Introducing Cats in EASY

In dit document wordt de huidige status van error handling in EASY geanalyseerd en wordt een eenvoudiger en minder foutgevoelige aanpak beschreven middels de Cats library. In de afgelopen maanden heb ik het e-book Scala with Cats bestudeerd en naar aanleiding daarvan een proof-of-concept gemaakt waarbij ik alle gebruik van `Try` in `easy-split-multi-deposit` heb vervangen door de datastructuren die Cats hiervoor biedt. Het resultaat hiervan is zeer geslaagd: alle tests werken nog precies hetzelfde, en zowel de output als de fout-rapportages zijn nog precies hetzelfde gebleven. Een logische volgende stap lijkt me om Cats, met name de onderdelen die binnen EASY van dienst kunnen zijn, nader toe te lichten in dit document en een voorstel te doen om Cats te introduceren aan de EASY developers.

Huidige situatie error handling in EASY

Tot nu toe wordt error handling in de EASY code base behandeld middels `scala.util.Try`. Een `Try[T]` kan òf een `Success[T]` of een `Failure` zijn. Semantisch gezien drukt een `Try` de mogelijkheid uit dat het een 'echte waarde' òf een `Throwable` bevat.

In de meeste omstandigheden, zoals we die in de praktijk binnen EASY tegenkomen, blijkt deze aanpak vrij effectief. Echter, in een aantal omstandigheden blijkt dat `Try` problemen/moeilijkheden oplevert.

- Aangezien niet getypeerd wordt om wat voor soort `Throwable` het gaat, is het soms lastig te bepalen welke errors te verwachten zijn. Voor een kleine applicatie is dit redelijk goed te bepalen, maar wanneer de applicatie groeit, wordt dit al wat lastiger. Ook wanneer op een later moment extra errors worden toegevoegd, is het lastig te bepalen op welke plaatsen deze error moet worden afgevangen. Veelal vraagt dit om een gedetaileerde code analyse. Ook leidt dit regelmatig tot bugs in de productie code.
- Wanneer we een `Seq[Try[T]]` maken en deze om willen zetten naar `Try[Seq[T]]`, zijn er verschillende mogelijkheden om dit te doen.
    - ***failslow:*** hierbij wordt altijd de collectie volledig doorlopen en worden de `Failure` elementen verzameld en samengevoegd tot één nieuwe `Failure`. Hiervoor is in `dans-scala-lib` een operator `.collectResults` toegevoegd. Errors worden hierbij samengevoegd in een `CompositeException`, die een collectie van `Throwable`s bevat. Hierdoor wordt de applicatie logica complexer dan omschreven in het voorgaande punt, aangezien nu rekening moet worden gehouden met de mogelijkheid dat er meerdere errors kunnen zijn. Vaak is op het moment van 'error handling' ook niet meer duidelijk waar een mogelijke `CompositeException` vandaan komt.

```scala
import nl.knaw.dans.lib.error._

def convert(s: String): Try[Int] = Try { s.toInt }
val input: Seq[String] = Seq("123", "45a6", "78b9")
val xs: Seq[Try[Int] = input.map(convert)
// Seq(Success(123), Failure(...), Failure(...))

val ys: Try[Seq[Int]] = input.map(convert).collectResults
// Failure(...) bevat errors voor "45a6" en "78b9"
```

    - ***failfast:*** hierbij wordt zodra het eerste `Failure` element in de collectie wordt gevonden, gestopt met verdere processing. Hier hebben we nog geen betere manier voor gevonden dan een `throw` te doen op de `Throwable`, op die manier uit de loop over alle elementen te breken en de error buiten de collection op te vangen middels een `Try`.

```scala
import nl.knaw.dans.lib.error._

def convert(s: String): Try[Int] = Try { s.toInt }
val input: Seq[String] = Seq("123", "45a6", "78b9")
val xs: Seq[Try[Int] = input.map(convert)
// Seq(Success(123), Failure(...), Failure(...))

val ys: Try[Seq[Int]] = Try { input.map(s => convert(s).unsafeGetOrThrow) }
// Failure(...) bevat alleen de error voor "45a6"
```

    - combinaties van 'failfast' en 'failslow' komen in enkele omstandigheden ook voor, bijvoorbeeld wanneer alle successresultaten tot aan de eerste error moeten worden teruggegeven, alsmede deze error zelf. Hiervoor is momenteel nog geen gestandaardiseerde operator in de code base gedefinieerd.

- `Try` is per definitie ***failfast*** onder sequentiële compositie middels `x.flatMap(f)`. Dit betekent dat in `Success(1).flatMap(x => ???)` de functie `f` *wel* wordt uitgevoerd, terwijl deze in `Failure(...).flatMap(x => ???)` niet wordt uitgevoerd. Zodra een `Failure(...)` wordt geconstateerd, wordt de rest van de compositie overgeslagen. Hoewel dit in de meeste omstandigheden zeker de bedoeling is, komen we binnen EASY ook verschillende malen tegen dat we eigenlijk een ***failslow*** compositie zouden willen gebruiken. Duidelijke voorbeelden hiervan zijn bijvoorbeeld de CSV parser in `easy-split-multi-deposit` en het hele validatie mechanisme in `easy-validate-dans-bag`. Hiervoor gebruiken we momenteel wel de `Try` middels `collectResults` en `CompositeException`, maar deze komen met hun eigen complicaties, zoals hierboven omschreven.

`scala.util.Either` en Cats

Een alternatief voor `Try` in Scala is om gebruik te maken van `scala.util.Either[A, B]`. Deze datastructuur kan òf een `Right[A]` òf een `Left[B]` zijn. `Try` en `Either` zijn equivalent aan elkaar, zoals getoond in de onderstaande tabel. Het belangrijkste verschil is dat een `Left` *wel* getypeerd is, waardoor duidelijk kan worden gemaakt welke type error kan worden teruggegeven.

	`Try[T]`	**`Either[A, B]`**
succesvol	`Success[T]`	`Right[A]`
error	`Failure`	`Left[B]`

We zouden dus `Try` kunnen definiëren als:

```scala
type Try[T] = Either[Throwable, T]
```

Hoewel de Scala SDK dus een `Either` type heeft, mist het een substantiële hoeveelheid nuttige syntax. Deze worden in Cats toegevoegd middels extension methods (gebruik `import cats.syntax.either._` voor deze methods). Voorbeelden hiervan zijn:

- `leftMap(f: A => A1)`, waar voor een `Either[A, B]` de functie `f` wordt toegepast op de value in `Left` en resulteert in een `Either[A1, B]`. (vergelijkbaar met `map(f: B => B1)` waar een `Either[A, B]` wordt omgezet in een `Either[A, B1]`)
- `recover` / `recoverWith`, vergelijkbaar in gedrag met de gelijknamige methoden in `Try`
- `valueOr`, vergelijkbaar met `getOrRecover` die wij zelf in `dans-scala-lib` hebben toegevoegd
- `traverse`, die gegeven een `Either[A, B]` en een functie `B => F[C]` een `F[Either[A, C]]` produceert (vergelijk met `map`, waarbij dezelfde input een `Either[A, F[C]]` zou produceren); deze methode zal cruciaal blijken in een oplossing voor ***failfast*** gedrag zoals hierboven omschreven in de context van `Seq[Try[T]]`.
- `Either.left("error!!!")` en `Either.right(2)`, die een value in een `Either` wrappen, resp. als een `Left` of `Right`.
- Equivalent aan voorgaande zijn: `"error!!!".asLeft` en `2.asRight`
- `Either.catchOnly[NumberFormatException] { "foo".toInt }` die een `Throwable` van uitsluitend het opgegeven type opvangt.
- `Either.catchNonFatal { "foo".toInt }` vangt ieder subtype (mits `NonFatal`) van `Throwable` op. Met deze en voorgaande functies voegt Cats dezelfde functionaliteit aan `Either` toe als die we in `Try` reeds zagen.

Cats biedt tevens een aantal type aliasses m.b.t. `Either` om een collectie van errors (in `Left`) uit te drukken:

- `type EitherNel[E, A] = Either[NonEmptyList[E], A]`
- `type EitherNec[E, A] = Either[NonEmptyChain[E], A]`
- `type EitherNes[E, A] = Either[NonEmptySet[E], A]`

Deze zijn in Cats te vinden onder `import cats.data.{ EitherNel, EitherNec, EitherNes }`.

Deze type aliasses worden daarnaast ook ondersteund middels operators op het type `Either[E, A]`, respectievelijk `.toEitherNel`, `.toEitherNec` en `.toEitherNes`, die de value in een `Left` wrappen in het betreffende collectie type. Ook zijn er operators om een value in een `Left` of `Right` te wrappen i.c.m. een dergelijke collectie: `.rightNel`, `.leftNel`, etc.

Validatie met Cats

Net als `Try`, is `Either` ***failfast*** onder compositie middels `flatMap`. Daardoor is deze niet geschikt voor ***failslow*** doeleinden zoals eerder omschreven.

Cats biedt een alternatieve datastructuur: `cats.data.Validated[E, A]` met subtypes `Valid[A]` en `Invalid[E]` die semantisch bedoelt is voor ***failslow*** gedrag. Het biedt in Cats vergelijkbare functionaliteit als voor `Either` (zie hierboven), maar geeft daarnaast syntax voor compositie waarbij errors verzameld worden (overeenkomstig met `collectResults` en `CompositeException`). Hiervoor zijn een aantal type alisasses gedefinieërd, vergelijkbaar in functionaliteit en ondersteuning met de hierboven genoemde varianten voor `Either`:

- `type ValidatedNel[E, A] = Validated[NonEmptyList[E], A]`
- `type ValidatedNec[E, A] = Validated[NonEmptyChain[E], A]`

Deze type aliasses zijn te vinden onder `import cats.data.{ ValidatedNel, ValidatedNec }`.

Onderstaand een voorbeeld waarbij een parser wordt gedefinieërd voor een `Person` object vanuit een `Map[String, String]`. Om de eventuele errors uit de `parseName` en `parseAge` te combineren, worden deze in een tuple gezet en wordt het `Person` object uiteindelijk gemaakt in een `mapN` functie.

Scala

```scala
import cats.data.{ Validated, ValidatedNel }
import cats.syntax.apply._
import cats.syntax.validated._

type Data = Map[String, String]
type Parsed[T] = ValidatedNel[String, T]

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

def parseName(data: Data): Parsed[String] = {
  data.get("NAME")
    .map(name => name.validNel)
    .getOrElse { "no value found for 'NAME'".invalidNel }
}

def parseAge(data: Data): Parsed[Int] = {
  data.get("AGE")
    .map(ageString => parseInt(ageString))
    .getOrElse { "no value found for 'AGE'".invalidNel }
}

def parseInt(num: String): Parsed[Int] = {
  Validated.catchOnly[NumberFormatException] { num.toInt }
    .leftMap(_ => s"value '$num' is not a number")
    .toValidatedNel
}

def parsePerson(data: Data): Parsed[Person] = {
  (
    parseName(data),
    parseAge(data),
  ).mapN((name, age) => Person(name, age))
}
```

Hieronder een aantal voorbeelden van input met daarbij de verwachtte output voor `parsePerson`:

input	output
`Map()`	`Invalid(List("no value found for 'NAME'", "no value found for 'AGE'"))`
`Map("NAME" -> "me", "AGE" -> "abc")`	`Invalid(List("value 'abc' is not a number"))`
`Map("NAME" -> "me", "AGE" -> "42")`	`Valid(Person("me", 42))`

Zoals hierin te zien is, worden de errors uit de verschillende onderdelen van het tuple verzameld middels `mapN` en in een `Invalid(List(...))` teruggegeven. Alleen wanneer alle onderdelen van de tuple succesvol (`Valid`) zijn, wordt werkelijk een `Person` object gemaakt.

Naast `mapN` biedt Cats nog andere manieren om `Validated` objecten te combineren, met name `tupled` , `contramapN` , `imapN` , `traverseN` en `apWith` . Behalve `tupled` worden deze in de praktijk echter zelden gebruikt.

Failfast en failslow voor collecties met Cats

Met `Validated` biedt Cats een manier om errors te verzamelen in plaats van zo snel mogelijk een berekening te laten falen. Dit gegeven kan ook worden gebruikt om errors in een collectie te verzamelen. Hiervoor wordt met name de `traverse` operator gebruikt in Cats. Voortbouwend op het voorbeeld met `parsePerson` , kunnen we hiermee een `parsePersons` (meervoud) implementeren:

Scala
```scala
import cats.instances.list._
import cats.syntax.traverse._

def parsePersons(datas: List[Data]): Parsed[List[Person]] = {
  datas.traverse(data => parsePerson(data))
}
```

Merk hierbij op dat `traverse` erg lijkt op `map` . Echter, waar `datas.map(parsePerson)` een `List[Parsed[Person]]` op zou leveren, resulteert een `datas.traverse(parsePerson)` in een `Parsed[List[Person]]` . De `traverse` operator draait dus de `Parsed` en `List` om. Hiermee biedt het exact het gedrag dat we binnen EASY ook bij `Try.collectResults` hebben geïmplementeerd, maar dan zonder de bijkomstigheid van een `CompositeException` en de complicaties die daaruit voortvloeien.

Verrassend genoeg blijkt dat failfast gedrag in Cats op dezelfde manier kan worden geïmplementeerd: ook hier wordt de `traverse` operator voor gebruikt. Echter, voor failfast wordt geen gebruik gemaakt van `Validated` , maar van `Either` . In het volgende voorbeeld wordt `traverse` gebruikt om voor ieder element van de `List` een functie `execute` aan te roepen, totdat een error (`Left`) wordt teruggegeven. Zodra dit het geval is, wordt `execute` niet meer aangeroepen met de overige elementen uit de `List` , maar wordt direct de error teruggegeven.

Scala
```scala
import cats.instances.either._
import cats.instances.list._
import cats.syntax.either._
import cats.syntax.traverse._

type FailFast[T] = Either[String, T]

def execute(s: String): FailFast[Int] = {
  // do something useful
  if (s.length > 2)
    s"invalid input '$s'".asLeft
  else
    s.length.asRight
}

def executeAll(input: List[String]): FailFast[List[Int]] = {
  input.traverse(s => execute(s))
}
```

input	output
`List("", "a", "ab")`	`Right(List(0, 1, 2))`
`List("a", "abcd", "ab")`	`Left("invalid input: 'abcd'")`

Merk op dat in dit laatste voorbeeld alleen `execute("a")` en `execute("abcd")` wordt aangeroepen. Omdat deze laatste een error (`Left`) teruggeeft, wordt `execute("ab")` *niet* aangeroepen.

Merk tevens op dat dit een nette vervanging is voor de hiervoor omschreven `Try.unsafeGetOrThrow` .

Overige opmerkingen

1. Er is een library (Cats-ScalaTest) beschikbaar die het eenvoudig maakt om met datastructuren zoals `Either` en `Validated` te testen. Voor het eerder beschreven `parsePerson` voorbeeld zouden tests geschreven kunnen worden als:

   ```Scala
   // voor geldige input
   parsePerson(input).value shouldBe Person(..., ...)

   // voor ongeldige input
   parsePerson(input).invalidValue should contain inOrderOnly ("...", "...")
   ```

   Hiervoor moet je aan de class definitie een of meerdere onderdelen van de volgende `extends` toevoegen:

   - `EitherMatchers` - biedt functionaliteit als:
     - `execute("abc") shouldBe left`
     - `execute("ab") shouldBe right`
   - `EitherValues` - biedt functionaliteit als:
     - `execute("a").value shouldBe 1`
     - `execute("abc").leftValue shouldBe "invalid input 'abc'"`
   - `ValidatedValues` - biedt functionaliteit als:
     - `parsePerson(input).value shouldBe Person(..., ...)`
     - `parsePerson(input).invalidValue should contain inOrderOnly ("...", "...")`

2. In sommige omstandigheden vraagt de Scala compiler om expliciet bepaalde types op te geven. Dit komt het vaakst voor bij het gebruik van `.traverse` en meestal betreft het een relatief complexe expressie, wat in zeker opzicht als een deal-breaker kan worden gezien voor het gebruik van Cats. Echter, met behulp van de 'kind-projector' compiler plugin kan deze complexe syntax ontweken worden en vervangen worden door een relatief eenvoudige syntax.

   ```Scala
   def parseToInt(s: String): Either[String, Int] = {
     Either.catchNonFatal{ s.toInt }.leftMap(_.getMessage)
   }

   // zonder compiler plugin
   List("1", "2", "abc", "4")
     .traverse[({type L[A] = Either[String, A]})#L, Int](s => parseToInt(s))

   // met compiler plugin
   List("1", "2", "abc", "4")
     .traverse[Either[String, *], Int](s => parseToInt(s))
   ```

3. Voor het toevoegen van Cats, Cats-ScalaTest en kind-projector aan de DANS Maven Parent, zijn inmiddels twee pull-requests ingediend: dans-mvn-plugin-defaults#1 en dans-mvn-lib-defaults#6. Deze zijn getest in combinatie met de proof-of-concept voor `easy-split-multi-deposit`.

Gebruikersgemak en leercurve in Cats

Zoals aangetoond biedt Cats eenvoudige syntax om de problemen op te lossen waar tijdens development in EASY regelmatig tegenaan gelopen wordt. Dit gaat echter niet zonder slag of stoot. Hoewel de syntax van Cats redelijk overeen komt met wat tot nu toe wordt gedaan met `Try`, vereist met name de notie/semantiek van `Either` en `Validated` wat tijd om hiermee bekend te raken.

Daarnaast vereist Cats een aantal imports. Dit heeft ermee te maken dat Cats volgens een structuur van 'type classes' is opgezet, waarbij syntax veelal middels extension methods is gedefinieerd vanuit diverse packages. Dit vergt dus kennis van welke packages geïmporteerd moeten worden. Naar verwachting zal het niet veel tijd kosten om hiermee om te leren gaan, maar zeker in het begin zou het tot wat rare foutmeldingen kunnen leiden wanneer deze imports vergeten worden.

Naast de voornoemde zaken biedt Cats nog veel andere syntax, datastructuren en functionaliteit. Voorlopig biedt dit weinig toegevoegde waarde binnen EASY en deze kan dan ook worden genegeerd.

De grootste hobbel is te voorzien in het gebruik van de kind-projector die in sommige situaties van `.traverse` nodig blijkt te zijn. Ik heb in de proof-of-concept echter gemerkt dat IntelliJ ons hierbij redelijk goed kan helpen, wat de pijn ietwat verzacht.

Al met al voorzie ik een kleine leercurve, zoals dit met iedere library het geval is. Daarna verwacht ik echter dat zal blijken dat het gemak wat Cats biedt sterk opweegt tegen het huidige gebruik van `Try`.

Adoptie van Cats in EASY

Ik wil benadrukken dat het zeker niet in mijn intentie ligt om voor te stellen om alle Scala services en applicaties in EASY om te schrijven en alle `Try` constructies te verwijderen. Dit zou niet alleen een enorme hoeveelheid extra werk opleveren, maar zou ook weinig tot geen functionaliteit toevoegen aan de reeds bestaande code.

Er zijn echter wel services en applicaties die bijzonder zouden kunnen profiteren van de voordelen van Cats. Ik heb reeds gezien in mijn proof-of-concept voor `easy-split-multi-deposit` dat het de code een stuk simpeler en begrijpelijker maakt en dat het duidelijker is welk type error kan worden verwacht bij het aanroepen van een methode. Een andere service die zeker voordeel zal halen uit de Cats library is `easy-validate-dans-bag`, waar het verzamelen van zoveel mogelijk errors over een bag centraal staat. Dit is een zeer typische use case voor `Validated`.

Cats biedt tevens operatoren om te schakelen tussen `Try`, `Either` en `Validated`, wat een combinatie van deze datastructuren mogelijk maakt. Hierdoor wordt het mogelijk om in een applicatie te besluiten nieuwe delen te implementeren in termen van `Either` of `Validated` en deze op een later punt om te vormen naar `Try` om dit deel in de rest van de applicatie te hangen.

Adoptie van Cats onder developers

Er is veel introductie materiaal voor Cats beschikbaar. Cats heeft zelf zeer uitgebreide documentatie, gecategoriseerd per datastructuur en voorzien van duidelijke voorbeelden. Daarnaast is er ook een e-book beschikbaar, waarin Cats vanuit ‘first principles’ wordt geïntroduceerd. Dit boek bevat tevens een aantal exercises en case studies (inclusief uitwerkingen en nadere toelichtingen achterin het boek).

Een mogelijkheid zou zijn om dit boek (met uitzondering van niet-relevante hoofdstukken en secties) door te werken tijdens een serie workshops. Hierdoor krijgen we meer begrip voor wat Cats ‘under the hood’ doet en wat de achterliggende concepten zijn, terwijl we ook de gewenste datastructuren (`Either`, `Validated`, etc.) langs zien komen. Hierdoor snappen we beter hoe Cats werkt en kunnen we eventuele compile errors eenvoudiger herkennen en oplossen.

Een andere aanpak is om voor nu tijdens één of twee workshops puur te focussen op de datastructuren die we in de praktijk willen gaan gebruiken. Hoe Cats werkt en waarom met name bepaalde imports noodzakelijk zijn, doen we hierbij af als ‘magic’ of ‘dat is nu eenmaal hoe Cats het wil’. In deze workshops zouden we wel gebruik kunnen maken van enkele exercises uit voornoemd boek, zei het in ietwat aangepaste vorm. Daarnaast zouden we (hetzij in dezelfde workshops, of in een opvolgende workshop) enkele repositories van EASY kunnen nemen en deze refactoren naar gebruikmaking van `Either` en `Validated` i.p.v. `Try`, `.collectResults` en `.unsafeGetOrThrow`. Enkele suggesties voor ‘refactoring kandidaten’ zijn:

- easy-change-accessrights-to-cc0
- easy-delete-dataset
- easy-ingest
- easy-update-fs-rdb
- easy-validate-dans-bag