Submission for IFL 2014

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 these two classes 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 (modulo efficiency) by the user defining only one of foldr or foldMap. For the purposes of this paper we consider the cleaner interface given below.

    class Foldable t where
        foldr :: (a → b → b) → b → t a → b
        foldMap :: Monoid m ⇒ (a → m) → t a → m

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 toList are all interdefinable and hence equal in expressive power on finite structures. On infinite structures, foldr captures a typical “cons-list” that can extend infinitely far to the right, while a foldl can capture a “snoc-list,” and foldMap is more powerful than both, capturing even structures that can have a notion of a center of contraction while extending indefinitely both right and left. However, for the purposes of this paper we will work directly with toList for expository purposes, with the understanding that it is future work to generalize these results.

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. 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. One motivation for our work is to provide a set of laws to begin to match our expectations of what Foldable “should” do.

3. Enter Buildable

Because Foldable has effectively only one operation, we cannot give it laws on its own. Rather, we must define how this operation interacts with other operations, and to do so we must introduce at least one other class. 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.[4, 20] 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.Ects.

    toList :: Foldable t ⇒ t a → [a]
    toList t = foldr (:) [] t
    build :: (forall b o (a → b → b) → b → [a])
    build g = g () []

Why this indirection? 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.”[1] This is an instance of “shortcut fusion” as introduced by Gill, Launchbury, and Peyton Jones.[12] Recent work by Hinze[16] has explored the relationship between shortcut fusion and the categorical notion of an adjoint, which we...
will come back to. 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 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
  build g = g insert unit

singleton :: a → f a
singleton x = build (λc n → c x n)

unit :: f a
unit = build (λacons nil → nil)

insert :: a → f a → f a
insert x xs = build (λacons nil → x · cons' xs)

fromList :: Buildable f a ⇒ [a] → f a
fromList xs = List.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. In fact, build is simply the Church encoding of the pair (insert, unit).

We can recognize such a pair as an F-algebra [31] of a list, or an f-valued catamorphism [25]. More intuitively, while a Foldable is “something that contains as to be folded over”, a Buildable a is simply “something that can be produced by folding over as”. Or, still another way, a Foldable is something that can be viewed as a list and a Buildable is a view on a list.

There is one important design decision here worth justifying – the choice to use a multi-parameter typeclass. This can be justified by examining a standard type that clearly should be buildable, the choice to use a multi-parameter typeclass, This can be justified on.

Submit for IFL 2014 2 2014/12/15

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[15]. 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.[8] 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 the work of functions occurs.

We recall the formal definition of an adjunction. Such a beast consists of a pair of functors, mapping from one category to a second and back. In the “hom-set adjunction” formulation, for two categories C and D and two functors F : D → C and G : C → D, we have the formula:

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

While such a definition appears very symmetric, this is deceptive. Informally, we can regard the functor F going rightwards as “forgetful” and the functor G going leftwards as “free”. The above isomorphism only applies to morphisms in C from objects induced by the functor F, and morphisms in D to objects induced by the functor G.

In our specific context, and taking our Buildable functor f to be right (forgetfully) adjoint to [], this is the condition that for all functions \( f : [a] \rightarrow [b] \), there is a function \( g : [a] \rightarrow [b] \) such that \( f \circ \text{fromList} \Rightarrow [a] \rightarrow [b] \) is isomorphic to \( \text{toList} \circ g : [a] \rightarrow [b] \). That is to say, all functions on our functor can be translated to functions on lists, and vice versa, such that even if they do not actually coincide, when we “move across” the types appropriately, they will. When our instances of foldable and buildable are lawful, we can in fact write functions to witness this directly, if not efficiently.

$$\text{toList} \circ (\text{Buildable } f a, \text{Foldable } f) \Rightarrow (f a \rightarrow f b) \Rightarrow [a] \rightarrow [b]$$

$$\text{mapToList} \circ \text{toList} \circ f \circ \text{fromList} \Rightarrow [a] \rightarrow [b]$$

$$\text{mapFromList} \circ (\text{Foldable } f, \text{Buildable } f b) \Rightarrow ([a] \rightarrow [b]) \rightarrow f a \rightarrow f b$$

$$\text{mapFromList } g \Rightarrow \text{toList } \circ f \circ \text{toList}$$

In this formulation, the law is that \( \text{mapFromList} \circ \text{mapToList} \) must be equivalent to identity. Expanding this out we see the condition is that for all functions f from \( [f a \rightarrow f b] \), it is the case that:

$$f \equiv \text{fromList } \circ \text{toList} \circ f \circ \text{toList}$$

Many adjoints follow the lead of Set and include some notion of a retract, in which case \( \text{fromList } \circ \text{toList} \) is itself an identity. In such a case, we call these creatures an “idempotent adjunction” and can make a stronger claim: Functions on lists may be seen as functions on functors right adjoint to list “factored through” lists, and dually that functions on functors right 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 (i.e. function f of type \([a] \rightarrow [b]\]) such that \( \forall x \exists y. f x \equiv \text{toList } y \) that cannot be transformed directly into a function on sets.

5. Lawful and Unlawful Builds and Folds

It is a lovely property of adjoints that given any functor F, both left and right adjoints, if they exist, are necessarily unique up to isomorphism. Hence, given any Foldable instance, we can check if a Buildable instance exists, and furthermore determine what such an instance must do, up to isomorphism. Similarly, any given Buildable instance uniquely determines a Foldable, if one exists.
Nonetheless, it is also true that there are many valid Foldable instances, corresponding to different strategies for “collecting the contents”, and in fact each gives rise to a different adjoint. Furthermore, to this point we have only considered “forgetful” adjoints to list (i.e. where they contain no more information than a list). In fact, we also want to consider left adjoints to list, which contain some information not able to be captured by a list. Here is a simple type with such a property:

```
data AnnotateL m a = AnnotateL m a

instance Foldable (AnnotateL m) where
  foldr f a (AnnotateL _ xs) = List.foldr f a xs

instance Monoid m ⇒ Buildable (AnnotateL m) a
  where
    mempty = AnnotateL m mempty
    insert x (AnnotateL m xs) = AnnotateL m (x : xs)
```

As we can see, this type is “bigger” than a plain list. And so it is no longer the case that `fromList ⇆ toList` is equivalent to identity. When so transliterated, any function of type `AnnotateL m a → AnnotateL m b` will preserve the action on the list component, but discard the action on the annotation. Thus we have an adjoint in the other direction. In general, such adjoints will arise when we have some notion of product in our functor. This is a specific instance of the general rule that right adjoint functors commute with limits, and left adjoint functors commute with colimits.[23] A common such case arises with trees, where a list equipped with branching structure is obviously bigger than a list with such structure flattened out.

So shall we require that Buildable be either a left or a right adjoint? At first glance, this situation seems problematic. The last thing we want is a typeclass where we can choose which set of laws to follow. However, things are not as bad as they appear. A functor is only both a left and a right adjoint if it is itself an equivalence. So we do not need to “choose” which laws to follow – they are chosen for us, and whenever both choices are applicable it is only because both choices coincide. Furthermore, even if we zig-and-zag over left and right adjoints freely, we still retain strong reasoning principles, as long as we restrict ourselves to idempotent adjunctions as described above – i.e. where they factor as embedding/projection pairs in either direction. This is because every idempotent adjunction induces equivalence between the full images of our categories C and D under our given functors. Thus if we require either the left or right idempotent adjoint condition, we still find they coincide in the following straightforward set of laws.

```
toList ◦ fromList ◦ toList ≡ toList
fromList ◦ toList ◦ fromList ≡ fromList
```

This is to say, going either direction may lose information, but no information may be lost repeatedly. Such a condition captures the neat intuition that Foldables and Buildables represent a general notion of computations that “factor through” lists, and furthermore that while we may “summarize” information at every step, we do not “repeatedly summarize” such that we eventually have no information left at all. For lack of better terminology call these laws a condition of weak idempotence.

It is still the case that, in general, we do not have unique Foldable and Buildable instances. For example, `Set.toList` is equally well adjoint if it produces things in either ascending or descending order. In fact, most Foldable instances can be paired with an appropriate Buildable and vice versa. And when they cannot be, this in itself is interesting, as we shall see. Nonetheless we have a strong set of laws which we can use to structure reasoning about our programs, where before we had none. In particular, most Buildable structures come with an obvious notion of the “right” way to reduce into them from lists, and this in itself is often enough to determine what the natural way to bring them back to lists should be.

To examine how this works out in practice, consider the product of two Buildables, with the derived Buildable instance derived as a product of the underlying instances.

```
data 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)
```

Assuming both functors are also Foldable, there are two potentially lawful Foldable instances in this case.

```
data Coproduct f g a = InL (f a) | InR (g a)

instance (Foldable f, Foldable g) ⇒ Foldable (Coproduct f g) where
  foldr f z = foldr f z f (InL a) ∘ foldr f z g (InR a)

instance (Buildable f, Buildable g) ⇒ Buildable (Coproduct f g) where
  unit = InL unit
  insert x (InL a) = InL (insert x a)
  insert x (InR a) = InR (insert x a)
```

Now, when our underlying structures are both left adjoints, we again fulfill neither set of adjoint laws but nonetheless preserve weak idempotence. Finally, in the presence of the Functor typeclass, we will add one more law to our Buildable instances. Foldable, which places no constraints on the a, has always induced toList as a natural transformation whenever f is a Functor – i.e it is necessarily the case that for all functions `g :: a → b`, `map g ◦ toList ≡ toList ◦ fmap g`. We now also require that when f is a Functor, then the Buildable instance place no constraints on a as well. This now yields the nice property that `fromList` is a natural transformation as well – i.e. it is the case that for all `g`, `fmap g ◦ fromList ≡ fromList ◦ fmap g`, and as a consequence `fmap g ◦ singleton ≡ singleton ◦ g`.

### 6. Relating Builds to Monads and Traversals

Equipping Foldable, Buildable pairs with an adjoint relationship allows us to generate a number of interesting derived properties. For one, as is well known, every adjunction gives rise to a Monad.

Submission for IFL 2014 3 2014/12/15
[23] With our equipment, we do not necessarily have a full member of the Monad typeclass in Haskell, in particular, because the elements of a Buildable are potentially subject to some restriction. Nonetheless, we possess enough power to build a Restricted Monad as proposed by Hughes[19, 28].

The following code listing, following the Restricted Monad implementation in the “rmonad” package [30], witnesses this relationship by producing for any right adjoint Buildable, Buildable pair, a corresponding monad.

```haskell
newtype WrapBuild f a = WrapBuild { getBuild :: f a }

instance Buildable f a ⇒ Buildable (WrapBuild f) a where
    unit = WrapBuild unit
    insert x (WrapBuild f) = WrapBuild (insert x f)

class Suitable m a where
    constraints :: Constraints m a

instance RMonad m where
    return :: Suitable m a ⇒ a → m a
    bind :: (Suitable m a, Suitable m b) ⇒ m a → (a → m b) → m b
    rmap :: (Suitable m a, Suitable m b) ⇒ (a → b) → m a → m b

data family Constraints (m :: * → *) :: * → *

data instance Constraints (WrapBuild f) a =
    (Buildable f a) ⇒ BC

instance (Buildable f a, Foldable f) ⇒
    Suitable (WrapBuild f) a

instance RMonad m where
    return :: Suitable m a ⇒ a → m a
    bind :: (Suitable m a, Suitable m b) ⇒ m a → (a → m b) → m b
    rmap :: (Suitable m a, Suitable m b) ⇒ (a → b) → m a → m b

data instance Constraints (WrapBuild f) a =
    (Suitable m a) ⇒ BC

instance Buildable (Ann Count) a

instance Buildable Ann m

instance Foldable Ann m where
    foldr f z x s = foldr f z (toList s)

newtype Count = Count Int deriving Show

instance Monoid Count where
    mempty = Count 0
    mappend (Count x) (Count y) = Count (x + y)

data Ann m a = Ann m

instance Buildable Ann Count a

instance Buildable Ann m

instance Foldable Ann m where
    foldr f z _ = z

newtype Sum a = Sum { getSum :: a }

instance Eq a ⇒ Eq (Sum a)

instance Ord a ⇒ Ord (Sum a)

instance Num a ⇒ Num (Sum a)

instance Foldable Sum where
    foldr f _ (Sum x) = f x

instance Product Sum (Ann Count) Double

sumCount :: Double →
    Product Sum (Ann Count) Double

sumCount = fromList
```

It is well known that Foldable is a superclass of Traversable. This demonstrates that right adjoint Buildable, Buildable pairs are a subclass, corresponding to those Traversable functors that can be built “incrementally” from zero elements upwards. Following [5], we can think of Traversable objects as uniquely decomposing into shape and contents. Right-adjoint Buildable functors are precisely those whose shape can be determined, piecemeal, by the stream of their contents.

7. Reducers as Builds

Hinze and Jeuring introduced a predecessor class to Foldable named Reduce.[17] 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 “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.

```haskell
newtype Count = Count Int deriving Show

instance Monoid Count where
    mempty = Count 0
    mappend (Count x) (Count y) = Count (x + y)

data Ann m a = Ann m

instance Buildable Ann Count a

instance Buildable Ann m

instance Foldable Ann m where
    foldr f z x s = foldr f z (toList s)

newtype Sum a = Sum { getSum :: a }

instance Eq a ⇒ Eq (Sum a)

instance Ord a ⇒ Ord (Sum a)

instance Num a ⇒ Num (Sum a)

instance Foldable Sum where
    foldr f _ (Sum x) = f x

instance Product Sum (Ann Count) Double

sumCount :: Double →
    Product Sum (Ann Count) Double

sumCount = fromList
```

The listing contains a few items of particular interest. First, we introduce a Ann 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. By construction our builds only require one pass, and so the use of our Product as defined previously allows the introduction of concurrent reductions while operating in constant space. Finally, the resultant sumCount function itself is trivial – all the work of describing the nature of the computation has been pushed entirely to a declarative level in the type, and the code itself is synthesized from the specification by the compiler’s typeclass resolution.

For completeness we have included Foldable definitions for Ann and Sum, though neither is of particular interest.
8. Varieties of Composition

Having introduced Product as one notion of composition, corresponding to parallel reduction, it seems appropriate to ask what other forms of composition we can define. For one, there is the traditional composition of functors. If we provide our outer \( f \) with a valid \( \text{Functor} \) instance, the following instance is possible.

\[
\text{newtype} \ \text{Compose} \ f \ a = \text{Compose} \ (f \ a) \\
\text{instance} \ (\text{Functor} \ f, \ \text{Buildable} \ g \ a, \ \text{Buildable} \ g \ a) \Rightarrow \ \text{Buildable} \ (\text{Compose} \ g \ f) \ a \\
\quad \text{where} \\
\quad \text{unit} = \text{Compose} \ \text{unit} \\
\quad \text{insert} \ x \ (\text{Compose} \ zs) = \text{Compose} \ (\text{fmap} \ (\text{insert} \ x) \ zs)
\]

Here, we perform an insert of each new element across \( all \) elements \( g \ a \) in our outer functor. This corresponds to a notion of simultaneous “fanning out” a computation across computational resources. We might use this form for evaluation to number a number of predicates on a substring simultaneously. While \( \text{Product} \) provides a notion of trivial concurrency, we can describe \( \text{Compose} \) as a notion of trivial parallelism.

But it is not enough to fork a computation out if we are not able to join it back. Furthermore, even without parallelism or concurrency, some forms of computation are innately sequential. Hence we introduce a type to capture sequential computation. While the structure is the same of our \( \text{Product} \) type, the \( \text{Buildable} \) instance is quite different, taking advantage of the relationship to \( \text{Foldable} \).

\[
\text{data} \ \text{Seq} \ f \ g \ a = \text{Seq} \ (f \ a) (g \ a) \\
\text{transform} :: (\text{Buildable} \ g \ a, \ \text{Foldable} \ f) \Rightarrow \ f \ a \rightarrow g \ a \\
\text{transform} \ zs = \text{build} \ (\lambda c \ n \rightarrow \text{foldr} \ c \ n \ zs) \\
\text{instance} \ (\text{Foldable} \ f, \ \text{Buildable} \ f \ a, \ \text{Buildable} \ g \ a) \Rightarrow \ \text{Buildable} \ (\text{Seq} \ f \ g) \ a \ where \\
\quad \text{unit} = \text{Seq} \ \text{unit} \ \text{unit} \\
\quad \text{insert} \ x \ (\text{Seq} \ zs \ ys) = \lambda \text{let} \ xzs' = \text{insert} \ x \ zs \ \\
\quad \text{in} \ \text{Seq} \ xzs' \ (\text{transform} \ xzs')
\]

This lets us build into \( f \), and, eventually, rebuild that \( f \) into a \( g \). Because we are in a lazy language, we do not in fact compute the transform (which is simply a fused from\( \text{List} \) \( \circ \) \( \text{toList} \) where the \( \text{Buildable} \) and \( \text{Foldable} \) instances need not coincide) at every step, but only when we force the value.

Our sequential and parallel composition operators lend themselves to a natural generalization that merges their behaviour in a nontrivial way. We can either generalize parallel composition by first preprocessing through some sort of “running counter”, or generalize sequential composition by “smearing” the transformation by appending our transform step rather than dropping it on the floor. The result is the same – a very general notion of a scan generated from a pair of list algebras.

\[
\text{data} \ \text{Scan} \ f \ g \ a = \text{Scan} \ (g \ a) (f \ (g \ a)) \ \text{deriving} \ Show \\
\text{instance} \ (\text{Buildable} \ f \ (g \ a), \ \text{Buildable} \ g \ a) \Rightarrow \ \text{Buildable} \ (\text{Scan} \ f \ g) \ a \ where \\
\quad \text{unit} = \text{Scan} \ \text{unit} \ \text{unit} \\
\quad \text{insert} \ x \ (\text{Scan} \ xzs \ ys) = \text{Scan} \ xzs' \ (\text{insert} \ xzs' \ ys) \\
\quad \text{where} \ xzs' = \text{insert} \ x \ xzs \\
\text{getScan} \ (\text{Scan} \ a \ _ \ x) = x
\]

We can witness that this captures, for example, the correct notion of a suffix sum.

\[
\text{suffizSum} :: \text{Num} \ a \Rightarrow \ [a] \rightarrow [\text{Sum} \ a] \\
\text{suffizSum} = \text{getScan} \circ \text{fromList}
\]

We can express the more common prefixSum by working on a reversed list, or performing a \( \text{foldl} \) instead of a \( \text{foldr} \).

Sharp-eyed readers may notice that this \( \text{Scan} \) is in fact a type of zygomorphism, yoking a pair of catamorphisms into a compound calculation. [24]

9. Nonempty Builds and Folds

The preceeding has operated with one significant simplification. \( \text{Foldable} \) is required to take both a unit and an action. The unit is necessary to provide a result if the structure is in some sense empty. However, there are a variety of things that are almost list algebras, but have no unital object. A trivial example would be a build into a nonempty list. Other examples are functions such as \( \text{maximum} \) or \( \text{mean} \). We can universally build into such things if we begin with a guaranteed nonempty structure. In his \textit{semigroupoids} library, Edward Kmett introduced such a class, named \( \text{Foldable1} \), which is a full subclass of \( \text{Foldable} \). Here we present a version with a simplified signature for expository purposes. Rather than a pair of a unit and an append function, \( \text{foldl1} \) instead only takes an append.

\[
\text{class} \ \text{Foldable} \ f \Rightarrow \ \text{Foldable1} \ f \ where \\
\quad \text{foldl1} :: (a \rightarrow a) \rightarrow f \ a \rightarrow a
\]

Similarly, we also have a class \( \text{Buildable1} \) that consists of things that do not provide a unital object, but only an insert function. Because we can consider \( \text{Buildable1} \) as the arguments to \( \text{Foldable1} \), by contravariance there are more instances of \( \text{Buildable1} \) than \( \text{Buildable} \), and hence it is a full superclass of the latter.

\[
\text{class} \ \text{Buildable1} f \ a \ where \\
\quad \text{build1} :: ((a \rightarrow f \ a) \rightarrow f \ a) \rightarrow f \ a \\
\quad \text{build1} \ g \ a = \text{insert1} \\
\quad \text{insert1} :: a \rightarrow f \ a \rightarrow f \ a \\
\quad \text{insert1} \ x \ zs = \text{build1} \ (\lambda \text{cons} \rightarrow \text{"cons" } zs) \\
\quad \text{fromList1} :: f \ a \rightarrow [a] \rightarrow f \ a \\
\quad \text{fromList1} \ u \ xs = \text{List.foldl} \ \text{insert1} \ u \ xs
\]

We can now characterize the laws for \( \text{Foldable1} \) as deriving directly from \( \text{Foldable} \), and the laws for \( \text{Buildable1} \) by the same adjoint relationship to \( \text{Foldable} \) as given in the laws for \( \text{Buildable} \). We also note in passing that while \( \text{Foldable} \), \( \text{Buildable} \) pairs provided a subclass of all \( \text{Traversals} \), the restriction to \( \text{Buildable1} \) allows us, albeit awkwardly, to capture the power of any \( \text{Traversable} \) structure. On exploration of this is beyond the scope of this paper.

With this in hand, we can for example provide a \( \text{Buildable1} \) instance for \( \text{Max} \), even though no general \( \text{Buildable} \) instance exists.

\[
\text{newtype} \ \text{Max} \ a = \text{Max} \ \{ \text{getMax} :: a \} \\
\text{deriving} \ (\text{Eq}, \ \text{Ord}, \ \text{Num}, \ \text{Show}) \\
\text{instance} \ \text{Ord} \ a \Rightarrow \ \text{Buildable1} \ \text{Max} \ a \ where \\
\quad \text{insert1} \ x \ (\text{Max} \ y) = \text{Max} \ (\text{max} \ x \ y)
\]

In general, throughout this paper, when \( \text{Buildable1} \) instances exist on \( \text{Buildable} \) objects, we will not provide such definitions, since they follow directly.

10. Extensions and Transformations

Following our compositional approach, we define new data constructors to handle these cases. In these data structures, it transpires that often our unit is more constrained than our insert operation, as the whole idea is that we can vary our inputs and outputs while requiring a properly buildable “data carrier”. So we now complicate things by introducing a \( \text{Buildable1} \) class, a strict superclass of \( \text{Buildable} \) that omits the unit operation. While \( \text{Buildable} \) gives an algebra on all lists, \( \text{Buildable1} \) gives algebras on non-empty lists. In a library situation, one would want a superclass constraint on \( \text{Buildable1} \).

If we recall our \( \text{Count} \) and \( \text{Sum} \) example from above, there’s an obvious missing component. Typically we take a count and a sum
such that we can divide the sum over the count and take the average. Our language of buildables describes how to create structures, but neither how to transform them, nor how to preprocess their inputs.

To perform transformations on our outputs, we introduce a `ThenDo` constructor:

```haskell
data ThenDo f a b = ThenDo (f b → a)

instance Buildable1 f b ⇒ Buildable1 (ThenDo f a) b
  where
    insert1 x (ThenDo xs) = ThenDo (xs ∘ insert1 x)

instance Buildable1 f a ⇒ Buildable (ThenDo f (f a)) a
  where
    unit = ThenDo id
    insert = insert1

mapTD :: (a → b) → ThenDo f a c → ThenDo f b c
mapTD k (ThenDo f) = ThenDo (k ∘ f)

pureTD :: a → ThenDo f a b
pureTD = ThenDo ∘ const

apTD :: ThenDo f (a → b) r → ThenDo f a r → ThenDo f b r
apTD (ThenDo f) (ThenDo x) = ThenDo (λa → (f z) (x z))

contramapTD :: Functor f ⇒ (d → c) → ThenDo f a c → ThenDo f d a
contramapTD f (ThenDo g) = ThenDo (g ∘ fmap f)
```

As `mapThenDo` illustrates, `ThenDo f` is covariantly functorial in its first argument. Furthermore, it is covariantly applicative as well. Finally, as `contramapThenDo` illustrates, if `f` is a `Functor`, then it is contravariantly functorial in its second argument. In fact, though the arguments are reversed from the typical order, `ThenDo f` is a `Profunctor`, and corresponds to `DownStar` in Edward Kmett’s `profunctors` package.

Thus far, the power we have gained with `Buildable` has been in tying the type we accept to the type we produce. Now, via `ThenDo`, we can separate that aspect out again, allowing the types accepted and produced to vary entirely independently, but while still properly tracking the type accepted.

An interesting and informative limitation is that `ThenDo` cannot itself be made `Foldable`. We have moved the parameters we accept into an explicitly contravariant position, and it is impossible to recover them directly. While we build into an `f b`, all we are now able to produce is an `a`. This tells us that, sensibly, `ThenDo` must serve as a “cap” on a chain of computations. Using this new tool, we can turn the running sum/count computation earlier and transform it into a genuine mean.

```haskell
mean :: [Double] → ThenDo (Product Sum (Ann Count)) Double Double
mean = fromList1 ∘ mapTD go unit
  where go (Product (Sum x) (Ann (Count y))) = x / fromIntegral y
```

The newtype noise, granted, makes this look uglier than one would desire. However, we see the same separation as in our earlier `sumCount` code – the core work of the computation, the running calculation, is declaratively expressed and captured in the type signature. Only the final step, which involves the introduction of an arbitrary computation, is written explicitly. Because that work is applied using a standard functorial approach, it is also open to easy algebraic reasoning and refactoring. For example, we can observe, although it does not necessarily make much difference here, than `fromList1 ∘ mapTD go` is in general, by parametricity, equivalent to `mapTD go ∘ fromList1`. When given a larger computation, built up incrementally, such algebraic reasoning can be a great aid in simplification and understanding.

Similarly, we may wish to equip our `Buildable` functors with a general notion of “pre-actions” that allow us to choose whether and how often values are `inserted` to begin with.

```haskell
data FirstDo f a b = FirstDo (b → [a]) (f a)

instance Buildable f a ⇒ Buildable (FirstDo f a) a
  where
    unit = FirstDo ([], (const a))
    insert x (FirstDo xs) = FirstDo (x : xs) (FirstDo x xs)

concatMapFD :: (c → [b]) → FirstDo f a b → FirstDo f a c
concatMapFD (FirstDo g xs) = FirstDo (concatMap g (fmap f xs))
```

This corresponds to some form of `contravariant` left Kan extension along the `List` functor. `FirstDo` is a contravariant functor, so we can now contravariantly map arbitrary functions over our input. Furthermore, it can act as a filter by choosing to return no elements in the list when they fail to pass a given test. And finally it can act to produce a large intermediate set of results that are again reduced. In fact it corresponds to precomposition with a Kleisli arrow, just as the `insert1` function corresponds to a single step of the `concatMap` function. In expressive power, `FirstDo` very closely resembles the “Transducers” introduced by Rich Hickey in Clojure, following on from work by Might and Shivers. [14, 29]

Assembling our entire toolkit, we are able to characterize both pre- and post-processing steps, while tracking the full computational structure of our reductions in the types.

### 11. Maximum Segment Sum

By associating list algebra directly with a typeclass, we have opened the way to reasoning about program transformation in the classic style of the Bird-Meertens Formalism [25]. While we do not claim to provide new insight on how to derive program transformations, our algebra of `Buildable` functors certainly allows us to express programs in this style concisely and declaratively, by pushing the actual code into typeclasses from which the resultant functions are then directly synthesized.

We demonstrate this power by tackling a classic exercise in program transformation – the Maximum Segment Sum problem.[27]. The statement of this problem is: given a list of numbers, find the maximum possible sum of any segment (contiguous run in the list). Of course, when the list of numbers is all positive, the answer is always the sum of the entire list. As an exercise in program transformation, one typically starts from the naive cubic time algorithm that corresponds directly to the specification, and then through a series of “correct moves” derives a linear algorithm. In our case, we will merely illustrate that the linear algorithm can be written cleanly with `Buildable` functors, and that doing so yields a statement of its “meaning” that is more obviously correct.

We begin by introducing a new `Buildable` functor that provides a notion of “bounded” behaviour.

```haskell
data Bound f a = Bound (f a → Bool) (f a) (f a)
getBound (Bound _ _ x) = x

instance Eq (f a) ⇒ Eq (Bound f a) where
  (≡) = (≡) ∘ `on` getBound

instance Ord (f a) ⇒ Ord (Bound f a) where
  compare = compare ∘ `on` getBound

instance Buildable f a ⇒ Buildable (Bound f a) where
  unit = Bound (const False) (const a) (const a)
  insert x (Bound p z z) = Bound p (z ∘ insert1 x) (z ∘ insert1 x)
```

of the first type, however, as an exercise in program transformation, one typically starts from the naive cubic time algorithm that corresponds directly to the specification, and then through a series of “correct moves” derives a linear algorithm. In our case, we will merely illustrate that the linear algorithm can be written cleanly with `Buildable` functors, and that doing so yields a statement of its “meaning” that is more obviously correct.

We begin by introducing a new `Buildable` functor that provides a notion of “bounded” behaviour.

```haskell
instance Ord (f a) ⇒ Ord (Bound f a) where
  compare = compare ∘ `on` getBound

instance Buildable f a ⇒ Buildable (Bound f a) where
  unit = Bound (const False) (const a) (const a)
  insert x (Bound p z z) = Bound p (z ∘ insert1 x) (z ∘ insert1 x)
```
\[ p \cdot x = \text{Bound} p \cdot z \]
\[ \text{otherwise} = \text{Bound} p \cdot z' \]
\[ \text{where} \quad z' = \text{insert} \cdot x \cdot z \]

We now follow the key observation of the efficient Maximum Segment Sum algorithm—our answer has a lower bound at zero, as even if all values are negative, the empty segment still has a zero sum.

Hence, we begin with building into a Bound Sum capped at zero. This yields us, going from the back of the list, a “resetting” running sum, that every time it dips below zero is “pulled back up”. Thinking inductively, adding values to the initial portion of that calculation (i.e. the tail of the list), could only give us a higher answer, not a lower. Hence, we need not consider cases where the initial segment is in any way truncated. Meanwhile, adding values to the final portion of that calculation (the head of our list) could create problems, if such values were negative. Hence, we must consider the “best” prefix possible out of all suffixes to the list. The suffixes are considered in turn by our Scan introduced above, yielding the following type and implementation:

\[
\text{maximumSegmentSum} :: (\text{Num} \cdot a, \text{Ord} \cdot a) \Rightarrow [a] \rightarrow \text{Scan} \cdot \text{Max} \cdot (\text{Bound} \cdot (\text{Sum})) \cdot a
\]
\[
\text{maximumSegmentSum} = \text{fromList1} \cdot (\text{Scan} \cdot (\text{Bound} < 0) \cdot 0 \cdot 0) \cdot (\text{Max unit})
\]

This is to say, as the types tell us, maximum segment sum is simply a scan into the maximum of a bounded sum.

The need to use Buildable1 make the above a bit uglier than we would like, in conjunction with our explicit use of Bound. We can also argue that we are projecting our \(a\) into a different number system, subject to saturation at zero, and give an explicit construction for this. This tidies up things quite a bit.

\[
\text{newtype Sat} \cdot a = \text{Sat} \cdot a \text{ deriving} \quad (\text{Show}, \text{Eq}, \text{Ord})
\]
\[
\text{sat} \cdot x \quad | \quad x < 0 = \text{Sat} \cdot 0
\]
\[
\text{otherwise} = \text{Sat} \cdot x
\]

\[
\text{instance} \quad (\text{Ord} \cdot a, \text{Num} \cdot a) \Rightarrow \text{Num} \cdot (\text{Sat} \cdot a) \text{ where}
\]
\[
\text{fromInteger} = \text{sat} \circ \text{fromInteger}
\]
\[
\text{Sat} \cdot x + (\text{Sat} \cdot y) = \text{sat} \cdot (x + y)
\]
\[
\text{Sat} \cdot x \cdot (\text{Sat} \cdot y) = \text{sat} \cdot (x \cdot y)
\]
\[
\text{Sat} \cdot x \cdot (\text{Sat} \cdot y) = \text{sat} \cdot (x \cdot y)
\]

\[
\text{msss} :: (\text{Num} \cdot a, \text{Ord} \cdot a) \Rightarrow [a] \rightarrow \text{Scan} \cdot \text{Max} \cdot \text{Sum} \cdot (\text{Sat} \cdot a)
\]
\[
\text{msss} = \text{fromList1} \cdot (\text{Scan} \cdot 0 \cdot 0) \circ \text{map Sat}
\]

In both cases, we have given a succinct representation of the classic efficient Maximum Segment Sum problem, where the “meaning” of the behaviour can be read right off the type. This rendition of the Maximum Segment Sum problem was in fact one of the motivating examples for the introduction of Zygomorphisms to begin with.[24]

We can compare this description of the algorithm with the explicit encoding, for example as given by Shin-Cheng Mu.[27]

\[
\text{msssClassic} = \text{snd} \circ \text{foldr} \cdot \text{step} \cdot (0, 0) \text{ where}
\]
\[
\text{step} \cdot x \cdot (p, s) =
\]
\[
(0 \cdot \text{max} \cdot (x + p), 0 \cdot \text{max} \cdot (x + p) \cdot \text{max} \cdot s)
\]

While this certainly is elegant, it would argue that it not necessarily as descriptive. Furthermore, the Buildable rendition of the algorithm lends itself to modular replacement of components, for the purpose of obtaining derived algorithms or simply of “unpacking” the behaviour of a given function. For example, suppose we do not understand the meaning of our running saturated sum, and wish to see the intermediate results. We can do this just by replacing our ‘Max’ Buildable by a list.

\[
\text{preMss} :: (\text{Ord} \cdot a, \text{Num} \cdot a) \Rightarrow [a] \rightarrow \text{Scan} \cdot [] \cdot \text{Sum} \cdot (\text{Sat} \cdot a)
\]
\[
\text{preMss} = \text{fromList} \circ \text{map Sat}
\]

If we want an even more basic insight into the data that the saturated sum is derived from, and how it relates to successive tails of our list, we need only swap out our other functor as well in order to obtain the “free” structure of a Scan:

\[
\text{prePreMss} :: [a] \rightarrow [] \cdot [] \cdot a
\]
\[
\text{prePreMss} = \text{fromList}
\]

This in turn corresponds to the tails function, and from our laws we can “read off” that the Maximum Segment Sum algorithm itself is related to the tails function by an adjunction.

12. Calculating Parallel Computations

We have imposed no constraints on our Buildable functors barring the adjoint relationship with Foldable functors. This is at variance with presentations of similar constructions. Edward Kmett’s reducers library requires that Reducers be instances of Semigroup – i.e. equipped with an associative binary operator. Similarly for reducers in Clojure, which are monoidal by nature, etc. In general, this line of work, all focused on associative operations, flows from Guy Steele’s 2009 ICFP invited talk “Organizing Functional Code for Parallel Execution”, which motivated monoidal structures and operations as providing a general skeleton for parallel decomposition of computations.

In particular, while we can consider a Buildable as the pair of a unit object and an insert operation, we have provided no way of relating two Buildable objects of the same type. However, we can produce a derived operation that specializes to something appropriate - \text{transpend}, which merges any pair of Buildable and Foldables. When we force the two functors to be the same, this operation becomes a “merge”.

\[
\text{transpend} :: (\text{Buildable} \cdot g \cdot a, \text{Foldable} \cdot f) \Rightarrow 
\]
\[
f \cdot a \rightarrow g \cdot a \rightarrow g \cdot a
\]
\[
\text{transpend} \cdot x \cdot ys = \text{build} \cdot (\lambda c \cdot n \rightarrow \text{foldr} \cdot c \cdot ys) \cdot x
\]
\[
\text{merge} :: (\text{Buildable} \cdot f \cdot a, \text{Foldable} \cdot f) \Rightarrow 
\]
\[
f \cdot a \rightarrow f \cdot a \rightarrow f \cdot a
\]
\[
\text{merge} = \text{transpend}
\]

One form of deriving parallel computations relies on a principled rearrangement and reassociation of operations from an existing serial implementation. In such a situation, it is helpful not only to have a Monoid, but also one that interacts with underlying lists in a particular specified way. In particular, if we have a retract such that \text{fromList} \circ \text{toList} \equiv \text{id}, then we can consider writing \text{insert} \cdot ax \text{ as fromList} \cdot (a \cdot \text{toList} \cdot zs), which is equivalent to \text{merge} \circ \text{singleton}. In the case where \text{merge} \circ \text{singleton} is indeed equivalent to \text{insert}, then we have the property that \text{fromList} is a list homomorphism – i.e. \text{fromList} \cdot x \cdot \text{merge} \cdot \text{fromList} \cdot ys \equiv \text{fromList} \cdot (xs + ys). This is the same condition as requiring \text{merge} to be associative – i.e. yielding a Monoid on \( f \cdot a \).

The existing Monoid instance for set is already such a list homomorphism. The obvious instances for things such as Ann, Count and Sum likewise. It is also the case that our compositions of Buildables can be equipped with Monoid instances derived from such instances on their underlying Buildables. For example:

\[
\text{instance Monoid} \circ f \cdot (g \cdot a) \Rightarrow 
\]
\[
\text{Monoid} \circ (\text{Compose} \cdot f \cdot g) \text{ where}
\]
\[
\text{mappend} \circ (\text{Compose} \cdot x) \cdot (\text{Compose} \cdot y) = 
\]
\[
\text{Compose} \circ (x \cdot \text{mappend} \cdot y)
\]
\[
\text{mempty} = \text{Compose} \cdot \text{mempty}
\]
\[
\text{instance} \quad (\text{Monoid} \circ f \cdot a, \text{Monoid} \circ (g \cdot a)) \Rightarrow 
\]
\[
\text{Monoid} \circ (\text{Product} \cdot f \cdot g) \text{ where}
\]
\[
\text{mappend} \cdot (\text{Product} \cdot x) \cdot (\text{Product} \cdot z') \cdot (\text{Product} \cdot y') = 
\]
\[
\text{Product} \circ (x \cdot \text{mappend} \cdot y') \cdot (y' \cdot \text{mappend} \cdot z')
\]
\[
\text{mempty} = \text{Product} \cdot \text{mempty} \cdot \text{mempty}
\]
We can write a naive function that chops a given list into a tree, and can operate as a list homomorphism; it suffices to establish that \( \text{mappend} \) in infix as provided by our laws.

As above, to prove that a \textit{Buildable} such as \textit{Scan} can operate as a list homomorphism, it suffices to establish that \( \text{mappend} \circ \text{singleton} \) is equivalent to \textit{insert}. For brevity, we write \( \text{mappend} \) infix as \( \langle \rangle \), and \textit{singleton} as \textit{inj}. We also make use of the property that \( \text{fmap} f \ (\text{inj} x) \equiv \text{inj} \circ f \) -- i.e., \textit{inj} is a natural transformation from the identity functor. This guarantee is provided by our laws.

The existence of a \textit{Monoid} instance for \textit{Scan} means that we can now build up our sequential scans from any parenthesisization of our list, and furthermore we can execute this in parallel. To witness this we can write a function that chops a given list into a tree, and then merges it.

```haskell
fromListPar :: (Buildable f a, Monoid (f a)) => [a] -> f a
fromListPar = go o map singleton
where
go [] = unit
  go [x] = x
  go xs = go (mp xs)

mp (a : b : zs) = a `mappend` b : mp zs
mp zs = zs
```

Now the suffix sum can be written as follows:

```haskell
suffixSumMerge :: Num a => [a] -> [a]
suffixSumMerge = getScan o fromListPar
```

If we annotate \textit{fromListPar} with parallelism operators, this yields a new derivation of the famous Parallel Prefix Sum algorithm, which given sufficient processors operates on \( O(\log n) \) time.

We can write a naive \textit{fromListPar} like so:

```haskell
chunksOf n = unfoldr go where
  go [] = Nothing
  go x = Just (splitAt n x)
fromListPar :: (Buildable f a, Monoid (f a), NFDData (f a)) => [a] -> f a
fromListPar = go o map fromList o chunksOf 1000
where go [] = unit
  go [x] = x
  go xs = go (mp xs)

mp (a : b : zs) = let ab = (a `mappend` b) in
  ab `deepseq` (mpzs `par` ab : mpzs)
mpzs = zs

suffixScanPar :: (Num a, NFDData a) => [a] -> Scan [] Sum a
suffixScanPar = fromListPar
```

Tuning code for efficient parallelism is an entirely different art than the pure parallel decomposition of algorithms. However, with lists of size \( 10^7 \), the above implementation was sufficient to provide a small parallel speedup on the author’s machine (from 18s with one processor to 14s with three).

As discussed, the \textit{Monoid} for \textit{Scan} is derived from monoidal properties of the underlying \textit{Buildables}. Returning to the Maximum Segment Sum structure from the previous section, we can observe that the \textit{Sat} representation makes very explicit why that structure is not a \textit{Monoid} and thus does not naively parallelize. In particular, saturated arithmetic is famously nonassociative -- i.e. in a saturated setting, \( (1 + -5) + 5 = 5 \) while \( 1 + (-5 + 5) = 1 \). Hence \textit{Sat} cannot be given a \textit{Monoid} instance, an nor can structures such as \textit{Scan} derived from it. This formulation also makes very clear that an associativity condition is equivalent to a requirement that computations forget directional structure. Hence, it also makes clear that the core of the Maximum Segment Sum algorithm is structured around essential use of precisely such structure.

### 13. Windowed Algorithms; Monoids Need Not Be List Homomorphisms

We have argued that many folds are best thought of naturally not as monoids--Maximum Segment Sum, for example, has an innately directional structure, although a more complex monoidal structure may be derived for it. Similarly while every monoid yields a list homomorphism, there are many useful instances of monoids are not list homomorphisms. This becomes evident when we turn our attention to windowed computation.

A test case here is the problem of a windowed sum (from which we can easily derive the commonly useful windowed mean). That is: given a list \([1..10]\) return a list such as \([1 + 2 + 3 + 4 + 3 + 4 + 5 + 3 + 4 + 5 + 6 \ldots]\). Algorithms exist for computing such results in \( O(n) \) time given any associative operation. Using the power of our monoidal buildable scan, we are able not only to do so, but also to, with sufficient processors, calculate the result in \( O(m + \log (n/m)) \) time, where \( m \) is our window size. To do so, we introduce a structure that is effectively the pullback of two symmetric windowed scans. This involves first creating a \textit{ScanL} to mirror our existing \textit{Scan}, and then introducing a \textit{ScanSeg} structure, that is a triple of a left scan (adjoint to \textit{inits}), a pure carrier, and a right scan (adjoint to \textit{tails}). The intuition is we have two “fringes” that consist of a portion of a completed windowed calculation, with all information available this far, and in between the completed section of calculations. When we compose such structures monoidally we line up the fringes and merge them using the underlying associative (monoidal) operation. One notion of such structures is shown below.

```haskell
data Rev f a = Rev [a] (f a) deriving Show
fromRev (Rev rs) = rs

instance (Buildable f a) => Buildable (Rev f) a where
  unit = Rev []
  insert x (Rev xs) =>
    Rev (x : xs) (fromList (reverse (x : xs)))

instance Monoid (f a) => Monoid (Rev f a) where
  mempty = Rev []
  mappend (Rev xs r) (Rev ys r') =>
    Rev (xs ++ ys) (r `deepseq` r')

instance Foldable f => Foldable (Rev f) where
  foldr c z (Rev xs) = foldr c z xs

type ScanL f g a = Rev (Scan (Rev f) (Rev g)) a
getScanL = fmap fromRev o fromList o getScan o fromRev

data ScanSeg g a =
```

Submission for IFL 2014
Submission for IFL 2014

cost of merging any two $O$-dowed sum as follows:

$$\text{ScanSeg} \quad (\text{ScanL} \, [] \, g \, a) \quad (g \, a) \quad (\text{Scan} \, [] \, g \, a)$$

deriving Show

instance (Buildable $g$ $a$) $\Rightarrow$ Buildable (ScanSeg $g$) $a$

where

insert $x$ (ScanSeg $h$ $m$ $n$ $t$) =

ScanSeg (insert $x$ $h$) $m$ (insert $x$ $t$)

$unit$ = ScanSeg $unit$ $unit$ $unit$

instance (Buildable $g$ $a$, Monad ($g$ $a$)) $\Rightarrow$

Monoid (ScanSeg $g$) $a$

$mempty$ = $unit$

$mappend$ $x$@((ScanSeg $h$ $m$ $n$ $t$)) $y$@((ScanSeg $h'$ $m'$ $n'$ $t'$))

| isEmpty $x$ = $y$
| isEmpty $y$ = $x$
| otherwise =

ScanSeg $h$ ($m$ <> $n$ <> merged <> $m'$) $t'$

where

merged = $zipMon$

$(getScan t)$

$(getScanL h')$

$zipMon$ ($x$ : $xs$) ($y$ : $ys$) =

$x$ <> $y$ : $zipMon$ $xs$ $ys$

$zipMon$ $[]$ $ys$ = $ys$

$zipMon$ $xs$ $[]$ = $xs$

isEmpty (ScanSeg (Rev $[]$) $[]$ (Scan $[]$ $[]$)) =

True

isEmpty _ = False

ScanSeg can be verified as a monoid by the following argument.

Definitionally, left and right appends to the unit object are identity. Outside of this case, the left fringe and left middle are always preserved by a right append, and the right fringe and right middle are always preserved by a left append. The “merged middle” is determined solely by the left fringe of the right object and the right fringe of the left object. Thus, we can verify that there is no “action at a distance” between $a$ and $c$ in the expression $a <> b <> c$. This in turn is enough to guarantee associativity.

Because our goal is to demonstrate the gist of an algorithm and not provide an implementation efficient in all respects, we have introduced some simplifications in the above code. First, we work with a mildly inefficient $Rev$ type to capture a left scan in terms our already given Scan, rather than introduce a new type that fuses this away, or coupling our Buildable class with a BuildableR that provides an insertRight or “snoc” function.

Second, the ScanSeg we give is specialized to work over lists as our “structure carrier” algebra, rather than any Buildable. We do so to avoid the need for a fully generalized notion of “zipping.” It is worth noting in passing that when $a$ is a Monoid, structures such as $[a]$ carry at least two monoidal structures – one derived from the Monoid on list given by append, and one derived from the structure induced by zipping; i.e., horizontal and vertical composition. However, an exploration of these two structures and their generalizations is outside of the scope of this paper.

Furthermore, the specialization to lists gives us a complexity cost in the price of monoidal append. An efficient implementation of the ScanSeg structure would be better served by a carrier with $O(1)$ mappend.

With those caveats, we can nonetheless write a sliding windowed sum as follows:

slidingSum :: Num $a$ $\Rightarrow$ Int $\rightarrow$ $[a]$ $\rightarrow$ ScanSeg $Sum$ $a$

slidingSum $n$ $xs$ =

$moneat$ (map fromList (chunksOf (n - 1) $xs$))

With an efficient implementation we can observe that the overall cost of merging any two ScanSegs is $O(m)$ in our window size, and furthermore that building a ScanSeg directly costs $O(m)$. The number of builds and merges we need to perform is $O(n/m)$ in the size of our list, and hence the entire operation is $O(n)$ with no varying cost due to window size. Additionally, unlike many implementations of sliding sums, ours requires no “subtract”, “negate”, or other inverse operation, and so generalizes across all monoids. Finally, we can observe that the monoidal “divide-and-conquer” strategy in play is also amenable to parallelization precisely as with Parallel Prefix Sum. In ideal conditions, this provides the $O(n \cdot \log (m/n))$ bounds as promised.

Not only is ScanSeg not a list homomorphism, the entire point is that it is not. In particular, different partitions of our input list give results of different window sizes. Hence we see that for associative computations operating only on the sequential ordering of a list, list homomorphisms are a key tool, but for computations requiring equipping lists with some stronger sort of overall global structure, they are not.

14. 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 $* \rightarrow *$, 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 we 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.” Steele’s talk in fact also provided a similar construction of Parallel Prefix Sum. In that case, he explored using “monoid-cached trees” – structures that are built with an associated summary Monoid. This is in contrast to our approach here that builds structures over Monoids (or more generally, arbitrary Buildables).

As discussed, the connection of Buildable functors to list algebras relates very strongly to the algebra of programming in the tradition of the Bird-Merteens Formalism [3, 24, 25]. The Parallel Prefix Sum algorithm has already been studied in this line of work as well [10] There is even more work in this tradition studying Maximum Segment Sum, with some notable examples being [6, 11, 24, 26]. The problem of windowed monoidal computation appears to be surprisingly less studied, although it is tackled in passing by [9]. In personal communication, Edward Kmett and Daniel Peebles have observed that this can be solved with the use of a monoid-indexed dequeue, such as provided by a Finger Tree [18]. However, we are not aware of prior work solving this problem in parallel as presented here.

15. Conclusion and Future Work

In the course of this paper, initially motivated by seeking to provide structure to Foldable objects, we have introduced a new typeclass, Buildable, and associated laws. By directly representing list algebras, Buildable turns out to be quite good to think with. Com-
position of Buildable functors has allowed us to capture notions of parallel, concurrent, and streaming computation. Furthermore, it has allowed us to represent algorithms in the program calculation tradition in a very direct form. Additionally, we have explored the power of loosening prior restrictions and assumptions—looking at Buildables that need not be Monoids and Monoids that need not be list homomorphisms in a unified framework. The typeclass representation of list algebras has been especially helpful here, allowing constraints to filter downwards, while instantiation of the typeclasses themselves synthesizes code bottom up.

A number of future lines of development are possible. The beginning of this paper observed that foldMap is more general than foldl or foldr as it allows summary of structures potentially extending infinitely in both directions. It would be interesting to explore how to alter Buildable to mirror that power, and to explore what structures and laws arise in such a case. Along similar lines, the relationship of Buildable1 and Foldable1 as well as their associated laws could stand further exploration. An outcome of this work would be a library that takes the main results of this paper into a general purpose toolkit.

One further avenue of exploration would be to move towards a Profunctor characterization of Buildable types—making use of binary type constructors separating out what they accept into the first type argument and what they “fold into” the second position. We believe the machinery explored in the “Extensions and Transformations” section would work out much more elegantly in such a case. Another common functional model of streaming computation comes via the tradition of work on Iterates and later Pipes and Machines [13, 21, 22]. We would like to explore how to slot these into the Buildable formulation and see if it provides a useful generalization—in such a situation, the Profunctor formulation again seems like a good candidate.

With regards to the algorithms explored, it seems like there should be a way to combine windowed decomposition techniques with the machinery introduced for Parallel Prefix Sum to obtain a derivation of a parallel version of Maximum Segment Sum and similar problems. This relates to the “near homomorphism” approach to such a problem introduced in [7]. We would also like to see if our approach could be extended to tackle problems such as Maximum Segment Density, as explored in [9].

A number of computations provided have followed the structure of a hylomorphism, first conceptually “unfolding” a list into a more deeply nested structure, and then performing nested reduction. It would be worthwhile to generalize that pattern within our framework, perhaps seeking to capture a general class of structures with a coalgebra f a → f (f a) derived in interaction with the list functor. This would in turn generalize notions of “mapReduce” to a more general “unfoldReduce” that captures the skeleton of a wider variety of computations.

Relatedly, the current work has focused on structures adjoint to list. It would be worth exploring structures adjoint to other functors, such as binary trees. Just as our current formulation allowed us to obtain a certain class of relative monads and restricted traversals, we suspect that other, related typeclass pairs could be used to explore a broader range of structures.

Acknowledgments

Thanks to Jost Berthold who encouraged me to pursue work on this project, and to Duncan Coutts, Edward Kmett, and Sjoerd Visscher for illuminating and encouraging discussions. Special thanks also to Jeff Polakow for providing invaluable editorial input, and to the attendees of IFL 2014 for their useful feedback and probing questions.

Note. All code in this paper compiles and runs in GHC 7.8.3 with standard imports, and with the requirement of hiding foldr from the Prelude. A few obvious instances such as Buildable1[] and some instances of Buildable1 have been omitted for brevity. Extensions used at various points in this paper include RankNTypes, MultiParamTypeClasses, FlexibleInstances, FlexibleContexts, StandaloneDeriving, GADTs, NoMonomorphismRestriction, GeneralizedNewtypeDeriving, and TypeFamilies.

References


