On Ringads and Foldables

(Extended Abstract)

James McKinna
LFCS, School of Informatics
University of Edinburgh
UK
james.mckinna@ed.ac.uk

Abstract

While trying to understand Torsten Grust's 2015 MPC keynote on comprehension syntax [Grust 2015], and Jeremy Gibbons's 2016 WadlerFest essay on "Ringad Comprehensions" [Gibbons 2016], and their relationship to Haskell's Foldable typeclass, I arrived at the following characterisations:

- A Functor f is Foldable iff Every Monoid instance a is an f-Algebra instance
- A Monad f is a Ringad iff Every f-Algebra instance a is a Monoid instance

The first is (perhaps) folklore, and appears in Uustalu's paper for the Oliveira Festschrift [Uustalu 2016], but was independently rediscovered during my research. The second is, as far as I know, new.

1 Introduction

Generic Programming has customarily concerned itself with abstraction over the kind of *types*. But as the abstract indicates, this paper concerns two related results, each characterising, in a *generic* way, a class of *higher kind* objects, in this case type *constructors*, representing *collections* or *containers*. Moreover, each characterisation takes the form of a *higher-order* constraint, expressible in the hereditary Harrop fragment. As such it is a contribution to the WGP strand of TyDe, but also indirectly, to the Haskell programing language, where such qualified class constraints are once again the subject of ongoing research [Bottu et al. 2017] (the paper freely abuses haskell's type class syntax as a shorthand to describe algebraic structure, but the reader should be in no doubt that what follows is neither legal haskell, nor conforms to existing definitions in the standard library).

The first concerns the well-known Haskell **Foldable** type class, or at least, that part of it concerned with **Functor** instances:

```
class Functor f => Foldable f where
    fold :: Monoid m =>
        (a -> m) -> f a -> m
```

abstracting over the list `fold` operation from lists to arbitrary collection functors f (true haskell uses the more prolix identitfier `foldMap`, and does not require f to be a **Functor**). Given a mapping from a type `a` to a **Monoid** instance `m`, the generic `fold` should be understood as a specification of 'rolling up' a collection `f a` of `a`-elements, by mapping them to the monoid, and using its operations to *reduce* the collection `f m` of monoid elements down to a single `m`-value. As such, `fold` is nothing more or less than a (non-commutative!) instance of Eindhoven Quantifier Notation, with trivial filtering function [Backhouse and Michaelis 2006].

The second concerns the much less well-known type class, **Ringad**, recently recuperated by Jeremy Gibbons [Gibbons 2016] following pioneering work by Phil Wadler. **Ringad**s are an attempt to capture a common pattern of monad comprehensions arising from functional language representations of SQL queries [Grust 2015], abstracted over the underlying type of collection functor. A key property of comprehensions in the database setting is that *null*, and *singleton* comprehensions should exist, but also that they may be *aggregated* via an abstract binary *union* operation. In other words, **Ringad**s are **Monad** instances (the `return` operation giving rise to singleton collections in the usual way; `bind` permits 'collections of collections' to be collapsed in exactly the ways we expect from comprehension notation) which moreover satisfy the **MonadZero** and **MonadPlus** constraints:

```
class (MonadZero f, MonadPlus f) => Ringad f where
```

In each case, the fundamental relationship, or at least, a more primitive one to which the complex class definition may be reduced, exists between **Monoid** structure on the one hand, and **Algebra** structure on the other, with:

- ```
  class Monoid a where
      0 :: a
      ⊕ :: a -> a -> a
  ```

- ```
  class Functor f => Algebra f a where
      alg :: f a -> a
  ```


TyDe 2017, September 03, 2017, Mathematical Institute, Oxford, UK

2 On Foldables

An alternative definition of **Foldable**, were haskell to support it [Bottu et al. 2017] may be given as the following empty class definition:

```
class (Functor f,  ∀ a. Monoid a => Algebra f a)
     => Foldable f where
```

by taking `fold h = alg . fmap h` Indeed, given a **Foldable** f, every **Monoid** m carries an f-**Algebra** structure by taking `alg = fold id` so, provided at least `fmap id = id`, the above definition indeed characterises **Foldable**s. Given a true **Functor** instance, satisfying also `fmap (f . g) = fmap f . fmap g`, we should then expect the following naturality property of **Foldable**s, viz.:

```
fold (h . f) = fold h . fmap f
```

We may note here in passing that none of the above makes any use whatsoever of the **Monoid** class: that is, we could generalise this result further to a defintion of **Foldable** which is parametrised over *any* class qualifier Q. But doing so would take us far outside the hereditary Harrop fragment.

3 On Ringads

After the preceding warmup, let us proceed directly to our second characterisation:

```
class (Monad f,  ∀ a. Algebra f a => Monoid a)
     => Ringad f where
```

The twist here is that we need to consider f-**Algebra** structure wrt f being a **Monad** instance, that is, the `alg` operation should additionally satisfy:

(η) `alg . return = id`
(μ) `alg . mult = alg . fmap alg`

The main technical idea, already present in Gibbons' beautiful reconstruction [Gibbons 2016] of Wadler's earlier ideas, is to see how f-**Algebra** structure, in the presence of **MonadZero** and **MonadPlus**, gives rise to **Monoid** structure:

```
instance (MonadZero f, MonadPlus f, Algebra f a) =>
    Monoid a where
      0 = alg zero
      a ⊕ b = alg ((return a) `plus` (return b))
```

As Gibbons notes, it is an nice exercise to show that these definitions do indeed give rise to **Monoid** structure, and that, in particular, associatitivity of `plus` implies that of the induced $\oplus$; similarly for 0 being a unit for $\oplus$ on the basis that `zero` is for `plus`.

The other direction of the equivalence is (perhaps) even easier: the distinguished (free) algebra structure `mult` obtained from the **Monad** f, coupled with the constraint `∀ a. Algebra f a => Monoid a`, directly yields the relevant instances of **MonadZero** and **MonadPlus**.

4 Where's the catch?

As pointed out by the anonymous referees, the issue of what equations should be imposed on the various operations remains unresolved by the definitions made here, as does the detailed verification of the round-trip laws needed to witness the identifications claimed here. This is future work!

5 Conclusions

Beyond being perhaps merely a cute 'trick', my interest in this work was to try to understand, and hopefully explain, two 'difficult' compound type classes in haskell in terms of simpler compponents. While the **Foldable** type class is familiar to all haskell programmers, it emerges as an evolutionary abstraction within the simple system of class constraints supported by haskell. By passing to the richer hereditary Harrop fragment of such constraints, exploiting universal quantification over implicational constraints, we have shown how to characterise it completely, and indeed to show it that it really has nothing to do with **Monoid** at all.

By contrast, Gibbons' reinvestigation of Wadler's **Ringad** class, together with the already rich literature relating (SQL-like) queries and monad comprehensions, has thrown up a number of questions regarding the further, non-haskell-expressible, constraints which should be imposed in order to capture queries-as-comprehensions. By reducing **Ringad** to its more lementary components, we hope to shed light on future investigations in this area.

Acknowledgments

I am grateful to the members and observers of IFIP Working Group 2.1 for their helpful feedback following a first presentation of this work. Special thanks to Jeremy Gibbons for some equational hand-holding. Any errors and misunderstandings remain, of course, my own.

References

Roland Carl Backhouse and Diethard Michaelis. 2006. Exercises in Quantifier Manipulation. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings (Lecture Notes in Computer Science)*, Tarmo Uustalu (Ed.), Vol. 4014. Springer, 69–81.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Haskell Symposium, Oxford, September 2017, Proceedings.*

Jeremy Gibbons. 2016. Comprehending Ringads - For Phil Wadler, on the Occasion of his 60th Birthday. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. LNCS 9600. Springer, 132–151.

Torben Grust. 2015. A Compilation of Compliments For a Compelling Companion: The Comprehension. In *Mathematics of Program Construction, 12th International Conference, MPC 2015, Königswinter, Germany, June 29-July 1, 2015, Proceedings*, Ralf Hinze and Janis Voigtländer (Eds.), Vol. LNCS 9129. Springer.

Tarmo Uustalu. 2016. A divertimento on MonadPlus and nondeterminism. *J. Log. Algebr. Meth. Program.* 85, 5 (2016), 1086–1094.