flatMap(Oslo) 9 May 2019

Galaxy Brain

type-dependence and state-dependence in property-based testing

Erik Osheim (@d6)

stripe

who am i?

— typelevel member λ

— errant maintainer of many scala libraries 🐴

— interested in expressiveness and performance 🌓

— work on ML infrastructure at stripe ✍

my github is at `http://github.com/non`

what will this talk cover?

1. property-based testing overview
2. complicating factors and motivating example
3. generating types to test generic programs
4. tracking state during generation
5. review and conclusion

overview

origins of property-based testing

"QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" (ICFP 2000)

by Koen Claessen and John Hughes

the basic idea

1. Describe how to generate input data
2. Describe properties for all input data
3. Generate cases until satisified (or until failure)

Generalizes the process of writing specific test cases.

```
test("renders are constant after maxWidth") {
  forAll { (d: Doc, ws: List[Int]) =>
    val m = d.maxWidth
    val maxR = d.render(m)
    val justAfter = (1 to 20).iterator
    val trials = (justAfter ++ ws.iterator)
    val goodW = trials.map(w => (m + w) max m)
    assert(goodW.forall(w => d.render(w) == maxR))
  }
}
```

(taken from http://github.com/typelevel/paiges/)

Why is this great?

We have a chance to catch bugs we didn't anticipate.

(Anecdotally, this happens a lot.)

What are the odds of this happening?

P(caught) = P(generated) * P(noticed)

What are the odds of this happening?

P(caught) = P(generated) * P(noticed)

What are the odds of this happening?

P(caught) = P(generated) * P(noticed)

Obvious bugs are often caught immediately.

Subtle bugs may be found after months of testing.

What are the odds of this happening?

P(caught) = P(generated) * P(noticed)

Obvious bugs are often caught immediately.

Subtle bugs may be found after months of testing.

What does this imply?

1. start simple

Start with simple generators and properties.

Often even a simple property will catch real bugs.

2. focus on the canary

What would signal a problem?

— Invariants work great here!
— Don't worry too much about root causes.
— The job of a property is to expose possible problems.

Often inspired by bugs seen in the wild.

3. coverage matters

— Improve quality of input data
— If P(generated) = 0, then P(caught) = 0
— Prioritize after (1) and (2) for maximum impact ☼

This talk will focus on (3).


complications

nota bene

These are real issues we hit at work. ☠

Fixing them exposed at least 5-6 serious bugs (YMMV).

I've created simplified examples for this talk.

Names changed to protect the innocent (and the guilty).

 But first, terminology! 

algebraic data types (ADTs)

ADTs may be recursive (as below).

```haskell
-- list in haskell, parametric
data List a = Nil | Cons a (List a)

-- tree in haskell, non-parametric
data Tree = Empty | Leaf Int | Node Tree Tree
```

Often encoded in Scala with `sealed trait` + classes.

generalized algebraic data types (GADTs)

GADTs generalize parametric ADTs, allowing the type of recursive references to vary.

Huh?

generalized algebraic data types (GADTs)

GADTs generalize parametric ADTs, allowing the type of recursive references to vary.

```
// List[Int] is a parametric ADT
val xs: List[Int] =
  1 :: 2 :: 3 :: Nil

// shapeless.HList is a GADT
val xs: Int :: Boolean :: Double :: shapeless.HNil =
  1 :: true :: 3.0 :: shapeless.HNil
```

generalized algebraic data types (GADTs)

```
sealed trait Expr[A]

object Expr {
  case class Const[A](value: A) extends Expr[A]

  case class Map[A, B](e: Expr[A], f: A => B) extends Expr[B]

  case class Plus(x: Expr[Double], y: Expr[Double])
    extends Expr[Double]

  case class Times(x: Expr[Double], y: Expr[Double])
    extends Expr[Double]
}
```

GADTs you may be familiar with

- shapeless.HList
- com.twitter.scalding.TypedPipe
- org.apache.spark.rdd.RDD [1]
- fs2.Stream
- cats.data.Free
- etc.

[1] RDD is not sealed, so this one is kind of sketchy.

case study: TypedPipe

We'll use Scalding's TypedPipe[A] as our case study. [2]

— Represents future MapReduce computation graph
— Separate compilation and evaluation
— Similarities to RDD or Stream

[2] The actual system these techniques were developed for is an ML feature system. It defines features that can be compiled to Scalding, Spark, or other platforms. But TypedPipe is sufficient to demonstrate the ideas.

property testing for GADTs can be challenging

Consider:

```
forAll { (pipe: TypedPipe[(String, Int)]) =>
  ...
}
```

How might you generate TypedPipe[(String, Int)]?

how about this?

```
def genPipe[A](g: Gen[A]): Gen[TypedPipe[A]] =
  Gen.listOf(g).map { (as: List[A]) =>
    TypedPipe.from(as)
  }
```

Now you can write properties using TypedPipe.

But...

we need to go deeper

There are many ways to create pipes:

— `map`, `flatMap`, `filter`

— `++`, `either`

— `distinct`, `grouped`

— `join`, `cogroup`

— ...and more!

Our previous appraoch won't exercise any of these.

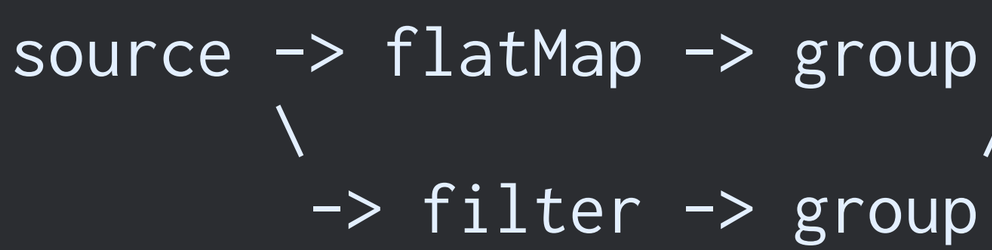
why not?

Because TypedPipe represents a computation graph. [3]

Many bugs will only be exposed with particular graphs.

[3] Spark's RDD does too but the details are messier.

An example TypedPipe graph (arrows are data flow):



(Not the actual GADT names, just a sketch.)

Imagine that we are implementing Scalding's TypedPipe:

— We are essentially compiler authors.
— We will be testing operations on TypedPipe.
— Any two operations on pipes might interact badly.
— We want to be able to generate buggy graphs.
— We also want to be able to test graph optimizations.

Some operations we'll want to test:

- `map`
- `filter`
- `flatMap`
- `distinct`
- `group`
- `join`
- `toTypedPipe`

Ideally our genPipe will generate all of these.

generator cheat sheet

— Gen[A] generates A values
— Cogen[A] consumes A values
— to generate (A, B) we need Gen[A] and Gen[B]
— to generate A => B we need a Cogen[A] and Gen[B]
— we can use map and flatMap to transform Gen

(Examples use an idealized implementation.)

toward on Gen[TypedPipe[A]]

```
// TypedPipe from a list is straightforward
def genFrom[A](g: Gen[A]): Gen[TypedPipe[A]] =
  for {
    lst <- Gen.listOf(g)
  } yield TypedPipe.from(lst)
```

toward on Gen[TypedPipe[A]]

```
// using our previous generator we can also filter
def genFilter[A](
    gpipe: Gen[TypedPipe[A],
    gp: Gen[A => Boolean]
): Gen[TypedPipe[A]] =
  for {
    pipe <- gpipe
    predicate <- gp
  } yield pipe.filter(predicate)
```

We're assuming we have `Gen[TypedPipe[A]]`.

toward on Gen[TypedPipe[A]]

```
// map is pretty similar to filter
def genMap[A, B](
    gpipe: Gen[TypedPipe[A]],
    gf: Gen[A => B]
): Gen[TypedPipe[B]] =
  for {
    pipe <- gpipe
    f <- gf
  } yield pipe.map(f)
```

So far so good, but do they compose?

```scala
def genPipe[A](g: Gen[A]): Gen[TypedPipe[A]] = {

  lazy val self: Gen[TypedPipe[A]] =
    Gen.defer(genFrom(g))

  self
}
```

```scala
def genPipe[A](g: Gen[A], c: Cogen[A]): Gen[TypedPipe[A]] = {

  val gp: Gen[A => Boolean] = Gen.function(c, Gen.boolean)

  lazy val self: Gen[TypedPipe[A]] =
    Gen.defer(Gen.oneOf(
      genFrom(g),
      genFilter(self, gp)))

  self
}
```

Filter seems fine (adding a `Cogen[A]` requirement).

```scala
def genPipe[A](g: Gen[A], c: Cogen[A]): Gen[TypedPipe[A]] = {

  val gp: Gen[A => Boolean] = Gen.function(c, Gen.boolean)
  val gf: Gen[A => A] = Gen.function(c, g)

  lazy val self: Gen[TypedPipe[A]] =
    Gen.defer(Gen.oneOf(
      genFrom(g),
      genFilter(self, gp),
      genMap(self, gf)))
}
```

```scala
def genPipe[A](g: Gen[A], c: Cogen[A]): Gen[TypedPipe[A]] = {

  val gp: Gen[A => Boolean] = Gen.function(c, Gen.boolean)
  val gf: Gen[A => A] = Gen.function(c, g)

  lazy val self: Gen[TypedPipe[A]] =
    Gen.defer(Gen.oneOf(
      genFrom(g),
      genFilter(self, gp),
      genMap(self, gf)))
}
```

Hmmm... something seems strange.

what's the problem?

Normally map goes between two types: `A => B`.

But we're always generating `A => A` functions.

Can we fix it?

```scala
def genPipe[A](g: Gen[A], c: Cogen[A]): Gen[TypedPipe[A]] = {

  val gp: Gen[A => Boolean] = Gen.function(c, Gen.boolean)
  val gf: Gen[A => A] = Gen.function(c, g)
  val gf1: Gen[Int => A] = Gen.function(Cogen.int, g)
  val self1: Gen[TypedPipe[Int]] = genPipe(Gen.int, Cogen.int)

  lazy val self: Gen[TypedPipe[A]] =
    Gen.defer(Gen.oneOf(
      genFrom(g),
      genFilter(self, gp),
      genMap(self, gf),
      genMap(self1, gf1)))
```

It works... sort of. Feels like a hack. ☂

Combinatorics are not in our favor,
and if you pull on this thread it gets worse:

```
// some operators need type class instances
distinct: (TypedPipe[A], Ordering[A]) => TypedPipe[A]

// some operators don't work for any type A.
group: (TypedPipe[(K, V)], Ordering[K]) => Grouped[K, V]

// some operators don't end with the same type of thing
// they started with.
join: (Grouped[K, V1], Grouped[K, V2]) => Grouped[K, (V1, V2)]
```

...and so on.

— Writing recursive Gen instances is already tricky.
— Recursive GADTs are significantly harder.
— We can create wimpy generators
— We can hardcode some types and pray
— We can stack hacks on hacks

These approaches all have major downsides.

generating types

cutting the gordian knot

Example-based testing uses hardcoded values.

By contrast, property-based testing generates values.

But with GADTs we're still hardcoding the types.

What if we… didn't? ✂

```
sealed abstract class TypeWith[Ev[_]] {
  type Type
  def evidence: Ev[Type]
}
```

```scala
sealed abstract class TypeWith[Ev[_]] {
  type Type
  def evidence: Ev[Type]
}

// we don't know what t.Type is.
val t0: TypeWith[Ordering] = ...

// but we DO know we have an Ordering[t.Type]
val o: Ordering[t0.Type] = t0.evidence

// similarly for Gen
val t1: TypeWith[Gen] = ...
val g: Gen[t1.Type] = t1.evidence
```

```scala
sealed abstract class TypeWith[Ev[_]] {
  type Type
  def evidence: Ev[Type]
}

object TypeWith {

  type Aux[A, Ev[_]] = TypeWith[Ev] { type Type = A }

  def apply[A, Ev[_]](implicit eva: Ev[A]): Aux[A, Ev] =
    new TypeWith[Ev] {
      type Type = A
      def evidence: Ev[Type] = eva
    }
}
```

```scala
// let's build some TypeWith instances
// the Ordering instances are picked up implicitly
val t0 = TypeWith[Int, Ordering]
val t1 = TypeWith[Byte, Ordering]
val t2 = TypeWith[String, Ordering]

// the vector only tracks the evidence type (Ordering),
// not the specific value types (Byte, Int, etc.)
val types: Vector[TypeWith[Ordering]] =
  Vector(t0, t1, t2)

// proving it really is path-dependent
val t = Random.shuffle(types).head
val ot: Ordering[t.Type] = t.evidence
```

What about when you need lots of evidence?

— We usually need both Gen and Cogen
— We also needed some Ordering instances
— Sometimes we'll also need ClassTag
— Many tests will want Eq or Equiv

There are (at least) two ways to do this.

1. build a case class

```
case class Testing[A](gen: Gen[A], cogen: Cogen[A])

object Testing {
  implicit def testing[A: Cogen: Gen]: Testing[A] =
    Testing(implicitly, implicitly)
}

// if we have implicit Gen, Cogen, this works
val t: TypeWith[Testing] = TypeWith[Int, Testing]
val g: Gen[t.Type] = t.evidence.gen
val c: Cogen[t.Type] = t.evidence.cogen
```

2. do something fancy

```
import elist._

// elist stands for evidence list.
// basically a version of HList with implicit definitions.
// elists are implicitly available if all member values are.

type Testing[A] = Gen[A] :/: Cogen[A] :/: Ordering[A] :/: ENil

// build as before
val t: TypeWith[Testing] = TypeWith[Int, Testing]

// use a destructuring match to extract evidence
val g :/: c :/: o :/: ENil = t.evidence
```

Either way, we can now generate types!

```
def genType: Gen[TypeWith[Testing]] = ...
```

Either way, we can now generate types!

```
def genType: Gen[TypeWith[Testing]] =
  Gen.sample(
    TypeWith[Unit, Testing],
    TypeWith[Unit, Boolean],
    TypeWith[Unit, Byte])
```

Either way, we can now generate types!

```
def genType: Gen[TypeWith[Testing]] =
  Gen.sample(
    TypeWith[Unit, Testing],
    TypeWith[Boolean, Testing],
    TypeWith[Byte, Testing],
    TypeWith[Char, Testing],
    TypeWith[Short, Testing],
    TypeWith[Int, Testing])
```

Unlike before this is extensible.

Let's see it in action!

```
def genMap(t1: TypeWith[Testing]): Gen[TypedPipe[t1.Type]] =
  genType.flatMap { t0 =>
    genPipe(t0).flatMap { pipe =>

      val gf: Gen[t0.Type => t1.Type] =
        Gen.function(t0.evidence.cogen, t1.evidence.gen)

      gf.map { f => pipe.map(f) }
    }
  }
```

Our type parameters are now path-dependent types.

Let's put it together!

```
def genPipe(t0: TypeWith[Testing]): Gen[TypedPipe[t0.Type]] =
  Gen.defer(Gen.oneOf(
    genFrom(t0), genFilter(t0),
    genMap(t0), genFlatMap(t0), ...))
```

So far so good. But what about `Ordering`?

We can have multiple type generators:

```
case class OrderedTesting[A](
    gen: Gen[A], cogen: Cogen[A], ordering: Ordering[A])

object OrderedTesting { ... }

def genOrderedType: Gen[TypeWith[OrderedTesting]] =
  Gen.sample(
    TypeWith[Int, OrderedTesting],
    TypeWith[String, OrderedTesting],
    ...)
```

Once you have several "testing" types elists are nice.

```
// we can generate a distinct operation when given an ordered type
def genDistinct(
    t0: TypeWith[OrderedTesting]
): Gen[TypedPipe[t0.Type]] =
  for {
    pipe <- genPipe(t0)
  } yield pipe.distinct(t0.evidence.ordering)

// or alternately, we can just do it ourselves...
def genFromDistinct(
    t1: TypeWith[Testing]
): Gen[TypedPipe[t1.Type]] =
  for {
    t0 <- genOrderedType
    pipe0 <- genDistinct(t0)
    f <- Gen.function(t0.evidence.cogen, t1.evidence.gen)
  } yield pipe0.map(f)
```

This approach is composable:

```
def genGrouped(
    k: TypeWith[OrderedTesting],
    v: TypedWith[Testing]
): Grouped[k.Type, v.Type] = {

  type KV = (k.Type, v.Type)

  val gkv: Gen[KV] = Gen.zip(k.gen, v.gen)
  val ckv: Cogen[KV] = Cogen.zip(k.cogen, v.cogen)
  val testing: Testing[KV] = Testing(gkv, ckv)
  val tkv: TypeWith[Testing] = TypeWith(testing)

  genPipe(tkv).map { pipe => pipe.group }
}
```

This approach is composable:

```
def genJoin(
    k: TypeWith[OrderedTesting],
    v1: TypedWith[Testing],
    v2: TypedWith[Testing]
): Grouped[k.Type, (v1.Type, v2.Type)] =
  for {
    g1 <- genGrouped(k, v1)
    g2 <- genGrouped(k, v1)
  } yield g1 join g2
}
```

This approach is composable:

```
def genFromJoin(
    t: TypedWith[Testing],
): TypedPipe[t.Type] =
  for {
    k <- genOrderedType
    v1 <- genType
    v2 <- genType
    gr <- genJoin(k, v1, v2)
    cv12 = Cogen.zip(v1.evidence.cogen, v2.evidence.cogen)
    ckv12 = Cogen.zip(k.evidence.cogen, cv12)
    f <- Gen.function(ckv12, t.evidence.dgen)
  } yield gr.toTypedPipe.map(f)
}
```

What's good about this approach?

— Removes hardcoded types.
— Often have a destination in mind but not a route.
— We define explicit type universes to test with.
— Type-generator could be pluggable.
— Can control distribution of types (not shown)
— Parametric functions should work for any type.
— Closely models GADT definitions in generators.

What is less good?

— Doesn't play well with implicit-driven generation.
— Requires a fair amount of project-specific setup.
— It can be hard to talk about the types sometimes.
— Sometimes for-comprehensions unexpectedly break.

This strategy also works well for law-testing:

```
// Ev1[T, F] => T[A] => T[F[A]]
trait Ev1[T[_], F[_]] {
  def apply[A](implicit fa: T[A]): T[F[A]]
}

type Testable1[F[_]] =
  Ev1[Gen, F] :/: Ev1[Cogen, F] :/: Ev1[Eq, F] :/: ENil

// parallels TypeWith[Testable]
type Kind = Type1With[Testable1]
```

```scala
def identity(k: Kind)(fk: Functor[k.Type1]): Gen[LawCase] =
  genType.flatMap { a =>
    val genk :/: _ :/: eqk :/: ENil = k.evidence
    val gena :/: _ :/: eqa :/: ENil = a.evidence
    genk(gena).map { (ka: k.Type1[a.Type]) =>
      val lhs: k.Type1[a.Type] = fk.map(ka)(Predef.identity)
      val rhs: k.Type1[a.Type] = ka
      LawCase(a, lhs, rhs)(eqk(eqa))
    }
  }
```

```
def composition(k: Kind)(fk: Functor[k.Type1]): Gen[LawCase] =
  genType.flatMap { a =>
    genType.flatMap { b =>
      genType.flatMap { c =>
        val genk :/: _       :/: eqk :/: ENil = k.evidence
        val gena :/: cogena :/: _   :/: ENil = a.evidence
        val genb :/: cogenb :/: _   :/: ENil = b.evidence
        val genc :/: _       :/: eqc :/: ENil = c.evidence
        Gen.function(cogena, genb).flatMap { f =>
          Gen.function(cogenb, genc).flatMap { g =>
            genk(gena).map { ka =>
              val lhs = fk.map(fk.map(ka)(f))(g)
              val rhs = fk.map(ka)(f andThen g)
              LawCase((a, b, c), lhs, rhs)(eqk(eqc))
            }}}}}}
```

law-checking recap

That was intense but:

— We tested the functor laws across all types.
— We control the universe of types
— Could combine with a list of instances to test (!)
— With more polish we could remove more boilerplate

(Incidentally, kind polymorphism might be helpful here.)

Using genType makes any polymorphic code easier to test.

But it is not a panacea.

Next, we'll see another technique to improve our generators.

tracking state

genType solves the problem of testing polymorphic code.

Is that the only issue we faced?

No!

We also faced issues around data distribution:

— generating trees instead of DAGs
— empty joins/intersections
— lack of heavy keys (or overly heavy keys)
— too much (or too little) data

generating trees vs DAGs

Remember our previous graph:

```
source -> flatMap -> group -> join -> map -> sink
        \                           /
         -> filter -> group
```

An interesting feature is its diamond subgraph.

Diamond subgraphs represent shared work:

— Simple recursive algorithms will "tree-ify" these
— Handling diamonds correctly is critical.
— Also represent an opportunity for contention.
— Without diamonds planning, etc. is easier.

It's important that we generate diamond subgraphs.

Will our previous generators produce these?

Not as written:

```
source0 -> flatMap -> group -> join -> map -> sink
                                /
source1 -> filter -> group -
```

— Our generators always produce "fresh" leaves

— Instead we want to be able to "reuse" subgraphs

Will our previous generators produce these?

Not as written:

```
source0 -> flatMap -> group -> join -> map -> sink
                                /
source1 -> filter -> group -
```

— Our generators always produce "fresh" leaves

— Instead we want to be able to "reuse" subgraphs

The solution? add state!

```scala
case class Cache(...) // stores generated subgraphs

import cats.data.StateT

type StGen[A] = StateT[Gen, Cache, A]

object StGen { ... } // mirrors Gen companion

// we can choose how often we want to reuse graphs
// versus generating new graphs.
val cache: StGen[Cache] =
  StateT((c: Cache) => Gen.const((c, c)))
```

```
type TT = TypeWith[Testable]

// cache operations
def get(c0: Cache, t: TT): List[TypedPipe[t.Type]] = ...
def add(c0: Cache, t: TT)(p: TypedPipe[t.Type]): Cache = ...

def tryReuse(t: TT): StGen[List[t.Type]] =
  StGen.oneIn(2).flatMap {
    case true => cache.map(c => get(c, t))
    case false => StGen.const(Nil)
  }
```

```scala
def genOrReuse(t: TT): StGen[TypedPipe[t.Type]] =
  tryReuse(t).flatMap {
    case Nil =>
      genPipe(t).flatMap { pipe =>
        StateT { (c0: Cache) =>
          val c1 = add(c0, t)(pipe)
          Gen.const((c1, pipe))
        }
      }
    case ps =>
      Gen.sample(ps)
  }
```

Similar tricks work other cases:

— Store previously generated values in Cache
— Bias towards (or against) previous values
— Index into cache by type or by generator.
— Check out type-aligned structures (e.g. HMap) [4]

[4] http://github.com/stripe/dagon


review

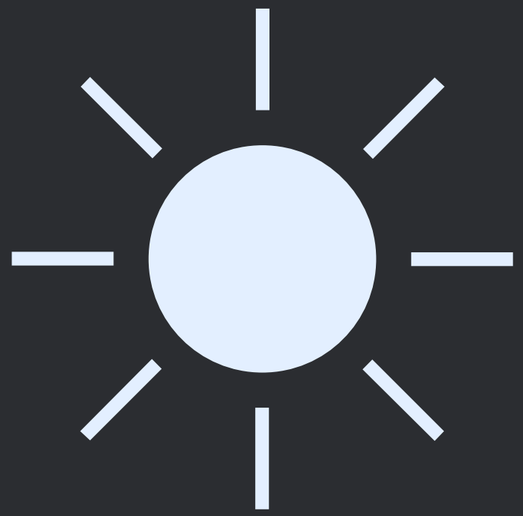
What did we do?

— Mostly we made our generators smarter.
— Comingled generation of different types
— Manually added state/distribution info.

In some sense, this is generator-based testing.

You can see a similar trend in libraries like Hedgehog, Hypothesis, etc. towards richer generators.

These strategies are new to me, but likely show up in other places.



What challenges remain?

— A fair amount of machinery to set this up.
— Type universes each have boilerplate
— Not the "standard path" for property-based testing
— Kinds require type class combinators
— How would you do this in Haskell?
— Still have challenges around recursive definitions
— Too many for one talk

I had hoped to tackle more of these in my talk.

There are major opportunities for new library designs in Scala (and elsewhere).

the end

thanks for listening!