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.Exts*.

```
toList :: Foldable t ⇒ t a → [a]
toList t = build (λc n → foldr c n t)

build :: (forall b ∘ (a → b → 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 *Foldable*s 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 (λc n → c x n)

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

  insert :: a → f a → f a
  insert x xs = build (λcons 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, 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.

What property should we expect from the interaction of our two functions *fromList* and *toList*? 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 $thereBack = toList \circ Set.fromList$ 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 $fromList \circ toList$ is a split idempotent. More generally, we can consider the functorial nature of *Foldable* and *Buildable* to produce a set of laws, proceeding from the notion of an *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[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 where 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 $F : D \to C$ and $G : C \to D$, 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 $f : f\ a \to f\ b$, there is a function $g : [a] \to [b]$ such that $f \circ fromList :: [a] \to f\ b$ is isomorphic to $fromList \circ g :: [a] \to f\ b$. 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.

```
mapToList :: (Buildable f a, Foldable f) ⇒
  (f a → f b) → [a] → [b]
mapToList f = toList ∘ f ∘ fromList
mapFromList :: (Foldable f, Buildable f b) ⇒
  ([a] → [b]) → f a → f b
mapFromList g = fromList ∘ g ∘ toList
```

In this formulation, the law is that $mapFromList \circ mapToList$ must be equivalent to identity. Expanding this out we see the condition is that for all functions f from $f\ a \to f\ b$, it is the case that:

$$f \equiv fromList \circ toList \circ f \circ fromList \circ toList$$

Many adjoints follow the lead of *Set* and include some notion of a retract, in which case $fromList \circ toList$ 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 $[a] \to [b]$ such that $\forall x.\exists y.\ f\ x \equiv toList\ y$) 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
    unit = AnnotateL 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 $fromList \circ toList$ is equivalent to identity. When so transliterated, any function of type $AnnotateL\ m\ a \to AnnotateL\ m\ b$ 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.

$$toList \circ fromList \circ toList \equiv toList$$
$$fromList \circ toList \circ fromList \equiv fromList$$

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.

```
instance (Foldable f, Foldable g) ⇒
  Foldable (Product f g) where
    foldr f z (Product xs _) = foldr f z xs

instance (Foldable f, Foldable g) ⇒
  Foldable (Product f g) where
    foldr f z (Product _ ys) = foldr f z ys
```

As *Product* is a generalization of our $AnnL$ above, we can observe by the same argument that this is a left adjoint. However, in the case that we take the product of two right adjoints, then our derived structure will obey neither the left nor right adjoint laws. However, if our underlying structures obey the weak idempotence laws, then so to will either instance.

Dually, we observe that Coproducts face the opposite choice – a single *Foldable* instance but two legitimate choices for *Buildable*.

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

instance (Foldable f, Foldable g) ⇒
  Foldable (Coproduct f g) where
    foldr f z (InL xs) = foldr f z xs
    foldr f z (InR ys) = foldr f z ys


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

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

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 $g :: a \to b$, $map\ g \circ toList \equiv toList \circ fmap\ g$. 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, $fmap\ g \circ fromList \equiv fromList \circ map\ g$, and as a consequence $fmap\ g \circ singleton \equiv singleton \circ g$.

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.

[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 *Foldable*, *Buildable* pair, a corresponding monad.

```
newtype WrapBuild f a = WrapBuild { getBuild :: f a }
  deriving (Foldable)
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
class RMonad m where
  rreturn :: Suitable m a ⇒ a → m a
  rbind :: (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 where
    constraints = BC
withResConstraints :: Suitable m a ⇒
  (Constraints m a → m a) → m a
withResConstraints f = f constraints
instance Foldable f ⇒ RMonad (WrapBuild f) where
  rreturn x = withResConstraints
    (λBC → singleton x)
  rbind s f = withResConstraints
    (λBC → foldr
      (λa s′ → foldr insert s′ (f a)) unit s)
  rmap f ma = withResConstraints
    (λBC → foldr
      (λx xs → insert (f x) xs) unit ma)
```

The generated instance for *WrapBuild Set* behaves just as one would hope a *Set* monad would behave. This in fact demonstrates that one adjoint the *Set* monad decomposes to is precisely that between Set and List. The class of monads that *WrapBuild* generates are in fact all "relative monads along the list functor", with the sense of relative monad being that described by Altenkirch, Chapman, and Uustalu. [2]. (We also note, in passing, that a left-adjoint *Foldable*, *Buildable* pair correspondingly yields a restricted comonad).

In a similar fashion we can define a restricted *Traversable* class, and equip any right adjoint *Foldable*, *Buildable* pair with an *RTraversable* instance.

```
withResConstraints1 :: Suitable m a ⇒
  (Constraints m a → f (m a)) → f (m a)
withResConstraints1 f = f constraints
class RTraversable t where
  rtraverse :: (Suitable t a, Suitable t b, Applicative f) ⇒
    (a → f b) → t a → f (t b)
instance Foldable f ⇒ RTraversable (WrapBuild f) where
  rtraverse g x = withResConstraints1
    (λBC →
      (fmap (foldr insert unit) ∘ traverse g ∘ toList) x)
```

It is well known that *Foldable* is a superclass of *Traversable*. This demonstrates that right adjoint *Foldable*, *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 Buildables

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.

```
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
  where
    unit = Ann (Count 0)
    insert x (Ann (Count m)) =
      Ann (Count (m + 1))
instance Foldable (Ann m) where
  foldr f z _ = z
newtype Sum a = Sum { getSum :: a }
  deriving (Eq, Ord, Num)
instance Num a ⇒ Buildable Sum a where
  unit = Sum 0
  insert x (Sum xs) = Sum (x + xs)
instance Foldable Sum where
  foldr f _ (Sum x) = f z x
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 computaiton 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 Compositon

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 *Functor* instance, the following instance is possible.

```
newtype Compose f g a = Compose (f (g a))
instance (Functor f, Buildable f (g a), Buildable g a) ⇒
  Buildable (Compose f g) a where
    unit = Compose unit
    insert x (Compose xs) =
      Compose (fmap (insert x) xs)
```

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 for example to evaluate a number of predicates on a substring simultaneously. While *Product* provides a notion of trivial concurrency, we can describe *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 *Product* type, the *Buildable* instance is quite different, taking advantage of the relationship to *Foldable*.

```
data Seq f g a = Seq (f a) (g a)
transform :: (Buildable g a, Foldable f) ⇒ f a → g a
transform xs = build (λc n → foldr c n xs)
instance (Foldable f, Buildable f a, Buildable g a) ⇒
  Buildable (Seq f g) a where
    unit = Seq unit unit
    insert x (Seq xs ys) =
      let xs′ = insert x xs
      in Seq xs′ (transform xs′)
```

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 $fromList \circ toList$ where the *Buildable* and *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" out the a 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.

```
data Scan f g a = Scan (g a) (f (g a)) deriving Show
instance (Buildable f (g a), Buildable g a) ⇒
  Buildable (Scan f g) a where
    unit = Scan unit unit
    insert x (Scan xs ys) = Scan xs′ (insert xs′ ys)
      where xs′ = insert x xs
getScan (Scan _ x) = x
```

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

```
suffixSum :: Num a ⇒ [a] → [Sum a]
suffixSum = getScan ∘ fromList
```

We can express the more common prefixSum by working on a reversed list, or performing a *foldl* instead of a *foldr*.

Sharp-eyed readers may notice that this *Scan* is in fact a type of *zygomorphism*, yoking a pair of catamorphisms into a compound calculation. [24]

9. Nonempty Builds and Folds

The preceding has operated with one significant simplification. *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 *maximum* or *mean*. We can universally build into such things if we begin with a guaranteed nonempty structure. In his *semigroupoids* library, Edward Kmett introduced such a class, named *Foldable1*, which is a full subclass of *Foldable*. Here we present a version with a simplified signature for expository purposes. Rather than a pair of a unit and an append function, *foldr1* instead only takes an append.

```
class Foldable f ⇒ Foldable1 f where
  foldr1 :: (a → a → a) → f a → a
```

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

```
class Buildable1 f a where
  build1 :: ((a → f a → f a) → f a) → f a
  build1 g = g insert1
  insert1 :: a → f a → f a
  insert1 x xs = build1 (λcons → x `cons` xs)
  fromList1 :: f a → [a] → f a
  fromList1 u xs = List.foldr insert1 u xs
```

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

With this in hand, we can for example provide a *Buildable1* instance for *Max*, even though no general *Buildable* instance exists.

```
newtype Max a = Max {getMax :: a}
  deriving (Eq, Ord, Num, Show)
instance Ord a ⇒ Buildable1 Max a where
  insert1 x (Max y) = Max (max x y)
```

In general, throughout this paper, when *Buildable1* instances exist on *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 buidable "data carrier". So we now complicate things by introducing a *Buildable1* class, a strict superclass of *Buildable* that omits the unit operation. While *Buildable* gives an algebra on all lists, *Buildable1* gives algebras on non-empty lists. In a library situation, one would want a superclass constraint on *Buildable1*.

If we recall our *Count* and *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:

```
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 (λz → (f z) (x z))
contramapTD :: Functor f ⇒
  (d → c) →
  ThenDo f a c →
  ThenDo f a d
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 $contraMapTD$ 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.

```
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 functoral 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 $fromList1 \circ mapTD\ go$ is in general, by parametricity, equivalent to $mapTD\ go \circ fromList1$. 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.

```
data FirstDo f a b = FirstDo (b → [a]) (f a)
instance Buildable f a ⇒ Buildable (FirstDo f a) a
  where
    unit = FirstDo (:[]) unit
    insert x (FirstDo f xs) = FirstDo f (insert x xs)
concatMapFD :: (c → [b]) → FirstDo f a b →
  FirstDo f a c
concatMapFD f (FirstDo g xs) =
  FirstDo (concatMap g ∘ 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 algebras directly with a typeclass, we have opened the way to reasoning about program transformation in the classic style of the Bird-Merteens 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.

```
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) unit unit
  insert x (Bound p z xs)
```

```
  | p xs' = Bound p z z
  | otherwise = Bound p z xs'
  where xs' = insert x xs
```

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:

```
maximumSegmentSum :: (Num a, Ord a) ⇒
  [a] → Scan Max (Bound (Sum)) a
maximumSegmentSum =
  fromList1 (Scan (Bound (<0) 0 0) (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.

```
newtype Sat a = Sat a deriving (Show, Eq, Ord)
sat x | x < 0 = Sat 0
  | otherwise = Sat x
instance (Ord a, Num a) ⇒ Num (Sat a) where
  fromInteger = sat ∘ fromInteger
  (Sat x) + (Sat y) = sat (x + y)
  (Sat x) − (Sat y) = sat (x − y)
  (Sat x) ∗ (Sat y) = sat (x ∗ y)
mss :: (Num a, Ord a) ⇒ [a] → Scan Max Sum (Sat a)
mss = fromList1 (Scan 0 0) ∘ 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]

```
mssClassic = snd ∘ foldr step (0, 0) where
  step x (p, s) =
    (0 `max` (x + p), 0 `max` (x + p) `max` s)
```

While this certainly is elegant, we 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.

```
preMss :: (Ord a, Num a) ⇒ [a] → Scan [] Sum (Sat a)
preMss = fromList ∘ 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*:

```
prePreMss :: [a] → Scan [] [] a
prePreMss = 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 *Reducer*s 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 – *transpend*, which merges any pair of *Buildable* and *Foldables*. When we force the two functors to be the same, this operation becomes a "merge".

```
transpend :: (Buildable g a, Foldable f) ⇒
  f a → g a → g a
transpend xs ys = build (λc n → foldr c ys xs)
merge :: (Buildable f a, Foldable f) ⇒
  f a → f a → f a
merge = 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 $fromList \circ toList \equiv id$, then we can consider writing *insert a xs* as $fromList\ (a : toList\ xs)$, which is equivalent to $merge \circ singleton$. In the case where $merge \circ singleton$ is indeed equivalent to *insert*, then we have the property that *fromList* is a list homomorphism – i.e. $fromList\ xs\ `merge`\ fromList\ ys \equiv fromList\ (xs +\!\!+ ys)$. This is the same condition as requiring *merge* to be associative – i.e. yielding a *Monoid* on *f 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 *Buildable*s can be equipped with *Monoid* instances derived from such instances on their underlying *Buildable*s. For example:

```
instance Monoid (f (g a)) ⇒
  Monoid (Compose f g a) where
    mappend (Compose x) (Compose y) =
      Compose (x `mappend` y)
    mempty = Compose mempty
instance (Monoid (f a), Monoid (g a)) ⇒
  Monoid (Product f g a) where
    mappend (Product x y) (Product x' y') =
      Product (x `mappend` x') (y `mappend` y')
    mempty = Product mempty mempty
```

The instance for *Seq* follows the same pattern. In all cases, the underlying list homomorphisms let us push through the proofs that the derived *Monoid* instances abide properly with our *Buildable* and *Foldable* instances. In the case of compose and product, such proofs are trivial. In the case of *Scan*, the merge is more subtle and the equational reasoning is worth spelling out.

```
instance (Monoid (f (g a)), Monoid (g a), Functor f) ⇒
  Monoid (Scan f g a) where
    mappend (Scan xs ys) (Scan xs' ys') =
      Scan (xs <> xs') (fmap (<>xs') ys <> ys')
    mempty = Scan mempty mempty
```

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

```
mappend (singleton x) (Scan xs ys) =
mappend (Scan (inj x) (inj (inj x))) (Scan xs ys) =
Scan (inj x <> xs) (fmap (<>xs) (inj (inj x) <> ys)) =
Scan (insert x xs) (inj (inj (x <> xs)) <> ys) =
Scan (insert x xs) (insert (inj (x <> xs)) ys) =
Scan (insert x xs) (insert (insert x xs) ys) =
insert x (Scan xs ys)
```

The existence of a *Monoid* instance for *Scan* means that we can now build up our sequential scans from any parenthisization 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.

```
fromListMerge :: (Buildable f a, Monoid (f a)) ⇒
  [a] → f a
fromListMerge = go ∘ map singleton
  where
    go [] = unit
    go [x] = x
    go xs = go (mp xs)
    mp (a : b : xs) = a `mappend` b : mp xs
    mp xs = xs
```

Now the suffix sum can be written as follows:

```
suffixSumMerge :: Num a ⇒ [a] → [Sum a]
suffixSumMerge = getScan ∘ fromListMerge
```

If we annotate *fromListMerge* with parallelism operators, this yields a new derivation of the famous Parallel Prefix Sum algorithm, which given sufficient processors operates on $\mathcal{O}(\log n)$ time. We can write a naive *fromListPar* like so:

```
chunksOf n = unfoldr go where
  go [] = Nothing
  go x = Just (splitAt n x)
fromListPar ::
  (Buildable f a, Monoid (f a), NFData (f a)) ⇒
  [a] → f a
fromListPar = go ∘ map fromList ∘ chunksOf 1000
  where go [] = unit
    go [x] = x
    go xs = go (mp xs)
    mp (a : b : xs) = let ab = (a `mappend` b)
      mpxs = mp xs
      in ab `deepseq` (mpxs `par` ab : mpxs)
    mp xs = xs
suffixScanPar :: (Num a, NFData 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 *Monoid* for *Scan* is derived from monoidal properties of the underlying *Buildable*s. Returning to the Maximum Segment Sum structure from the previous section, we can observe that the *Sat* representation makes very explicit why that structure is not a *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 *Sat* cannot be given a *Monoid* instance, an nor can structures such as *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 $[1 + 2 + 3 + 4, 2 + 3 + 4 + 5, 3 + 4 + 5 + 6...]$. Algorithms exist for computing such results in $\mathcal{O}(n)$ 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 $\mathcal{O}(m * \log(n/m))$ 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 *ScanL* to mirror our existing *Scan*, and then introducing a *ScanSeg* structure, that is a triple of a left scan (adjoint to *inits*), a pure carrier, and a right scan (adjoint to *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.

```
data Rev f a = Rev [a] (f a) deriving Show
fromRev (Rev _ xs) = xs
instance (Buildable f a) ⇒ Buildable (Rev f) a where
  unit = Rev [] unit
  insert x (Rev xs _) =
    Rev (x : xs) (fromList (reverse (x : xs)))
instance Monoid (f a) ⇒ Monoid (Rev f a) where
  mempty = Rev [] mempty
  mappend (Rev xs r) (Rev ys r') =
    Rev (xs ++ ys) (r <> 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 ∘ fromRev ∘ getScan ∘ fromRev
data ScanSeg g a =
```

```
  ScanSeg
    (ScanL [] g a)
    [g a]
    (Scan [] g a)
      deriving Show
instance (Buildable g a) ⇒ Buildable (ScanSeg g) a
  where
    insert x (ScanSeg h m t) =
      ScanSeg (insert x h) m (insert x t)
    unit = ScanSeg unit unit unit
instance (Buildable g a, Monoid (g a)) ⇒
  Monoid (ScanSeg g a) where
    mempty = unit
    mappend x@(ScanSeg h m t) y@(ScanSeg h' m' t')
        | isEmpty x = y
        | isEmpty y = x
        | otherwise =
          ScanSeg h (m <> 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 f*. 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 $\mathcal{O}(1)$ *mappend*.

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

```
slidingSum :: Num a ⇒ Int → [a] → ScanSeg Sum a
slidingSum n xs =
  mconcat (map fromList (chunksOf (n − 1) xs))
```

With an efficient implementation we can observe that the overall cost of merging any two *ScanSeg*s is $\mathcal{O}(m)$ in our window size, and furthermore that building a *ScanSeg* directly costs $\mathcal{O}(m)$. The number of builds and merges we need to perform is $\mathcal{O}(n/m)$ in the size of our list, and hence the entire operation is $\mathcal{O}(n)$ 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 $\mathcal{O}(m * \log(m/n))$ 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 $* \to *$, 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 Monoid*s (or more generally, arbitrary *Buildable*s).

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 been studied in this line of work as well [10] There is even more work in this tradition studying Maxiumum 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 $Buildable$s that need not be $Monoid$s and $Monoid$s 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 $foldr$ or $foldl$ 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" 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 $Iteratee$s and later $Pipe$s and $Machine$s [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 $f\ a \to f\ (f\ a)$ 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 $Buildable\ []$ 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

[1] Ghc.exts haddock documentation. http://hackage.haskell.org/package/base-4.7.0.1/docs/GHC-Exts.html.

[2] T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, pages 297–311. Springer, 2010.

[3] R. Bird and O. De Moor. The algebra of programming. In *NATO ASI DPD*, pages 167–203, 1996.

[4] R. Bird, J. Gibbons, S. Mehner, J. Voigtländer, and T. Schrijvers. Understanding idiomatic traversals backwards and forwards. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2013.

[5] R. Bird, J. Gibbons, S. Mehner, J. Voigtländer, and T. Schrijvers. Understanding idiomatic traversals backwards and forwards. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2013.

[6] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[7] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. Technical report, Proceedings of Parco 93. Elsevier Series in Advances in Parallel Computing, 1993.

[8] D. Coutts. *Stream Fusion: Practical shortcut fusion for coinductive sequence types*. PhD thesis, University of Oxford, 2010.

[9] S. Curtis and S.-C. Mu. Functional pearl: Finding a densest segment. preprint, 2014.

[10] J. Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction*, pages 122–138. Springer, 1993.

[11] J. Gibbons. Maximum segment sum, monadically (distilled tutorial). *EPTCS*, 66:181–194, 2011.

[12] A. Gill, J. Launchbury, and S. P. Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232. ACM Press, 1993.

[13] G. Gonzalez. pipes. http://hackage.haskell.org/package/pipes, 2012-14.

[14] R. Hickey. Transducers. Strange Loop, 2014.

[15] R. Hinze. Adjoint folds and unfolds. In *Mathematics of Program Construction*, pages 195–228. Springer, 2010.

[16] R. Hinze. Type fusion. In *Algebraic Methodology And Software Technology*, pages 92–110. Springer, 2011.

[17] R. Hinze and J. Jeuring. Generic haskell: Practice and theory. In *Generic Programming*, pages 1–56. Springer, 2003.

[18] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(02):197–217, 2006.

[19] J. Hughes. Restricted data types in haskell. In *Haskell Workshop*, volume 99, 1999.

[20] M. Jaskelioff and O. Rypacek. An investigation of the laws of traversals. In *MSFP'12*, pages 40–49, 2012.

[21] O. Kiselyov. Iteratees. In *Functional and Logic Programming*, pages 166–181. Springer, 2012.

[22] E. Kmett, R. Bjarnason, and J. Cough. machines. http://hackage.haskell.org/package/machines, 2012-14.

[23] S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971. ISBN 0387900357.

[24] G. Malcolm. Algebraic data types and program transformation. *Science of Computer Programming*, 14, 1990.

[25] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.

[26] S.-C. Mu. Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 31–39. ACM, 2008.

[27] S.-C. Mu. The maximum segment sum problem: Its origin, and a derivation. http://www.iis.sinica.edu.tw/~scm/2010/maximum-segment-sum-origin-and-derivation/, 2010.

[28] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 287–298. ACM, 2013.

[29] O. Shivers and M. Might. Continuations and transducer composition. *ACM SIGPLAN Notices*, 41(6):295–307, 2006.

[30] G. Sittampalam and P. Gavin. rmonad. http://hackage.haskell.org/package/rmonad, 2008-9.

[31] P. Wadler. Recursive types for free! unpublished manuscript, July 1990.