# Proving Algebraic Properties with Stainless

Romain Ruetschi

June 9th, 2017

## Contents

1 Introduction 2

2 Typeclasses 2

2.1 Definition . . . . . . . . . . 2

2.2 Laws . . . . . . . . . . 3

2.3 Associated methods . . . . . . . . . . 4

2.4 Typeclass inheritance . . . . . . . . . . 4

2.5 Typeclasses in Haskell . . . . . . . . . . 4

2.6 Typeclasses in Scala . . . . . . . . . . 5

2.7 Typeclasses in *Pure Scala* . . . . . . . . . . 6

2.8 Coherence . . . . . . . . . . 6

3 Implementation 6

4 Results 6

5 References 7

6 Appendix 8

# 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 [5] 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:

```haskell
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:

```haskell
-- 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.

```haskell
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.

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 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.

```
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

```
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 1 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:

```
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 def ordInt: Ord[Int] = new 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(1, 2)) // true
}
```

### 2.7 Typeclasses in *Pure Scala*

### 2.8 Coherence

## 3 Implementation

## 4 Results

## 5 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] 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.

## 6 Appendix