NYC Freestyle Enthusiasts

NYC Science, Industry and Business Library • 09.06.2018

Our first Meetup

- Welcome
  - Introductions and getting to know each other
  - Experience with funcional coding ?
- I am learning too - not an expert - looking for others to join me in this learning journey
- Goals for this meetup
  - Study Group?
  - Online Meetings??
  - Take turns presenting?

This Talk

NOTE: This is not all Original Content - I have assembled examples from several sources and links to original content are provided

Category Theory - base concepts

Free and Tagless - compare / contrast

Freestyle Macros - inside Freestyle

Code Review - using Freestyle / Freestyle-RPC

Why Functional ?

- Concurrency
- Composability
- Side Effects don’t scale
- Divide and Conquer
  - matches limits of human brain
- Testing vs Proof

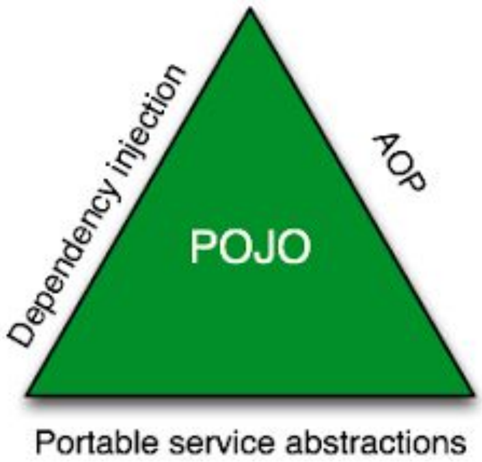
[Why Functional Programming From A Developer Productivity Perspective]

[Benefits of Functional Programming]

Why Freestyle ?

- Provide structure / Generate boilerplate
  - Algebras
  - Modules
  - Handlers
  - Parallel Execution (in monadic context)
- Dependency Injection - Free and Tagless
- Free Monads for cross cutting concerns - AOP
- Integrations
  - Ready to use algebras - think Spring Templates
- Algebras and Modules - think Spring Beans

I come from a Spring / J2EE background: like Spring in the Java world, freestyle provides building blocks for writing code



Category Theory

- Composability (p. 9)
- Operational vs Denotational Semantics (p. 20)
  - Proof via mathematical theorem
- Computational effects can be mapped to Monads (p. 21)
- Pure vs Dirty Functions (p. 22)

Category Theory for Programmers

Typeclassopedia - **Design Patterns**

Typelevel Ecosystem (partial)

- Dsl.scala: The !-notation for creating Cats monadic expressions
- eff: functional effects and effect handlers (alternative to monad transformers)
- Freestyle: pure functional framework for Free and Tagless Final apps & libs
- iota: Fast [co]product types with a clean syntax
- Monocle: an optics library for Scala (and Scala.js) strongly inspired by Haskell Lens.
- shims: seamless interoperability for cats and scalaz typeclasses and datatypes
- doobie: a pure functional JDBC layer for Scala
- Fetch: efficient data access to heterogeneous data sources
- finch: Scala combinator library for building Finagle HTTP services
- Frameless: Expressive types for Spark
- FS2: compositional, streaming I/O library
- http4s: A minimal, idiomatic Scala interface for HTTP
- Monix: high-performance library for composing asynchronous and event-based programs

Algebraic Data Types

Sum - “is a”

```
sealed trait A
final case class B() extends A
final case class C() extends A
```

A is a B *or* C

Product - “has a”

```
final case class A(b: B, c: C)
```

A has a B *and* C

[Essential Scala Six Core Concepts]

Structural recursion

[Short ebook]

[*Generalised Abstract Data Type (GADT)*], an interface to construct the algebra’s operations into a type F[_].

Free vs Tagless Final

Reference Article Video

```
case class User(id: UUID, email: String, loyaltyPoints: Int)
trait UserRepository {
  def findUser(id: UUID): Future[Option[User]]
  def updateUser(u: User): Future[Unit]
}
class LoyaltyPoints(ur: UserRepository) {
  def addPoints(userId: UUID, pointsToAdd: Int): Future[Either[String, Unit]] = {
    ur.findUser(userId).flatMap {
      case None => Future.successful(Left("User not found"))
      case Some(user) =>
        val updated = user.copy(loyaltyPoints = user.loyaltyPoints + pointsToAdd)
        ur.updateUser(updated).map(_ => Right(()))
    }
  } // other methods ...
}
```

Base Case is tied to Future

https://softwaremill.com/free-tagless-compared-how-not-to-commit-to-monad-too-early/

Free

Create case classes for algebra

```
sealed trait UserRepositoryAlg[T]
case class FindUser(id: UUID) extends UserRepositoryAlg[Option[User]]
case class UpdateUser(u: User) extends UserRepositoryAlg[Unit]

import cats.free.Free

type UserRepository[T] = Free[UserRepositoryAlg, T]
def findUser(id: UUID): UserRepository[Option[User]] = Free.liftF(FindUser(id))
def updateUser(u: User): UserRepository[Unit] = Free.liftF(UpdateUser(u))
```

@free - macro that will generate boilerplate associated with Free

“we return a **data structure** - a value - which uses abstract instructions, without specifying in any way how to interpret those instructions”

https://softwaremill.com/free-tagless-compared-how-not-to-commit-to-monad-too-early/

Tagless Final

Trait instead of case class

"we have parameterized both the UserRepository and LoyaltyPoints classes with the resulting container"

```
trait UserRepositoryAlg[F[_]] {
  def findUser(id: UUID): F[Option[User]]
  def updateUser(u: User): F[Unit]
}
class LoyaltyPoints[F[_]: Monad](ur: UserRepositoryAlg[F]) {
  def addPoints(userId: UUID, pointsToAdd: Int): F[Either[String, Unit]] = {
    ur.findUser(userId).flatMap {
      case None => implicitly[Monad[F]].pure(Left("User not found"))
      case Some(user) =>
        val updated = user.copy(loyaltyPoints = user.loyaltyPoints + pointsToAdd)
        ur.updateUser(updated).map(_ => Right(()))
    }
  }
}
```

@tagless - macro to generate boilerplate

https://softwaremill.com/free-tagless-compared-how-not-to-commit-to-monad-too-early/

Summary of Differences

- The biggest advantage of free is that programs become **values**, which can be passed as an argument, returned, combined, sequenced etc. (optimization)
- While final-tagless requires less boilerplate, and makes it much **easier to combine multiple languages**, it requires making everything generic in the resulting container `F[_]`. Free again benefits from the fact that it's just a value: it doesn't need to live in a parametrized "environment".
- More on the site… see link below

https://softwaremill.com/free-tagless-compared-how-not-to-commit-to-monad-too-early/

Algebras

http://frees.io/docs/core/algebras/

In Freestyle, an algebra is a trait or abstract class annotated with `@free` or `@tagless`:

```scala
import freestyle.free._

case class User(id: Long, name: String)

// defined class User

@free trait Users {

  def get(id: Long): FS[User]

  def save(user: User): FS[User]

  def list: FS[List[User]]

}
```

"Smart Constructors"

```scala
import freestyle.tagless._

@tagless(true) trait Validation {
  def minSize(s: String, n: Int): FS[Boolean]
  def hasNumber(s: String): FS[Boolean]
}
// defined trait Validation
// defined object Validation

@tagless(true) trait Interaction {
  def tell(msg: String): FS[Unit]
  def ask(prompt: String): FS[String]
}
```

Modules

```
object algebras {
  @tagless(true) trait Database {
    def get(id: Int): FS[Int]
  }
  @tagless(true) trait Cache {
    def get(id: Int): FS[Option[Int]]
  }
  @free trait Presenter {
    def show(id: Int): FS[Int]
  }
  @free trait IdValidation {
    def validate(id: Option[Int]): FS[Int]
  }
}
```

```
object modules {
  @module trait Persistence {
    val database: Database.StackSafe
    val cache: Cache.StackSafe
  }
  @module trait Display {
    val presenter: Presenter
    val validator: IdValidation
  }
  @module trait App {
    val persistence: Persistence
    val display: Display
  }
}
```

See Coproduct in Category Theory for Programmers

Things get more complicated once the number of Algebras grows. Fortunately, Freestyle automatically aligns all those for you and gives you an already aligned Coproduct of all algebras contained by a Module, whether directly referenced or transitively through its modules dependencies.

Handlers

http://frees.io/docs/core/handlers/

```
@free trait KVStore {
  def put[A](key: String, value: A): FS[Unit]
  def get[A](key: String): FS[Option[A]]
  def delete(key: String): FS[Unit]
  def update[A](key: String, f: A => A): FS.Seq[Unit] =
    get[A](key).freeS flatMap {
      case Some(a) => put[A](key, f(a)).freeS
      case None => ().pure[FS.Seq]
    }
}
// defined trait KVStore
// defined object KVStore
```

To define a runtime interpreter for this, we simply extend KVStore.Handler[M[_]] and implement its abstract members:

```
type KVStoreState[A] = State[Map[String, Any], A]
// defined type alias KVStoreState

implicit val kvStoreHandler:
KVStore.Handler[KVStoreState] = new
KVStore.Handler[KVStoreState] {
  def put[A](key: String, value: A): KVStoreState[Unit]
=
    State.modify(_.updated(key, value))
  def get[A](key: String): KVStoreState[Option[A]] =
    State.inspect(_.get(key).map(_.asInstanceOf[A]))
  def delete(key: String): KVStoreState[Unit] =
    State.modify(_ - key)
}
```

Target implementation

1. **Create type alias**
2. **Supply to “new”**

Build Programs from Modules

```
import modules._

def program[F[_]](implicit app: App[F]): FreeS[F, Int] = {
  import app.display._, app.persistence._
  for {
    cachedToken <- cache.get(1)
    id          <- validator.validate(cachedToken)
    value       <- database.get(id)
    view <- presenter.show(value)
  } yield view
}
```

import
modules

Farewell to Free ?

https://typelevel.org/blog/2017/12/27/optimizing-final-tagless.html

https://youtu.be/E9iRYNuTIYA

- Free Applicatives allow us to inspect inner structure allows optimization (peek ahead) cannot do with Free Monads
- Tagless Final has no boilerplate - no intermediate CoProduct structure is more performant
- Tagless final not bound to Applicative / Monad - principle of least power

- Interpret twice in tagless allows peek ahead - instead of IO → nested IO
- Sphynx optimizer

Reference Material in Depth

Related Articles

- Farewell to Free
- Free and Tagless Compared

Freestyle / Freestyle - RPC

Examples

- todo list-lib
- todo list - rpc
- tagless-vs-free
- tagless-vs-free-w-modules
- More

References

- http://frees.io/
- https://typelevel.org/cats/
- 3 books
- Category theory for programmers videos

Next steps

Group Coding?

Scala Exercises?

Online Meetups?

Goals for next meeting

1.
2.
3.