The Essence of the Iterator Pattern

Jeremy Gibbons and Bruno C. d. S. Oliveira
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
http://www.comlab.ox.ac.uk/jeremy.gibbons/
http://www.comlab.ox.ac.uk/bruno.oliveira/

Abstract

The ITERATOR pattern gives a clean interface for element-by-element access to a collection, independent of the collection's shape. Imperative iterations using the pattern have two simultaneous aspects: *mapping* and *accumulating*. Various existing functional models of iteration capture one or other of these aspects, but not both simultaneously. We argue that McBride and Paterson's *applicative functors*, and in particular the corresponding *traverse* operator, do exactly this, and therefore capture the essence of the ITERATOR pattern. Moreover, they do so in a way that nicely supports modular programming. We present some axioms for traversal, discuss modularity concerns, and illustrate with a simple example, the *wordcount* problem.

1 Introduction

Perhaps the most familiar of the so-called Gang of Four design patterns (Gamma *et al.*, 1995) is the ITERATOR pattern, which 'provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation'. Traditionally, this is achieved by identifying an ITERATOR interface that presents operations to initialize an iteration, to access the current element, to advance to the next element, and to test for completion; collection objects are expected to implement this interface, usually indirectly via a subobject. Essential to the pattern is the idea that elements are accessed sequentially, but independently of their 'position' in the collection; for example, labelling each element of a tree with its index in left-to-right order fits the pattern, but labelling each element with its depth does not.

This traditional version of the pattern is sometimes called an EXTERNAL ITERATOR. An alternative INTERNAL ITERATOR approach assigns responsibility for managing the traversal to the collection instead of the client: the client needs only to provide an operation, which the collection applies to each of its elements. The latter approach is simpler to use, but less flexible; for example, it is not possible for the iteration to affect the order in which elements are accessed, nor to terminate the iteration early. By 'iteration' in this paper we mean the INTERNAL ITERATOR approach — not EXTERNAL ITERATORS, nor iteration in the sense of Pascal's **for** loop.

An external iterator interface has been included in the Java and the C# libraries since their inception. Syntactic sugar supporting use of the interface, in the form of the **foreach**

```
public static int loop⟨MyObj⟩ (IEnumerable⟨MyObj⟩ coll){
   int n = 0;
   foreach (MyObj obj in coll){
      n = n + 1;
      obj.touch ();
   }
   return n;
}
```

Fig. 1. Iterating over a collection in C#.

construct, has been present in C# since the first version and in Java since version 1.5. This syntactic sugar effectively represents internal iterators in terms of external iterators; its use makes code cleaner and simpler, although it gives privileged status to the specific iteration interface chosen, entangling the language and its libraries.

In Figure 1 we show an application of C#'s **foreach** construct: a method *loop* that iterates over a collection, counting the elements but simultaneously interacting with each of them. The method is parametrized by the type *MyObj* of collection elements; this parameter is used twice, once to constrain the collection *coll* passed as a parameter, and again as a type for the local variable *obj*. The collection itself is rather unconstrained; it only has to implement the *IEnumerable*⟨*MyObj*⟩ interface.

In this paper, we investigate the structure of iterations over collection elements like that shown in Figure 1. We emphasize that we want to capture both aspects of the method *loop* and iterations like it: *mapping* over the elements, and simultaneously *accumulating* some measure of those elements. Moreover, we aim to do so *holistically*, treating the iteration as an abstraction in its own right; this leads us naturally to a higher-order presentation. We also want to develop an *algebra* of such iterations, with combinators for composing them and laws for reasoning about them; this leads us towards a functional approach. We argue that McBride and Paterson's *applicative functors* (McBride & Paterson, 2008), and in particular the corresponding *traverse* operator, have exactly the right properties. Finally, we will argue that *traverse* and its laws are ideally suited to modular development, whereby more complex programs can be obtained by composing simpler ones together, and compositions may be transformed by the application of the laws.

The rest of this paper is structured as follows. Section 2 reviews a variety of earlier approaches to capturing the essence of iterations functionally. Section 3 presents McBride and Paterson's notions of applicative functors and traversals. These two sections summarise previous work; our present contribution starts in Section 4, with a more detailed look at traversals. In Section 5 we propose a collection of laws of traversal, and in Section 6 we illustrate the use of some of these laws in the context of a simple example, the *wordcount* problem. Section 7 concludes.

2 Functional iteration

In this section, we review a number of earlier approaches to capturing the essence of iteration. In particular, we look at a variety of datatype-generic recursion operators: maps, folds, unfolds, crushes, and monadic maps. The traversals we discuss in Section 4 generalise most of these.

2.1 Origami

In the *origami* style of programming (Meijer *et al.*, 1991; Gibbons, 2002; Gibbons, 2003), the structure of programs is captured by higher-order recursion operators such as *map*, *fold* and *unfold*. These can be made *datatype-generic* (Jansson & Jeuring, 1997; Gibbons, 2006a), parametrised by the shape of the underlying datatype, as shown below.

```
class Bifunctor s where
  bimap :: (a → b) → (c → d) → s a c → s b d

data Fix s a = In{ out :: s a (Fix s a) }

map      :: Bifunctor s ⇒ (a → b) → Fix s a → Fix s b
map f    = In ∘ bimap f (map f) ∘ out

fold     :: Bifunctor s ⇒ (s a b → b) → Fix s a → b
fold f   = f ∘ bimap id (fold f) ∘ out

unfold   :: Bifunctor s ⇒ (b → s a b) → b → Fix s a
unfold f = In ∘ bimap id (unfold f) ∘ f
```

For a suitable binary type constructor *s*, the recursive datatype *Fix s a* is the fixpoint (up to isomorphism) in the second argument of *s* for a given type *a* in the first argument; the constructor *In* and destructor *out* witness the implied isomorphism. The type class *Bifunctor* captures those binary type constructors appropriate for determining the shapes of datatypes: the ones with a *bimap* operator that essentially locates elements of each of the two type parameters. Technically, *bimap* should also satisfy the laws

```
bimap id id          = id                   -- identity
bimap (f ∘ h) (g ∘ k) = bimap f g ∘ bimap h k  -- composition
```

but this constraint is not expressed in the type class declaration.

The recursion pattern *map* captures iterations that modify each element of a collection independently; thus, *map touch* captures the mapping aspect of the C# loop in Figure 1, but not the accumulating aspect.

At first glance, it might seem that the datatype-generic *fold* captures the accumulating aspect; but the analogy is rather less clear for a non-linear collection. In contrast to the C# method above, which is sufficiently generic to apply to non-linear collections, a datatype-generic counting operation defined using *fold* would need a datatype-generic numeric algebra as the fold body. Such a thing could be defined polytypically (Jansson & Jeuring, 1997; Hinze & Jeuring, 2003), but the fact remains that *fold* in isolation does not encapsulate the datatype genericity.

Essential to iteration in the sense we are using the term is linear access to collection elements; this was the problem with *fold*. One might consider a datatype-generic operation to yield a linear sequence of collection elements from possibly non-linear structures, for example by *unfold*ing to a list. This could be done (though as with the *fold* problem, it requires additionally a datatype-generic sequence coalgebra as the unfold body); but even then, this would address only the accumulating aspect of the C# iteration, and not the mapping aspect — it discards the shape of the original structure. Moreover, for some datatypes the sequence of elements is not definable as an unfold (Gibbons *et al.*, 2001).

We might also explore the possibility of combining some of these approaches. For example, it is clear from the definitions above that *map* is an instance of *fold*. Moreover, the *banana split theorem* (Fokkinga, 1990) states that two folds in parallel on the same data structure can be fused into one. Therefore, a map and a fold in parallel fuse to a single fold, yielding both a new collection and an accumulated measure, and might therefore be considered to capture both aspects of the C# iteration. However, we feel that this is an unsatisfactory solution: it may indeed simulate or implement the same behaviour, but it is no longer manifest that the shape of the resulting collection is related to that of the original.

2.2 Crush

Meertens (1996) generalised APL's 'reduce' to a *crush* operation, $\langle\!\langle \oplus \rangle\!\rangle :: t\ a \rightarrow a$ for binary operator $(\oplus) :: a \rightarrow a \rightarrow a$ with a unit, polytypically over the structure of a regular functor t. For example, $\langle\!\langle + \rangle\!\rangle$ polytypically sums a collection of numbers. For projections, composition, sum and fixpoint, there is an obvious thing to do, so the only ingredients that need to be provided are the binary operator (for products) and a constant (for units). Crush captures the accumulating aspect of the C# iteration in Figure 1, accumulating elements independently of the shape of the data structure, but not the mapping aspect.

2.3 Monadic map

One aspect of iteration expressed by neither the origami operators nor crush is the possibility of effects, such as stateful operations or exceptions. Seminal work by Moggi (1991), popularised by Wadler (1992), showed how such computational effects can be captured in a purely functional context through the use of *monads*.

```
class Functor f where
  fmap  :: (a → b) → f a → f b

class Functor m ⇒ Monad m where
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a
```

satisfying the following laws:

```
fmap id              = id                          -- identity
fmap (f ∘ g)         = fmap f ∘ fmap g             -- composition

return a >>= f       = f a                         -- left unit
mx >>= return        = mx                          -- right unit
(mx >>= f) >>= g     = mx >>= (λx → f x >>= g)     -- associativity
```

Roughly speaking, the type *m a* for a monad *m* denotes a computation returning a value of type *a*, but in the process possibly having some computational effect corresponding to *m*; the *return* operator lifts pure values into the monadic domain, and the 'bind' operator $\gg\!=$ denotes a kind of sequential composition.

Haskell's standard library (Peyton Jones, 2003) defines a *monadic map* for lists, which lifts an effectful computation on elements to one on lists:

```
mapM :: Monad m ⇒ (a → m b) → ([a] → m [b])
```

Fokkinga (1994) showed how to generalise this from lists to an arbitrary regular functor, polytypically. Several authors (Meijer & Jeuring, 1995; Moggi *et al.*, 1999; Jansson & Jeuring, 2002; Pardo, 2005; Kiselyov & Lämmel, 2005) have observed that monadic map is a promising model of iteration. Monadic maps are very close to the *idiomatic traversals* that we propose as the essence of imperative iterations; indeed, for monadic applicative functors, traversal reduces exactly to monadic map. However, we argue that monadic maps do not capture accumulating iterations as nicely as they might. Moreover, it is well-known (Jones & Duponcheel, 1993; King & Wadler, 1993) that monads do not compose in general, whereas applicative functors do; this will give us a richer algebra of traversals. Finally, monadic maps stumble over products, for which there are two reasonable but symmetric definitions, coinciding only when the monad is commutative. This stumbling block forces either a bias to left or right, or a restricted focus on commutative monads, or an additional complicating parametrisation; in contrast, applicative functors generally have no such problem, and in fact can exploit it to provide traversal reversal.

Closely related to monadic maps are operations like Haskell's *sequence* function

$sequence :: Monad\ m \Rightarrow [m\ a] \rightarrow m\ [a]$

and its polytypic generalisation to arbitrary datatypes. Indeed, *sequence* and *mapM* are interdefinable: $mapM\ f = sequence \circ map\ f$, and so $sequence = mapM\ id$. Most writers on monadic maps have investigated such an operation; Moggi *et al.* (1999) call it *passive traversal*, Meertens (1998) calls it *functor pulling*, and Pardo (2005) and others have called it a *distributive law*. McBride and Paterson introduce the function *dist* playing the same role, but as we shall see, more generally.

3 Applicative Functors

McBride and Paterson (2008) recently introduced the notion of an *applicative functor* or *idiom* as a generalisation of monads. ('Idiom' was the name McBride originally chose, but he and Paterson now favour the less evocative term 'applicative functor'. We have a slight preference for the former, not least because it lends itself nicely to adjectival uses, as in 'idiomatic traversal'. However, out of solidarity, we will mostly use 'applicative functor' as the noun in this paper, resorting to 'idiomatic' as the adjective. Note that Leroy's parametrised modules that map equal type parameters to equal abstract types (Leroy, 1995) are a completely different kind of 'applicative functor'.) Monads allow the expression of effectful computations within a purely functional language, but they do so by encouraging an *imperative* programming style (Peyton Jones & Wadler, 1993); in fact, Haskell's monadic **do** notation is explicitly designed to give an imperative feel. Since applicative functors generalise monads, they provide the same access to effectful computations; but they encourage a more *applicative* programming style, and so fit better within the functional programming milieu. Moreover, as we shall see, applicative functors strictly generalise monads; they provide features beyond those of monads. This will be important to us in capturing a wider variety of iterations, and in providing a richer algebra of those iterations.

Applicative functors are captured in Haskell by the following type class, provided in recent versions of the GHC hierarchical libraries (GHC Team, 2006).

```
class Functor m ⇒ Applicative m where
  pure :: a → m a
  (⊛)  :: m (a → b) → m a → m b
```

Informally, *pure* lifts ordinary values into the idiomatic world, and $\circledast$ provides an idiomatic flavour of function application. We make the convention that $\circledast$ associates to the left, just like ordinary function application.

In addition to those of the *Functor* class, applicative functors are expected to satisfy the following laws.

$$\begin{array}{lll}
pure\ id \circledast u & = u & \text{-- identity} \\
pure\ (\circ) \circledast u \circledast v \circledast w & = u \circledast (v \circledast w) & \text{-- composition} \\
pure\ f \circledast pure\ x & = pure\ (f\ x) & \text{-- homomorphism} \\
u \circledast pure\ x & = pure\ (\lambda f \rightarrow f\ x) \circledast u & \text{-- interchange}
\end{array}$$

In case the reader feels the need for some intuition for these laws, we refer them forwards to the stream Naperian applicative functor discussed in Section 3.1 below, which we believe provides the most accessible instance of them.

These four laws are sufficient to rewrite any expression built from the applicative functor operators into a canonical form, consisting of a pure function applied to a series of idiomatic arguments: $pure\ f \circledast u_1 \circledast \cdots \circledast u_n$. (The composition law read right to left re-associates applications to the left; the interchange law moves pure functions to the left; and the homomorphism and identity laws combine multiple or zero occurrences of *pure* into one.) Hence the sequencing of effects of any applicative computation is fixed; in contrast, the ‘bind’ operation of a monad allows the result of one computation to affect the choice and ordering of effects of subsequent computations, a feature therefore not supported by applicative functors in general.

3.1 Monadic applicative functors

Applicative functors generalise monads; every monad induces an applicative functor, with the following operations.

$$\begin{array}{l}
\textbf{newtype}\ \mathbb{M}\ m\ a = Wrap\{ unWrap :: m\ a \} \\
\textbf{instance}\ Monad\ m \Rightarrow Applicative\ (\mathbb{M}\ m)\ \textbf{where} \\
\quad pure\ = Wrap \circ return \\
\quad f \circledast x = Wrap\ (unWrap\ f\ `ap`\ unWrap\ x)
\end{array}$$

(The wrapper $\mathbb{M}$ lifts a monad to an applicative functor, and is needed to avoid overlapping type class instances.) The *pure* operator for a monadic applicative functor is essentially just the *return* of the monad, and idiomatic application $\circledast$ is essentially monadic application, $mf\ `ap`\ mx = mf \ggg\!= \lambda f \rightarrow mx \ggg\!= \lambda x \rightarrow return\ (f\ x)$, here with the effects of the function preceding those of the argument — there is another, completely symmetric, definition, with the effects of the argument preceding those of the function (see Section 4.3). We leave the reader to verify that the monad laws thus entail the applicative functor laws.

For example, the *State* monad uses the following type declaration:

$$\textbf{newtype}\ State\ s\ a = State\{ runState :: s \rightarrow (a, s) \}$$

and induces a monadic applicative functor $\mathbb{M}\ (State\ s)$.

A particular subclass of monadic applicative functors corresponds to datatypes of fixed shape, and is exemplified by the stream functor:

$$\textbf{data}\ Stream\ a = SCons\ a\ (Stream\ a)$$

The *pure* operator lifts a value to a stream, with infinitely many copies of it; idiomatic application is a pointwise ‘zip with apply’, taking a stream of functions and a stream of arguments to a stream of results:

```
instance Applicative Stream where
  pure x                     = xs where xs = SCons x xs
  (SCons f fs) ⊛ (SCons x xs) = SCons (f x) (fs ⊛ xs)
```

This applicative functor turns out to be equivalent to the one induced by the *Reader* monad:

```
newtype Reader r a = Reader{ runReader :: r → a }
```

where the environment type r is the natural numbers. Computations within the stream applicative functor tend to perform a transposition of results; they are related to what Kühne (1999) calls the *transfold* operator. We find that this applicative functor is the most accessible one for providing some intuition for the applicative functor laws.

A similar construction works for any fixed-shape datatype: pairs, vectors of length n, matrices of fixed size, infinite binary trees, and so on. Peter Hancock calls such datatypes *Naperian*, because they support a notion of logarithm. That is, datatype t is Naperian if $t\ a \simeq a^p \simeq p \rightarrow a$ for some type p of positions, called the logarithm $\log\ t$ of t. Then $t\ 1 \simeq 1^p \simeq 1$, so the shape is fixed, and familiar properties of Napier’s logarithms arise — for example, $\log\ (t \times u) \simeq \log\ t + \log\ u$. Naperian functors generally are equivalent to *Reader* monads, with the logarithm as environment; nevertheless, we feel that it is worth identifying this particular subclass of monadic applicative functors as worthy of special attention. We expect some further connection with data-parallel and numerically intensive computation, in the style of Jay’s language FISh (Jay & Steckler, 1998), but we leave the investigation of that connection for future work.

3.2 Monoidal applicative functors

Applicative functors strictly generalise monads; there are applicative functors that do not arise from monads. A second family of applicative functors, this time non-monadic, arises from constant functors with monoidal targets. McBride and Paterson call these *phantom applicative functors*, because the resulting type is a phantom type, as opposed to a container type of some kind. Any monoid $(\emptyset, \oplus)$ induces an applicative functor, where the *pure* operator yields the unit $\emptyset$ of the monoid and application uses the binary operator $\oplus$.

```
newtype Const b a = Const{ unConst :: b }

instance Monoid b ⇒ Applicative (Const b) where
  pure _ = Const ∅
  x ⊛ y  = Const (unConst x ⊕ unConst y)
```

Computations within this applicative functor accumulate some measure: for the monoid of integers with addition, they count or sum; for the monoid of lists with concatenation, they collect some trace of values; for the monoid of booleans with disjunction, they encapsulate linear searches; and so on.

Note that the ‘repeat’ and ‘zip with apply’ operations of the stream Naperian applicative functor can be adapted for ordinary lists (Fridlender & Indrika, 2000) (although this instance does not seem to arise from a monad):

```
instance Applicative [ ] where
  pure x        = xs where xs = x : xs
```

$$\begin{aligned}(f : fs) \circledast (x : xs) &= f\ x : (fs \circledast xs)\\ \_ \quad \circledast \_ \quad &= [\,]\end{aligned}$$

Therefore, lists form applicative functors in three different ways: monadic in the usual way using cartesian product, when they model non-deterministic evaluation; monoidal using concatenation, when they model tracing of outputs; and Naperian-inspired using zip, when they model data-parallel computations.

3.3 Combining applicative functors

Like monads, applicative functors are closed under products; so two independent idiomatic effects can generally be fused into one, their product.

$\mathbf{data}\ (m \boxtimes n)\ a = Prod\{pfst :: m\ a, psnd :: n\ a\}$

$(\otimes) :: (Functor\ m, Functor\ n) \Rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow n\ b) \rightarrow (a \rightarrow (m \boxtimes n)\ b)$
$(f \otimes g)\ x = Prod\ (f\ x)\ (g\ x)$

$\mathbf{instance}\ (Applicative\ m, Applicative\ n) \Rightarrow Applicative\ (m \boxtimes n)\ \mathbf{where}$
$\quad pure\ x \quad = Prod\ (pure\ x)\ (pure\ x)$
$\quad mf \circledast mx = Prod\ (pfst\ mf \circledast pfst\ mx)\ (psnd\ mf \circledast psnd\ mx)$

Unlike monads in general, applicative functors are also closed under composition; so two sequentially-dependent idiomatic effects can generally be fused into one, their composition.

$\mathbf{data}\ (m \boxdot n)\ a = Comp\{unComp :: m\ (n\ a)\}$

$(\odot) :: (Functor\ n, Functor\ m) \Rightarrow (b \rightarrow n\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow (m \boxdot n)\ c)$
$f \odot g = Comp \circ fmap\ f \circ g$

$\mathbf{instance}\ (Applicative\ m, Applicative\ n) \Rightarrow Applicative\ (m \boxdot n)\ \mathbf{where}$
$\quad pure\ x \qquad\qquad\qquad = Comp\ (pure\ (pure\ x))$
$\quad (Comp\ mf) \circledast (Comp\ mx) = Comp\ (pure\ (\circledast) \circledast mf \circledast mx)$

The two operators $\otimes$ and $\odot$ allow us to combine idiomatic computations in two different ways; we call them *parallel* and *sequential composition*, respectively. We will see examples of both in Sections 4.1 and 6.

3.4 Idiomatic traversal

Two of the three motivating examples McBride and Paterson provide for idiomatic computations — sequencing a list of monadic effects and transposing a matrix — are instances of a general scheme they call *traversal*. This involves iterating over the elements of a data structure, in the style of a ‘map’, but interpreting certain function applications idiomatically.

$traverseList :: Applicative\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b]$
$traverseList\ f\ [\,] \qquad = pure\ [\,]$
$traverseList\ f\ (x : xs) = pure\ (:) \circledast f\ x \circledast traverseList\ f\ xs$

A special case is for traversal with the identity function, which distributes the data structure over the idiomatic structure:

$distList :: Applicative\ m \Rightarrow [m\ a] \rightarrow m\ [a]$
$distList = traverseList\ id$

The ‘map within the applicative functor’ pattern of traversal for lists generalises to any (finite) functorial data structure, even non-regular ones (Bird & Meertens, 1998). We capture this via a type class of *Traversable* data structures (a slightly more elaborate type class *Data.Traversable* appears in recent GHC hierarchical libraries (GHC Team, 2006)):

```
class Functor t ⇒ Traversable t where
  traverse :: Applicative m ⇒ (a → m b) → t a → m (t b)
  traverse f = dist ∘ fmap f

  dist :: Applicative m ⇒ t (m a) → m (t a)
  dist = traverse id
```

For example, here is a datatype of binary trees:

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)

instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Bin t u) = Bin (fmap f t) (fmap f u)
```

The corresponding *traverse* closely resembles the simpler *map*, with judicious uses of *pure* and ⊛:

```
instance Traversable Tree where
  traverse f (Leaf x) = pure Leaf ⊛ f x
  traverse f (Bin t u) = pure Bin ⊛ traverse f t ⊛ traverse f u
```

McBride and Paterson propose a special syntax involving ‘idiomatic brackets’, which would have the effect of inserting the occurrences of *pure* and ⊛ implicitly; apart from these brackets, the definition then looks exactly like a definition of *fmap*. This definition could be derived automatically (Hinze & Peyton Jones, 2000), or given datatype-generically once and for all, assuming some universal representation of datatypes such as sums and products (Hinze & Jeuring, 2003) or (using the definitions of *Bifunctor*, *Fix* and *fold* from Section 2.1) regular functors (Gibbons, 2003):

```
class Bifunctor s ⇒ Bitraversable s where
  bidist :: Applicative m ⇒ s (m a) (m b) → m (s a b)

instance Bitraversable s ⇒ Traversable (Fix s) where
  traverse f = fold (fmap In ∘ bidist ∘ bimap f id)
```

When *m* is specialised to the identity applicative functor, traversal reduces precisely (modulo the wrapper) to the functorial map over lists.

```
newtype Id a = Id{ unId :: a }

instance Applicative Id where
  pure x     = Id x
  mf ⊛ mx = Id ((unId mf) (unId mx))
```

In the case of a monadic applicative functor, traversal specialises to monadic map, and has the same uses. In fact, traversal is really just a slight generalisation of monadic map: generalising in the sense that it applies also to non-monadic applicative functors. We consider this an interesting insight, because it reveals that monadic map does not require the full power of a monad; in particular, it does not require the ‘bind’ or ‘join’ operators, which are unavailable in applicative functors in general.

For a Naperian applicative functor, traversal transposes results. For example, interpreted in the pair Naperian applicative functor, *traverseList id* unzips a list of pairs into a pair of lists.

For a monoidal applicative functor, traversal accumulates values. The function *reduce* performs that accumulation, given an argument that assigns a value to each element:

$reduce :: (Traversable\ t, Monoid\ m) \Rightarrow (a \to m) \to t\ a \to m$
$reduce\ f = unConst \circ traverse\ (Const \circ f)$

The special case *crush* (named after Meertens' operator discussed in Section 2.2, but with an additional monoidal constraint) applies when the elements are their own values:

$crush :: (Traversable\ t, Monoid\ m) \Rightarrow t\ m \to m$
$crush = reduce\ id$

For example, when the monoid is that of integers and addition, traversal sums the elements of a collection.

$tsum :: Traversable\ t \Rightarrow t\ Integer \to Integer$
$tsum = crush$

4 Traversals as iterators

In this section, we show some representative examples of traversals over data structures, and capture them using *traverse*.

Before we look at traversals, however, we will introduce a convenient piece of notation. Recall the identity and constant functors introduced in Section 3:

newtype $Id\ a = Id\{unId :: a\}$
newtype $Const\ b\ a = Const\{unConst :: b\}$

We will have a number of new datatypes with coercion functions like *Id*, *unId*, *Const* and *unConst*. To reduce clutter, we introduce a common notation for such coercions:

class $Coerce\ a\ b \mid a \to b$ **where**
  $\Downarrow :: a \to b$
  $\Uparrow :: b \to a$

The idea is that an instance of *Coerce a b* indicates that type *a* is a new datatype built on top of an underlying type *b*; the function $\Downarrow$ reveals the underlying value, and the function $\Uparrow$ wraps it up. The identity functor is an instance of this type class, of course:

instance $Coerce\ (Id\ a)\ a$ **where**
  $\Downarrow = unId$
  $\Uparrow = Id$

and so are constant functors:

instance $Coerce\ (Const\ a\ b)\ a$ **where**
  $\Downarrow = unConst$
  $\Uparrow = Const$

Moreover, instances may be propagated through product:

instance $(Coerce\ (m\ a)\ b, Coerce\ (n\ a)\ c) \Rightarrow Coerce\ ((m \boxtimes n)\ a)\ (b,c)$ **where**
  $\Downarrow mnx\ \ = (\Downarrow (pfst\ mnx), \Downarrow (psnd\ mnx))$
  $\Uparrow (x,y) = Prod\ (\Uparrow x)\ (\Uparrow y)$

through composition:

$\textbf{instance}\ (Functor\ m, Functor\ n, Coerce\ (m\ b)\ c, Coerce\ (n\ a)\ b) \Rightarrow$
$\quad Coerce\ ((m \boxdot n)\ a)\ c\ \textbf{where}$
$\Downarrow = \Downarrow \circ fmap \Downarrow \circ unComp$
$\Uparrow = Comp \circ fmap \Uparrow \circ \Uparrow$

and through monad wrapping:

$\textbf{instance}\ Coerce\ (m\ a)\ c \Rightarrow Coerce\ (\mathbb{M}\ m\ a)\ c\ \textbf{where}$
$\Downarrow = \Downarrow \circ unWrap$
$\Uparrow = Wrap \circ \Uparrow$

We will introduce other instances of *Coerce* as we need them.

4.1 Shape and contents

In addition to being parametrically polymorphic in the collection elements, the generic *traverse* operation is parametrised along two further dimensions: the datatype being traversed, and the applicative functor in which the traversal is interpreted. Specialising the latter to lists as a monoid yields a generic *contents* operation:

$contentsBody :: a \rightarrow Const\ [a]\ b$
$contentsBody\ x = \Uparrow [x]$

$contents :: Traversable\ t \Rightarrow t\ a \rightarrow Const\ [a]\ (t\ b)$
$contents = traverse\ contentsBody$

To obtain a function of the expected type $t\ a \rightarrow [a]$, we need to remove the type coercions. The type class *Coerce* allows this to be done generically:

$run :: (Coerce\ b\ c, Traversable\ t) \Rightarrow (t\ a \rightarrow b) \rightarrow t\ a \rightarrow c$
$run\ program = \Downarrow \circ program$

Now we can define the function we expect:

$runContents :: Traversable\ t \Rightarrow t\ a \rightarrow [a]$
$runContents = run\ contents$

(so that $runContents = reduce\ (:[\,])$). The function *run* is applicable to all the other traversals we define as well, but for the sake of brevity we usually omit the routine definitions.

The *contents* operation is in turn the basis for many other generic operations, including non-monoidal ones such as indexing. Moreover, it yields one half of Jay's decomposition of datatypes into shape and contents (Jay, 1995). The other half of the decomposition is obtained simply by a map, which is to say, a traversal interpreted in the identity idiom:

$shapeBody :: a \rightarrow Id\ ()$
$shapeBody\ \_ = \Uparrow ()$

$shape :: Traversable\ t \Rightarrow t\ a \rightarrow Id\ (t\ ())$
$shape = traverse\ shapeBody$

This pair of traversals nicely illustrates the two aspects of iterations that we are focussing on, namely mapping and accumulation. Of course, it is trivial to compose them in parallel to obtain both halves of the decomposition as a single function, but doing this by tupling in the obvious way

$decompose :: Traversable\ t \Rightarrow t\ a \rightarrow (Id \boxtimes Const\ [a])\ (t\ ())$
$decompose = shape \otimes contents$

entails two traversals over the data structure. Is it possible to fuse the two traversals into one? The product of applicative functors allows exactly this, and Section 5.3 justifies this decomposition of a data structure into shape and contents in a single pass:

$\quad decompose = traverse\ (shapeBody \otimes contentsBody)$

Moggi *et al.* (1999) give a similar decomposition, but using a customised combination of monads; we believe that the above component-based approach is simpler.

A similar benefit can be found in the reassembly of a full data structure from separate shape and contents. This is a stateful operation, where the state consists of the contents to be inserted; but it is also a partial operation, because the number of elements provided may be less than the number of positions in the shape. We therefore make use of both the *State* monad and the *Maybe* monad, and so we incorporate these two in our framework for coercions:

$\quad \mathbf{instance}\ Coerce\ (Maybe\ a)\ (Maybe\ a)\ \mathbf{where}$
$\quad\quad \Downarrow = id$
$\quad\quad \Uparrow = id$

$\quad \mathbf{instance}\ Coerce\ (State\ s\ a)\ (s \to (a,s))\ \mathbf{where}$
$\quad\quad \Downarrow = runState$
$\quad\quad \Uparrow = State$

This time, we form the composition of the functors, rather than their product. (As it happens, the composition of the *State* and *Maybe* monads in this way does in fact form another monad, but that is not the case for monads in general.)

The central operation in the solution is the partial stateful function that strips the first element off the list of contents, if this list is non-empty:

$\quad reassembleBody :: () \to (\mathbb{M}\ (State\ [a]) \boxdot \mathbb{M}\ Maybe)\ a$
$\quad reassembleBody = \Uparrow \circ takeHead$
$\quad\quad \mathbf{where}\ takeHead\ \_\ [\,] \qquad = (Nothing, [\,])$
$\quad\quad\quad\quad takeHead\ \_\ (y : ys) = (Just\ y, ys)$

This is a composite monadic value, using the composition of the two monads *State* $[a]$ and *Maybe*; traversal using this operation yields a stateful function for the whole data structure.

$\quad reassemble :: Traversable\ t \Rightarrow t\ () \to (\mathbb{M}\ (State\ [a]) \boxdot \mathbb{M}\ Maybe)\ (t\ a)$
$\quad reassemble = traverse\ reassembleBody$

Now it is simply a matter of running this stateful function and discarding any leftover elements:

$\quad runReassemble :: Traversable\ t \Rightarrow (t\ (), [a]) \to Maybe\ (t\ a)$
$\quad runReassemble = fst \circ uncurry\ (run\ reassemble)$

Decomposition and reassembly are partial inverses, in the following sense:

$\quad run\ decompose\ t = (s,c) \Leftrightarrow run\ reassemble\ s\ c = (Just\ t, [\,])$

Moreover, traversal of any data structure may be expressed in terms of list-based traversal of its contents:

$\quad runDecompose\ xs = (ys, zs) \Rightarrow$
$\quad\quad fmap\ (curry\ runReassemble\ ys)\ (traverseList\ f\ zs) = fmap\ Just\ (traverse\ f\ xs)$

This reinforces the message that traversal concerns the linear processing of contents, preserving but independent of the shape.

4.2 Collection and dispersal

We have found it convenient to consider special cases of effectful traversals, in which the mapping aspect is independent of the accumulation, and vice versa. The first of these traversals accumulates elements effectfully, with an operation of type $a \to m\ ()$, but modifies those elements purely and independently of this accumulation, with a function of type $a \to b$.

$collect :: (Traversable\ t, Applicative\ m) \Rightarrow (a \to m\ ()) \to (a \to b) \to t\ a \to m\ (t\ b)$
$collect\ f\ g = traverse\ (\lambda a \to pure\ (\lambda () \to g\ a) \circledast f\ a)$

The C# iteration in Figure 1 is an example, using the applicative functor of the *State* monad to capture the counting:

$loop :: Traversable\ t \Rightarrow (a \to b) \to t\ a \to \mathbb{M}\ (State\ Integer)\ (t\ b)$
$loop\ touch = collect\ (\lambda a \to Wrap\ (\mathbf{do}\ \{n \leftarrow get; put\ (n+1)\}))\ touch$

The second kind of traversal modifies elements purely but dependent on the state, with a binary function of type $a \to b \to c$, evolving this state independently of the elements, via a computation of type $m\ b$:

$disperse :: (Traversable\ t, Applicative\ m) \Rightarrow m\ b \to (a \to b \to c) \to t\ a \to m\ (t\ c)$
$disperse\ mb\ g = traverse\ (\lambda a \to pure\ (g\ a) \circledast mb)$

An example of this family of traversals is a kind of converse of counting, labelling every element with its position in order of traversal.

$label :: Traversable\ t \Rightarrow t\ a \to \mathbb{M}\ (State\ Integer)\ (t\ Integer)$
$label = disperse\ (Wrap\ step)\ (curry\ snd)$

$step :: State\ Integer\ Integer$
$step = \mathbf{do}\ \{n \leftarrow get; put\ (n+1); return\ n\}$

4.3 Backwards traversal

In contrast to pure maps, the order in which elements are visited in an effectful traversal is significant; in particular, iterating through the elements backwards is observably different from iterating forwards, because the effects happen in the opposite order. We can capture this reversal quite elegantly as an *applicative functor adapter*:

$\mathbf{newtype}\ Backwards\ m\ a = Backwards\{runBackwards :: m\ a\}$

$\mathbf{instance}\ Applicative\ m \Rightarrow Applicative\ (Backwards\ m)\ \mathbf{where}$
$\quad pure = Backwards \circ pure$
$\quad f \circledast x = Backwards\ (pure\ (flip\ (\$)) \circledast runBackwards\ x \circledast runBackwards\ f)$

Informally, *Backwards m* is an applicative functor if *m* is, but any effects happen in reverse; this provides the symmetric 'backwards' embedding of monads into applicative functors referred to in Section 3.1.

Such an adapter can be parcelled up existentially:

$\mathbf{data}\ AppAdapter\ m\ \mathbf{where}$
$\quad AppAdapter :: Applicative\ (g\ m) \Rightarrow$
$\qquad\qquad (\forall a.\ m\ a \to g\ m\ a) \to (\forall a.\ g\ m\ a \to m\ a) \to AppAdapter\ m$

$backwards :: Applicative\ m \Rightarrow AppAdapter\ m$
$backwards = AppAdapter\ Backwards\ runBackwards$

It can be used to define a parametrised traversal:

$ptraverse :: (Applicative\ m, Traversable\ t) \Rightarrow$
$\quad AppAdapter\ m \to (a \to m\ b) \to t\ a \to m\ (t\ b)$
$ptraverse\ (AppAdapter\ insert\ retrieve)\ f = retrieve \circ traverse\ (insert \circ f)$

For example, reverse labelling is just labelling, adapted to run backwards:

$lebal = ptraverse\ backwards\ (\lambda a \to step)$

Of course, there is a trivial *forwards* adapter too:

newtype $Forwards\ m\ a = Forwards\{\ runForwards :: m\ a\ \}$

instance $Applicative\ m \Rightarrow Applicative\ (Forwards\ m)$ **where**
$\quad pure\ = Forwards \circ pure$
$\quad f \circledast x = Forwards\ (runForwards\ f \circledast runForwards\ x)$

instance $Functor\ m \Rightarrow Functor\ (Forwards\ m)$ **where**
$\quad fmap\ f = Forwards \circ fmap\ f \circ runForwards$

$forwards :: Applicative\ m \Rightarrow AppAdapter\ m$
$forwards = AppAdapter\ Forwards\ runForwards$

5 Laws of traverse

In line with other type classes such as *Functor* and *Applicative*, we should consider also what properties the various datatype-specific definitions of *traverse* ought to enjoy.

5.1 Free theorems of traversal

In addition to his popularisation of Moggi's work on monads, Wadler made Reynolds' work on parametricity (Reynolds, 1983) more accessible under the slogan 'theorems for free' (Wadler, 1989). This principle states that a parametrically polymorphic function enjoys a property that follows entirely from its type, without any consideration of its implementation. The free theorem arising from the type of *dist* is

$dist \circ fmap\ (fmap\ k) = fmap\ (fmap\ k) \circ dist$

As corollaries, we get the following two free theorems of *traverse*:

$traverse\ (g \circ h) \qquad = traverse\ g \circ fmap\ h$
$traverse\ (fmap\ k \circ f) = fmap\ (fmap\ k) \circ traverse\ f$

These laws are not constraints on the implementation of *dist* and *traverse*; they follow automatically from their types.

5.2 Sequential composition of traversals

We have seen that applicative functors compose: there is an identity applicative functor *Id* and, for any two applicative functors *m* and *n*, a composite applicative functor $m \boxdot n$. We impose on implementations of *dist* the constraint of respecting this compositional structure. Specifically, the distributor *dist* should respect the identity applicative functor:

$dist \circ fmap\ Id = Id$

and the composition of applicative functors:

$dist \circ fmap\ Comp = Comp \circ fmap\ dist \circ dist$

As corollaries, we get analogous properties of *traverse*:

$$\begin{aligned} &\mathit{traverse}\ (\mathit{Id} \circ f) &&= \mathit{Id} \circ \mathit{fmap}\ f \\ &\mathit{traverse}\ (\mathit{Comp} \circ \mathit{fmap}\ f \circ g) &&= \mathit{Comp} \circ \mathit{fmap}\ (\mathit{traverse}\ f) \circ \mathit{traverse}\ g \end{aligned}$$

Both of these consequences have interesting interpretations. The first says that *traverse* interpreted in the identity applicative functor is essentially just *fmap*, as mentioned in Section 3.4. The second provides a fusion rule for the sequential composition of two traversals; it can be written equivalently as:

$$\mathit{traverse}\ (f \odot g) = \mathit{traverse}\ f \odot \mathit{traverse}\ g$$

5.3 Idiomatic naturality

We also impose the constraint that the distributor *dist* should be *natural in the applicative functor*, as follows. An *applicative functor transformation* $\phi :: m\ a \rightarrow n\ a$ from applicative functor m to applicative functor n is a homomorphism over the structure of applicative functors, that is, a polymorphic function (categorically, a natural transformation between functors m and n) that respects the applicative functor structure, as follows:

$$\begin{aligned} &\phi\ (\mathit{pure}_m\ a) &&= \mathit{pure}_n\ a \\ &\phi\ (\mathit{mf} \circledast_m \mathit{mx}) &&= \phi\ \mathit{mf} \circledast_n \phi\ \mathit{mx} \end{aligned}$$

(Here, the idiomatic operators are subscripted by their idiom for clarity.)

Then *dist* should satisfy the following naturality property: for applicative functor transformation ϕ,

$$\mathit{dist}_n \circ \mathit{fmap}\ \phi = \phi \circ \mathit{dist}_m$$

One consequence of this naturality property is a ‘purity law’:

$$\mathit{traverse}\ \mathit{pure} = \mathit{pure}$$

This follows, as the reader may easily verify, from the observation that $\mathit{pure}_m \circ \mathit{unId}$ is an applicative functor transformation from applicative functor *Id* to applicative functor *m*. This is an entirely reasonable property of traversal; one might say that it imposes a constraint of shape preservation. (But there is more to it than shape preservation: a traversal of pairs that flips the two halves necessarily ‘preserves shape’, but breaks this law.) For example, consider the following definition of *traverse* on binary trees, in which the two children are swapped on traversal:

```
instance Traversable Tree where
  traverse f (Leaf a) = pure Leaf ⊛ f a
  traverse f (Bin t u) = pure Bin ⊛ traverse f u ⊛ traverse f t
```

With this definition, $\mathit{traverse}\ \mathit{pure} = \mathit{pure} \circ \mathit{mirror}$, where *mirror* reverses a tree, and so the purity law does not hold; this is because the corresponding definition of *dist* is not natural in the applicative functor. Similarly, a definition with two copies of *traverse f t* and none of *traverse f u* makes *traverse pure* purely return a tree in which every right child has been overwritten with its left sibling. Both definitions are perfectly well-typed, but (according to our constraints) invalid.

On the other hand, the following definition, in which the traversals of the two children are swapped, but the *Bin* operator is flipped to compensate, is blameless. The purity law still applies, and the corresponding distributor is natural in the applicative functor; the effect of the reversal is that elements of the tree are traversed ‘from right to left’.

instance *Traversable Tree* **where**
  $traverse\ f\ (Leaf\ a) = pure\ Leaf \circledast f\ a$
  $traverse\ f\ (Bin\ t\ u) = pure\ (flip\ Bin) \circledast traverse\ f\ u \circledast traverse\ f\ t$

We consider this to be a reasonable, if rather odd, definition of *traverse*.

Another consequence of naturality is a fusion law for the parallel composition of traversals, as defined in Section 3.3:

$traverse\ f \otimes traverse\ g = traverse\ (f \otimes g)$

This follows from the fact that *pfst* and *psnd* are applicative functor transformations from *Prod m n* to *m* and to *n*, respectively.

5.4 Sequential composition of monadic traversals

A third consequence of naturality is a fusion law specific to monadic traversals. The natural form of composition for monadic computations is called *Kleisli composition*:

$(\bullet) :: Monad\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c)$
$(f \bullet g)\ x = \mathbf{do}\ \{y \leftarrow g\ x; z \leftarrow f\ y; return\ z\}$

The monad *m* is *commutative* if, for all *mx* and *my*,

$\mathbf{do}\ \{x \leftarrow mx; y \leftarrow my; return\ (x,y)\} = \mathbf{do}\ \{y \leftarrow my; x \leftarrow mx; return\ (x,y)\}$

When interpreted in the applicative functor of a commutative monad *m*, traversals with bodies $f :: b \rightarrow m\ c$ and $g :: a \rightarrow m\ b$ fuse:

$traverse\ f \bullet traverse\ g = traverse\ (f \bullet g)$

This follows from the fact that $\mu \circ unComp$ forms an applicative functor transformation from $m \boxdot m$ to *m*, for a commutative monad *m* with 'join' operator μ (that is, $\mu = (\gg\!= id)$).

This fusion law for the Kleisli composition of monadic traversals shows the benefits of the more general idiomatic traversals quite nicely. Note that the corresponding more general fusion law for applicative functors in Section 5.2 allows two different applicative functors rather than just one; moreover, there are no side conditions concerning commutativity, in contrast to the situation with Kleisli composition. For example, consider the following programs:

$update_1 :: a \rightarrow State\ Integer\ a$
$update_1\ x = \mathbf{do}\ \{var \leftarrow get; put\ (var * 2); return\ x\}$

$update_2 :: a \rightarrow State\ Integer\ a$
$update_2\ x = \mathbf{do}\ \{var \leftarrow get; put\ (var + 1); return\ x\}$

$monadic_1 = traverse\ update_1 \bullet traverse\ update_2$
$monadic_2 = traverse\ (update_1 \bullet update_2)$

$applicative_1 = traverse\ update_1 \odot traverse\ update_2$
$applicative_2 = traverse\ (update_1 \odot update_2)$

Because $update_1$ and $update_2$ do not commute, $monadic_1 \neq monadic_2$ in general; nevertheless, $applicative_1 = applicative_2$. The only advantage of the monadic law is that there is just one level of monad on both sides of the equation; in contrast, the idiomatic law has two levels of applicative functor, because there is no analogue of the 'join' operator μ.

We conjecture that the monadic traversal fusion law also holds even if *m* is not commutative, provided that *f* and *g* themselves commute ($f \bullet g = g \bullet f$); but this no longer follows from naturality of the distributor in any simple way, and it imposes the alternative constraint that the three types a, b, c are equal.

5.5 No duplication of elements

Another way in which a definition of *traverse* might cause surprises would be to visit elements multiple times. (A traversal that skips elements would violate the purity law in Section 5.3.) For example, consider this definition of *traverse* on lists, which visits each element twice:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Traversable}\ [\,]\ \textbf{where} \\
\quad \mathit{traverse}\ f\ [\,] \qquad = \mathit{pure}\ [\,] \\
\quad \mathit{traverse}\ f\ (x : xs) = \mathit{pure}\ (\mathit{const}\ (:)) \circledast f\ x \circledast f\ x \circledast \mathit{traverse}\ f\ xs
\end{array}
$$

Note that this definition still satisfies the purity law. However, it behaves strangely in the following sense: if the elements are indexed from zero upwards, and then the list of indices is extracted, the result is not an initial segment of the natural numbers. To make this precise, we define:

$$
\begin{array}{l}
\mathit{index} :: \mathit{Traversable}\ t \Rightarrow t\ a \to (t\ \mathit{Integer}, \mathit{Integer}) \\
\mathit{index}\ xs = \mathit{run}\ \mathit{label}\ xs\ 0
\end{array}
$$

where *label* was given in Section 4.2. We might expect for any *xs* that if $\mathit{index}\ xs = (ys, n)$ then $\mathit{runContents}\ ys = [0\,..\,n-1]$; however, with the duplicating definition of traversal for lists above, we get *index* `"abc"` $= (ys, 6)$ where $\mathit{runContents}\ ys = [1,1,3,3,5,5]$.

We might impose ‘no duplication’ as a further constraint on traversal, but the characterisation of the constraint in terms of indexing feels rather ad hoc; we are still searching for a nice theoretical treatment of this condition. For the time being, therefore, we propose to leave as an observation the fact that some odd definitions of traversal may duplicate elements.

6 Modular programming with applicative functors

In Section 4, we showed how to model various kinds of iteration — both mapping and accumulating, and both pure and impure — as instances of the generic *traverse* operation. The extra generality of applicative functors over monads, capturing monoidal as well as monadic behaviour, is crucial; that justifies our claim that idiomatic traversal rather than monadic map is the essence of the ITERATOR pattern.

However, there is an additional benefit of applicative functors over monads, which concerns the modular development of complex iterations from simpler aspects. Hughes (1989) argues that one of the major contributions of functional programming is in providing better glue for plugging components together. In this section, we make a corresponding case for applicative traversals: the improved compositionality of applicative functors over monads provides better glue for fusion of traversals, and hence better support for modular programming of iterations.

6.1 An example: wordcount

As an illustration, we consider the Unix word-counting utility `wc`, which computes the numbers of characters, words and lines in a text file. The program in Figure 2, based on

```
public static int [ ] wc⟨char⟩ (IEnumerable⟨char⟩ coll){
  int nl = 0, nw = 0, nc = 0;
  bool state = false;
  foreach (char c in coll){
    ++ nc;
    if (c ≡ '\n') ++ nl;
    if (c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t'){
      state = false;
    } else if (state ≡ false){
      state = true;
      ++ nw;
    }
  }
  int [ ] res = {nc, nw, nl};
  return res;
}
```

Fig. 2. Kernighan and Ritchie's `wc` program in C#

Kernighan and Ritchie's version (1988), is a translation of the original C program into C#. This program has become a paradigmatic example in the program comprehension community (Gallagher & Lyle, 1991; Villavicencio & Oliveira, 2001; Gibbons, 2006b), since it offers a nice exercise in re-engineering the three separate slices from the one monolithic iteration. We are going to use it in the other direction: fusing separate simple slices into one complex iteration.

6.2 Modular iterations, idiomatically

The character-counting slice of the `wc` program accumulates a result in the integers-as-monoid applicative functor:

```
type Count = Const Integer
count :: a → Count b
count _ = Const 1
```

The body of the iteration simply yields 1 for every element:

```
cciBody :: Char → Count a
cciBody = count
```

Traversing with this body accumulates the character count:

```
cci :: String → Count [a]
cci = traverse cciBody
```

(Note that the element type of the output collection is unconstrained for traversal in a monoidal applicative functor, because the result has a phantom type.)

Counting the lines (in fact, the newline characters, thereby ignoring a final 'line' that is not terminated with a newline character) is similar: the difference is simply what number to use for each element, namely 1 for a newline and 0 for anything else.

```
test :: Bool → Integer
test b = if b then 1 else 0
```

With the help of this function, we define:

```
lciBody :: Char → Count a
lciBody c = ⇑ (test (c ≡ '\n'))

lci :: String → Count [a]
lci = traverse lciBody
```

Counting the words is trickier, because it necessarily involves state. Here, we use the *State* monad with a boolean state, indicating whether we are currently within a word, and compose this with the applicative functor for counting:

```
wciBody :: Char → (𝕄 (State Bool) ⊡ Count) a
wciBody c = ⇑ (updateState c) where
  updateState :: Char → Bool → (Integer, Bool)
  updateState c w = let s = not (isSpace c) in (test (not w ∧ s), s)

wci :: String → (𝕄 (State Bool) ⊡ Count) [a]
wci = traverse wciBody
```

The wrapper actually to extract the word count runs this traversal from an initial state of *False*, and discards the final boolean state:

```
runWci :: String → Integer
runWci s = fst (run wci s False)
```

These components may be combined in various ways. For example, character- and line-counting may be combined to compute a pair of results, using the product of applicative functors:

```
clci :: String → (Count ⊠ Count) [a]
clci = cci ⊗ lci
```

This composition is inefficient, though, since it performs two traversals over the input. Happily, the two traversals may be fused into one, as we saw in Section 5.3, giving

```
clci = traverse (cciBody ⊗ lciBody)
```

in a single pass rather than two.

It so happens that both character- and line-counting use the same applicative functor, but that is not important here. Exactly the same technique works to combine these two components with the third:

```
clwci :: String → ((Count ⊠ Count) ⊠ (𝕄 (State Bool) ⊡ Count)) [a]
clwci = traverse (cciBody ⊗ lciBody ⊗ wciBody)
```

Note that character- and line-counting traversals are monoidal, whereas word-counting is monadic. For a related example using a Naperian applicative functor, consider conducting an experiment to determine whether the distributions of the letters ‘q’ and ‘u’ in a text are correlated. This might be modelled as follows:

```
quiBody :: Char → Pair Bool
quiBody c = P (c ≡ 'q', c ≡ 'u')

qui :: String → Pair [Bool]
qui = traverse quiBody
```

where *Pair* is a datatype of pairs:

```
newtype Pair a = P (a, a)
```

made into a Naperian applicative functor in the obvious way. Applying *qui* to a string yields a pair of boolean sequences, representing the graphs of the distributions of these two letters in the string:

$run\ qui$ `"qui"` $= ([True, False, False], [False, True, False])$

Moreover, *qui* combines nicely with character-counting:

$ccqui :: String \to (Count \boxtimes Pair)\ [Bool]$
$ccqui = cci \otimes qui = traverse\ (cciBody \otimes quiBody)$

We can also combine *qui* with the word-counting traversal — although the product of two applicative functors requires them to agree on the element type, the word-counting body *wci* is agnostic about this type and so combines with anything:

$wcqui :: String \to (Pair \boxtimes (\mathbb{M}\ (State\ Bool) \boxdot Count))\ [Bool]$
$wcqui = qui \otimes wci = traverse\ (quiBody \otimes wciBody)$

In general, however, component traversals may not be so amenable to composition, and product may not be the appropriate combinator. Such a situation calls for sequential composition $\odot$ rather than parallel composition $\otimes$ of applicative functors alone. Here, however, we can't directly compose querying with counting, because counting discards its argument; and neither can we compose counting with querying, because querying produces booleans and counting consumes characters. Instead, we have to use both sequential and parallel composition, preserving a copy of the input for querying in addition to counting it:

$wcqui' :: String \to ((Id \boxtimes (\mathbb{M}\ (State\ Bool) \boxdot Count)) \boxdot Pair)\ [Bool]$
$wcqui' = traverse\ (quiBody \odot (Id \otimes wciBody))$

6.3 Modular iterations, monadically

It is actually possible to compose the three slices of `wc` using monads alone. Let us explore how that works out, for comparison with the approach using applicative functors.

The first snag is that none of the three slices is actually monadic; we have to cast them in the monadic mold first. The simple counting slices can be expressed using the *Writer* monad:

$ccmBody :: Char \to Writer\ Integer\ Char$
$ccmBody\ c = \mathbf{do}\ \{tell\ 1; return\ c\}$

$ccm :: String \to Writer\ Integer\ String$
$ccm = mapM\ ccmBody$

$lcmBody :: Char \to Writer\ Integer\ Char$
$lcmBody\ c = \mathbf{do}\ \{tell\ (test\ (c \equiv$ `'\n'`$)); return\ c\}$

$lcm :: String \to Writer\ Integer\ String$
$lcm = mapM\ lcmBody$

Word-counting is stateful, acting on a state of type $(Integer, Bool)$:

```
$wcmBody :: Char \to State\ (Integer, Bool)\ Char$
$wcmBody\ c = \mathbf{let}\ s = not\ (isSpace\ c)\ \mathbf{in\ do}$
                $(n, w) \leftarrow get$
                $put\ (n + test\ (not\ w \wedge s), s)$
                $return\ c$
```

$wcm :: String \to State\ (Integer, Bool)\ String$
$wcm = mapM\ wcmBody$

This rewriting is a bit unfortunate; however, having rewritten in this way, we can compose the three traversals into one, and even fuse the three bodies:

$\mathit{clwcm} = \mathit{ccm} \otimes \mathit{lcm} \otimes \mathit{wcm} = \mathit{mapM}\ (\mathit{ccmBody} \otimes \mathit{lcmBody} \otimes \mathit{wcmBody})$

Now let us turn to the Naperian traversal. That too can be expressed monadically: as observed in Section 3.1, a Naperian functor is equivalent to a *Reader* monad with the position being the ‘environment’. In particular, the Naperian applicative functor for the functor *Pair* is equivalent to the monad *Reader Bool*.

$\mathit{qumBody} :: \mathit{Char} \rightarrow \mathit{Reader\ Bool\ Bool}$
$\mathit{qumBody}\ c = \mathbf{do}\ \{ b \leftarrow \mathit{ask}; \mathit{return}\ (\mathbf{if}\ b\ \mathbf{then}\ (c \equiv \texttt{'q'})\ \mathbf{else}\ (c \equiv \texttt{'u'}))\}$

$\mathit{qum} :: \mathit{String} \rightarrow \mathit{Reader\ Bool}\ [\mathit{Bool}]$
$\mathit{qum} = \mathit{mapM\ qumBody}$

We can’t form the parallel composition of this with word-counting, for the same reason as with the idiomatic approach: the element return types differ. But with monads, we can’t even form the sequential composition of the two traversals either: the two monads differ, and Kleisli composition requires two computations in the same monad.

It is sometimes possible to work around the problem of sequential composition of computations in different monads, using *monad transformers* (Jones, 1995). A monad transformer *t* turns a monad *m* into another monad *t m*, typically adding some functionality in the process; the operation *lift* embeds a monadic value from the simpler space into the more complex one.

$\mathbf{class}\ \mathit{MonadTrans}\ t\ \mathbf{where}$
$\quad \mathit{lift} :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow t\ m\ a$

With this facility, there may be many monads providing a certain kind of functionality, so that functionality too ought to be expressed in a class. For example, the functionality of the *State* monad can be added to an arbitrary monad using the monad transformer *StateT*, yielding a more complex monad with this added functionality:

$\mathbf{newtype}\ \mathit{StateT}\ s\ m\ a = \mathit{StateT}\{\mathit{runStateT} :: s \rightarrow m\ (a, s)\}$
$\mathbf{instance}\ \mathit{MonadTrans}\ (\mathit{StateT}\ s)\ \mathbf{where} \ldots$

$\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \mathbf{where}$
$\quad \mathit{get} :: m\ s$
$\quad \mathit{put} :: s \rightarrow m\ ()$
$\mathbf{instance}\ \mathit{MonadState}\ s\ (\mathit{State}\ s)\ \mathbf{where} \ldots$
$\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ (\mathit{StateT}\ s\ m)\ \mathbf{where} \ldots$

Now, in the special case of the composition of two different monads in which one is a monad transformer applied to the other, progress is possible:

$(^{\ulcorner}\bullet) :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow$
$\qquad (b \rightarrow t\ m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow t\ m\ c)$
$p1\ ^{\ulcorner}\bullet\ p2 = p1 \bullet (\mathit{lift} \circ p2)$

$(\bullet^{\urcorner}) :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow$
$\qquad (b \rightarrow m\ c) \rightarrow (a \rightarrow t\ m\ b) \rightarrow (a \rightarrow t\ m\ c)$
$p1 \bullet^{\urcorner} p2 = (\mathit{lift} \circ p1) \bullet p2$

We can use these constructions to compose sequentially the ‘q’–‘u’ experiment and word-counting. We need to generalise the type of *wcmBody* from the *State* monad specifically to any monad with the appropriate functionality (and in particular, one with *State* functionality added to the *Reader* monad):

$$\begin{array}{l}
\mathit{wcmBody'} :: \mathit{MonadState}\ (\mathit{Integer},\mathit{Bool})\ m \Rightarrow \mathit{Char} \rightarrow m\ \mathit{Char} \\
\mathit{wcmBody'}\ c = \mathbf{let}\ s = \mathit{not}\ (\mathit{isSpace}\ c)\ \mathbf{in\ do} \\
\qquad\qquad\qquad\qquad (n,w) \leftarrow \mathit{get} \\
\qquad\qquad\qquad\qquad \mathit{put}\ (n + \mathit{test}\ (\mathit{not}\ w \wedge s), s) \\
\qquad\qquad\qquad\qquad \mathit{return}\ c
\end{array}$$

(Notice that the definition is identical; only the type has changed.) Now querying and word-counting compose monadically:

$$\begin{array}{l}
\mathit{quwcm} :: \mathit{String} \rightarrow \mathit{StateT}\ (\mathit{Integer},\mathit{Bool})\ (\mathit{Reader}\ \mathit{Bool})\ [\mathit{Bool}] \\
\mathit{quwcm} = \mathit{mapM}\ \mathit{qumBody} \bullet\!\urcorner\ \mathit{mapM}\ \mathit{wcmBody'} = \mathit{mapM}\ (\mathit{qumBody} \bullet\!\urcorner\ \mathit{wcmBody'})
\end{array}$$

This particular pair of monads composes just as well the other way around, because the types *State s* (*Reader r a*) and *Reader r* (*State s a*) are isomorphic. So we could instead use the *ReaderT* monad transformer to add *Reader* behaviour to the *State* monad, and use the dual composition operation $\ulcorner\!\bullet$. However, both cases are rather awkward, because they entail having to generalise (perhaps previously-written) components from types involving specific monads (such as *State*) to general monad interfaces (such as *StateT*). Writing the components that way in the first place might be good practice, but that rule is little comfort when faced with a body of code that breaks it. Moreover, the monad transformer approach works only for certain monads, not for all of them; in contrast, composition of applicative functors is universal.

The upshot is that composition of applicative functors is more flexible than composition of monads.

7 Conclusions

Monads have long been acknowledged as a good abstraction for modularising certain aspects of programs. However, composing monads is known to be difficult, limiting their usefulness. One solution is to use monad transformers, but this requires programs to be designed initially with monad transformers in mind. Applicative functors have a richer algebra of composition operators, which can often replace the use of monad transformers; there is the added advantage of being able to compose applicative but non-monadic computations. We thus believe that applicative functors provide an even better abstraction than monads for modularisation.

We have argued that idiomatic traversals capture the essence of imperative loops — both mapping and accumulating aspects. We have stated some properties of traversals and shown a few examples, but we are conscious that more work needs to be done in both of these areas.

This work grew out of an earlier discussion of the relationship between design patterns and higher-order datatype-generic programs (Gibbons, 2006a). Preliminary versions of that work argued that pure datatype-generic maps are the functional analogue of the ITERATOR design pattern. It was partly while reflecting on that argument — and its omission of imperative aspects — that we came to the more refined position presented here. Note that idiomatic traversals, and even pure maps, are more general than object-oriented ITERATORS in at least one sense: it is trivial with our approach to change the type of the collection elements with a traversal, whereas with an approach based on mutable objects, this is essentially impossible.

As future work, we are exploring properties and generalisations of the specialised traversals *collect* and *disperse*. We hope that such specialised operators might enjoy richer composition properties than do traversals in general, and for example will provide more insight into the *repmin* example discussed in the conference version of this paper (Gibbons & Oliveira, 2006). We also hope to investigate the categorical structure of *dist* further: naturality in the applicative functor appears to be related to Beck's distributive laws (Beck, 1969), and 'no duplication' to linear type theories.

8 Acknowledgements

We are grateful to the members of IFIP WG2.1, the *Algebra of Programming* research group at Oxford, the *Datatype-Generic Programming* project at Oxford and Nottingham, and the anonymous MSFP and JFP referees, whose insightful comments have improved this paper considerably. Thanks are due especially to Conor McBride and Ross Paterson, without whose elegant work on applicative functors this would never have happened. As well as the clear debt we owe to (McBride & Paterson, 2008), we thank McBride for pointing us to Hancock's notion of Naperian functors, and Paterson for the observation that *dist* should be natural in the applicative functor.

References

Beck, Jon. (1969). Distributive laws. *Pages 119–140 of:* Eckmann, B. (ed), *Seminar on triples and categorical homology theory*. Lecture Notes in Mathematics, vol. 80.

Bird, Richard S., & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of:* Jeuring, Johan (ed), *Proceedings of mathematics of program construction*. Lecture Notes in Computer Science, vol. 1422. Marstrand, Sweden: Springer-Verlag.

Fokkinga, Maarten. (1994). *Monadic maps and folds for arbitrary datatypes*. Department INF, Universiteit Twente.

Fokkinga, Maarten M. (1990). Tupling and mutumorphisms. *The Squiggolist*, **1**(4), 81–82.

Fridlender, Daniel, & Indrika, Mia. (2000). Do we need dependent types? *Journal of functional programming*, **10**(4), 409–415.

Gallagher, K. B., & Lyle, J. R. (1991). Using program slicing in software maintenance. *IEEE transactions on software engineering*, **17**(8), 751–761.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

GHC Team. (2006). *Haskell hierarchical libraries*. `http://www.haskell.org/ghc/docs/latest/html/libraries/`.

Gibbons, Jeremy. (2002). Calculating functional programs. *Pages 148–203 of:* Backhouse, Roland, Crole, Roy, & Gibbons, Jeremy (eds), *Algebraic and coalgebraic methods in the mathematics of program construction*. Lecture Notes in Computer Science, vol. 2297. Springer-Verlag.

Gibbons, Jeremy. (2003). Origami programming. *Pages 41–60 of:* Gibbons, Jeremy, & de Moor, Oege (eds), *The fun of programming*. Cornerstones in Computing. Palgrave.

Gibbons, Jeremy. (2006a). Design patterns as higher-order datatype-generic programs. *Workshop on generic programming*.

Gibbons, Jeremy. (2006b). Fission for program comprehension. *Pages 162–179 of:* Uustalu, Tarmo (ed), *Mathematics of program construction*. Lecture Notes in Computer Science, vol. 4014. Springer-Verlag.

Gibbons, Jeremy, & Oliveira, Bruno C. d. S. (2006). The essence of the Iterator pattern. McBride, Conor, & Uustalu, Tarmo (eds), *Mathematically-structured functional programming*.

Gibbons, Jeremy, Hutton, Graham, & Altenkirch, Thorsten. (2001). When is a function a fold or an unfold? *Electronic notes in theoretical computer science*, **44**(1). Coalgebraic Methods in Computer Science.

Hinze, Ralf, & Jeuring, Johan. (2003). Generic Haskell: Practice and theory. *Pages 1–56 of:* Backhouse, Roland, & Gibbons, Jeremy (eds), *Summer school on generic programming*. Lecture Notes in Computer Science, vol. 2793.

Hinze, Ralf, & Peyton Jones, Simon. (2000). Derivable type classes. *International conference on functional programming*.

Hughes, John. (1989). Why functional programming matters. *Computer journal*, **32**(2), 98–107.

Jansson, Patrick, & Jeuring, Johan. (1997). PolyP – a polytypic programming language extension. *Pages 470–482 of: Principles of programming languages*.

Jansson, Patrik, & Jeuring, Johan. (2002). Polytypic data conversion programs. *Science of computer programming*, **43**(1), 35–75.

Jay, Barry, & Steckler, Paul. (1998). The functional imperative: Shape! *Pages 139–53 of:* Hankin, Chris (ed), *European symposium on programming*. Lecture Notes in Computer Science, vol. 1381.

Jay, C. Barry. (1995). A semantics for shape. *Science of computer programming*, **25**, 251–283.

Jeuring, Johan, & Meijer, Erik (eds). (1995). *Advanced functional programming*. Lecture Notes in Computer Science, vol. 925.

Jones, Mark P. (1995). Functional programming with overloading and higher-order polymorphism. *In:* (Jeuring & Meijer, 1995).

Jones, Mark P., & Duponcheel, Luc. (1993). *Composing monads*. Tech. rept. RR-1004. Department of Computer Science, Yale.

Kernighan, Brian W., & Ritchie, Dennis M. (1988). *The C programming language*. Prentice Hall.

King, David J., & Wadler, Philip. (1993). Combining monads. Launchbury, J., & Sansom, P. M. (eds), *Functional programming, Glasgow 1992*. Springer.

Kiselyov, Oleg, & Lämmel, Ralf. (2005). *Haskell's Overlooked Object System*. Draft; submitted for publication.

Kühne, Thomas. (1999). Internal iteration externalized. *Pages 329–350 of:* Guerraoui, Rachid (ed), *European conference on object-oriented programming*. Lecture Notes in Computer Science, vol. 1628.

Leroy, Xavier. (1995). Applicative functors and fully transparent higher-order modules. *Pages 142–153 of: Principles of programming languages*.

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *Journal of functional programming*, **18**(1), 1–13.

Meertens, Lambert. (1996). Calculate polytypically! *Pages 1–16 of:* Kuchen, H., & Swierstra, S. D. (eds), *Programming language implementation and logic programming*. Lecture Notes in Computer Science, vol. 1140.

Meertens, Lambert. (1998). Functor pulling. Backhouse, Roland, & Sheard, Tim (eds), *Workshop on generic programming*.

Meijer, Erik, & Jeuring, Johan. (1995). Merging monads and folds for functional programming. *In:* (Jeuring & Meijer, 1995).

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of:* Hughes, John (ed), *Functional programming languages and computer architecture*. Lecture Notes in Computer Science, vol. 523. Springer-Verlag.

Moggi, E., Bellè, G., & Jay, C. B. (1999). Monads, shapely functors and traversals. Hoffman, M., Pavlovic, D., & Rosolini, P. (eds), *Category theory in computer science*.

Moggi, Eugenio. (1991). Notions of computation and monads. *Information and computation*, **93**(1).

Pardo, Alberto. (2005). Combining datatypes and effects. *Advanced functional programming*. Lecture Notes in Computer Science, vol. 3622.

Peyton Jones, Simon. (2003). *The Haskell 98 language and libraries: The revised report*. Cambridge University Press.

Peyton Jones, Simon L., & Wadler, Philip. (1993). Imperative functional programming. *Pages 71–84 of: Principles of programming languages*.

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of: Information processing 83*. Elsevier.

Villavicencio, Gustavo, & Oliveira, José Nuno. (2001). Reverse program calculation supported by code slicing. *Pages 35–48 of: Eighth working conference on reverse engineering*. IEEE.

Wadler, Philip. (1989). Theorems for free! *Pages 347–359 of: Functional programming languages and computer architecture*. ACM.

Wadler, Philip. (1992). Monads for functional programming. Broy, M. (ed), *Program design calculi: Proceedings of the Marktoberdorf summer school*.