Proving Algebraic Properties with Stainless

Romain Ruetschi

June 21th, 2017

Algebra

Algebra is the study of *algebraic structures*.

Algebraic structure

A carrier set with one or more operations defined on it that satisfies a list of axioms (*laws*).[1]

Examples

Abstract algebra
Eq, Ord, Poset, Monoid, Ring, Group, Semilattice, Field

Category Theory
Functor, Monad, Comonad, Profunctor, Adjunctions

Programming
Foldable, Traversable, Alternative

[1] https://en.wikipedia.org/wiki/Algebraic_structure

Example

Monoid

A set S with some binary operation $\oplus : S \times S \to S$ is a **monoid** if the following axioms are satisfied:

- **Associativity:** $\forall a, b, c \in S.(a \oplus b) \oplus c = a \oplus (b \oplus)$.
- **Identity:** $\exists e \in S.\forall a \in S.e \oplus a = a \oplus e = a$

Examples

- Integers under addition/multiplication ($oplus = +/\times$, $e = 0/1$)
- Lists ($\oplus =$ ++, $e = Nil$)
- Endomorphisms ($\oplus = \circ$, $e = \backslash x \to x$)
- And many others...

Why do we care?

Algebraic structures...

- enable equational reasoning at scale[2]
- give rise to powerful, composable abstractions[3]
- allow us to build (and verify!) programs the same way we do mathematics[4]

[2] https://haskellforall.com/2014/07/equational-reasoning-at-scale.html

[3] https://haskellforall.com/2014/04/scalable-program-architectures.html

[4] https://haskellforall.com/2012/08/the-category-design-pattern.html

Programs as proofs

Mathematics	Programming
Build small proofs that we can prove correct in isolation.	Build small components that we can verify in isolation.
Compose smaller proofs into larger proofs.	Compose smaller components into larger components.

Algebra and programming

How to take advantage of these structures to actually build programs?

Algebra and programming

How to take advantage of these structures to actually build programs?

Typeclasses!

Typeclasses

- Introduced by Wadler [5] in 1989 as principled way to implement overloading (*ad-hoc polymorphism*) in functional languages
- Very well suited for expressing algebraic structures found in datatypes

```haskell
class Monoid a where
  empty  :: a
  append :: a -> a -> a

instance Monoid [a] where
  empty  = []
  append = (++)
```

Typeclasses

What about the laws (*axioms*)?

Typeclasses

What about the laws (*axioms*)?

Three options:

Typeclasses

What about the laws (*axioms*)?

Three options:

- Check them manually using equational reasoning (actual proof, error-prone)

Typeclasses

What about the laws (*axioms*)?

Three options:

- Check them manually using equational reasoning (actual proof, error-prone)
- Check them automatically using property-based testing (not a real proof, less error-prone)

Typeclasses

What about the laws (*axioms*)?

Three options:

- Check them manually using equational reasoning (actual proof, error-prone)
- Check them automatically using property-based testing (not a real proof, less error-prone)
- **Prove them statically with Stainless** (best of both worlds)

Typeclass in Scala

Unlike Haskell [2], Scala does not have first-class support for typeclasses. However, it is possible to represent them using Scala's powerful implicit resolution mechanism [3].

- Typeclasses are represented as abstract classes.
- Operations are represented by abstract methods of these classes.
- Instances are represented by implicit values of these types.

Typeclasses in Pure Scala I

```scala
abstract class Monoid[A] {
  def empty: A
  def append(a: A, b: A): A

  @law def law_identity = forall { (x: A) =>
    append(empty, x) == x && append(x, empty) == x
  }

  @law def law_assoc = forall { (x: A, y: A, z: A) =>
    append(append(x, y), z) == append(x, append(y, z))
  }
}

implicit def bigIntMonoid = new Monoid[BigInt] {
  def empty: BigInt = 0
  def append(a: BigInt, b: BigInt): BigInt = a + b
}
```

Typeclasses in Pure Scala I

```
+-| Verification Summary |----------------------------+
|                                                     |
| bigIntMonoid   adt invariant   valid   nativez3   1.001 |
+-----------------------------------------------------+
```

Typeclasses in Pure Scala II

```scala
sealed trait Nat {
  def +(m: Nat): Nat = /* ... */
}
case class Zero()       extends Nat
case class Succ(n: Nat) extends Nat

implicit def natMonoid = new Monoid[Nat] {
  def empty: Nat = Zero()
  def append(a: Nat, b: Nat): Nat = a + b
}
```

Typeclasses in Pure Scala III

```scala
@induct def lemma_identity(n: Nat) = {
  Zero() + n == n && n + Zero() == n
} holds

@induct def lemma_assoc(n: Nat, m: Nat, l: Nat) = {
  (n + m) + l == n + (m + l)
} holds

implicit def natMonoid = new Monoid[Nat] {
  /* ... */
  override def law_assoc =
    super.law_assoc because forall { (n: Nat) =>
      lemma_assoc(n)
    }
}
```

Encoding I

Typeclasses are encoded as a case class with the appropriate invariant.

```
case class Monoid[A](empty: A, append: (A, A) => A) {
  require {
    forall { (x: A) =>
      append(empty, x) == x && append(x, empty) == x
    }
    &&
    forall { (x: A, y: A, z: A) =>
      append(append(x, y), z) == append(x, append(y, z))
    }
  }
}
```

Encoding II

Instances are converted to instances of the case class + relevant assertions.

```
implicit def natMonoid = {
  require {
      forall { (n: Nat) => lemma_identity(n) } &&
      forall { (n: Nat) => lemma_assoc(n) }
  }

  Monoid[Nat](Zero(), (n: Nat, m: Nat) => n + m)
}
```

Typeclass inheritance

```scala
abstract class Semigroup[A] {
  def append(a: A, b: A): A
}

abstract class Monoid[A](implicit semigroup: Semigroup[A]) {
  def empty: A

  def append(a: A, b: A): A = semigroup.append(a, b)
}
```

Typeclass inheritance

```scala
case class Semigroup[A](append: (A, A) => A)

case class Monoid[A](semigroup: Semigroup[A], empty: A) {
  def append(a: A, b: A): A = semigroup.append(a, b)
}
```

Coherence

```scala
object Set {
  def apply[A](xs: A*)(implicit ord: Ord[A]): Set[A]
}

// module A
implicit val lessThanOrd: Ord[Int]
val foo = Set(1, 2, 3)

// module B
implicit val greaterThanOrd: Ord[Int]
val bar = Set(4, 5, 6)

// module C
val baz = foo union bar // ???
```

Coherence

This is a problem known as *coherence* [4]. Haskell partially enforces it by triggering an error whenever it encounters more than one instance of a typeclass at link time.

Scala somewhat enforces it too by yielding an *ambiguous implicits* error when two instances of equal priority are in scope, but this does not cover all cases, as we have seen previously.

Coherence

Because Stainless operates under a closed-world assumption, we can actually enforce coherence by statically disallowing the creation of multiple instances of a typeclass for the same type.

Coherence

Because Stainless operates under a closed-world assumption, we can actually enforce coherence by statically disallowing the creation of multiple instances of a typeclass for the same type.

This allows to reason about typeclass instances equality:

$$\forall \text{ typeclass } TC.\ \forall \text{ type } A.\ \forall a, b : \texttt{TC[A]}.\ a = b$$

Partially enforced by rewriting equality test between instances of the same type to `true`.

Coherence

Because Stainless operates under a closed-world assumption, we can actually enforce coherence by statically disallowing the creation of multiple instances of a typeclass for the same type.

This allows to reason about typeclass instances equality:

$$\forall \text{ typeclass } TC.\ \forall \text{ type } A.\ \forall a, b : \texttt{TC[A]}.\ a = b$$

Partially enforced by rewriting equality test between instances of the same type to `true`.

But need more work to figure out how to properly enforce it for composite datatypes and collections which contains such instances.

Limitations

- Currently unable to express some of the more interesting typeclasses such as Functor or Monad because of Stainless' lack of proper support for higher-kinded types.
- Verification of obviously broken instances most often timeouts instead of yielding `invalid`.
- Some typeclasses have many or complex associated laws, and verification thus timeouts as well.
- Have not been able to prove algebraic properties about inductive datatypes neither in plain Pure Scala or with our extension. Might be related to ADT invariants + lambdas.

Results

We have tested our implementation over a corpus of typeclasses[5] and a few of their possible instances.

Instance	Result	ADT invariant (s)
`Monoid[Any]`	valid	0.222
`Monoid[First]`	valid	1.107
`Monoid[Sum]`	valid	0.214
`Monoid[BigInt]` (additive)	valid	2.746
`Semigroup[Int]` (additive)	valid	1.044
`Newtype[Sum, BigInt]`	valid	0.133
`Eq[BigInt]` / `Ord[BigInt]` (partial)	valid	1.536 / 1.725

Timeout: `EqOrd[BigInt]`, `Monoid[Endo]`, `Monoid[List]`, `Monoid[Nat]`, `Monoid[Option]`, `Semigroup[NonEmpty]`, `Uniplate[Expr]`, `Semiring[BigInt]`, `Semiring[Boolean]`

[5] github.com/romac/LARA-MscSemesterProject/tree/master/Testcases

References

[1] Claessen, K. and Hughes, J. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), 268–279.

[2] Hall, C.V. et al. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (Mar. 1996), 109–138.

[3] Oliveira, B.C. et al. 2010. Type Classes As Objects and Implicits. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), 341–360.

[4] Peyton Jones, S. et al. 1997. Type classes: an exploration of the design space. *Haskell workshop* (Amsterdam, january 1997).

[5] Wadler, P. and Blott, S. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), 60–76.

Map-Reduce

```scala
trait Monoid[A] {
  def empty: A
  def append(a: A, b: A): A

  def concat(xs: List[A]): A = {
    xs.foldRight(empty)(append)
  }
}
```

Map-Reduce

```scala
case class Sum(value: BigInt)

implicit def sumMonoid = new Monoid[Sum] {
  def empty = Sum(0)
  def append(a: Sum, b: Sum): Sum = Sum(a.value + b.value)
}

def foldMap[A, M : Monoid](f: A => M)(xs: List[A]): M = {
  Monoid[M].concat(xs.map(f))
}

val sum = foldMap(Sum(_))(List(1, 2, 3, 4)).value // 10
```

Map-Reduce

```scala
def mapReduce[A, M : Monoid](f: A => M)(xs: List[A]): M = {
  Monoid[M].concat(xs.par.map(f))
}

case class Document(/* ... */)
case class WordCount(value: Map[String, Int]) extends AnyVal

implicit val wcm = new Monoid[WordCount] { /* ... */ }

def countWords(doc: Document): WordCount = /* ... */

val docs: List[Document] = /* ... */

val wordsCounts: Map[String, Int] =
  mapReduce(countWords)(docs).value
```