Proving Algebraic Properties with Stainless

Romain Ruetschi

June 9th, 2017

Contents

1 Introduction

Stainless aims to help developers build verified Scala software [**???**]. While Stainless does not support the full Scala language, it understands a purely functional subset of it, *Pure Scala*. 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 [4] 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

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.

```haskell
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 not foolproof, nor does it provide any type of mathematical guarantee.

2.3 Typeclass inheritance

2.4 Typeclasses in Scala

2.5 Typeclasses in *Pure Scala*

3 Coherence

4 Implementation

5 Results

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] 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