Flipping Fold, Reformulating Reduction

An Exercise in Categorical Design

Gershom Bazerman
S&P/CapitalIQ
gershomb at gmail

1. Introduction

We begin this paper by considering the Haskell ‘Foldable‘ typeclass, a stalwart of the standard libraries. Unlike many other typeclasses, most famously Monad, Foldable itself has been equipped with no required laws. This is rather surprising, as folds themselves are some of the most well understood and studied aspects of functional programming, and the universal properties of folds, in general, are what we often use to prove laws. We will explore why it is hard to give laws to Foldable on its own. From there we will define a naturally arising class, adjoint to Foldable, which we name Buildable. In turn, we will explore how Foldable and Buildable in conjunction, each individually lawless, nonetheless are mutually constrained by an elegant set of laws arising from categorical principles. We will then explore Buildable as an independently useful class that allows us to compose systems of streaming and parallel computation, and explore its relationship to a prior, similar formulation. The aim of this paper is then threefold; to provide laws to Foldable, to provide a new, useful class of Buildable types, and along the way, to illustrate a way in which categorical thinking can give rise to practical results.

2. Recalling Foldable

The ‘Data.Foldable‘ library, written by Ross Paterson, and part of the standard libraries that ship with the Glasgow Haskell Compiler, provides a Foldable typeclass. While it has many methods, all methods can be derived by the user defining only one of foldr or foldMap. So we consider the cleaner interface given below.

```
class Foldable t where
    foldr ::
        (a -> b -> b) -> b -> t a -> b Source
    foldMap ::
        Monoid m => (a -> m) -> t a -> m Source
```

In fact, there is a further function, not in the class, but included in the file, which also provides a complete implementation of ‘foldable‘. We can consider its definition as follows:

```
toList :: Foldable t => t a -> [a]
toList t = foldr (:) [] t
```

Without much work, one can see how ‘foldr‘, ‘foldMap‘, and ‘build‘ are all interdefinable and hence equal in expressive power. When one considers folds in general, one typically expects them to universally characterize the meaning of a particular data structure in terms of all operations possible on it – in fact that is the very definition of a proper fold. However, we can observe that the ‘foldr‘ given here in fact characterizes all operations possible on a data structure ***when considered as a list***. So the laws of folds themselves follow naturally from our usual constructions, and are given directly. However, what it means to consider a data structure to a list is left completely undefined. For example, we could equip all type constructors of arity one with the Foldable instance who acts as the empty list. This would violate user expectations, but not any particular given typeclass law. Clearly something must be done.

3. Enter Buildable

If the laws of Foldable won’t come from the class itself, then they must come from interaction with other classes. This is the pattern we have seen elsewhere, recently where work by Jaskelioff, and later Bird and Gibbons has provided ‘Traversable‘ functors with laws as given by their interrelationship with Applicative actions.[6][1] Much earlier, of course, we had to define the relationship of ‘Eq‘ and ‘Ord‘ instances such that they agreed. Other examples also abound.

What class shall we use to interact with ‘Foldable‘? A clue is provided in the genuine definition of ‘toList‘, which in turn is defined in terms of ‘build‘, imported from ‘GHC.Exts‘.

```
toList :: Foldable t => t a -> [a]
toList t = build (\ c n -> foldr c n t)

build ::
    forall a.
    (forall b. (a -> b -> b) -> b -> b)
    -> [a]
build g = g (:) []
```

Why this indirection? Well, as the documentation tells us, ”GHC’s simplifier will transform an expression of the form ‘foldr k z (build g)‘, which may arise after inlining, to ‘g k z‘, which avoids producing an intermediate list.”. This is an instance of ”shortcut fusion” as introduced by Gill, Launchbury, and Peyton Jones.[2] Recent work by Hinze[**?**] has explored the relationship between shortcut fusion and the categorical notion of an ***adjoint***, which we will come back to. In any case, in the special case of ‘foldr‘ and ‘build‘ on lists, we observe that they correspond to providing a full isomorphism between lists and the partial application of the fold function to lists, which is to say between lists seen ”initially” and lists seen ”finally” as characterized by their universal property.

Just as as the ‘Foldable‘ typeclass simply wraps up ‘fold‘, we now introduce a ‘Buildable‘ typeclass to wrap up ‘build‘. As all Foldables can provide a ‘toList‘, we also provide a ‘fromList‘ to help examine the behaviour of Buildables.

```
class Buildable f a where
    build :: ((a -> f a -> f a) -> f a -> f a) -> f a
    build g = g insert unit

    singleton :: a -> f a
    singleton x = build (\c n -> c x n)

    unit ::  f a
    unit = build (\cons nil -> nil)

    insert :: a -> f a -> f a
    insert x xs = build (\cons nil -> x ‘cons‘ xs)

fromList :: Buildable f a => [a] -> f a
fromList xs = foldr insert unit xs
```

A minimal complete definition is given by ‘build‘, or by ‘insert‘ coupled with ‘unit‘. The ‘build‘ function can be seen as providing the concrete constructors to a partially applied fold, and the ‘insert‘ and ‘unit‘ functions as just introducing the two constructors (the binary and unary operations) explicitly.

There are a few design decisions here worth justifying. First, the choice to use a multi-parameter typeclass, and second the choice (only implicitly present here) not to require any sort of monoidal behaviour, and instead a looser notion of “adjoint” laws. Both decisions can be justified by examining a standard type that clearly should be buildable, but nonetheless is not isomorphic to list – ‘Set‘. We can write a Buildable instance for ‘Set‘ like so:

```
instance Ord a => Buildable Set a where
    unit = Set.empty
    insert = Set.insert
```

Here the purpose of the extra type variable becomes clear – while the ‘Ord‘ constraint is not necessary to “tear down” a set, it certainly is necessary to build one up, and thus must be included in our typeclass. While this costs us in terms of verbosity, at least it introduces no loss in expressiveness.

Now we consider the behaviour of the interaction of fromList and toList on ‘Set‘. Whatever laws we introduce must surely not rule out such a basic instance. Clearly we expect ‘`thereBack xs :: toList . Set.fromList`‘ to reorder our elements. Furthermore, we expect it to merge duplicate elements. However, we also know that if we iterate ‘thereBack‘ repeatedly, it is idempotent. In this case, ‘toList‘ is a retraction of ‘fromList‘, and the composition ‘fromList . toList‘ is a split idempotent. More generally, we can consider the functorial nature of ‘Foldable‘ and ‘Buildable‘ to produce a set of laws claiming that when both instances exist, they should be ***adjoint***.

4. Folds, Builds, and Adjunctions

The connection of adjointness to folds, unfolds and fusion laws has been explored in the recent work of Ralf Hinze[4,5]. In general, fusion laws are about moving to an “adjoint space” where composition is directly given, and then shifting back to the original space to present the result. Although the movement between regular and church-encoded lists given in fold/build fusion is an isomorphism, in general there is no such restriction. Streams including ‘yield‘ are a bigger space than lists, etc. The purpose behind such adjunctions is, loosely speaking, to allow us to capture “only what matters” about a computation. When “moving across” the two functors which make up an adjoint, we are able to transport where the work of functions occurs.

To examine how this plays out in the terrain of ‘Foldable‘ and ‘Buildable‘ we can translate a version of the adjoint laws to our specialized usecase. In the ”hom-set adjunction” formulation, for two categories C and D and two functors $F : D \to C$ and $G : C \to D$, we have the formula:

$$C(FX, Y) \cong D(X, GY)$$

Take C to be some ‘Buildable‘ and ‘Foldable‘ functor f, and D to be ‘List‘, and we arrive at the following Haskell claim: For all functions ‘`f : f a -> f b`‘, there is a function ‘`g : [a] -> [b]`‘ such that ‘`f . fromList :: [a] -> f b`‘ is isomorphic to ‘`toList . g :: [a] -> f b`‘. That is to say, all functions on ‘Foldables‘ can be translated to functions on ‘Buildables‘, and vice versa, such that even if they do not actually coincide, when we ”move across” the types appropriately, they will.

When our ‘Buildable‘ and ‘Foldable‘ instances are lawful, we can in fact write functions to witness this directly, if not efficiently.

```
f2g f = toList . f . fromList
g2f g = fromList . g . toList
```

And so we see that functions on lists may be seen as functions on functors adjoint to list ”factored through” lists, and dually that functions on functors adjoint to list may be viewed as actions on lists ”factored” through the adjoint, and that such notions coincide. In the specific case of ‘Set‘, this means that there is no function on sets that cannot be written as a function on the list underlying a set, and furthermore that there is no function yielding a list that underlies a Set that cannot be transformed directly into a function on sets.

5. Reducers as Buildables

Hinze and Jeuring introduced a predecessor class to ‘Foldable‘ named ‘Reduce‘.[5] However, it is in fact ‘Buildable‘ that really provides the ”reduction” component directly – with ‘Foldable‘ describing the ”shape” of a reduction but ‘Buildable‘ providing the actual ***target semantics*** of any given fold. ‘Foldable‘ describes how to fold, but it is ‘Buildable‘ that fixes a fold to a concrete meaning. In fact, ‘Buildable‘ provides a very close analog, though more theoretically motivated, to the ‘Monoidal Reducers‘ available in Edward Kmett’s reducers package.

The following code listing demonstrates the “basic” functionality that all notions of reduction should share – the ability to define multiple aggregations such as sum and count, and the ability to zip them into one pass. Here the aggregations we define happen to be in fact monoidal. But in general, no such restriction applies.

```
newtype Sum a = Sum {getSum :: a}
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x + y)

newtype Count = Count Int deriving Show
instance Monoid Count where
    mempty = Count 0
    mappend (Count x) (Count y) = Count (x + y)

newtype Const m a = Const m

instance Num a => Buildable (Const (Sum a)) a where
    unit = Const (Sum 0)
    insert x (Const xs) = Const (Sum x ‘mappend‘ xs)

instance Buildable (Const Count) a where
```

```
    unit = Const (Count 0)
    insert x (Const xs) = Const (Count 1 `mappend` xs)

newtype Product f g a = Product (f a, g a)

instance (Buildable f a, Buildable g a) =>
                Buildable (Product f g) a where
    unit = Product (unit, unit)
    insert x (Product (xs,ys)) =
                  Product (insert x xs, insert x ys)
```

The listing contains two items of particular interest. First, we introduce a 'Const' type to carry around explicit information about what should be "fed in" to a 'Buildable', and more generally to lift an aggregation into a functorial context. Second, we introduce a traditional product of functors, and give it a 'Buildable' instance directly. By construction our builds only require one pass, and so we can introduce concurrent reductions while operating in constant space.

6. Composing Buildables Horizontally and Vertically

7. Extensions and Transformations

8. Serial and Parallel Computation

9. Relating Builds to Traversals

10. Related Work

As discussed, the closest analogue to the work presented here is Edward Kmett's monoidal reducers package. The concrete difference is that rather than generalize over things of kind '`* -> *`', Monoidal Reducers are equipped with two type parameters, each of kind '`*`' – the things that reducers "accept", and the things that reducers "reduce to." Furthermore, these reducers, as one would infer from the name, are required to operate as a monoid does, i.e. associatively. (Less importantly for our purposes, Monoidal Reducers, as one would not infer from the name, are in fact generalized as to work over semigroups [i.e. they do not require an "empty" value equivalent to 'unit' as presented here]). In the absence of any other constraints, requiring associative structure is about the minimal law one can require such a structure to hold. However, as have seen, in the presence of an interaction with 'Foldable', we can get a looser but still sufficient notion of a lawful structure even without requiring associativity – and in fact, there are very good reasons we should not!

Rich Hickey also arrived at similar formulations to Kmett's, though in an untyped context, in the 'reducers' library for Clojure. The inspiration for both lines of work is owed to Guy Steele's 2009 ICFP invited talk "Organizing Functional Code for Parallel Execution."

11. Conclusion

Acknowledgments

References

[1] Bird, R., Gibbons, J., et al. Haskell 2013: 25-36

[2] Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993) A short cut to deforestation. Proceedings, Conference on Functional Languages and Computer Architecture, 223-232.

[3] Hinze, R. Adjoint Folds and Unfolds. MPC 2010: 195-228

[4] Hinze, R. Type Fusion. AMAST 2010: 92-110

[5] Hinze, R. and Jeuring, J. Generic Haskell: Practice and theory. Technical Report UU-CS-2003-15, Department of Computer Science, Utrecht University, 2003.

[6] Jaskelioff, M., Rypacek, O. MSFP 2012: 40-49