Automatic law checking with Tagless (using Discipline & ScalaCheck)

lance.gatlin@gmail.com 19Feb19

Who am I?

- Scala developer, 7 years, 7 teams
  - Prefer data-flow, service-oriented, no frills, no magic, least power Scala
- Independent consultant
  - Doer, fixer, closer, truth-seeker, adventurer
  - I work within the bounds of team's skills & culture
  - Available!

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

My Code Principles (read later)

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

- Review tagless algebra/API
  - Users from last presentation
- What/Why generic law-based auto-testing?
- Quick reviews:
  - ScalaCheck
  - Discipline
- Overall auto-test architecture
  - Writing generic laws
- Modeling effects
- Writing an auto-test for a specific implementation and monad
- Questions

Follow along?

- I posted a link to the github repo for this presentation in today's meetup
- Runnable demo code
  - sbt test
- Also linked here for later
  - https://github.com/lancegatlin/tagless-final-autotest-talk-19feb19

Users algebra/API review

```
01 trait Users[E[_]] {
02   def findById(id: UUID) : E[Option[User]]
03   def create(id: UUID, username: String, plainTextPassword: String) : E[Boolean]
04   def remove(userId: UUID) : E[Boolean]
05   …
06 }
```

```
01 object Users {
02   case class User(
03     id: UUID,
04     username: String,
05     passwordDigest: String,
06     created: Instant,
07     removed: Option[Instant]
08   )
09 }
```

What is generic law-based auto-testing?

- Builds on automatic property checking of ScalaCheck
- Written only in terms of the tagless algebra/API (and the expected effects)
- Leverage ScalaCheck to dynamically inject random boundary-case test cases
- Verify all laws hold for any combination of algebra implementation and monad
- Re-use laws across algebras
- Everything is tested automatically

Example explicit Users test with mocking

```
01 "UsersImpl.create" should "create a new user when id & username does not already exist" in {
02     val fixture = new Fixture
03     import fixture._
04     val id = UUID.randomUUID()
05     val newUserData = UserData(
06       username = "test-user",
07       passwordDigest = "test-digest"
08     )
09     (usersDao.findById _).expects(id).returns(None).once
10     (usersDao.findByNativeQuery _).expects("`username`='test-user'").returns(Seq.empty).once
11     (passwords.mkDigest _).when("test-password").returns("test-digest")
12     (usersDao.insert _).expects(id, newUserData).returns(true).once
13     (logger.info _).expects(s"Created user $id with username test-user").once
14
15     users.create(id,"test-user","test-password") shouldBe true
16   }
```

Why generic law-based auto-testing?

- Writing explicit tests is time consuming and error prone
- Must write unit tests and separate integration tests
- Boundary condition testing is generally skipped
- Generic law-based auto-testing :
  - Same code used for:
    - Unit testing (Id monad)
    - Integration testing (Future, IO, Task, etc)
    - With or without backend database, in-memory, etc
    - Verify any implementation
    - Verify all laws hold for any combination of implementation and monad

ScalaCheck review

- ScalaCheck has properties (org.scalacheck.Prop)
- Properties are functions that can accept zero or more typed parameters that verify some property is true
- ScalaCheck generates a few random valid values to plug into property functions based on the Arbitrary type-class
- ScalaCheck also ensures boundary condition values are tested (e.g. -1, 0, 1, Int.MaxValue, empty string, etc)

http://www.scalacheck.org/

ScalaCheck review

```
01  import org.scalacheck.Properties
02  import org.scalacheck.Prop.forAll
03
04  object StringSpecification extends Properties("String") {
05    property("startsWith") = forAll { (a: String, b: String) =>
06      (a+b).startsWith(a)
07    }
08
09    property("concatenate") = forAll { (a: String, b: String) =>
10      (a+b).length > a.length && (a+b).length > b.length
11    }
12
13    property("substring") = forAll { (a: String, b: String, c: String) =>
14      (a+b+c).substring(a.length, a.length+b.length) == b
15    }
16  }
```

http://www.scalacheck.org/

Discipline review

- Allows writing RuleSets:
  - Consists of named “laws” (i.e. ScalaCheck Prop)
  - Allows for re-using laws from other RuleSets (parents & bases)
    - Not utilized here
- checkAll tests all properties in RuleSets

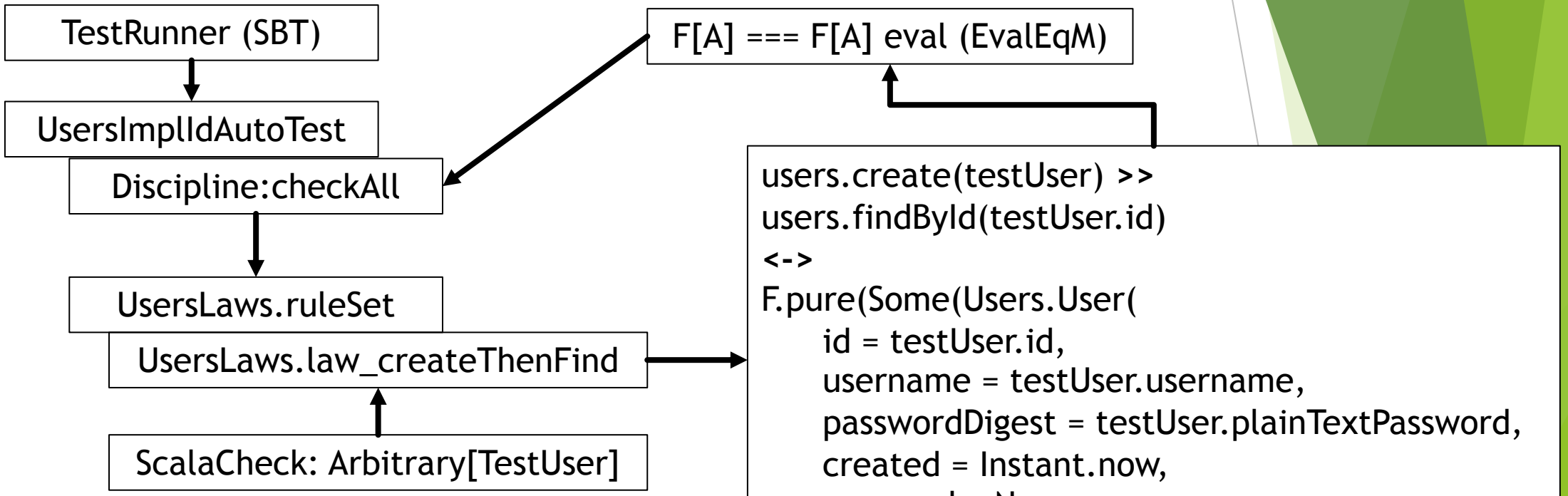
https://typelevel.org/blog/2013/11/17/discipline.html

Discipline review

```
01 trait Laws {
02   trait RuleSet {
03     def name: String
04     def bases: Seq[(String, Laws#RuleSet)] = Seq()
05     def parents: Seq[RuleSet] = Seq()
06     def props: Seq[(String, Prop)] = Seq()
07     // ...
08   }
09 }
10 class SomeTypeclassLaws extends Laws {
11    val laws : RuleSet = ???
12 }
13 checkAll(”SomeTypeClass", SomeTypeClassLaws.laws)
```

https://typelevel.org/blog/2013/11/17/discipline.html

Overall auto-test flow



****But what about effects???**

Note: a >> b is a.flatMap(_ => b)

Note: a <-> b means ‘a’ must be “equal” to ’b’ (i.e. cats.Eq)

Writing generic laws

- To allow laws to be run against any implementation and any monad
  - Must write laws only in terms of algebra
  - Must verify results and expected effects
  - Must be able to eval (i.e. run) Monad in current thread
    - Due to design of ScalaCheck & Discipline
    - For async Monad this means blocking (only in tests)

Modeling effects

- Previous law tests output, but what about effects?
- What are effects exactly?
  - Structured modifications to some underlying “effect system”
  - We can model as an accumulation of state (e.g. in-memory or database)
  - Or we could model as an accumulation of effect objects themselves (like free monad)
- To test effects, I’ve added a test extension algebra that allows explicitly specifying the expected effects inside laws

Test UsersEfx extension

```
01  trait UsersEfx[F[_]] { self:Users[F] =>
02    def efx_state : F[List[Users.User]]
03    def efx_createUser(testUser: TestUser) : F[Unit]
04    def efx_removeUser(testUser: TestUser) : F[Unit]
05  }
```

- Note: how these are implemented depends on the Users implementation backend
  - Could modify a database
  - Could update in memory map
  - Could accumulate effect ADT

Explicitly test accumulation of effects

```
01  def law_createEfx(testUser: TestUser) = {
02      {
03        val users : Users[F] with UsersEfx[F] = mkFixture()
04        users.create(testUser.id, testUser.username,
05  testUser.plainTextPassword) >>
06        users.efx_state
07      } <-> {
08        val users : Users[F] with UsersEfx[F] = mkFixture()
09        users.efx_createUser(testUser) >>
10        users.efx_state
11      }
12    }
```

UsersLaws

Source:

https://github.com/lancegatlin/tagless-final-autotest-talk-19feb19/blob/master/src/test/scala/org/ldg/UsersLaws.scala

UsersImplIdAutoTest

Source:

https://github.com/lancegatlin/tagless-final-autotest-talk-19feb19/blob/master/src/test/scala/org/ldg/UsersImplIdAutoTest.scala

Future work

- Improving ScalaCheck & Discipline to handle Monads properly to avoid EvalEqM
- Extend Discipline to mix laws
  - If A <-> B and B <-> C then why not try A <-> C?
- Once a law is verified, it only needs to be checked again if the code changes (test caching)
- Build a tighter DSL and testing fixture that:
  - Makes it easier to test results & effects in the same law
  - Makes it easier to specify the expected effects and the result in the law

Questions?

lance.gatlin@gmail.com

https://github.com/lancegatlin

References

- https://medium.com/iterators/tagless-with-discipline-testing-scala-code-the-right-way-e74993a0d9b1
- http://www.scalacheck.org/
- https://github.com/typelevel/discipline