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
{*jg,bruno*}*@comlab.ox.ac.uk*

Abstract

The ITERATOR pattern gives a clean interface for element-by-element access to a collection. Imperative iterations using the pattern have two simultaneous aspects: *mapping* and *accumulating*. Various existing functional iterations model one or other of these, 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. We present some axioms for traversal, 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, perhaps indirectly via a subobject. This traditional approach is sometimes called an EXTERNAL ITERATOR, to distinguish it from an alternative INTERNAL ITERATOR implementation that delegates responsibility for managing the traversal to the collection instead of the client, thereby requiring the client only to provide a single operation to apply to each element. This latter style of iteration is our focus in this paper.

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* 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 the **foreach** construct: a C# 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, to constrain the collection *coll* passed as a parameter, and as a type for the local variable *obj*. The collection itself is rather unconstrained; it only has to implement the $IEnumerable\langle MyObj\rangle$ interface.

```
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#.

In this paper, we investigate the structure of such iterations over collection elements. 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, 2007), and in particular the corresponding *traverse* operator, have exactly the right properties. Finally, we will argue that *traverse* and its laws are ideally suited for the development of modular programs, where more complex programs can be obtained by composing simpler ones together and efficient programs can derived by mechanical application of these laws.

The rest of this paper is structured as follows. Section 2 reviews a variety of earlier approaches to capturing the essence of such iterations functionally. Section 3 presents McBride and Paterson's notions of applicative functors and traversals. 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 generalize all 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), parametrized 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)}
```

$$
\begin{array}{ll}
\mathit{map} & :: \mathit{Bifunctor}\ s \Rightarrow (a \to b) \to \mathit{Fix}\ s\ a \to \mathit{Fix}\ s\ b \\
\mathit{map}\ f & = \mathit{In} \circ \mathit{bimap}\ f\ (\mathit{map}\ f) \circ \mathit{out} \\
\mathit{fold} & :: \mathit{Bifunctor}\ s \Rightarrow (s\ a\ b \to b) \to \mathit{Fix}\ s\ a \to b \\
\mathit{fold}\ f & = f \circ \mathit{bimap}\ \mathit{id}\ (\mathit{fold}\ f) \circ \mathit{out} \\
\mathit{unfold} & :: \mathit{Bifunctor}\ s \Rightarrow (b \to s\ a\ b) \to b \to \mathit{Fix}\ s\ a \\
\mathit{unfold}\ f & = \mathit{In} \circ \mathit{bimap}\ \mathit{id}\ (\mathit{unfold}\ f) \circ f
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{bimap}\ \mathit{id}\ \mathit{id} & = \mathit{id} & \text{-- identity} \\
\mathit{bimap}\ (f \circ h)\ (g \circ k) & = \mathit{bimap}\ f\ g \circ \mathit{bimap}\ h\ k & \text{-- composition}
\end{array}
$$

but this constraint is not captured 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 above, 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# program 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 loses the shape of the original structure. Moreover, the sequence of elements is not always 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 (Meertens, 1996) generalized APL's 'reduce' to a *crush* operation, $\langle\langle\oplus\rangle\rangle :: t\ a \to a$ for binary operator $(\oplus) :: a \to a \to a$ 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 above, 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) popularized 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 $>\!\!>\!\!=$ denotes a kind of sequential composition.

Haskell's standard prelude 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 (Fokkinga, 1994) showed how to generalize 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 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 parametrization; in contrast, applicative functors generally have no such problem, and in fact turn it into a virtue.

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

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

and its polytypic generalization to arbitrary datatypes. Indeed, *sequence* and *mapM* are interdefinable:

$$mapM\ f = sequence \circ map\ f$$

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 Idioms

McBride and Paterson (2007) recently introduced the notion of an *applicative functor* (or *idiom*) as a generalization 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'. For solidarity, we will mostly use 'applicative functor' as the noun in this paper, but 'idiomatic' as the adjective.) Monads allow the expression of effectful computations within a purely functional language, but they do so by encouraging an *imperative* (Peyton Jones & Wadler, 1993) programming style; in fact, Haskell's monadic **do** notation is explicitly designed to give an imperative feel. Since applicative functors generalize 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 generalize 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.

```
pure id ⊛ u            = u                    -- identity
pure (∘) ⊛ u ⊛ v ⊛ w   = u ⊛ (v ⊛ w)          -- composition
pure f ⊛ pure x        = pure (f x)           -- homomorphism
u ⊛ pure x             = pure (λf → f x) ⊛ u  -- interchange
```

These two collections of laws are together sufficient to allow any expression built from the applicative functor operators to be rewritten into a canonical form, consisting of a

pure function applied to a series of idiomatic arguments: $pure\ f \circledast u_1 \circledast \cdots \circledast u_n$. (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 3.2 below.)

3.1 Monadic applicative functors

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

```
newtype WrappedMonad m a = WrapMonad { unwrapMonad :: (m a) }

instance Monad m ⇒ Applicative (WrappedMonad m) where
  pure                                = WrapMonad ∘ return
  WrapMonad f ⊛ WrapMonad v = WrapMonad (f `ap` v)
```

(The wrapper *WrappedMonad* lifts a monad to an applicative functor, avoiding the need for overlapping instances.) The *pure* operator for a monadic applicative functor is just the *return* of the monad; idiomatic application $\circledast$ is monadic application, here with the effects of the function preceding those of the argument (that is, $mf\ `ap`\ mx = mf \gg\!= \lambda f \to mx \gg\!= \lambda x \to return\ (f\ x)$). There is another, completely symmetric, definition, with the effects of the argument before those of the function (see Section 4.3). We leave the reader to verify that the monad laws entail the applicative functor laws.

For example, one of McBride and Paterson's motivating examples of an applicative functor arises from the 'environment' or 'reader' monad:

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

With this definition, *WrappedMonad* (*Reader r*) is a monadic applicative functor.

3.2 Naperian applicative functors

The 'bind' operation of a monad allows the result of one computation to affect the choice and ordering of effects of subsequent operations. Applicative functors in general provide no such possibility; indeed, as we have seen, every expression built just from the idiomatic combinators is equivalent to a pure function applied to a series of idiomatic arguments, and so the sequencing of any effects is fixed. This is reminiscent of Jay's *shapely operations* (1995), which separate statically-analysable shape from dynamically-determined contents. Static shape suggests another class of applicative functors, exemplified by the stream functor.

```
data Stream a = SCons a (Stream a)

instance Applicative Stream where
  pure a    = srepeat a
  mf ⊛ mx = szipWith ($) mf mx

srepeat  :: a → Stream a
srepeat x = xs where xs = SCons x xs

szipWith :: (a → b → c) → Stream a → Stream b → Stream c
szipWith f (SCons x xs) (SCons y ys) = SCons (f x y) (szipWith f xs ys)
```

The *pure* operator lifts a value to a stream, with infinitely many copies of that value; idiomatic application is a pointwise 'zip with apply', taking a stream of functions and a

stream of arguments to a stream of results. We find this applicative functor is the most accessible one for providing some intuition for the applicative functor laws. Computations within the stream applicative functor tend to perform a transposition of results; there appears to be some connection with what Kühne (1999) calls the *transfold* operator.

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 $t\,a \simeq a^p \simeq p \to a$ 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 logarithms arise — for example, $\log\ (t \circ u) \simeq \log\ t \times \log\ u$. Naperian functors turn out to be equivalent to environment monads, with the logarithm as environment.) We therefore 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.

3.3 Monoidal applicative functors

Applicative functors strictly generalize monads; there are applicative functors that do not arise from monads. A third 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 of the monoid and application uses the binary operator.

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

instance Monoid b ⇒ Applicative (Const b) where
  pure _ = Const ∅
  x ⊛ y  = Const (getConst x ⊕ getConst 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 sequences of one kind or another form applicative functors in three different ways: monadic with cartesian product, modelling non-determinism; Naperian with zip, modelling data-parallelism; and monoidal with concatenation, modelling tracing.)

3.4 Combining applicative functors

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

```
data Prod m n a = Prod {pfst :: m a, psnd :: n a}

(⊗) :: (Functor m, Functor n) ⇒ (a → m b) → (a → n b) → a → Prod m n b
(f ⊗ g) a = Prod (f a) (g a)

instance (Applicative m, Applicative n) ⇒ Applicative (Prod m n) where
  pure x    = Prod (pure x) (pure x)
  mf ⊛ mx = Prod (pfst mf ⊛ pfst mx) (psnd mf ⊛ 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.

$$\begin{aligned}
&\textbf{data}\ \mathit{Comp}\ m\ n\ a = \mathit{Comp}\{\mathit{unComp} :: m\ (n\ a)\}\\
&(\odot) :: (\mathit{Functor}\ n, \mathit{Functor}\ m) \Rightarrow (b \to n\ c) \to (a \to m\ b) \to a \to \mathit{Comp}\ m\ n\ c\\
&f \odot g = \mathit{Comp} \circ \mathit{fmap}\ f \circ g\\
&\textbf{instance}\ (\mathit{Applicative}\ m, \mathit{Applicative}\ n) \Rightarrow \mathit{Applicative}\ (\mathit{Comp}\ m\ n)\ \textbf{where}\\
&\quad \mathit{pure}\ x \quad = \mathit{Comp}\ (\mathit{pure}\ (\mathit{pure}\ x))\\
&\quad mf \circledast mx = \mathit{Comp}\ (\mathit{pure}\ (\circledast) \circledast \mathit{unComp}\ mf \circledast \mathit{unComp}\ mx)
\end{aligned}$$

The two operators $\otimes$ and $\odot$ allow us to combine two different 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.

3.5 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.

$$\begin{aligned}
&\mathit{traverseList} :: \mathit{Applicative}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b]\\
&\mathit{traverseList}\ f\ [\,] \quad\ = \mathit{pure}\ [\,]\\
&\mathit{traverseList}\ f\ (x : xs) = \mathit{pure}\ (:) \circledast f\ x \circledast \mathit{traverseList}\ f\ xs
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{distList} :: \mathit{Applicative}\ m \Rightarrow [m\ a] \to m\ [a]\\
&\mathit{distList} = \mathit{traverseList}\ \mathit{id}
\end{aligned}$$

The 'map within the applicative functor' pattern of traversal for lists generalizes 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)):

$$\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ t \Rightarrow \mathit{Traversable}\ t\ \textbf{where}\\
&\quad \mathit{traverse} :: \mathit{Applicative}\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)\\
&\quad \mathit{traverse}\ f = \mathit{dist} \circ \mathit{fmap}\ f\\
&\quad \mathit{dist} :: \mathit{Applicative}\ m \Rightarrow t\ (m\ a) \to m\ (t\ a)\\
&\quad \mathit{dist} = \mathit{traverse}\ \mathit{id}
\end{aligned}$$

For example, here is a datatype of binary trees:

$$\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\\
&\textbf{instance}\ \mathit{Functor}\ \mathit{Tree}\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathit{Leaf}\ a) = \mathit{Leaf}\ (f\ a)\\
&\quad \mathit{fmap}\ f\ (\mathit{Bin}\ t\ u) = \mathit{Bin}\ (\mathit{fmap}\ f\ t)\ (\mathit{fmap}\ f\ u)
\end{aligned}$$

The corresponding *traverse* closely resembles the simpler *map*:

$$\textbf{instance}\ \mathit{Traversable}\ \mathit{Tree}\ \textbf{where}$$

$$traverse\ f\ (Leaf\ a) = pure\ Leaf \circledast f\ a$$
$$traverse\ f\ (Bin\ t\ u) = pure\ Bin \circledast traverse\ f\ t \circledast traverse\ f\ u$$

McBride and Paterson propose a special syntax involving 'idiomatic brackets', which would have the effect of inserting the occurrences of *pure* and $\circledast$ 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 polytypically once and for all, assuming some universal representation of datatypes such as sums and products (Hinze & Jeuring, 2003) or regular functors (Gibbons, 2003):

class *Bifunctor s* $\Rightarrow$ *Bitraversable s* **where**
$$bidist :: Applicative\ m \Rightarrow s\ (m\ a)\ (m\ b) \to m\ (s\ a\ b)$$

instance *Bitraversable s* $\Rightarrow$ *Traversable* $(Fix\ s)$ **where**
$$traverse\ f = fold\ (fmap\ In \circ bidist \circ bimap\ f\ id)$$

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

newtype $Identity\ a = Identity\{\ runIdentity :: a\ \}$

instance *Applicative Identity* **where**
$$pure\ a \quad = Identity\ a$$
$$mf \circledast mx = Identity\ ((runIdentity\ mf)\ (runIdentity\ mx))$$

In the case of a monadic applicative functor, traversal specializes to monadic map, and has the same uses. In fact, traversal is really just a slight generalization of monadic map: generalizing 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. For example, interpreted in the integer monoidal applicative functor, traversal accumulates a sum of integer measures of the elements.

$$tsum :: (a \to Int) \to [a] \to Int$$
$$tsum\ f = getConst \circ traverseList\ (Const \circ f)$$

4 Traversals as iterators

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

4.1 Shape and contents

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

$$\begin{aligned}
&\mathit{contentsBody} :: a \to \mathit{Const}\ [a]\ b \\
&\mathit{contentsBody} = \lambda a \to \mathit{Const}\ [a] \\
&\mathit{contents} :: \mathit{Traversable}\ t \Rightarrow t\ a \to \mathit{Const}\ [a]\ (t\ b) \\
&\mathit{contents} = \mathit{traverse}\ \mathit{contentsBody} \\
&\mathit{run_contents} :: \mathit{Traversable}\ t \Rightarrow t\ a \to [a] \\
&\mathit{run_contents} = \mathit{get_contents} \circ \mathit{contents} \\
&\mathit{get_contents} = \mathit{getConst}
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{shapeBody} :: a \to \mathit{Identity}\ () \\
&\mathit{shapeBody} = \lambda a \to \mathit{Identity}\ () \\
&\mathit{shape} :: \mathit{Traversable}\ t \Rightarrow t\ a \to \mathit{Identity}\ (t\ ()) \\
&\mathit{shape} = \mathit{traverse}\ \mathit{shapeBody} \\
&\mathit{run_shape} :: \mathit{Traversable}\ t \Rightarrow t\ a \to t\ () \\
&\mathit{run_shape} = \mathit{get_shape} \circ \mathit{shape} \\
&\mathit{get_shape} = \mathit{runIdentity}
\end{aligned}$$

Of course, it is trivial to compose these two traversals in parallel to obtain both halves of the decomposition as a single function, but doing this by tupling in the obvious way

$$\begin{aligned}
&\mathit{decompose} :: \mathit{Traversable}\ t \Rightarrow t\ a \to \mathit{Prod}\ \mathit{Identity}\ (\mathit{Const}\ [a])\ (t\ ()) \\
&\mathit{decompose} = \mathit{shape} \otimes \mathit{contents}
\end{aligned}$$

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 5.4 shows how to obtain the decomposition of a data structure into shape and contents in a single pass:

$$\begin{aligned}
&\mathit{decompose} = \mathit{traverse}\ (\mathit{shapeBody} \otimes \mathit{contentsBody}) \\
&\mathit{run_decompose} :: \mathit{Traversable}\ t \Rightarrow t\ a \to (t\ (), [a]) \\
&\mathit{run_decompose} = \mathit{get_decompose} \circ \mathit{decompose}\ \textbf{where} \\
&\quad \mathit{get_decompose}\ xy = (\mathit{get_shape}\ (\mathit{pfst}\ xy), \mathit{get_contents}\ (\mathit{psnd}\ xy))
\end{aligned}$$

Moggi *et al.* (1999) give a similar decomposition, but using a customized combination of monads; we believe 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 not agree with the number of positions in the shape. We therefore make use of both the *State* monad and the *Maybe* monad; but this time, we form their composition 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 (Jones & Duponcheel, 1993; King & Wadler, 1993).)

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:

$$\begin{aligned}
&\mathit{takeHead} :: \mathit{State}\ [a]\ (\mathit{Maybe}\ a) \\
&\mathit{takeHead} = \textbf{do}\ \{ xs \leftarrow \mathit{get}; \textbf{case}\ xs\ \textbf{of}
\end{aligned}$$

$$\begin{array}{l}[\,] \quad\quad \to \mathit{return}\ \mathit{Nothing}\\ (y : \mathit{ys}) \to \mathbf{do}\ \{\mathit{put}\ \mathit{ys}; \mathit{return}\ (\mathit{Just}\ y)\}\}\end{array}$$

This is a composite monadatic value, using the composition of the two monads $\mathit{State}\ [a]$ and *Maybe*; traversal using this operation (suitably wrapped) returns a stateful function for the whole data structure.

$$\begin{array}{l}\mathit{reassemble} :: \mathit{Traversable}\ t \Rightarrow t\ () \to \mathit{Comp}\ (\mathit{WrappedMonad}\ (\mathit{State}\ [a]))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{WrappedMonad}\ \mathit{Maybe})\ (t\ a)\\ \mathit{reassemble} = \mathit{traverse}\ (\lambda() \to \mathit{Comp}\,\$\,\mathit{WrapMonad}\,\$\,\mathit{fmap}\ \mathit{WrapMonad}\ \mathit{takeHead})\end{array}$$

Now it is simply a matter of running this stateful function, and checking that the contents are entirely consumed:

$$\begin{array}{l}\mathit{run\_reassemble} :: \mathit{Traversable}\ t \Rightarrow (t\ (), [a]) \to \mathit{Maybe}\ (t\ a)\\ \mathit{run\_reassemble}\ (x, \mathit{ys}) = \mathit{allGone}\,\$\\ \quad \mathit{runState}\ (\mathit{unwrapMonad}\,\$\,\mathit{unComp}\,\$\,\mathit{reassemble}\ x)\ \mathit{ys}\end{array}$$

where

$$\begin{array}{l}\mathit{allGone} :: (\mathit{WrappedMonad}\ \mathit{Maybe}\ (t\ a), [a]) \to \mathit{Maybe}\ (t\ a)\\ \mathit{allGone}\ (\mathit{mt}, [\,]) \quad\quad = \mathit{unwrapMonad}\ \mathit{mt}\\ \mathit{allGone}\ (\mathit{mt}, (\_ : \_)) = \mathit{Nothing}\end{array}$$

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.

$$\begin{array}{l}\mathit{collect} \quad :: (\mathit{Traversable}\ t, \mathit{Applicative}\ m) \Rightarrow (a \to m\ ()) \to (a \to b) \to t\ a \to m\ (t\ b)\\ \mathit{collect}\ f\ g = \mathit{traverse}\ (\lambda a \to \mathit{pure}\ (\lambda() \to g\ a) \circledast f\ a)\end{array}$$

$$\begin{array}{l}\mathit{disperse} :: (\mathit{Traversable}\ t, \mathit{Applicative}\ m) \Rightarrow m\ b \to (a \to b \to c) \to t\ a \to m\ (t\ c)\\ \mathit{disperse}\ \mathit{mb}\ g = \mathit{traverse}\ (\lambda a \to \mathit{pure}\ (g\ a) \circledast \mathit{mb})\end{array}$$

The first of these traversals accumulates elements effectfully, but modifies those elements purely and independently of this accumulation. The C# iteration in Section 1 is an example, using the applicative functor of the *State* monad to capture the counting:

$$\begin{array}{l}\mathit{loop} :: \mathit{Traversable}\ t \Rightarrow (a \to b) \to t\ a \to \mathit{State}\ \mathit{Int}\ (t\ b)\\ \mathit{loop}\ \mathit{touch} = \mathit{collect}\ (\lambda a \to \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+1)\})\ \mathit{touch}\end{array}$$

The second kind of traversal modifies elements purely but dependent on the state, evolving this state independently of the elements. An example of this is a kind of converse of counting, labelling every element with its position in order of traversal.

$$\begin{array}{l}\mathit{label} :: \mathit{Traversable}\ t \Rightarrow t\ a \to \mathit{State}\ \mathit{Int}\ (t\ (a, \mathit{Int}))\\ \mathit{label} = \mathit{disperse}\ \mathit{step}\ (,)\end{array}$$

$$\begin{array}{l}\mathit{step} :: \mathit{State}\ \mathit{Int}\ \mathit{Int}\\ \mathit{step} = \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+1); \mathit{return}\ n\}\end{array}$$

4.3 Backwards traversal

Unlike the case with 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. We can capture this reversal quite elegantly as an *applicative functor adapter*:

```
newtype Backwards m a = Backwards{ runBackwards :: m a }
instance Applicative m ⇒ Applicative (Backwards m) where
  pure = Backwards ∘ pure
  f ⊛ x = Backwards (pure (flip ($)) ⊛ runBackwards x ⊛ 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:

```
data AppAdapter m = ∀g. Applicative (g m) ⇒
  AppAdapter (∀a. m a → g m a) (∀a. g m a → m a)
backwards :: Applicative m ⇒ AppAdapter m
backwards = AppAdapter Backwards runBackwards
```

and used in a parametrized traversal, for example to label backwards:

```
ptraverse :: (Applicative m, Traversable t) ⇒
  AppAdapter m → (a → m b) → t a → m (t b)
ptraverse (AppAdapter wrap unwrap) f = unwrap ∘ traverse (wrap ∘ f)
lebal = ptraverse backwards (λa → step)
```

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

```
newtype Forwards m a = Forwards{ runForwards :: m a }
instance Applicative m ⇒ Applicative (Forwards m) where
  pure = Forwards ∘ pure
  f ⊛ x = Forwards (runForwards f ⊛ runForwards x)
instance Functor m ⇒ Functor (Forwards m) where
  fmap f = Forwards ∘ fmap f ∘ runForwards
forwards :: Applicative m ⇒ AppAdapter m
forwards = AppAdapter Forwards runForwards
```

5 Laws of traverse

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

5.1 Free theorems of traversal

The free theorem (Wadler, 1989) arising from the type of *dist* is

```
dist ∘ fmap (fmap k) = fmap (fmap k) ∘ dist
```

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

```
traverse (g ∘ h)       = traverse g ∘ fmap h
traverse (fmap k ∘ f) = fmap (fmap k) ∘ 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 *Identity* and, for any two applicative functors *m* and *n*, a composite applicative functor *Comp m n*. We impose on implementations of *dist* the constraint of respecting this compositional structure. Specifically, the distributor *dist* should respect the identity applicative functor:

$$dist \circ fmap\ Identity = Identity$$

and the composition of applicative functors:

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

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

$$traverse\ (Identity \circ f) = Identity \circ fmap\ f$$
$$traverse\ (Comp \circ fmap\ f \circ g) = Comp \circ fmap\ (traverse\ f) \circ traverse\ g$$

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 3.5. The second provides a fusion rule for the sequential composition of two traversals; it can be written equivalently as:

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

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* $\phi :: m\ a \to n\ a$ from applicative functor *m* to applicative functor *n* is a polymorphic function (natural transformation) that respects the applicative functor structure:

$$\phi\ (pure_m\ a) = pure_n\ a$$
$$\phi\ (mf \circledast_m mx) = \phi\ mf \circledast_n \phi\ mx$$

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

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

$$dist_n \circ fmap\ \phi = \phi \circ dist_m$$

One consequence of this naturality property is a 'purity law':

$$traverse\ pure = pure$$

This follows, as the reader may easily verify, from the observation that $pure_m \circ runIdentity$ is an applicative functor transformation from applicative functor *Identity* 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, $traverse\ pure = pure \circ mirror$, 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*.

5.4 Parallel composition of traversals

Another consequence of naturality is a fusion law for the parallel composition of traversals, as defined in Section 3.4:

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

This follows from the fact that *pfst* is an applicative functor transformation from *Prod m n* to *m*, and symmetrically for *psnd*.

5.5 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*:

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

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 $f :: b \to m\ c$ and $g :: a \to m\ b$ 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 *Comp m m* to *m*, for a commutative monad *m* with join operator μ.

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. 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 μ operator of a monad for collapsing two levels to one.

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.

5.6 No duplication of elements

Another constraint we impose upon a definition of *traverse* is that it should visit each element precisely once. For example, we consider this definition of *traverse* on lists to be bogus, because it visits each element twice.

```
instance Traversable [] where
  traverse f []     = pure []
  traverse f (x:xs) = pure (const (:)) ⊛ f x ⊛ f x ⊛ traverse f xs
```

Note that this definition satisfies the purity law above; but we would still like to rule it out.

This axiom is harder to formalize, and we do not yet have a nice theoretical treatment of it. One way of proceeding is in terms of indexing. We require that the function *labels* returns an initial segment of the natural numbers, where

```
labels :: Traversable t ⇒ t a → [Int]
labels t = get_contents $ contents $ fmap snd $ fst $ runState (label t) 0
```

and *label* is as defined in Section 4.2. The bogus definition of *traverse* on lists given above is betrayed by the fact that we get instead *labels* `"abc"` $= [1,1,3,3,5,5]$.

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 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 argue that the fact that applicative functors compose more readily than 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 Kernighan and Ritchie's version (Kernighan & Ritchie, 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.

```
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#

6.2 Modular iterations, idiomatically

The character-counting slice of the `wc` program accumulates a result in the integers-as-monoid applicative functor. The body of the iteration simply yields 1 for every element:

```
cciBody :: Char → Const Int a
cciBody c = Const 1
```

Traversing with this body accumulates the character count:

```
cci :: String → Const Int [a]
cci = traverse cciBody
```

The count itself — which is just the length of the list — is easily extracted:

```
run_cci :: String → Int
run_cci = getConst ∘ cci
```

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.

```
lciBody :: Char → Const Int a
lciBody c = Const (fromEnum (c ≡ '\n'))

lci :: String → Const Int [a]
lci = traverse lciBody
```

The actual line count can be extracted in the same way as for the character count:

```
run_lci :: String → Int
run_lci = getConst ∘ lci
```

Counting the words is trickier, because it involves state. We therefore use the *State* monad, maintaining both an *Int* (for the count) and a *Bool* (indicating whether we are currently within a word).

```
wciBody :: Char → WrappedMonad (State (Int, Bool)) Char
wciBody = λc → let s = not (isSpace c) in WrapMonad $ do
  (n, w) ← get
```

```
    put (n + fromEnum (not w ∧ s), s)
    return c
wci :: String → WrappedMonad (State (Int, Bool)) String
wci = traverse wciBody
```

As before, here is a simple wrapper to extract the count:

```
run_wci :: String → Int
run_wci s = fst $ (execState $ unwrapMonad $ wci s) (0, False)
```

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

```
clci :: String → Prod (Const Int) (Const Int) String
clci = cci ⊗ lci
```

where $\otimes$ denotes the product of applicative functors, defined in Section 3.4. 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 5.4, 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's not important here. Exactly the same technique works to combine these two components with the third:

```
clwci = traverse ((cciBody ⊗ lciBody) ⊗ wciBody)
```

Character- and line-counting traversals use a monoidal applicative functor, and word-counting a monadic applicative functor. 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 an applicative functor in the obvious way. Applying *qu* to a string yields a pair of boolean sequences, modelling graphs of the distributions of these two letters in the string. Moreover, *qui* combines with character-counting nicely:

```
ccqui = cci ⊗ qui = traverse (cciBody ⊗ quiBody)
```

Unfortunately, *qui* does not combine with the word-counting traversal: the element type returned is *Bool* rather than *Char*, whereas the product of two applicative functors requires the element types to agree. (This wasn't a problem with the two monoidal applicative functors, which are agnostic about the element return type.) This situation calls for sequential composition $\odot$ rather than parallel composition $\otimes$ of applicative functors, giving

```
wcqui = wci ⊙ qui = traverse (wciBody ⊙ quiBody)
```

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 two of the three slices are not monadic it all; we have to cast them in the monadic mold first.

```
ccmBody :: Char → State Int Char
ccmBody = λc → do {n ← get; put (n+1); return c}

ccm :: String → State Int String
ccm = mapM ccmBody

lcmBody :: Char → State Int Char
lcmBody = λc → do {n ← get; put (n + fromEnum (c ≡ '\n')); return c}

lcm :: String → State Int String
lcm = mapM lcmBody
```

Word-counting is almost in monadic form already; all that is needed is to strip off the wrapper.

```
wcmBody :: Char → State (Int, Bool) Char
wcmBody = λc → unwrapMonad $ wciBody c

wcm :: String → State (Int, Bool) String
wcm = mapM wcmBody
```

This rewriting is a bit unfortunate, as it blurs the distinction between the different varieties of iteration that we could previously express. However, having rewritten in this way, we can compose the three traversals into one, and even fuse the three bodies:

```
clwcm = (ccm ⊗ lcm) ⊗ wcm = mapM ((ccmBody ⊗ lcmBody) ⊗ wcmBody)
```

Now let us turn to the Naperian traversal. That too can be expressed monadically: 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*.

```
qumBody :: Char → Reader Bool Bool
qumBody c = do b ← ask
  return $ if b then (c ≡ 'q') else (c ≡ 'u')

qum :: String → Reader Bool [Bool]
qum = 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, but 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.

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → 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:

```
newtype StateT s m a = StateT {runStateT :: s → m (a, s)}
instance MonadTrans (StateT s) where ...

class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
instance MonadState s (State s) where ...
instance Monad m ⇒ MonadState s (StateT s m) where ...
```

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.

```
(⌜•) :: (Monad m, MonadTrans t, Monad (t m)) ⇒
        (b → t m c) → (a → m b) → a → t m c
p1 ⌜• p2 = p1 • (lift ∘ p2)
(•⌝) :: (Monad m, MonadTrans t, Monad (t m)) ⇒
        (b → m c) → (a → t m b) → a → t m c
p1 •⌝ p2 = (lift ∘ p1) • p2
```

We can use these constructions to compose sequentially the ‘q’–‘u’ experiment and word-counting. We need to generalize 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).

```
wcmBody′ :: MonadState (Int, Bool) m ⇒ Char → m Char
wcmBody′ = λc → let s = not (isSpace c) in do
  (n, w) ← get
  put (n + fromEnum (not w ∧ s), s)
  return c
```

(Notice that the definition is essentially identical; only the type has changed.)

```
quwcm :: String → StateT (Int, Bool) (Reader Bool) [Bool]
quwcm = mapM qumBody •⌝ mapM wcmBody′ = mapM (qumBody •⌝ wcmBody′)
```

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 symmetric composition operation ⌜•.

However, both cases are rather awkward, because they entail having to generalize (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.

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

7 Conclusions

Monads have long been acknowledged as a good abstraction for modularizing certain aspects of programs. However, composing monads is known to be hard, 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 modularization.

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 paper 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 a less holistic approach one is left worrying about the state of a partially-complete type-changing traversal.

As future work, we are exploring properties and generalizations of the specialized traversals *collect* and *disperse*. We hope that such specialized 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 referees, whose valuable 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, 2007), 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), *LNM 80: Seminar on triples and categorical homology theory*.

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

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

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

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 (Oct.). *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), *LNCS 2297: Algebraic and coalgebraic methods in the mathematics of program construction*. 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 (Sept.). 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), *LNCS 2793: Summer school on generic programming*.

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), *LNCS 1381: European symposium on programming*.

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

Jeuring, Johan, & Meijer, Erik (eds). (1995). *LNCS 925: Advanced functional programming*.

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

Jones, Mark P., & Duponcheel, Luc. 1993 (Dec.). *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), *LNCS 1628: European conference on object-oriented programming*.

McBride, Conor, & Paterson, Ross. (2007). Applicative programming with effects. *Journal of functional programming*.

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

Meertens, Lambert. 1998 (June). 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. *LNCS 3622: Advanced functional programming*.

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

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*.