Finally! Tagless and Fancy-Free Monads

Lance Gatlin

03Oct18

Who am I?

- Scala developer, 6 years, 7 teams
  - No background in functional programming before Scala
  - I prefer data-flow, service-oriented, no frills, no magic, least power Scala
- Independent consultant
  - Doer, fixer, closer, truth-seeker, adventurer
  - I don’t often greenfield projects
  - I work within the bounds of team’s skills & culture

Who am I?

- My philosophy:
  - Everyday, less wrong; Everyday, better
- If you disagree with me, please let me know!*
  - You might teach me something (thanks!)
  - *We have limited time, so I may defer talking about your question to later

Who am I?

- Trouble-maker:
  - I remain unconvinced that reifying purity in Scala (i.e. non-async IO monad) is worth the complexity trade off
    - What makes code “easier to reason about” is an opinion
    - Marking pure (or impure) functions in documentation or in naming convention is a design pattern I’m all for!

My Manifesto

1. Write readable code
   - Humans matter more (write once, read many)
   - Write code team can read today, push to expand that (code reviews, brown bags, tech talks, etc)
2. Keep it simple
   - The domain & its problems are hard enough
   - Love your future-self now, and you'll always love your past-self
   - Always understand the cost/benefit of introducing a new non-standard library concept
3. Be connected to the needs of users
   - Coding is the art of trading time for features & fixes
   - When shortcuts & comprises are needed (they always are), knowing users' needs allows for better choices
4. Incrementally, deliver the right value, at the right time
   - Talk about anything, but only work on what users/stakeholders care about right now
   - Avoid treating job as a technical playground
5. Success = 50% hard work, 50% perception of that hard work
   - Be an active participant in influencing that perception
   - Don't work hard if no one is paying attention, instead first work hard on getting someone to pay attention

Overview

- What is tagless-final?
- Tagless-final example
  - Cats-tagless
- Why does it matter?
- How does it compare to the free monad?
- When should I use use it?
- When should I use tagless-final or free monad?
- Questions

What is tagless-final?

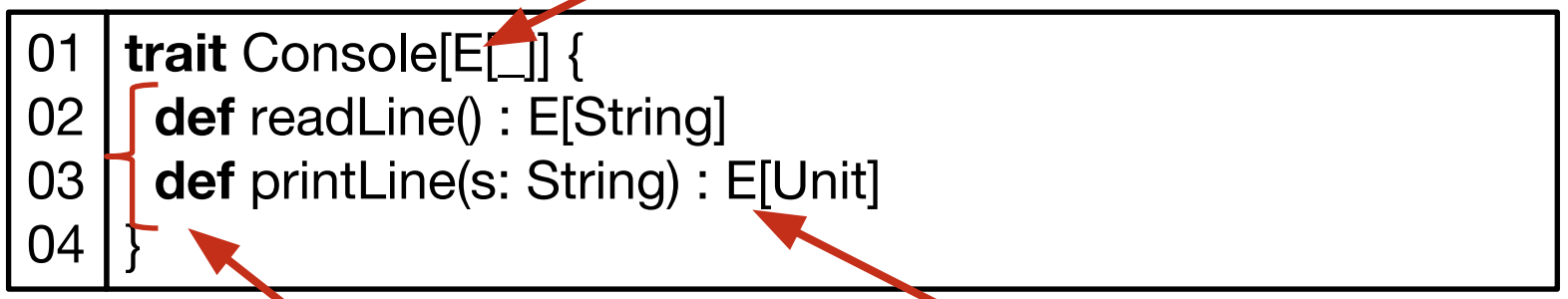
- Tagless-final is a pattern for embedding DSLs (i.e. algebras, services, APIs, etc) in typed functional languages like Scala
- Example DSL:

```
01 trait Console[E[_]] {
02   def readLine() : E[String]
03   def printLine(s: String) : E[Unit]
04 }
```

http://okmij.org/ftp/tagless-final/

What is tagless-final?

- Tagless-final is a pattern for embedding DSLs (i.e. algebras, services, APIs, etc) in typed functional languages like Scala
- Example DSL:



http://okmij.org/ftp/tagless-final/

Why is it called tagless-final?

- “Tagless” since we don’t reify operations as values
- “Final” since execution occurs immediately inside monad
  - (As opposed to free monad which suspends execution until later)
- Found a few sources that call it “finally tagless” or “final tagless” too

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

What is tagless-final?

- Interpreter: implements DSL for concrete monad

```
01  val console = new Console[Id] {
02    def readLine() : Id[String] = System.console.readLine
03    def printLine(s: String) : Id[Unit] = println(s)
04  }
```

What is tagless-final?

- Interpreter: implements DSL for concrete monad

```
01 val console = new Console[Future] {
02   def readLine() : Future[String] = Future {
03     System.console.readLine
04   }
05   def printLine(s: String) : Future[Unit] = Future {
06     println(s)
07   }
08 }
```

What is tagless-final?

- Interpreter: implements DSL for concrete monad

```
01  val console = new Console[Future] {
02    def readLine() : Future[String] = Future {
03      System.console.readLine
04    }
05    def printLine(s: String) : Future[Unit] = Future {
06      println(s)
07    }
08  }
```

- Note: could also implement E as Free Monad!

What is tagless-final?

- Stack safety
  - Tagless-final has the stack safety of the target monad
  - Id => no stack safety
  - Future/IO/Monix Task => stack safety
  - Free => stack safety

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

What is tagless-final?

- Bridge is a DSL written in terms of other DSLs

```
01 class Logger[E[_]] extends Console[E] {
02   def info(message: String) : E[Unit] =
03     printLine(message)
04 }
```

What is tagless-final?

- Bridge (same but compose Console)

```
01 class Logger[E[_]](console: Console[E]) {
02   def info(message: String) : E[Unit] =
03     printLine(message)
04 }
```

What is tagless-final?

- Bridge (same but more generalized)

```
01 trait Logger {
02   def info[E[_]](message: String)(implicit console: Console[E]) : E[Unit] =
03     printLine(message)
04 }
```

What is tagless-final?

- Program: uses DSL to do work

```
01 def program[E:Monad](console: Console[E]) : E[Unit] =
02   for {
03     s <- console.readLine()
04     _ <- console.printLine(s)
05   } yield ()
```

What is tagless-final?

- Program: uses DSL to do work

```
01 def program[E:Monad](console: Console[E]) : E[Unit] =
02   for {
03     s <- console.readLine()
04     _ <- console.printLine(s)
05   } yield ()
```

MONAD: have to commit to somebody's type-class (Cats, Scalaz, ???)

FLATMAP: Could use just FlatMap typeclass here (or other FlatMap derived)

What is tagless-final?

- Tagless-final was created as an answer to the “expression problem” in DSLs:
  - Functional only: adding a new operation is easy (function), but adding a new expression type (ADT) is difficult (update all functions)
  - OOO only: adding a new expression type is easy (extend object), but adding a new operation is difficult (update all objects)
  - Scala(both OO and functional), is uniquely suited to easily implement tagless-final

https://oleksandrmanzyuk.wordpress.com/2014/06/18/from-object-algebras-to-finally-tagless-interpreters-2/

Slightly more interesting example

- Tagless-final “service”

```
1  trait Users[E[_]] {
2    def findByUsername(username: String) : E[Option[User]]
3    def findById(id: UUID) : E[Option[User]]
4    def findAll(start: Int, batchSize: Int) : E[Seq[User]]
5    def create(id: UUID,username: String,plainTextPassword: String) : E[Boolean]
6    def rename(userId: UUID, newUsername: String) : E[Boolean]
7    def setPassword(userId: UUID, plainTextPassword: String) : E[Boolean]
8    def remove(userId: UUID) : E[Boolean]
9  }
```

Declaration

```
1  trait Users[E[_]] {
2    def findByUsername(username: String) : E[Option[User]]
3    def findById(id: UUID) : E[Option[User]]
4    …
5  }
```

Declaration

DATA-IN: Input parameters are typically simple data types (e.g. String, Int, collection, case class)

```
1  trait Users[E[_]] {
2    def findByUsername(username: String) : E[Option[User]]
3    def findById(id: UUID) : E[Option[User]]
4    …
5  }
```

DATA-OUT: return values are typically simple data types

Implementation

```
01 class UsersImpl[E[_]:Monad](
02   usersDao: SqlDocDao[UUID,UserData,E],
03   passwords: Passwords[E],
04   logger: Logger[E]
05 ) extends Users[E] {
06   …
07   def findById(id: UUID) =
08     usersDao.findById(id)).map(toUser)
09   …
10 }
```

Implementation

TYPECLASS-DEP: Declare minimum type-classes needed for monad-shape

```
01  class UsersImpl[E[_]:Monad](
02    usersDao: SqlDocDao[UUID,UserData,E],
03    passwords: Passwords[E],
04    logger: Logger[E]
05  ) extends Users[E] {
06    …
07    def findById(id: UUID) =
08      usersDao.findById(id)).map(toUser)
09    …
```

COMPOSE: Inject instances of DSLs required & propagate the generic monad-shape to composed DSLs

Method implementation

```
12  …
13  def create(id: UUID, username: String, plainTextPassword: String): E[Boolean] =
14    for {
15      optUser <- findByIdOrUsername(id,username)
16      createOk <- optUser match {
17        case Some(_) => E.pure(false)
18        case None =>
19          for {
20            digest <- passwords.mkDigest(plainTextPassword)
21            insertResult <- usersDao.insert(id,UserData(
22              username = username,
23              passwordDigest = digest
24            ))
25            _ <- logger.info(s"Created user $id with username $username")
26          } yield insertResult
27      }
28    } yield createOk
```

Test

```
01  class UsersImplTest extends FlatSpec with Matchers with MockFactory {
02    class Fixture(
03     val usersDao: SqlDocDao[UUID,UserData,Id] = stub[SqlDocDao[UUID,UserData,Id]],
04     val passwords: Passwords[Id] = stub[Passwords[Id]],
05     val logger: Logger[Id] = mock[Logger[Id]]
06    ) {
07      val users = new UsersImpl[Id](
08        usersDao = usersDao,
09        passwords = passwords,
10        logger = logger
11      )
12    }
13  …
```

Test

```
14  …
15  "UsersImpl.create" should "create a new user when id & username does not already exist" in
16  {
17    val fixture = new Fixture
18    import fixture._
19    val id = UUID.randomUUID()
20    val newUserData = UserData(
21      username = "test-user",
22      passwordDigest = "test-digest"
23    )
24    (usersDao.findById _).when(id).returns(None)
25    (usersDao.findByNativeQuery _).when(s"`username`='test-user'").returns(Seq.empty)
26    (passwords.mkDigest _).when("test-password").returns("test-digest")
27    (usersDao.insert _).when(id,newUserData).returns(true)
28    (logger.info _).expects(s"Created user $id with username test-username").once()
29
30    users.create(id,"test","password") shouldBe true
31  }
    …
```

Create an instance (Future)

```
01  val sqlDocDao : SqlDocDao[UUID,User,Future] = …
02  val passwords : Passwords[Future] = …
03  val logger : Logger[Future] = …
04
05  val users = new UsersImpl[Future](
06    sqlDocDao = sqlDocDao,
07    passwords = passwords,
08    logger = logger
09  )
10  …
11  val result : Future[Boolean] = users.create(
12    UUID.randomUUID(),
13    ”SomeUser”,
14    ”password”
15  )
```

Create an instance (Monix Task)

```
01  val sqlDocDao : SqlDocDao[UUID,User,Task] = …
02  val passwords : Passwords[Task] = …
03  val logger : Logger[Task] = …
04
05  val users = new UsersImpl[Task](
06    sqlDocDao = sqlDocDao,
07    passwords = passwords,
08    logger = logger
09  )
10  …
11  val result : Task[Boolean] = users.create(
12    UUID.randomUUID(),
13    ”SomeUser”,
14    ”password”
15  )
```

What if composed service is wrong E?

```
01  val sqlDocDao : SqlDocDao[UUID,User,Future] = …
02  val passwords : Passwords[Future] = …
03  val logger : Logger[Id] = …
04
05  val users = new UsersImpl[Future](
06    sqlDocDao = sqlDocDao,
07    passwords = passwords,
08    logger = logger
09  )
10  …
11  val result : Future[Boolean] = users.create(
12    UUID.randomUUID(),
13    ”SomeUser”,
14    ”password”
15  )
```

Cats Tagless

```
01  import cats.tagless._
02
03  @autoFunctorK
04  trait ExpressionAlg[F[_]] {
05    def num(i: String): F[Float]
06    def divide(dividend: Float, divisor: Float): F[Float]
07  }
```

- Cats' FunctionK can transform an ExpressionAlg[F] to a ExpressionAlg[G]
  - Using a FunctionK[F, G], a.k.a. F ~> G.
- https://github.com/typelevel/cats-tagless

Why does it matter?

- Because final-tagless pattern is simple and regular, code generation based on DSLs can be created to auto-generate:
  - REST client/server stub/proxies/docs
  - Service deployment framework
  - Auto “lift” of monadic type (e.g. Cats-tagless FunctionK)
- E type-class can also include more interesting effects:
  - Parallelization
  - Built in logging
  - Custom exception/error handling

Why does it matter?

- We (the Scala community) can (finally!) begin to write ”middleware” code that can be shared across all silos:
  - Lightbend (Futures, actors, Kafka)
  - Scalaz/Typelevel (pure functional programming)
  - Spark (Spark DSL)
  - Better-python (Li Haoyi, blocking)
  - Better-java (blocking, reflection, null)
- Like https://github.com/mohiva/play-silhouette
  - But for all silos!

Why does it matter?

- This makes me very excited about tagless-final and its potential for Scala!
- Silo'd Scala has led to huge fragmentation!
  - Every team I've worked with does Scala differently!
    - Those choices often make their code incompatible with different silos!
  - New concepts embedded in the type system leak everywhere
    - All or nothing for concepts like IO monad
  - Open source from silos often doesn't interop with other silos!
- Other languages don't have this problem!
  - Leads to a lot of teams finding it easier to just create their own eco-system

Should I use it?

- If your team is comfortable with monadic programming, it’s worth considering
- OR if you are writing open source and you want to make code re-useable by most everyone in the wider community

Tagless-final or Free monad?

- Use tagless-final
  - If team is unsure exactly how code will be used (e.g. open source)
    - Can render to Free later
  - Also if team is ok with delegating stack safety to monad
- Free monad
  - If you need stack safety (monadic loops, deep workflows)
  - If you know your use case includes serialization of code (e.g. pass to workers or remote server)
  - Need transaction semantics
- Prefer tagless-final since it is less complex than Free monad in Scala

When NOT to use tagless-final or free

- If team isn't comfortable with monadic coding
  - It's a very big conceptual jump
- Of if monadic coding isn't a good fit for requirements
  - CPU bound tasks
  - Low memory footprint

Questions?

lance.gatlin@gmail.com

https://github.com/lancegatlin

References

- https://softwaremill.com/free-tagless-compared-how-not-to-commit-to-monad-too-early/
- http://okmij.org/ftp/tagless-final/
- https://github.com/typelevel/cats-tagless
- https://blog.scalac.io/exploring-tagless-final.html
- https://typelevel.org/blog/2017/12/27/optimizing-final-tagless.html
- https://www.beyondthelines.net/programming/introduction-to-tagless-final/
- https://medium.com/@agaro1121/free-monad-vs-tagless-final-623f92313eac
- https://pchiusano.github.io/2014-05-20/scala-gadts.html