Step 2

Modes of Composition in functional Scala programming

Ben Hutchison
Lambdajam Online 2020

Step 2: Validating an Order Message

- We will assume that the system shouldn't fully trust incoming orders and will validate their Customer and Skus
  - Could be a B2B (business-to-business) or franchised integration, or simply an team boundary in a large oranisation

- Customers and Skus are managed by different data stores, abstracted by interfaces `CustomerLookup[F]` and `SkuLookup[F]` respectively
  - These would likely be databases or key-value stores in production, but well use in-memory implementations for testing

- Some Skus can only be sold to customers in particular regions. Each customer has a region associated with them. We'll load this info from the lookups as well.

Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

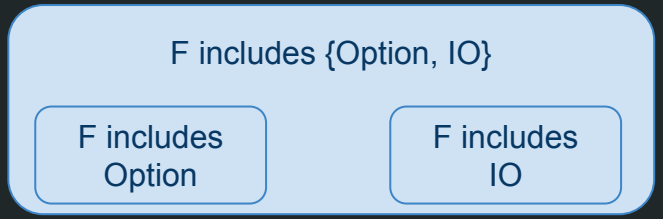
F, the effect type, has four typeclass constraints this time

We'll go through them one by one

Modes of Composition

- *An effectful value can incorporate several different effects*
  - *Tagless Final style and/or Monad Transformers*

- If part of our program includes an Option effect, and another includes an IO effect, how can we compose them together?
  - The effects present in our program are a *Set of Constraints.* We can add to the set when needed.



Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

Sync is a key typeclass from Cats-effect. Its ability is self-described as to “suspend the execution of side-effects” in the F context.

It implies capabilities similar to Monix’s Task, or the “IO monad” from Cats Effect or Haskell

Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

The Parallel typeclass from Cats library allows parts of a task or computation to be run in parallel with each other

Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

SkuLookup is a domain capability trait defined by the application (rather than from a library)

It provides a single capability: looking up a Sku record from a Sku code string

Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

CustomerLookup is another domain capability trait: looking up a Customer record from a Customer ID string

Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

OrderMsg is the record we parsed from JSON in the previous exercise

Zooming in on the problem

```
object OrderProcessor {

 def resolveOrderMsg[F[_]: Sync: Parallel: SkuLookup: CustomerLookup](msg: OrderMsg): F[CustomerOrder] =
???

}
```

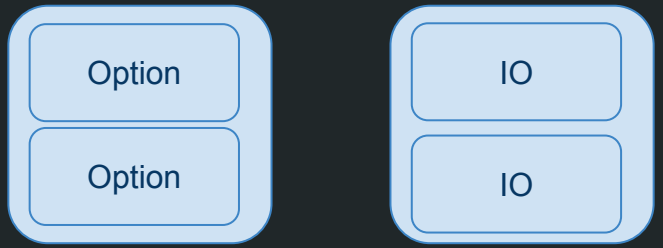
CustomerOrder is a validated domain object with invariants enforced by the type system

Customer and Sku Lookups

- Both Customer and Sku Lookups are effectful queries
  - There is one Customer lookup, but potentially many Sku lookups
- None depends on any other, so they should be done in parallel
- How can we compose multiple effectful queries in parallel?

Modes of Composition

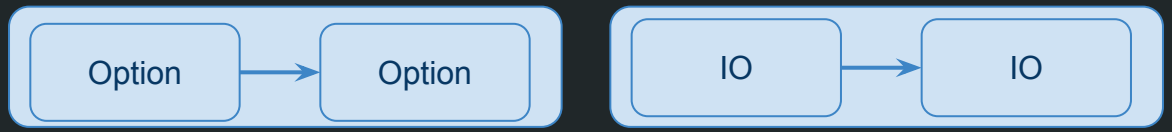
- *Independent effectful values can be run together concurrently*
  - *Applicative (or Semigroupal) operators*
- Multiple independent effectful values can be composed in parallel into a larger value of the same type using `Semigroupal.tupledN` operators
  - Operations can’t depend upon each other’s outputs and happen in an undefined order
- The “meaning” of the composition still depends upon the type of effect present in the value



Modes of Composition

- *An effectful value depends upon the result of another, or must happen after*
  - *Monadic operators*

- `FlatMap`, aka `Bind`, composes two effectful values into a larger value of the same type
  - The second operations happens after the first and may depend upon the value it yields
  - Flatmap operator is written symbolically as **>>=**
- The “meaning” of the composition depends upon the type of effect present in the value



Parallel Composition

- When we have a fixed number of effectful values that we wish to compose in parallel
  - Put them in a tuple
  - Invoke parTupled on the tuple
    - The components are all run concurrently and a tuple of the results is returned
  - Common variant: parMapN(f)
    - The components are all run concurrently, and the function f is invoked on the results (ie simply maps over the result)

Sku Lookups: Parallel Traversal

- Lets look at Sku lookups. We have a variable number of items to work with
  - `case class OrderMsg(customerId: String, skuQuantities: NonEmptyChain[(String, Int)])`
  - `trait SkuLookup[F[_]] { def resolveSku(s: String): F[Either[String, Sku]] }`

- We can invoke **parTraverse** on a `NonEmptyChain` to run an effectful function over each item, returning a `NonEmptyChain` of the results

Sku Lookups: Getting Rid of the Either

- SkuLookup returns us an Either[String, Sku]
  - How can we get at the String and deal with the Either?

- We can take advantage of the error facility built-in to the Sync effect
  - Sync[F] extends MonadError[F, Throwable]
  - The same combinator we used in previous problem, errorValueFromEither, can lift the left side of the Either into an error effect

- To feed the result of parTraverse over resolveSku into errorValueFromEither, we need to compose them sequentially (ie using monads)
  - Use **flatMap** or its alias **>>=** ie <expression1>.>>=(result => <expression2>)

PosInt: A look at Refinement Types

- Note SkuQuantity.quantity is a PosInt
  - A positive int is an Int certified by the type system to be > 0
  - Cannot order zero or a negative quantity of something

- PosInt is a *refinement type* provided by the Refined library
  - Refined types have a base type (Int in this case) refined with extra constraints (positiveness in this case)

- How do we obtain a PosInt?
  - PosInt provides a PosInt.fromF[F](<an int>): F[PosInt] that will raise an error if the passed Int isnt positive

PosInt: A look at Refinement Types

- Note SkuQuantity.quantity is a PosInt
  - A positive int is an Int certified by the type system to be > 0
  - Cannot order zero or a negative quantity of something

- PosInt is a *refinement type* provided by the Refined library
  - Refined types have a base type (Int in this case) refined with extra constraints (positiveness in this case)

- How do we obtain a PosInt?
  - PosInt provides a PosInt.fromF[F](<an int>): F[PosInt] that will raise an error if the passed Int isnt positive

Have a go

- **resolveOrderMsg** has been broken out into building block parts
  - The final composition of the parts is already filled in
- Try to implement the building blocks to implement **resolveOrderMsg**
- Run the unit tests to check your work
  - step2/test

Advice:

- Scala’s Type Inference is limited
  - Methods that take the effect type F as a type parameter will likely need to have F specified
  - If not, likely to see compiler error messages about diverging or ambiguous implicits