Trabajo Práctico - Cuarta Parte

[75.27] Algoritmos y Programación IV

del Mazo, Federico	Lafroce, Matías	Mermet, Javier
100029	91378	98153

26 de Febrero del 2021

En este trabajo revisamos los trabajos prácticos de Cecilia Hortas y Martín Coll que se pueden encontrar en `https://github.com/chortas/7527-AlgoritmosIV`

1 Primera parte

En la primera parte del TP vemos varias cosas positivas para analizar del programa. Por empezar nos gustó como se valida el DataSetRow utilizando for comprehension para validar cada uno de sus atributos, y cómo cada uno de estos validadores están implementados (y reutilizados).

```
val converter: Stream[IO, Either[Throwable, Int]] = for {
  blocker <- Stream.resource(Blocker[IO])
  results <- io.file
    .readAll[IO](Paths.get("train.csv"), blocker, 4096)
    .through(text.utf8Decode)
    .through(text.lines)
    .drop(1) // remove header
    .dropLastIf(_.isEmpty)
    .map(
      DataSetRow.toDataSetRowEither(_).map(QueryConstructor.constructInsert)
    )
    .evalMap { // collect errors from both parsing and transacting
      case Right(query) => query.run.transact(transactor).attempt
      case Left(error)  => IO.pure[Either[Throwable, Int]](Left(error))
    }
} yield results
```

También esta muy bien pensada la manera en la que se lidia con los errores utilizando `IO.pure`, para levantar el `Either` con una mónada y así mantener su valor.

Una cosa que se podría mejorar en esta parte del trabajo es el encapsulamiento; se ven distintas capas de abstracción en `Run.scala`, y ahí mismo se tiene el `converter`, que podría haber sido parte de su propia clase. De este modo se separarían responsabilidades.

2 Segunda parte

En esta entrega del TP vemos mucha lógica dentro de `Run`, al igual que en el primer TP. Tanto el manejo de la DB como de PMML podrían haber sido separados en sus propios módulos. Cada uno de estos podría ser un servicio que sea luego llamado en `Run`.

Algo que llama la atención es la implementación de Split. Un punto a destacar como positivo es que se utiliza una State Monad para encapsular `shuffle`. Pero por otro lado, hasta el mejor de nuestro entendimiento, no serviría en un stream considerando el siguiente snippet:

```
1  def split[T](list: List[T]): State[Seed, DataSet[T]] =
2    shuffle(list).map { l =>
3      val (l1, l2) = l.splitAt((list.length * 0.7).round.toInt)
4      DataSet(l1, l2)
5    }
```

Vemos que llama a `splitAt`, el cual requiere conocer el largo total de la lista. Considerando esto, su ejecución no podría ser lazy, y requiere tener cargado todo el dataset en memoria.

Nos queda la duda si no sería mas conveniente evitar el uso de `ScalaReflection` para el armado del `DataFrame`. Al usar reflexión se pierde abstracción y la capacidad de manejar distintos errores que pudieran surgir, con abstracciones del paradigma. Al trabajar con reflexión se trabaja sobre instancias particulares, de modo que se pierde la capacidad de trabajar con traits.

En el siguiente snippet podemos ver a lo que nos referíamos al principio de la sección:

```
1  val pipeline = new Pipeline().setStages(stages)
2  Try(pipeline.fit(dataSetTrain))
3    .fold(_ => println("The model creation failed"), p => {
4      val pipelinePredictionDf = p.transform(dataSetTest)
5      pipelinePredictionDf.show(10)
6      val pmml = new PMMLBuilder(schema, p).build
7      val os: OutputStream = new FileOutputStream("model.pmml");
8      MetroJAXBUtil.marshalPMML(pmml, os);
9    })
```

Tanto el entrenamiento, como inferencia sobre el set de testing, com el armado y dumpeado del PMML están altamente acoplados.

3 Tercera parte

En esta ultima entrega no notamos el problema de la entrega anterior, y la cantidad de código en main es noblemente menor.

Sobre el siguiente snippet, consideramos que hay soluciones mas idiomáticas/paradigmáticas para `if -> return 0`.

```
45 def valueFromFieldName(name: String): Any = {
46   val elementName = productElementNames.toList.find(_.toLowerCase == name.toLowerCase).get
47   if (elementName == "date") {
48     return date.toEpochSecond(ZoneOffset.UTC)
49   }
50   val index = productElementNames.indexOf(elementName)
51   val value = productElement(index)
52   if (value == None || value == null) {
53     return 0
54   }
55   value
56 }
```

Los tests son muy completos: utilizan mocks de distinta índole para realizarlos.

Resulta interesante ver que el `transactor` esta encapsulado en `F`, de este modo esta desligado de su tiempo de ejecución hasta que se le indica el interprete. De este modo es consistente con el uso de `F` en el resto del TP.

4 Notas generales

Notamos que en los 3 TPs, pocos `val` están tipados. Se soporta mucho en la inferencia de tipos.