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. This function is implemented by composed of several other existing functions. It first uses a Scala compiler (scalac or Dotty) to parse and type check the initial input. From there, Stainless generates verification conditions (that is, mathematical propositions) which are then fed 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, similarly to the ones provided by the Haskell standard library. In addition to these type classes, Cats contains a lot of infrastructure around them, such as type class instance for commonly used data types, the associated laws that define each type class and an elaborate testing infrastructure to assess that instances respect the associated type class laws. Figure 1 shows a subset of the hierarchy of Cats type classes.

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

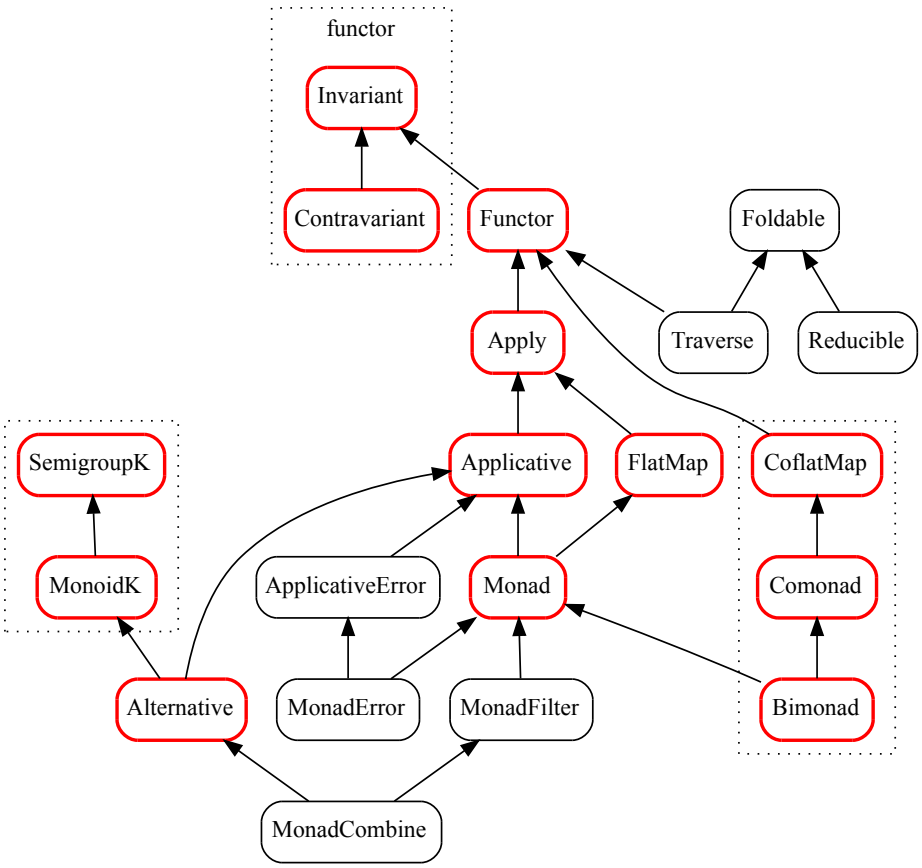

Figure 1: Hierarchy of Cats type classes. Highlighted nodes are covered in this project.

```
package scala.collection.immutable

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

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

```
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

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. These laws are essential to allow equational reasoning on functional programs. In a sense, type class laws constitute the contract associated with every type class instances. Whenever a data type implements a type class instance, it implicitly implies that this instance respects the associated laws. Cats defines `Functor` laws similarly to the follows:

```
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]) = ??? }
```

A method similar to the `FunctorLaws.check` sketched above is packaged and made available to 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:

```
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 makes heavy use of 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 structure and primitives that are covered in this project. The right-hand side of this table explicits the concrete data type that is used. The equivalent data types defined in the standard library (and used with cats instance) use constructs that are out of 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 contain most of the data types contain default type class instances implemented in the core of the Cats library.

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 defined in Cats were covered in this project. See the nodes in highlighted in Figure 1.

3 Userland proofs

In this section, we show how laws we implemented definitions and verification of type class laws using the Stainless framework. Each code snippet presented here is carefully packaged-scoped to reflect the fact that each file lies in a different package. This allows files to be compiled and packaged separately, allows users to maintain the important separation between application and testing/proving. Having a testing/proving to affect the binaries publish in production would be an unacceptable requirement.

Laws are implemented in a separate package, as a set of traits that mirror the hierarchy of type classes. This example shows the interface for the `MonoidKLaws`. `MonoidK` extends the `SemigroupK` type class, and so does their laws. Note that this **`trait`** has two implemented `Boolean` returning 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 this `ListInstance` respects the `MonoidKLaws`, users needs to mix the law interface and the type class instance together to write the proof in a context with definitions from both:

```
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 right identity require an explicit 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 it allows users to compose proofs. For instance, a proof for the left identity Monad law can be written by reusing the proof of 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 writing Stainless proof for type class laws looks like from a user perspective. In this section, we discuss how the proving framework could be useful for library authors.

The graph from Figure 1 models how type classes relate together with from an object-oriented perspective. In type class hierarchy implementation, edges correspond to an "is a" relationship: every `MonoidK` is a `SemigroupK`. This inheritance relation also reflects when looking at law: every `MonoidK` obeys the laws for `SemigroupK`.

The depth of graph in Figure 1 implies that type classes near the bottom of the graph are constrained by a large number of laws. For instance, the `Monad` type class will inherit 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 that might help to detect additional bugs. However, from a mathematical vue-point this abundance of law is not very appealing: more laws implies more proofs to write for every instance.

This point raises an interesting theoretical question: for any given type class, what is the minimum set of laws that is sufficient to fully specify the behavior type class? It is common knowledge that `Monad` type class is defined

by "the three `Monad` laws", but what does that mean formally? It means that assuming these three `Monad` laws, it's possible to prove all the inherited laws.

Such implication can be modeled in Stainless. For instance, we can show that the `monadRightIdentity` is sufficient to prove the `covariantFunctorIdentity` identity law given that `map` is non-overridden from its definition inside `Monad`.

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

// 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 the proof without any user input. A hand written proof is reported as a comment to illustrate the type of reasoning that Stainless is able to derive:

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

  def covariantFunctorIdentityProof[A, B, C](fa: F[A]) = {
    monadRightIdentity(fa) ==> covariantFunctorIdentity(fa)
    //     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 for type class laws! 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 code for this project is available at on GitHub [1].

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