Proving the Cats Library

Olivier Blanvillain

June 9, 2017

Abstract

The goal of this project is to prove laws of Cats type classes with Stainless. In long term, this work aims at becoming a pull request against the Cats repository to incorporate proofs into the project's testing infrastructure. The main contributions of this work is to provide a large-scale usability test that could be used to raise the awareness of the Scala community about the Stainless.

1 Context

Stainless is a verification framework for Pure Scala, a subset of the Scala programming languages. From a high-level point of view, Stainless is a function taking Scala files as inputs and producing verification and termination analysis. Under the hood, Stainless uses a Scala compiler (scalac or Dotty) to parse and type check the input program. From there, it generates mathematical propositions that are into an SMT solver such as Z3 or SMT-LIB.

Cats is a Scala library for functional programming. At its core, Cats defines a set of commonly used type classes, similar to the ones provided by the Haskell standard library.

Cats also contains some infrastructure around these type classes, such as instance for commonly used data types, laws associated with each type class and an elaborate testing infrastructure to assess that type class instances conform to the associated laws. Figure 1 shows a subset of the hierarchy of type classes implemented in Cats.

Let's have a look at an example with the `List` data type and the `Functor` type class. `List` is defined and compiled independently as part of the Scala standard library:

```
package scala.collection.immutable
```

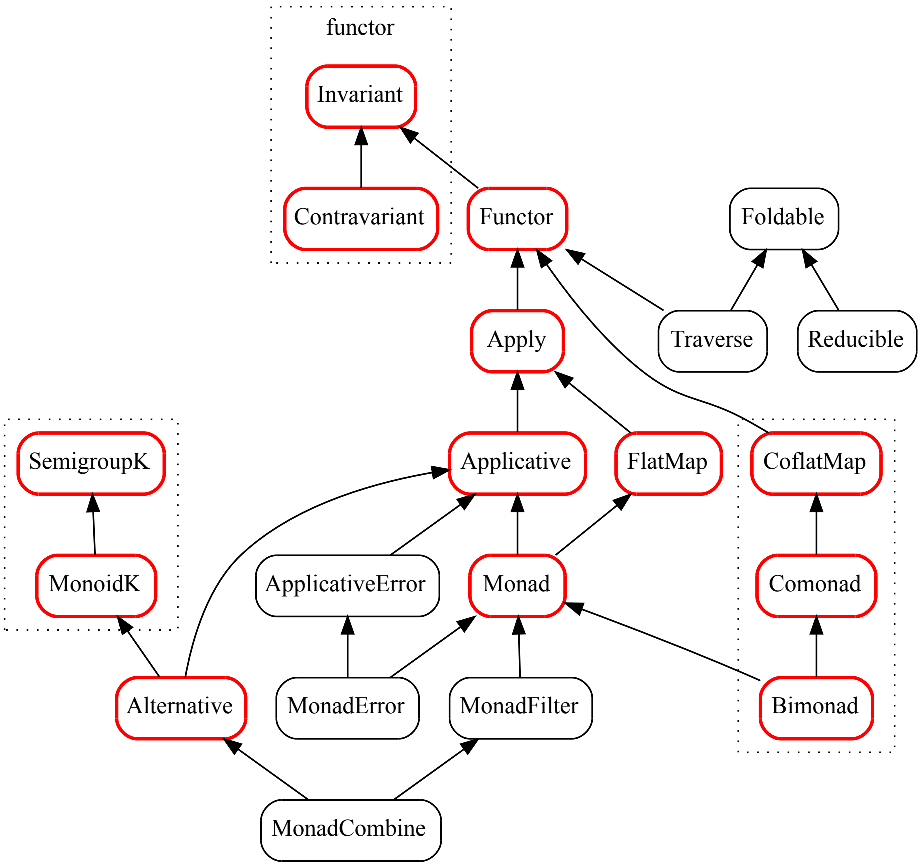

Figure 1: Hierarchy of Cats type classes. Highlighted nodes are covered in this project.

```
class List[A] {
  def map[B](f: A => B): List[B] = ???
}
```

The `Functor` type class capture the signature (and meaning) of the `map` method:

```
package cats

trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

In Scala, type classes (or ad-hoc polymorphism) is implemented using implicits. An instance of the `Functor` type class for `List` is an implicit value of type `Functor[List]`:

```
package cats.instances
```

```scala
implicit val listfunctor: Functor[List] = new Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa.map(f)
}
```

To be valid `Functor` instance, the above definition needs to fulfill a set of laws associated with the `Functor` type class. These laws are one of the key property that allow functional programmers to use equational reasoning on their programs. Type class laws constitute a contract associated with every type class instances. Cats defines laws in a way similar to the following:

```scala
package cats.laws

class FunctorLaws[F[_]](implicit F: Functor[F])  {

  def identity[A](fa: F[A]): Boolean =
    F.map(fa)(x => x) == fa

  def associative[A, B, C](fa: F[A], f: A => B, g: B => C): Boolean =
    F.map(F.map(fa)(f))(g), F.map(fa)(g compose f)
}
object FunctorLaws { def check[F[_]](i: Functor[F]) = ??? }
```

`FunctorLaws.check` is packaged and published for users of the Cats library, it provides a simple way to check the correctness of type classes instance using property based testing. The implement of `FunctorLaws.check` is analogous to the following:

```scala
def check[F[_]](i: Functor[F]): Unit = {
  val laws = new FunctorLaws(i)
  assert {
    (1 to 100).forall { _ =>
      type A = Int                       // Random!
      val fa: F[A] = Arbitrary[F[A]].next // Random!
      laws.law1(fa) && laws.law2(fa, ..., ...)
    }
  }
}
```

The actual implementation uses the ScalaCheck testing framework to randomly generate inputs for the laws.

2 Project goal

The goal of this project is to replace tests with proofs.

Table 1 shows the data structures and primitives covered in this project. The right-hand side of this table explicits the concrete data type that is used in the implementation. The equivalent data types defined in the standard library use programming constructs that are out of the subset of Scala supported by Stainless. Table 2 lists the data structures that were not covered in this project. Put together, Table 1 and Table 2 have most of the data types which have default type class instances implemented in Cats.

Covered	Concrete Data Type
`Int`	`stainless.lang.BigInt`
`Real`	`stainless.lang.Real`
`List`	`stainless.collection.List`
`Either[X, ?]`	`stainless.lang.Either`
`Option`	`stainless.lang.Option`
`Set`	`as Function1[?, Boolean]`
`()=> ?`	`scala.Function0`
`? => X`	`scala.Function1`
`X => ?`	`scala.Function1`

Table 1: Data structured & primitives (covered)

Not covered	Why not?
`java.lang.String`	Java construct
`java.util.UUID`	Java construct
`scala.util.Try`	Exceptions
`Future`	Exceptions
`Stream`	No pure implementation
`Bitset`	No pure implementation

Table 2: Data structured & primitives (not covered)

In term of type class coverage, the majority of type classes in Figure 1 were covered in this project (highlighted nodes).

3 Userland proofs

In this section, we show how laws we implement verify type class laws using the Stainless framework. Each code snippet presented here is carefully packaged-scoped to reflect that each file lies in a different project. This allows files to be compiled, packaged and published separately, which is fundamental is having a separation between application and testing/proving sources. Having a testing/proving framework to affect the binaries publish in production would obviously be unacceptable.

Laws are implemented in their own package as a set of traits that mirror type class hierarchy. The example below shows the interface for `MonoidKLaws`. The `MonoidK` type class extends the `SemigroupK` type class, and so does their laws. Note that this **`trait`** has two concrete methods, one for each law, and abstract methods for the proofs, to be filled by the user.

```
package cats.laws

trait MonoidKLaws[F[_]] extends SemigroupKLaws[F] { self: MonoidK[F] =>
  // Inherited: semigroupKAssociative & semigroupKAssociativeProof
  def monoidKLeftIdentity[A](a: F[A])  = combineK(empty, a) == a
  def monoidKRightIdentity[A](a: F[A]) = combineK(a, empty) == a
  def monoidKLeftIdentityProof[A](a: F[A]): Boolean
  def monoidKRightIdentityProof[A](a: F[A]): Boolean
}
```

`MonoidK` stands for Higher (K)inded Monoid, a Monoid for an entire family of type. `List` belongs to this type class with `++` and `Nil`:

```
package userland.runtime

trait ListInstance extends MonoidK[List] {
  def empty[A]: List[A] = Nil[A]()
  def combineK[A](f1: List[A], f2: List[A]): List[A] = f1 ++ f2
}
```

To prove that `ListInstance` respects the `MonoidKLaws`, users needs to mix the law interface and the type class instance together to write the proof in a scope with all these definitions available.

```
package userland.tests

class ListProofs extends ListInstance with MonoidKLaws[List] {
  // Stainless is able to derive this proof without any help!
  def monoidKLeftIdentityProof[A](a: List[A]): Boolean =
    monoidKLeftIdentity(a).because(trivial).holds

  // The proof for the right identity law uses structural induction:
  def monoidKRightIdentityProof[B](b: List[B]): Boolean = {
```

```
    listMonoidKRightIdentity(b) because {
      b match { case Nil()       => trivial
                case Cons(x, xs) => listMonoidKRightIdentityProof(xs)
    }}}.holds

  def semigroupKAssociativeProof[A](a: List[A], b: List[A], c: List[A]) = ...
}
```

On benefits of mixing laws and implementations together is that enables proof composition. For instance, the proof for the left identity Monad law can be written by reusing the right identity MonoidK law:

```
def listMonadLeftIdentityProof[A, B](a: A, f: A => List[B]): Boolean = {
  listMonadLeftIdentity(a, f) because {
    listMonoidKRightIdentityProof(f(a))
  }
}.holds
```

4 Meta proofs

We've now seen how users can write Stainless proof for type class laws. In this section, we discuss how the proving framework can be useful for library authors.

The graph in Figure 1 models an inheritance relationship between type classes. An `A` □`B` edge correspond to an "`A` is a `B`" relationship. For instance, every `MonoidK` is a `SemigroupK`. This inheritance relation directly translates to laws: every `MonoidK` obeys the laws for `SemigroupK`.

The depth this graph implies that type classes near the bottom of the graph are constrained by a large number of laws. For instance, the `Monad` type class inherits laws from `Invariant`, `Functor`, `Apply` and `Applicative`, to which is added another set of laws for the `Monad` itself. This potential explosion of laws is not a problem for property based testing: having more laws only implies slightly longer execution time for the tests, but might also help detecting additional bugs. However, from a mathematical view point, this abundance of law is unpleasant: every extra law implies an additional proofs per instance.

This observation raises an interesting theoretical question: for every type class in this hierarchy, what is the minimum set of laws that is sufficient to fully specify it's behavior? It is common knowledge that `Monad` type class is defined by "the three `Monad` laws". This means that, assuming these three `Monad` laws, it is possible to prove all the inherited laws.

These meta proofs, proofs on the law them-selfs, can be modeled in Stainless. For instance, we can show that the `monadRightIdentity` is sufficient to prove the `covariantFunctorIdentity` given the `Monad`'s definition of `map`.

The `monadRightIdentity` and `covariantFunctorIdentity` laws are defined as follows:

```
def covariantFunctorIdentity[A](fa: F[A]): Boolean =
  map(fa)(x => x) == fa

def monadRightIdentity[A](fa: F[A]): Boolean =
  flatMap(fa)(pure) == fa
```

The `map` method on a `Monad` is defined using `pure` and `flatMap`:

```
// As defined in Monad
def map[A, B](fa: F[A])(f: A => B): F[B] =
  flatMap(fa)(a => pure(f(a)))
```

Stainless is able to derive a proof of the `covariantFunctorIdentity` from the `monadRightIdentity`:

```
trait MetaProofs[F[_]] extends MonadLaws[F] with FunctorLaws[F] {
  self: Monad[F] =>

  def covariantFunctorIdentityProof[A, B, C](fa: F[A]) = {
    monadRightIdentity(fa) ==> covariantFunctorIdentity(fa)

    // Hand written version of this proof; illustrates the
    // reasoning derived by Stainless:

    //     map(fa)(x => x)                      lhs of covariantFunctorIdentity
    // <-> flatMap(fa)(a => pure((x => x)(a)))  By definition of map
    // <-> flatMap(fa)(a => pure(a))            Beta reduction on identity
    // <-> flatMap(fa)(pure)                    Syntactic rewrite
    // <-> fa                                   From monadRightIdentity
  }.holds
}
Raw
```

5 Conclusion

In conclusion, we can say that Stainless works very well with type classes! At the time of writing a few bugs prevent the implementation from being as nice as presented in this report, but with a few workarounds, it's possible to use Stainless to replace most of the tests implemented in Cats with formal proofs. The complete implementation of this project is available at on GitHub [1].

[1] https://github.com/OlivierBlanvillain/cats/tree/proofs