Brave New World of Haskell Type Classes

František Farka

School of Computing
University of Dundee
School of Computer Science
University of St Andrews
ffarka@dundee.ac.uk

Abstract. Type classes are Haskell way to ad–hoc polymorphism. They provide a mean of specification of a methods with certain semantics and a mean of implementing these methods for arbitrary data type through class instances. Semantics of these methods is expressed in a form of class laws. However, the laws are expressed only informally and a programmer may break them when providing the instance—by accident or by intention.

This paper shows that with use of certain language extensions some of the instances in class hierarchy can be provided in a generic way directly in a class definition and thus some of the laws are guaranteed by class definition itself. The language extensions also allow more flexible approach to changes in class hierarchy that remains backward compatibility and enables author of the class definitions to provide simpler interface to a complex hierarchy of classes.

1 Introduction

We live in a wonderful brave new world of GHC 7.10 and beyond. We don't need to ask ourselves any more "How often did you say: A Monad is always an Applicative but ..."[2]. No buts. A Monad is always an Applicative. `liftM` is `fmap`, really. The notoriously known Haskell type class hierarchy from `base` package now looks like:

```
class Functor f where
        fmap   :: (a → b) → f a → f b

class Functor f ⇒ Applicative f where
        pure   :: a → f a
        (<*>)    :: f (a → b) → f a → f b

class Applicative f ⇒ Monad m where
        (>>=)  :: m a → (a → m b) → m b
        return :: a → m a
```

we can provide instances and use these type classes in our code. Of course, the method type signature is not the only thing we need to keep in mind when implementing instances of these type classes for our custom data types. There are are `Functor`, `Applicative`, and `Monad` laws that instances should abide. In particular, we are interested in laws that are induced by context constraint. McBride and Paterson described the laws in [1]:

$$\text{fmap}\ f\ x = \text{pure}\ f <\!\!*\!\!> x \tag{1}$$

$$\text{pure}\ x = \text{return}\ x \tag{2}$$

$$\text{fmap}\ f\ xs = xs \gg\!\!= \text{return} \circ f \tag{3}$$

for `fmap`, `pure`, `return`, and `(<*>)`. When we closely examine laws (1) and (3) we notice that these compose to another law:

$$\text{pure}\ f <\!\!*\!\!> x = xs \gg\!\!= \text{return} \circ f \tag{4}$$

With use of law (2) and equational reasoning we can state that:

$$pf <\!\!*\!\!> px = px \gg\!\!= \lambda x \to pf \gg\!\!= \lambda f \to \text{return}\ (f\ x) \tag{5}$$

This equation together with (2) and (3) shows that instances of `Functor` and `Applicative` of particular data type are solely determined by an instance of `Monad` for this data type. Same holds for an instance of `Functor` given law(1) where an instance of `Applicative` exists. However, if we provide only an instance of `Monad` for our data type `MyData` compiler refuses to process the code with following error:[1]

```
No instance for (Applicative MyData)
  arising from the superclasses of an instance
  declaration
In the instance declaration for 'Monad MyData'
```

It is necessary to manually provided instances for both `Functor` and `Applicative` even though the behavior is known. This example is not unique among type classes in `base` libraries or elsewhere. In Section 3.3 we show another case—the `Traversable` class.

However, this necessity can be avoided. With additional language extension *Default Superclass Instances* that was described in [4] the instances for the `Functor` and the `Applicative` classes in the above example can be provided in polymorphic form—as the laws that specify their behavior in presence of `Monad` instance are also polymorphic—directly in class definition:

```
class Applicative m ⇒ Monad m where
    (>>=)  :: m a → (a → m b) → m b
    return :: a → m a
```

[1] This example is produced with The Glorious Glasgow Haskell Compilation System, version 7.8.3

```
default instance Applicative m where
        pure x = return x
        pf (<*>) px = px >>= λ x → pf
               >>= λ f → return (f x)
```

This saves programmer of some boilerplate code and can even be done in backward compatible way – without any breakage in existing source code that makes use of these classes. Magalahães et al. described general deriving mechanism for Haskell [6]. This mechanism restricts derivable instances to classes for which exist some notion of default implementation of class method including polymorphic definition that is structurally recursive on its arguments and definition for a set of basic data types. Our approach differs as the definition of method in a default instance is is carried out in terms of other methods of the class hierarchy.

We describe appropriate modification of these classes in full detail in Section 2.1 of this paper.

But this three class hierarchy – `Functor`, `Applicative`, and `Monad` – is not the only class layout that provides methods `return`, `(>>=)`, `pure`, `<*>`, and `fmap`. One may go forth and seek, further up, he will find even newer and brighter world of `Bind` and `Pointed` [5], that a variant of is described by Figure 1. However,

```
class Functor where
  fmap :: (a → b) → f a → f b

class Functor f ⇒ Pointed f where
  pure :: a → f a

class Functor f ⇒ Bind f where
  bind :: f a → (a → f b) → f b

class Pointed f ⇒ Applicative f where
  (<*>) :: f (a → b) → f a → f b

class (Bind m, Applicative m) ⇒ Monad where
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a
```

Fig. 1. Pointed and Bind classes

this finer–grained structure of classes has certain drawbacks. Now it is necessary to define five instances just to work with `Monad`. Since similar laws as in previous example apply some boilerplate can be scraped out with *Default Superclass Instances*. But there is also a new law:

$$\text{bind}\ x\ f = x \gg\!= f \tag{6}$$

This laws relates `bind` and `(>>=)` methods in the same way as `return` and `pure` are bound. In case of `Applicative` and `Monad` we cane simply state that the duplicity of `return` and `pure` is there for historical reasons and for backward compatibility with existing code that uses these two classes.

The first reason – the historical one – does not hold for `bind` for sure. There are no historical premises for this method. Assume that `(>>=)` is moved to the `Bind` class and `Monad` class contains just the `return` method:

```
class Functor f ⇒ Bind f where
        (>>=) :: f a → (a → f b) → f b

class (Bind m, Applicative m) ⇒ Monad m
        return :: a → m a
```

Now, if we provide all the 5 instances necessary to use `Monad` we get the same functionality with `(>>=)` and `return` as before. Data types that instantiate only `Bind` and `Functor` (due to superclass constraint) can be used with `(>>=)` method. But as a side–effect of this move of the method we have lost backward compatibility. Existing instances of `Monad` define the `(>>=)` method, which does no longer exist in the class as the method now resides in the `Bind` class. Compilation fails with following error:

```
'>>=' is not a (visible) method of class 'Monad'
```

John Meachem described a proposal [3] for language extension that allows programmer to specify *Class Aliases*. We give a summary of this proposal in Section 2.2. In principle, the extension allows now language construct – a class alias:

```
class alias Monad m = (Bind m, Applicative m)
```

This class alias can occur wherever ordinary class can and is simply expanded into the two classes it consists of. In particular, it is possible to provide an instance of this alias in a same way an instance of class is provided and define implementations of methods from both `Bind` and `Applicative`. Instances of these classes are generated by compiler and method definitions are supplied from class alias instance.

Section 3.2 shows that this alias can be conveniently used with Default Superclass Instances extension. It is possible to retain `Monad` as a class alias that is backward compatible and still provide standalone class `Bind` with method `(>>=)` without any duplication in form of `bind` method. In addition it is sufficient to manually instantiate only `Monad` alias and instance for `Functor`, `Pointed`, `Applicative`, and `Bind` are generated by compiler. There is also another aspect of the `Monad` alias—the alias abstract from the finer-grained classes and provides simpler interface to underlying concepts. It is fine to use ordinary old `Monad` in all the places where it used to be but we now have the full power and expressiveness of the new hierarchy. We give a more detailed example in Section 3.4.

1.1 Contribution

This paper describes an approach to class hierarchy modifications in Haskell. We point out two language extensions that make the modifications easier: *Default Superclass Instances* and *Class Aliases*. In particular it is possible with these extensions to

- easily change type class hierarchies in a backward compatible way;
- provide some instances automatically, this holds for general type classes, no per-class compiler support is necessary; and
- it is no longer problem to provide fine-grained type class hierarchies—simpler interface to hierarchy can be provided.

We provide several examples that demonstrate use of default superclass instances and class aliases, e. g. `Functor`, `Applicative` and `Monad` hierarchy , extension of previous with `Bind` and `Pointed`, and `Traversable`.

References

1. Conor Mcbride and Ross Paterson. 2008. Applicative programming with effects. J. Funct. Program. 18, 1 (January 2008), pp. 1–13.
2. Functor–Applicative–Monad Proposal, `https://wiki.haskell.org/Functor\-Applicative-Monad_Proposal`
3. John Meacham. Class Alias Proposal for Haskell. `http://repetae.net/recent/out/classalias.html`
4. František Farka. 2015. Maintainable Type Classes for Haskell. Submitted to ACM SIGPLAN symposium on Haskell (Haskell '15).
5. Marina Lenisa, John Power, and Hiroshi Watanabe. 2000. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. Electronic Notes in Theoretical Computer Science, 33, pp. 230–260.
6. José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A generic deriving mechanism for Haskell. In Proceedings of the third ACM Haskell symposium on Haskell (Haskell '10). ACM, New York, NY, USA, 37-48.