Step 5

Modes of Composition in functional Scala programming

Ben Hutchison
Lambdajam Online 2020

Step 5: Processing an Order Stream

- In step 4, we completed the logic to process a single Order.
- In step 5 we'll introduce FS2 Streams to model a flow of many orders coming in and being processed concurrently.
- While we use synthetic data in this workshop, there exist libraries that can represent Kafka, Google PubSub and similar message streams as FS2 streams
    - https://github.com/fd4s/fs2-kafka
    - https://github.com/permutive/fs2-google-pubsub

Stream[F, O]

- FS2's Stream is a widely useful abstraction representing repeated effectful computations or work

- A **Stream[F, O]** is an ordered series of effectful values, with effect **F** and payload **O**
  - When F is the type **Nothing**, the Stream is *pure* (ie just data values in memory)
  - When O is **Unit**, the stream contains effectful actions that emit no data

- Stream consumers *pull* (aka poll, fetch) from their input
  - Contrast with Reactive Streams which is push-based
  - Events source like mouse clicks or web requests need to be *queued* to work with FS2
  - Consumer pull model naturally handles backpressure

Zooming in on the problem

```
def processMsgStream

    [F[_]: Concurrent: Parallel: Clock: UuidRef: SkuLookup: CustomerLookup: Inventory: Publish](

    msgs: fs2.Stream[F, Array[Byte]], maxParallel: Int = 20): fs2.Stream[F, Unit]
```

Zooming in on the problem

```
def processMsgStream

    [F[_]: Concurrent: Parallel: Clock: UuidRef: SkuLookup: CustomerLookup: Inventory: Publish](

    msgs: fs2.Stream[F, Array[Byte]], maxParallel: Int = 20): fs2.Stream[F, Unit]
```

All the constraints on **F** required to process an individual message, except **Sync** has been strengthened to **Concurrent**.
The **Concurrent** typeclass allows green threads (Fibers) to be spawned for running subtasks

Zooming in on the problem

```
def processMsgStream

    [F[_]: Concurrent: Parallel: Clock: UuidRef: SkuLookup: CustomerLookup: Inventory: Publish](

    msgs: fs2.Stream[F, Array[Byte]], maxParallel: Int = 20): fs2.Stream[F, Unit]
```

Processing works by transforming an input stream of byte-array messages into tasks that will decode, validate and action each of them

Zooming in on the problem

```
def processMsgStream

    [F[_]: Concurrent: Parallel: Clock: UuidRef: SkuLookup: CustomerLookup: Inventory: Publish](

    msgs: fs2.Stream[F, Array[Byte]], maxParallel: Int = 20): fs2.Stream[F, Unit]
```

The **maxParallel** parameter caps how many concurrent Fibers should be running at any point. Defaults to 20 if unspecified. Should be approx equal to CPU thread count.

Zooming in on the problem

```
def processMsgStream

    [F[_]: Concurrent: Parallel: Clock: UuidRef: SkuLookup: CustomerLookup: Inventory: Publish](

    msgs: fs2.Stream[F, Array[Byte]], maxParallel: Int = 20): fs2.Stream[F, Unit]
```

Returns a stream of actions, each of which has pulled and processed one message

See the unit tests for example of how we can run the overall stream

Zooming in on the problem

```
def processMsgStream

    [F[_]: Concurrent: Parallel: Clock: UuidRef: SkuLookup: CustomerLookup: Inventory: Publish](

    msgs: fs2.Stream[F, Array[Byte]], maxParallel: Int = 20): fs2.Stream[F, Unit]
```

parEvalMapUnordered is the suggested stream operator to process messages most efficiently.
It processes messages in parallel, and doesn't worry about maintaining ordering. Allowing processing of messages out-of-order is a business rule decision.