Proving Algebraic Properties with Stainless

Romain Ruetschi

June 9th, 2017

Contents

1 Introduction

Stainless aims to help developers build verified Scala software.[1] While it does not support the full Scala language yet, Stainless understands a substantial, purely functional subset of it: *Pure Scala*.[2] This subset, while already very expressive and powerful, still lacks some features commonly used in the Scala community. Of those, typeclasses are one of the most notable ones. The aim of this project was thus to enrich Pure Scala with typeclasses and modify Stainless to properly handle them.

2 Typeclasses

2.1 Definition

Typeclasses were introduced by Wadler [6] as an extension to the Hindley/Milner type system to implement a certain kind of overloading, known as *ad-hoc* polymorphism.

A typeclass is identified by its name, and is associated with a set of (usually polymorphic) functions signatures, its *methods*. It can then be *instanciated* at various types, given that the user is able to provide a concrete implementation for each method. A user can then apply these methods to any type for which there is a corresponding instance, which essentially corresponds to *overloading*.

In Haskell notation, a typeclass `Num`, which allows to overload the arithmetic operations `+` and `*`, can be defined as follows:

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
```

An instance for the `Int` and `Float` types can then be provided in the following way:

```
-- Assuming these functions are provided by the standard library
addInt, mulInt     :: Int -> Int -> Int
addFloat, mulFloat :: Float -> Float -> Float

instance Num Int where
  (+) = addInt
  (*) = mulInt

instance Num Float where
  (+) = addFloat
  (*) = mulFloat
```

allowing the user to call such functions on either integers or floating-point values.

```
twice :: Int -> Int
twice x = x + x
```

[1] http://leon.epfl.ch/doc/intro.html
[2] http://leon.epfl.ch/doc/purescala.html

```
square :: Float -> Float
square x = x * x
```

A more general version of these two user-defined functions can be written, by abstracting over the specific type they are applied to, provided it is an instance of the `Num` typeclass.

```
twice :: Num a => a -> a
twice x = x + x

square :: Num a => a -> a
square x = x * x

a :: Float
a = twice (square 1.337f)

b :: Int
b = square (twice 42)
```

2.2 Laws

Let's now imagine that some savy programmer comes across the following piece of code during a refactoring:

```
compute :: Num a => a -> a -> a -> a
compute a b c = c * b + b * a
```

While this is a perfectly good piece of code, she rightly realises that it could expressed a more concise manner, namely:

```
compute :: Num a => a -> a -> a -> a
compute a b c = b * (a + c)
```

However, for this refactoring to preserve the existing semantics of the `compute` function, some assumptions are needed. In this case, these assumptions correspond to the laws of arithmetic. That is, any type which is a instance of the `Num` class must provide an implementation of `+` and `*` which adheres to the following laws:

- Associativity: $(a + b) + c = a + (b + c)$ and $(a * b) * c = a * (b * c)$
- Commutativity: $a + b = b + a$ and $a * b = b * a$
- Distributivity: $a * (b + c) = a * b + a * c$

Should these properties not hold, the above refactoring might end up changing the meaning of the program, and potentially introduce a bug or break an existing unit test.

This is why typeclasses are often informally associated with a set of laws which programmers must make sure that their implementation adheres to. These laws often correspond to the very algebraic laws which govern how an algebraic structure can be used.

Most mainstream languages, amongst which Haskell and Scala, do not provide a way to ensure the validity of a typeclass instance at compile time. Rather, programmers must delay this task to runtime, often by the means of property-based testing [2]. That is, the have their tests generate various runtime values and feed those to functions which check that the aforementioned laws hold for these values. While this technique yields fairly good results in general, it is neither foolproof, nor does it provide any type of mathematical guarantee. We would therefore like to

discharge these obligations to Stainless, in order to let programmers semi-automatically prove the correctness of their instances, an endeavour we describe further on in this document.

2.3 Associated methods

On top of the abstract operations we have seen above, a typeclass can also introduces concrete methods which do not need to (but can) be (re-)defined by the programmer at instance declaration time. One could e.g. add a `square` method to the `Num` class we introduced above:

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a

  square :: a -> a
  square x = x * x
```

While this method could be defined as a standalone function, like we did above, having it be a part of the class allows users to override it with e.g. a more efficient implementation specific to the datatype they are instantiating the class for.

2.4 Typeclass inheritance

Much like regular object-oriented classes, typeclasses can inherit from each other. Let's take for example the `Ord` typeclass, which describes totally ordered datatypes. This class is defined as follows:

```
class Eq a => Ord a where
  (<=) :: a -> a -> Boolean

  (<) :: a -> a -> Boolean
  x < y = x != y && x <= y
```

where the `Eq` class itself is defined as:

```
class Eq a where
  (==) :: a -> a -> Boolean

  (!=) :: a -> a -> Boolean
  x != y = not (x == y)
```

Looking at the implementation of `<`, we see that it uses the `!=` method provided by the `Eq` class. Hence why, the class declaration is provided with a constraint on the type `a`, namely that it must be a instance of the `Eq` class as well. This can also be read as: if `a` is an instance of `Ord`, then it also is a instance of `Eq`.

2.5 Typeclasses in Haskell

In Haskell, typeclasses are desugared during compilation into a simpler form which fits in the standard Hindley/Milner type system [3].

```haskell
class Eq a => Ord a where
  (<=) :: a -> a -> Boolean

  (<) :: a -> a -> Boolean
  x < y = x != y && x <= y

instance Ord Int where
  (<=) = lteInt

gt :: Ord a => a -> a -> Boolean
gt x y = not (x <= y)

main = print (gt 1 2)
```

becomes

```haskell
data OrdD a
  = OrdD
  { eqD  :: EqD a
  , (<=) :: a -> a -> Boolean
  , (<)  :: a -> a -> Boolean
  }

mkOrdD :: EqD a -> (a -> a -> Boolean) -> Maybe (a -> a -> Boolean) -> OrdD a
mkOrdD eqD lte Nothing   = OrdD { eqD = eqD, (<=) = lte, (<) = \x y -> eqD.(!=) x y && lte x y }
mkOrdD eqD lte (Just le) = OrdD { eqD = eqD, (<=) = lte, (<) = le }

OrdD_Int :: OrdD Int
OrdD_Int = mkOrdD EqD_Int lteInt Nothing

gt :: OrdD a -> a -> a -> Boolean
gt ordD x y = not (ordD.(<=) x y)

main :: IO ()
main = print (gt OrdD_Int 100 2) -- True
```

We see here that classes are represented as a dictionary holding the superclasses, if any, as well as the methods' implementations. Instances then just becomes values of this type, and are passed explicitly to functions which require them.

2.6 Typeclasses in Scala

Unlike Haskell, Scala does not provide first-class support for typeclasses. Fortunately, its powerful implicit resolution mechanism for implicit parameters allow an encoding of these in a way which is very reminiscent of the desugared Haskell version [4]. The example above would be written in Scala as:

```scala
abstract class Eq[A] {
  def eq(x: A, y: A): Boolean
  def neq(x: A, y: A): Boolean = !eq(x, y)
}
```

```
abstract class Ord[A] extends Eq[A] {
  def lte(x: A, y: A): Boolean
  def lt(x: A, y: A): Boolean = lte(x, y) && neq(x, y)
}

implicit object ordInt extends Ord[Int] {
  override def eq(x: A, y: A): Boolean = x == y
  override def lte(x: Int, y: Int): Boolean = x <= y
}

def gt[A](x: A, y: A)(implicit O: Ord[A]): Boolean = !O.lte(x, y)

def main(): Unit = println(gt(100, 2)) // true
```

In Scala, typeclasses are represented as abstract classes, typeclass inheritance as regular class inheritance, instances as implicit objects. Functions which require a type to be an instance of some typeclass are provided with an implicit parameter of the corresponding type.

2.7 Ambiguity errors

One issue with the encoding above based on inheritance is that a programmer might encounter *ambiguous implicits errors* when . Such an error arise when the Scala compiler sees to equally valid implicit values in scope and thus cannot decide which one to use, as in this example, adapted from the Typelevel blog.[3]

```
trait Functor[F[_]] { /* ... */ }
trait Applicative[F[_]] extends Functor[F] { /* ... */ }
trait Monad[F[_]] extends Applicative[F] { /* ... */ }
trait Traverse[F[_]] extends Functor[F] { /* ... */ }

implicit object monadInt extends Monad[Int] { /* ... */ }
implicit object traverseInt extends Traverse[Int] { /* ... */ }

// The fact we don't actually use `Functor` here is irrelevant.
def bar[F[_]: Applicative: Functor]: F[Int] = Applicative[F].pure(10)

def callBar[F[_]](implicit M: Monad[F], T: Traverse[F]): F[Int] = bar[F]
// <console>:19: error: ambiguous implicit values:
//  both value T of type cats.Traverse[F]
//  and value M of type cats.Monad[F]
//  match expected type cats.Functor[F]
```

Here, both `M` and `T` satisfy the requirement for a `Functor[F]` instance because `Monad[F]` and `Traverse[F]` are both subtypes of `Functor[F]`. Although both instances should be perfectly valid,[4] the typesystem has no way of knowing that and can do nothing but bail out. In order to alleviate the problem, the Scala community has settled on a different encoding,[5] which is

[3] *Subtype type classes don't work* http://typelevel.org/blog/2016/09/30/subtype-typeclasses.html
[4] But they might in general not be, an issue which we go over in the next section.
[5] *Scalaz: Subtyping-free encoding for typeclasses* https://github.com/scalaz/scalaz/issues/1084

much closer to the way typeclasses are implemented in Haskell. Rather than having typeclasses inherit from their superclasses, they instead hold an instance of the superclass, much like the desugared dictionaries we have seen previously. Here is how a part of the hierarchy above is expressed with this encoding:[6]

```
trait Applicative[F[_]] {
  def functor: Functor[F]

  def pure[A](a: A): F[A]
  def map2[A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]
}

trait Traverse[F[_]] { def functor: Functor[F] }

trait FunctorConversions1 {
  implicit def applicativeIsFunctor[F[_]: Applicative]: Functor[F] =
    implicitly[Applicative[F]].functor
}

trait FunctorConversions0 extends FunctorConversions1 {
  implicit def traverseIsFunctor[F[_]: Traverse]: Functor[F] =
    implicitly[Traverse[F]].functor
}

object Prelude extends FunctorConversions0
```

This encoding makes use of implicit conversions and Scala's facility for prioritizing them to work around the ambiguity issues posed by subtyping. That is, if both an `Applicative` and `Traverse` instances are in scope, the Scala compiler will extract the `Functor` instance from the latter.

2.8 Coherence

Let's now look at another issue that might arise when working with typeclasses in Scala.

```
abstract class Monoid[A] {
    def empty: A
    def combine(x: A, y: A): A
}

implicit object intAddMonoid extends Monoid[Int] {
    override def empty: Int = 0
    override def combine(x: Int, y: Int): Int = x + y
}

implicit object intProdMonoid extends Monoid[Int] {
    override def empty: Int = 1
    override def combine(x: Int, y: Int): Int = x * y
}
```

[6] *Subtype type classes don't work* http://typelevel.org/blog/2016/09/30/subtype-typeclasses.html

```
def fold[A](list: List[A])(implicit M: Monoid[A]): A = {
    list.foldRight(M.empty)(M.combine)
}

val list: List[Int] = List(2, 3)
val res: Int = fold(list) // ?
```

Once again, the Scala compiler bails out with an *ambiguous implicits* error but for good reasons this time. Indeed, depending on which instance of `Monoid[Int]` it picks, `res` can either be equal to 5 or 6. This issue arise because the two instances above are *overlapping*, which has the effect of making the type system *incoherent* [5]. For a type system to be coherent, "every valid typing derivation for a program [must lead] to a resulting program that has the same dynamic semantics", which is clearly not the case here. While in this specific example, the compiler will rightfully reject the program, this might always be possible as one could import a different instance in scope in two different modules, and then a third module might assume that these modules actually make use of the same instance, silently breaking the program. Imagine trying to merge two `Sets` that have been created with two different `Ord` instances in scope.

Haskell partially solves this problem by enforcing that instances defined in the same module do not overlap, that is to say that the compiler would reject the program above. We deem Haskell's approach only partial as it allows the programmer to define two or more overlapping instances, provided that they are not defined in the same module. A program is then only rejected when the programmer makes imports such overlapping instances in scope and attempts to make use of them. This decision stems from the will to allow linking together two different libraries which both define a class instance for the same type. If one were to operate under a closed-world assumption, one could go further and disallow overlapping instances altogether which, as we will see, is the approach we have chosen in Stainless.

While current versions of Scala provides no such mechanism to enforce coherence, its version 3.0 will at least improve over the existing situation by letting the programmer declare a class as being coherent.[7]

With a coherent type system, one might still want to provide both an additive and multiplicative `Monoid` instance for integers. To this end, one can wrap values of type `Int` with two different wrapper (value-)classes for which we will define the respective instances.

```
case class Sum(value: Int)     extends AnyVal
case class Product(value: Int) extends AnyVal

implicit object intSumMonoid extends Monoid[Sum] {
    override def empty: Sum = Sum(0)
    override def combine(x: Int, y: Int): Sum = Sum(x.value + y.value)
}

implicit object intProductMonoid extends Monoid[Product] {
    override def empty: Product = Product(1)
    override def combine(x: Int, y: Int): Product = Product(x.value * y.value)
}
```

[7] *Allow Typeclasses to Declare Themselves Coherent* https://github.com/lampepfl/dotty/issues/2047

```
def fold[A](list: List[A])(implicit M: Monoid[A]): A = {
    list.foldRight(M.empty)(M.combine)
}

def foldMap[A, B](list: List[A])(f: A => B)(implicit M: Monoid[B]): B = {
    fold(list.map(f))
}
```

It then becomes possible to unambiguously pick which instance to use depending on the semantics we want.

```
val list: List[Int] = List(2, 3)

val sum: Int     = foldMap(list)(Sum(_)).value     // 5
val product: Int = foldMap(list)(Product(_)).value // 6
```

3 Our contributions

3.1 Typeclasses in *Pure Scala*

```
import stainless.lang._
import stainless.annotation._
import stainless.collection._

@coherent
abstract class Semigroup[A] {
    def combine(x: A, y: A): A

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

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

    @inline def combine(x: A, y: A): A = semigroup.combine(x, y)

    @law def law_leftIdentity = forall { (x: A) =>
        combine(empty, x) == x
    }

    @law def law_rightIdentity = forall { (x: A) =>
        combine(x, empty) == x
    }
}

case class Sum(value: Int)
```

```
def lemma_sumAssociative(x: Sum, y: Sum, z: Sum): Boolean = {
    Sum((x.value + y.value) + z.value) == Sum(x.value + (y.value + z.value))
} holds

implicit def intSumSemigroup: Semigroup[Sum] = new Semigroup[Sum] {
    override def combine(x: Sum, y: Sum): Sum = Sum(x.value + y.value)

    override def law_associative = super.law_associative because {
        forall { (x: Sum, y: Sum, z: Sum) => lemma_sumAssociative(x, y, z) }
    }

}

implicit def intSumMonoid: Monoid[Sum] = new Monoid[Sum] {
    override def empty: Sum = Sum(0)
}
```

```
import stainless.lang._
import stainless.annotation._
import stainless.collection._

case class Semigroup[A](combine: (A, A) => A) {
    require(law_associative)

    def law_associative = forall { (x: A, y: A, z: A) =>
        combine(combine(x, y), z) == combine(x, combine(y, z))
    }
}

case class Monoid[A](val semigroup: Semigroup[A], empty: A) {
    require(law_leftIdentity && law_rightIdentity)

    @inline
    def combine(x: A, y: A): A = semigroup.combine(x, y)

    def law_leftIdentity = forall { (x: A) =>
        combine(empty, x) == x
    }

    def law_rightIdentity = forall { (x: A) =>
        combine(x, empty) == x
    }
}

case class Sum(value: Int)

def lemma_sumAssociative(x: Sum, y: Sum, z: Sum): Boolean = {
    Sum((x.value + y.value) + z.value) == Sum(x.value + (y.value + z.value))
} holds
```

```
def intSumSemigroup: Semigroup[Sum] = {
    require {
        forall { (x: Sum, y: Sum, z: Sum) => lemma_sumAssociative(x, y, z) }
    }
    Semigroup[Sum]((x: Sum, y: Sum) => Sum(x.value + y.value))
}

def intSumMonoid: Monoid[Sum] = Monoid[Sum](intSumSemigroup, Sum(0))
```

4 Results

Testcase	Instance	Result	ADT invariant (s)	Total time (s)
AnyMonoid	`Monoid[Any]`	valid	0.222	5.236
FirstMonoid	`Monoid[First]`	valid	1.107	3.796
IntMonoid	`Monoid[Int]` (additive)	valid	2.746	4.951
SumMonoid	`Monoid[Sum]`	valid	0.214	5.811
SemigroupMonoid	`Monoid[Int]` (additive)	valid	0.431	4.856
SemigroupMonoid	`Semigroup[Int]` (additive)	valid	1.044	4.856
Newtype	`Newtype[Sum, BigInt]`	valid	0.133	10.178
EqOrd_partial	`Eq[Int]`	valid	1.536	5.835
EqOrd_partial	`Ord[Int]`	valid	1.725	5.835
EqOrd_full	`Eq[Int]`	timeout	-	-
EqOrd_full	`Ord[Int]`	timeout	-	-
EndoMonoid	`Monoid[Endo]`	timeout	-	-
ListMonoid	`Monoid[List]`	timeout	-	-
NatMonoid	`Monoid[Nat]`	timeout	-	-
OptionMonoid	`Monoid[Option]`	timeout	-	-
NonEmptySemigroup	`Semigroup[NonEmpty]`	timeout	-	-
Uniplate	`Uniplate[Expr]`	timeout	-	-
Semiring	`Semiring[Int]` (disjunctive)	timeout	-	-
Semiring	`Semiring[Boolean]` (disjunctive)	timeout	-	-
Semiring	`Semiring[Boolean]` (conjunctive)	timeout	-	-

5 Further work

6 References

[1] Arvidsson, A. and Touche, R. 2016. *Proving Type Class Laws in Haskell.* Chalmers University of Technology, University of Gothenburg.

[2] 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.

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

[4] 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.

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

[6] 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.

7 Appendix