  for {
    locData <- survey.get("location")
    latData <- locData.get("lat")
    lat     <- // interpret as a Double
  } yield lat
```

Reclaiming type information

Scala types ⇔ Data values

Type classes

Scala ⇒ Data

```scala
trait ToData[A] {
  def apply(value: A): Data
}
```

```scala
trait ToData[A] {
  def apply(value: A): Data
}

object ToData {

  implicit val booleanToData: ToData[Boolean] =
    new ToData[Boolean] {
      def apply(value: Boolean): Data =
        BooleanData(value)
    }

  implicit val intToData: ToData[Int] =
    new ToData[Int] {
      def apply(value: Int): Data =
        IntData(value)
    }

}
```

Type classes + Extension methods

```scala
implicit class ToDataOps[A](value: A) {

  def toData(implicit toData: ToData[A]): Data =
    toData(value)

}
```

123.toData

```
new ToDataOps(123).toData
```

```
new ToDataOps(123).toData(intToData)
```

```
123.toData
// IntData(123)
```

```
List(1, 2, 3).toData
// ListData(List(IntData(1), IntData(2), IntData(3)))
```

waterQualitySurvey.toData

```scala
implicit val wqToData: ToData[WaterQuality] =
  new ToData[WaterQuality] {
    def apply(wq: WaterQuality): Data =
      ProductData(ListMap(
        "location"    -> wq.location.toData,
        "timestamp"   -> wq.timestamp.toData,
        "river"       -> wq.river.toData,
        "temperature" -> wq.temperature.toData,
        "ph"          -> wq.ph.toData,
        "turbidity"   -> wq.turbidity.toData
      ))
  }
```

Data ⇒ Scala

```scala
trait FromData[A] {
  def apply(data: Data): Either[String, A]
}
```

```scala
sealed trait Data {

  def as[A](implicit from: FromData[A]): Either[String, A] =
    from(this)

}
```

```scala
val latitude: Either[String, Double] =
  for {
    locData <- survey.get("location")
    latData <- locData.get("lat")
    lat     <- latData.as[Double]
  } yield lat
```

```scala
sealed trait Data {

  def getAs[A](fields: String *)
        (implicit from: FromData[A]): Either[String, A] =
    ???

}
```

```scala
val latitude: Either[String, Double] =
  survey.getAs[Double]("location", "lat")
```

Either $\Rightarrow$ Validated

Chapters 4 and 6

underscore.io/books/scala-with-cats

JSON Codecs

```scala
import io.circe._
import io.circe.syntax._

val survey: WaterQuality =
  // ...

val surveyJson: Json =
  survey.asJson

val surveyCopy: Decoder.Result[WaterQuality] =
  surveyJson.as[WaterQuality]
```

```scala
import io.circe._
import io.circe.syntax._

val survey: WaterQuality =
  // ...

val surveyJson: Json =
  survey.asJson(Encoder[WaterQuality])

val surveyCopy: Decoder.Result[WaterQuality] =
  surveyJson.as[WaterQuality](Decoder[WaterQuality])
```

```scala
import io.circe._
import io.circe.syntax._

val survey: Data =
  // ...

val surveyJson: Json =
  survey.asJson(Encoder[Data])

val surveyCopy: Decoder.Result[Data] =
  surveyJson.as[Data](Decoder[Data])
```

```scala
val surveyJson: Json =
  survey.asJson
// {
//    "location"    : { "lat": 52.0, "lng": 0.0 },
//    "timestamp"   : "2017-12-10T10:15:00Z",
//    "river"       : "Thames",
//    "temperature" : 10.0,
//    "ph"          : null,
//    "turbidity"   : { "QualitativeTurbidity": {
//                      "description": "Cloudy"
//                    }}
//    }
// }

val surveyCopy: Decoder.Result[Data] =
  surveyJson.as[Data]
```

```scala
val surveyJson: Json =
  survey.asJson
// {
//    "location"    : { "lat": 52.0, "lng": 0.0 },
//    "timestamp"   : "2017-12-10T10:15:00Z",
//    "river"       : "Thames",
//    "temperature" : 10.0,
//    "ph"          : null,
//    "turbidity"   : { "QualitativeTurbidity": {
//                      "description": "Cloudy"
//                    }}
//    }
// }


val surveyCopy: Decoder.Result[Data] =
  surveyJson.as[Data]
```

Discarding information

Gaining information

```scala
val surveyJson: Json =
  survey.asJson
// {
//     "location"    : { "lat": 52.0, "lng": 0.0 },
//     "timestamp"   : "2017-12-10T10:15:00Z",
//     "river"       : "Thames",
//     "temperature" : 10.0,
//     "ph"          : null,
//     "turbidity"   : { "QualitativeTurbidity": {
//                        "description": "Cloudy"
//                      }}
//     }
// }
```

Product or Sum?

```scala
val surveyCopy: Decoder.Result[Data] =
  surveyJson.as[Data]
```

We need the information
from WaterQuality

But we can't define WaterQuality
at compile time!

Types ⇒ Schemas

```scala
sealed trait Schema

case object BooleanSchema extends Schema
case object IntSchema extends Schema
case object DoubleSchema extends Schema
case object StringSchema extends Schema
case object TimestampSchema extends Schema

case class ListSchema(child: Schema) extends Schema
case class OptionSchema(child: Schema) extends Schema

case class ProductSchema(children: ListMap[String, Schema])
  extends Schema
case class SumSchema(children: ListMap[String, Schema])
  extends Schema
```

```scala
sealed trait Schema

case object BooleanSchema extends Schema
case object IntSchema extends Schema
case object DoubleSchema extends Schema
case object StringSchema extends Schema
case object TimestampSchema extends Schema

case class ListSchema(child: Schema) extends Schema
case class OptionSchema(child: Schema) extends Schema

case class ProductSchema(children: ListMap[String, Schema])
  extends Schema
case class SumSchema(children: ListMap[String, Schema])
  extends Schema
```

Schema-level encoding of ADTs

```scala
case class Location(
    lat: Double,
    lng: Double)

sealed trait Turbidity
case class QualitativeTurbidity(description: String)
    extends Turbidity
case class QuantitativeTurbidity(value: Int)
    extends Turbidity

case class WaterQuality(
    location    : Location,
    timestamp   : ZonedDateTime,
    river       : String,
    temperature : Double,
    ph          : Option[Double],
    turbidity   : Turbidity)
```

```scala
val locationSchema = ProductSchema(ListMap(
  "lat" -> DoubleSchema,
  "lng" -> DoubleSchema))

val turbiditySchema = SumSchema(ListMap(
  "QualitativeTurbidity"  -> ProductSchema(ListMap(
    "description" -> StringSchema)),
  "QuantitativeTurbidity" -> ProductSchema(ListMap(
    "value"         -> IntSchema))))

val waterQualitySchema = ProductSchema(ListMap(
  "location"    -> locationSchema,
  "timestamp"   -> TimestampSchema,
  "river"       -> StringSchema,
  "temperature" -> DoubleSchema,
  "ph"          -> OptionSchema(DoubleSchema),
  "turbidity"   -> turbiditySchema))
```

Back to JSON codecs

```scala
val surveyJson: Json =
  survey.asJson
// {
//    "location"    : { "lat": 52.0, "lng": 0.0 },
//    "timestamp"   : "2017-12-10T10:15:00Z",
//    "river"       : "Thames",
//    "temperature" : 10.0,
//    "ph"          : null,
//    "turbidity"   : { "QualitativeTurbidity": {
//                        "description": "Cloudy"
//                      }}
//    }
// }
```

Product or Sum?

```scala
val surveyCopy: Decoder.Result[Data] =
  surveyJson.as[Data]
```

```scala
val surveyJson: Json =
  survey.asJson
// {
//    "location"     : { "lat": 52.0, "lng": 0.0 },
//    "timestamp"    : "2017-12-10T10:15:00Z",
//    "river"        : "Thames",
//    "temperature"  : 10.0,
//    "ph"           : null,
//    "turbidity"    : { "QualitativeTurbidity": {
//                        "description": "Cloudy"
//                      }}
//    }
// }
```

Product or Sum?

```scala
val surveyCopy: Decoder.Result[Data] =
  surveyJson.as(decoder(waterQualitySchema))
```
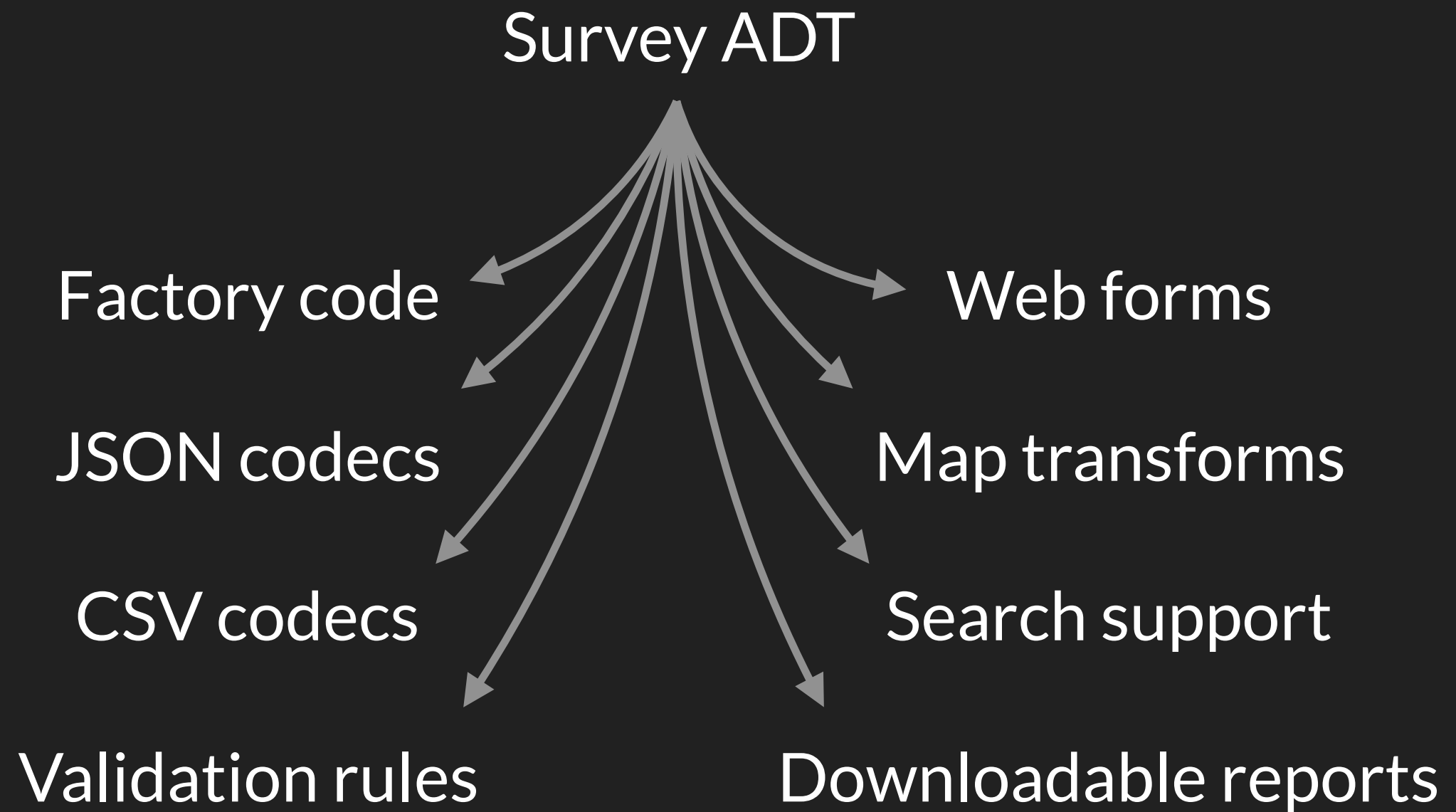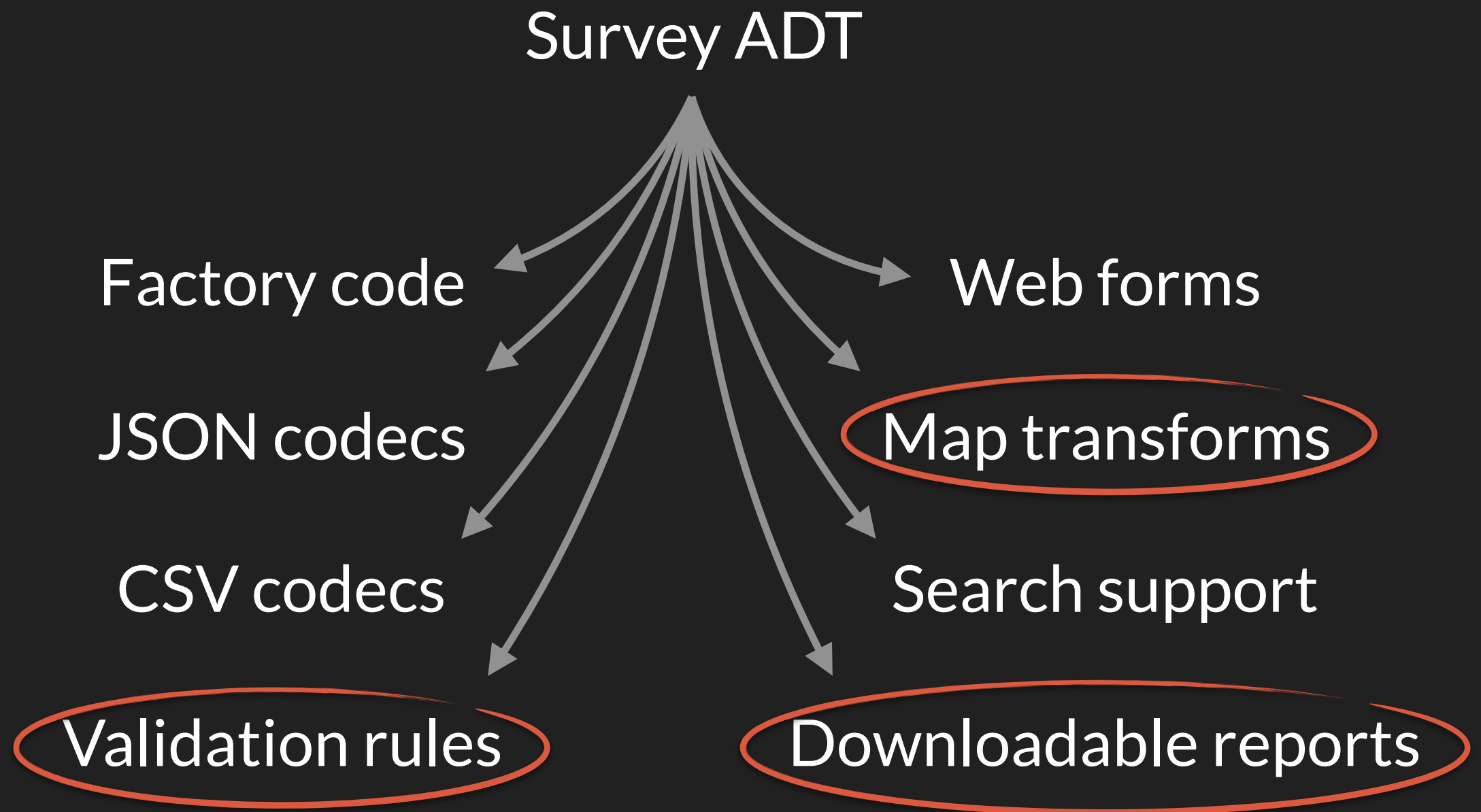
```scala
def decoder(schema: Schema): Decoder[Data] =
  schema match {
    case BooleanSchema      => booleanDecoder
    case IntSchema          => intDecoder
    case DoubleSchema       => doubleDecoder
    case StringSchema       => stringDecoder
    case TimestampSchema    => timestampDecoder
    case ListSchema(s)      => listDecoder(decoder(s))
    case OptionSchema(s)    => decoder(s).or(nullDecoder)
    case s: ProductSchema   => productDecoder(s)
    case s: SumSchema       => sumDecoder(s)
  }
```

One last complication…

We've replaced types with schemas

What about code that
relies on those schemas?

Two options…

Either enrich the schemas…

Or represent code as values

Code ⇒ Values

Access fields in data

Apply functions to fields

Generate results of fixed Scala types
(e.g. validation rules return Boolean)

```scala
sealed trait Expr

final case class Const(data: Data)
  extends Expr

final case class Select(path: List[String])
  extends Expr

final case class Apply(func: String, args: List[Expr])
  extends Expr
```

```scala
def dataAt(path: String *): Expr =
  Select(path.toList)

implicit class ExprOps[A](value: A) {

  def toExpr(implicit toData: ToData[A]): Expr =
    Const(value.toData)

}
```

```scala
sealed trait Expr {
  def unary_- : Expr         = Apply("unary_-", List(this))
  def +(that: Expr): Expr    = Apply("+", List(this, that))
  def -(that: Expr): Expr    = Apply("-", List(this, that))
  def *(that: Expr): Expr    = Apply("*", List(this, that))
  def /(that: Expr): Expr    = Apply("/", List(this, that))
  def >(that: Expr): Expr    = Apply(">", List(this, that))
  def <(that: Expr): Expr    = Apply("<", List(this, that))
  def >=(that: Expr): Expr   = Apply(">=", List(this, that))
  def <=(that: Expr): Expr   = Apply("<=", List(this, that))
  def ===(that: Expr): Expr  = Apply("===", List(this, that))
  def =!=(that: Expr): Expr  = Apply("=!=", List(this, that))
  def unary_! : Expr         = Apply("unary_!", List(this))
  def &&(that: Expr): Expr   = Apply("&&", List(this, that))
  def ||(that: Expr): Expr   = Apply("||", List(this, that))
  def ++(that: Expr): Expr   = Apply("++", List(this, that))
  def combineAll: Expr       = Apply("combineAll", List(this))

  def getOrElse(that: Expr): Expr =
    Apply("getOrElse", List(this, that))
}
```

```scala
val expr: Expr =
  dataAt("temperature") >= 0.toExpr &&
  dataAt("temperature") <= 100.toExpr
```

```scala
val expr: Expr =
  dataAt("temperature") >= 0.toExpr &&
  dataAt("temperature") <= 100.toExpr
// Apply("&&", List(
//   Apply(">=", List(
//     Select(List("temperature")),
//     Const(IntData(0))
//   )),
//   Apply("<=", List(
//     Select(List("temperature")),
//     Const(IntData(100))
//   ))
// ))
```

```scala
sealed trait Expr {

  def eval(data: Data): Either[String, Data] =
    ???

  def evalAs[A](data: Data)
      (implicit f: FromData[A]): Either[String, A] =
    eval(data).flatMap(f.apply)

}
```

```scala
val expr: Expr =
  dataAt("temperature") >= 0.toExpr &&
  dataAt("temperature") <= 100.toExpr

expr.evalAs[Boolean](data)
// Right(true)
// Right(false)
// Left("field not found: temperature")
// etc...
```

We can also represent functions and higher order functions

```
val expr = fn { data =>
  data.ph.fold(true)(ph => ph >= 0 && ph <= 14)
}

expr.evalAs[Boolean](data)
```

We can serialize simple expressions

We can run them against data

We can type check them against schemas

Summary

Allowed users to edit types

Changed representations
static $\Rightarrow$ dynamic
concrete $\Rightarrow$ generic

Lost type information

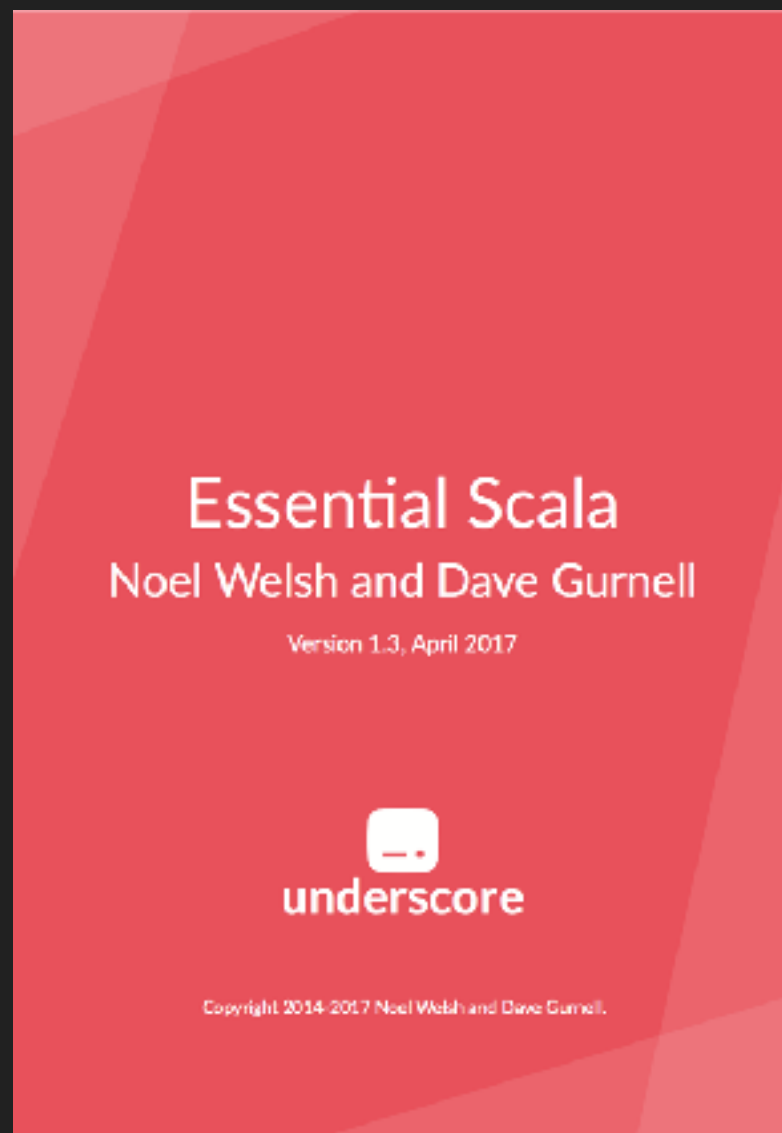# Regained type safety with the Either monad

Regained lost information with schemas

Created serializable
type information, data, and code

But it was a lot of work

So don't do this

Further reading

underscore.io/books/
essential-scala

underscore.io/books/
scala-with-cats

underscore.io/books/
shapeless-guide

Algebraic Data Types
Essential Scala, Chapter 2

flatMap and map
Essential Scala, Chapter 4

Type classes and extension methods
Essential Scala, Chapter 7
Scala with Cats, Chapters 1 and 2

Either and Validated
Scala with Cats, Chapters 4 and 6

Deriving ToData and FromData automatically
Shapeless Guide, Chapters 1, 2, 3, and 5

15:15 today
Andrew Gustafson
Moving Away from Hope-Driven Development

16:15 today
Ben Parker
Almost Type-Safe Error Handling

15:00 tomorrow
Dan Porter
Holophrase: Baby's First DSL

Kris Nuttycombe

Describing Data
with free applicative functors
(and more)

https://www.youtube.com/watch?v=oRLkb6mqvVM

Ionuț G. Stan

A Type Inferencer for ML
in 200 Lines of Scala

https://www.youtube.com/watch?v=H7x4THVU4BQ

Come chat at
the Underscore booth

Or maybe a workshop at
#Scalax2gether
(Saturday 9am - 4pm)

Thank you

https://github.com/davegurnell/away-with-the-types

Dave Gurnell, @davegurnell

underscore