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 type-class, 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 programming 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 \textbf{Foldable} type class, or at least, that part of it concerned with \textbf{Functor} instances:

\[
\text{class } \text{Functor } f \Rightarrow \text{Foldable } f \text{ where}\\
\quad \text{fold :: Monoid } m \Rightarrow\\n\quad \quad (a \rightarrow m) \rightarrow f a \rightarrow m\\
\]

abstracting over the list \texttt{fold} operation from lists to arbitrary collection functors \( f \) (true haskell uses the more prolix identifier \texttt{foldMap}, and does not require \( f \) to be a \textbf{Functor}).

Given a mapping from a type \( a \) to a \textbf{Monoid} instance \( m \), the \texttt{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, \texttt{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, \textbf{Ringad}, recently recuperated by Jeremy Gibbons [Gibbons 2016] following pioneering work by Phil Wadler. \textbf{Ringads} 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 \texttt{null}, and \texttt{singleton} comprehensions should exist, but also that they may be aggregated via an abstract binary \texttt{union} operation. In other words, \textbf{Ringads} are \textbf{Monad} instances (the \texttt{return} operation giving rise to singleton collections in the usual way; \texttt{bind} permits ‘collections of collections’ to be collapsed in exactly the ways we expect from comprehension notation) which moreover satisfy the \textbf{MonadZero} and \textbf{MonadPlus} constraints:

\[
\text{class } (\text{MonadZero } f, \text{MonadPlus } f) \Rightarrow \text{Ringad } f \text{ where}\\
\quad \text{alg :: Algebra } f a \text{ where}\\
\quad \quad \text{alg :: } f a \rightarrow a\\
\]

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

\[
\quad \text{class } \text{Monoid } a \text{ where}\\
\quad \quad 0 :: a\\
\quad \quad \oplus :: a \rightarrow a \rightarrow a\\
\]

\[
\quad \text{class } \text{Functor } f \Rightarrow \text{Algebra } f a \text{ where}\\
\quad \quad \text{alg :: } f a \rightarrow a\\
\]
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:

\[
\text{class } (\text{Functor } f, \ \forall a. \text{Monoid } a \Rightarrow \text{Algebra } f \ a) \\
\Rightarrow \text{Foldable } f \text{ where}
\]

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

\[
\text{fold } (h \ . \ f) = \text{fold } h \ . \ \text{fmap } f
\]

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 definition 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:

\[
\text{class } (\text{Monad } f, \ \forall a. \text{Algebra } f \ a \Rightarrow \text{Monoid } a) \\
\Rightarrow \text{Ringad } f \text{ where}
\]

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:

\[
\begin{align*}
(\eta) \text{ alg . return } &= \text{id} \\
(\mu) \text{ alg . mult } &= \text{alg . fmap alg}
\end{align*}
\]

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:

\[
\text{instance } (\text{MonadZero } f, \text{MonadPlus } f, \text{Algebra } f \ a) \Rightarrow \\
\text{Monoid } a \text{ where}
\begin{align*}
0 &= \text{alg zero} \\
a \oplus b &= \text{alg } ((\text{return } a) \ \text{\textquoteright plus } \ (\text{return } b))
\end{align*}
\]

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

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 \( \forall a. \text{Algebra } f \ a \Rightarrow \text{Monoid } a \), 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 components. 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


