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 *idioms*, 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 *repmin* problem.

Keywords: Iterator, traversal, design pattern, map, fold, monad, idiom.

1. INTRODUCTION

Perhaps the most familiar of the so-called Gang of Four design patterns [5] is the ITERATOR pattern, which 'provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation'. This is achieved by identifying an ITERATOR interface (for example, presenting operations to initialize an iteration, to access the current element, to advance to the next element, and to test for completion), that collection objects are expected to implement — perhaps indirectly, via a subobject. (The *Iterator* interface has uses beyond the ITERATOR pattern — for example, for generators — but those uses are not our concern in this paper.)

Just such an interface has been included in the Java and the C# libraries since their inception. C# has some syntactic sugar (matched in Java in version 1.5) to avoid the need to write the boilerplate code to manage an iteration over collection elements; we show an example below. This makes code cleaner and simpler, but gives privileged status to the specific iteration interface chosen, and entangles the language and its libraries. Recently (in Java 1.5 and C# 2.0), both languages also introduced *generics*, giving a kind of parametric polymorphism.

The code below shows a C# method *loop* that iterates over a collection, counting the elements but simultaneously interacting with each of them.

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

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*⟨*MyObj*⟩ interface.

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. Finally, we 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 *idioms* [20], and in particular the corresponding *traverse* operator, have exactly the right properties.

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 idioms 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 *repmin* problem.

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 [6, 7], the structure of programs is captured by higher-order recursion operators such as *map*, *fold* and *unfold*. These can be made *datatype-generic* [8], 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)}
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 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, satisfying the laws

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

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 [12, 10], 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 [9].

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* [4] 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 [21] 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

Haskell's standard prelude defines a *monadic map* for lists, which lifts the standard map on lists to the Kleisli category:

$$mapM :: Monad\ m \Rightarrow (a \to m\ b) \to ([a] \to m\ [b])$$

Fokkinga [3] showed how to generalize this from lists to an arbitrary regular functor, polytypically. Several authors [23, 25, 13, 26, 18] 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 idioms, 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 [16, 17] that monads do not compose in general, whereas idioms 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, idioms 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.* [25] call it *passive traversal*, Meertens [22] calls it *functor pulling*, and Pardo [26] 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 [20] recently introduced the notion of an *idiom* or *applicative functor* 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 prefer the original term, not least because it lends itself nicely to adjectival uses, as in ‘idiomatic traversal’.) Monads [24, 29] allow the expression of effectful computations within a purely functional language, but they do so by encouraging an *imperative* [27] programming style; in fact, Haskell’s monadic **do** notation is explicitly designed to give an imperative feel. Since idioms 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, idioms 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.

Idioms are captured in Haskell by the following type class. (In contrast to McBride and Paterson, but without loss of generality, we make make *Idiom* a subclass of *Functor*.)

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

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

In addition to those of the *Functor* class, idioms 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 idiom operators to be rewritten into a canonical form, consisting of a pure function applied to a series of idiomatic arguments: $\mathit{pure}\ f \circledast u_1 \circledast \ldots \circledast u_n$. (In case the reader feels the need for some intuition for these laws, we refer them forwards to the stream Naperian idiom discussed in Section 3.2 below.)

3.1. Monadic idioms

Idioms generalize monads; every monad induces an idiom, with the following operations.

```
instance Monad m ⇒ Idiom m where
  pure a    = do {return a}
  mf ⊛ mx = do {f ← mf; x ← mx; return (f x)}
```

(Taken literally as a Haskell declaration, this code yields overlapping instances; consider it therefore as a statement of intent instead.) The *pure* operator for a monadic idiom is just the *return* of the monad; idiomatic application ⊛ is monadic application, here with the effects of the function preceding those of the argument. 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 idiom laws.

One of McBride and Paterson’s motivating examples of an idiom arises from the environment monad, for which *pure* and ⊛ turn out to be the *K* and *S* combinators, respectively.

```
newtype Env e a = Env{unEnv :: e → a}
```

3.2. Naperian idioms

The ‘bind’ operation of a monad allows the result of one computation to affect the choice and ordering of effects of subsequent operations. Idioms in general provide no such possibility; indeed, as we have seen, every expression built just from the idiom 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* [15], which separate statically-analysable shape from dynamically-determined contents. Static shape suggests another class of idioms, exemplified by the stream functor.

```
data Stream a = SCons a (Stream a)
instance Idiom 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 idiom is the most accessible one for providing some intuition for the idiom laws. Computations within the stream idiom tend to perform a transposition of results; there appears to be some connection with what Kühne [19] 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 a datatype *Naperian*, because it supports 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 [14], but we leave the investigation of that connection for future work.

3.3. Monoidal idioms

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

```
newtype K b a = K {unK :: b}
instance Monoid b ⇒ Idiom (K b) where
  pure _ = K ∅
  x ⊛ y  = K (unK x ⊕ unK y)
```

Computations within this idiom 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 idioms in three different ways: monadic, with cartesian product; Naperian, with zip; monoidal, with concatenation.)

3.4. Combining idioms

Like monads, idioms 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}
fork :: (a → m b) → (a → n b) → a → Prod m n b
fork f g a = Prod (f a) (g a)
instance (Idiom m, Idiom n) ⇒ Idiom (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, idioms are also closed under composition; so two sequentially-dependent idiomatic effects can generally be fused into one, their composition.

```
data Comp m n a = Comp{unComp :: m (n a)}
instance (Idiom m, Idiom n) ⇒ Idiom (Comp m n) where
  pure x    = Comp (pure (pure x))
  mf ⊛ mx = Comp (pure (⊛) ⊛ unComp mf ⊛ unComp mx)
```

We will see examples of both of these combinations in Section 4.1.

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 within the idiom.

```
traverseList :: Idiom m ⇒ (a → m b) → [a] → m [b]
traverseList f []       = pure []
traverseList f (x : xs) = pure (:) ⊛ f x ⊛ traverseList f xs
```

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

```
distList :: Idiom m ⇒ [m a] → m [a]
distList = traverseList id
```

The ‘map within the idiom’ pattern of traversal for lists generalizes to any (finite) functorial data structure, even non-regular ones. We capture this via a type class of *Traversable* data structures (again, unlike McBride and Paterson, but without loss of generality, we subclass *Functor*):

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

Although *traverse* and *dist* are interdefinable (intuitively, *dist* is to *traverse* as monadic join μ is to bind ≫=), so only one needs to be given, defining *traverse* and inheriting *dist* is usually simpler and more efficient than vice versa.

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
instance Traversable Tree where
  traverse f (Leaf a)  = pure Leaf ⊛ f a
  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 $\circledast$ implicitly; apart from these brackets, the definition then looks exactly like a definition of *fmap*. This definition could be derived automatically [11], or given polytypically once and for all, assuming some universal representation of datatypes such as sums and products [10] or regular functors [7]:

```
class Bifunctor s ⇒ Bitraversable s where
  bidist :: Idiom 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 specialized to the identity idiom, traversal reduces to the functorial map over lists.

```
newtype Id a = Id{unId :: a}
instance Idiom Id where
  pure a    = Id a
  mf ⊛ mx = Id ((unId mf) (unId mx))
```

In the case of a monadic idiom, 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 idioms. 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 idioms in general.

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

For a monoidal idiom, traversal accumulates values. For example, interpreted in the integer monoidal idiom, traversal accumulates a sum of integer measures of the elements.

```
tsum :: (a → Int) → [a] → Int
tsum f = unK ∘ traverseList (K ∘ f)
```

4. TRAVERSALS AS ITERATORS

In this section, we show some representative examples of ITERATORS 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 traversal above is parametrized along two further dimensions: the datatype being traversed, and the idiom in which the traversal is interpreted. Specializing the latter to the lists-as-monoid idiom yields a generic *contents* operation:

```
contents :: Traversable t ⇒ t a → [a]
contents = unK ∘ traverse (K ∘ single)
single :: a → [a]
single a = [a]
```

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 [15]. The other half is obtained simply by a map, which is to say, a traversal interpreted in the identity idiom:

```
shape :: Traversable t ⇒ t a → t ()
shape = unId ∘ traverse (Id ∘ bang)
```

$$\begin{aligned}&\mathit{bang} :: a \to ()\\&\mathit{bang} = \mathit{const}\ ()\end{aligned}$$

Of course, it is trivial to combine these two traversals to obtain both halves of the decomposition as a single function, but doing this by tupling in the obvious way entails two traversals over the data structure. Is it possible to fuse the two traversals into one? The product of idioms allows exactly this, yielding the decomposition of a data structure into shape and contents in a single pass:

$$\begin{aligned}&\mathit{decompose}' :: \mathit{Traversable}\ t \Rightarrow t\ a \to \mathit{Prod}\ \mathit{Id}\ (K\ [a])\ (t\ ())\\&\mathit{decompose}' = \mathit{traverse}\ (\mathit{fork}\ (\mathit{Id} \circ \mathit{bang})\ (K \circ \mathit{single}))\end{aligned}$$

It is then a simple matter of removing the tags for the product of idioms and the idioms themselves:

$$\begin{aligned}&\mathit{decompose} :: \mathit{Traversable}\ t \Rightarrow t\ a \to (t\ (), [a])\\&\mathit{decompose} = \mathit{getPair} \circ \mathit{decompose}'\\&\mathit{getPair} :: \mathit{Prod}\ \mathit{Id}\ (K\ b)\ a \to (a, b)\\&\mathit{getPair}\ \mathit{xy} = (\mathit{unId}\ (\mathit{pfst}\ \mathit{xy}), \mathit{unK}\ (\mathit{psnd}\ \mathit{xy}))\end{aligned}$$

Jay [25] gives a similar decomposition, but using a customized combination of monads; we believe the above 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 forms another monad, but that is not the case for monads in general.)

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

$$\begin{aligned}&\mathit{takeHead} :: \mathit{State}\ [a]\ (\mathit{Maybe}\ a)\\&\mathit{takeHead} = \mathbf{do}\ \{\mathit{xs} \leftarrow \mathit{get}; \mathbf{case}\ \mathit{xs}\ \mathbf{of}\\&\qquad\qquad\qquad [\,] \qquad\quad \to \mathit{return}\ \mathit{Nothing}\\&\qquad\qquad\qquad (y : \mathit{ys}) \to \mathbf{do}\ \{\mathit{put}\ \mathit{ys}; \mathit{return}\ (\mathit{Just}\ y)\}\}\end{aligned}$$

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

$$\begin{aligned}&\mathit{reassemble}' :: \mathit{Traversable}\ t \Rightarrow t\ () \to \mathit{State}\ [a]\ (\mathit{Maybe}\ (t\ a))\\&\mathit{reassemble}' = \mathit{unComp} \circ \mathit{traverse}\ (\lambda() \to \mathit{Comp}\ \mathit{takeHead})\end{aligned}$$

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

$$\begin{aligned}&\mathit{reassemble} :: \mathit{Traversable}\ t \Rightarrow (t\ (), [a]) \to \mathit{Maybe}\ (t\ a)\\&\mathit{reassemble}\ (x, \mathit{ys}) = \mathit{allGone}\ (\mathit{runState}\ (\mathit{reassemble}'\ x)\ \mathit{ys})\\&\mathit{allGone} :: (\mathit{Maybe}\ (t\ a), [a]) \to \mathit{Maybe}\ (t\ a)\\&\mathit{allGone}\ (\mathit{mt}, [\,]) \qquad\ = \mathit{mt}\\&\mathit{allGone}\ (\mathit{mt}, (\_ : \_)) = \mathit{Nothing}\end{aligned}$$

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.

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

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

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 idiom of the *State* monad to capture the counting:

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

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.

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

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

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 *idiom adapter*:

$$\begin{array}{l} \mathbf{newtype}\ \mathit{Backwards}\ m\ a = \mathit{Backwards}\{\mathit{runBackwards} :: m\ a\} \end{array}$$

$$\begin{array}{l} \mathbf{instance}\ \mathit{Idiom}\ m \Rightarrow \mathit{Idiom}\ (\mathit{Backwards}\ m)\ \mathbf{where} \\ \quad \mathit{pure} \;\; = \mathit{Backwards} \circ \mathit{pure} \\ \quad f \circledast x = \mathit{Backwards}\ (\mathit{pure}\ (\mathit{flip}\ (\$)) \circledast \mathit{runBackwards}\ x \circledast \mathit{runBackwards}\ f) \end{array}$$

Informally, *Backwards m* is an idiom if *m* is, but any effects happen in reverse; this provides the symmetric ‘backwards’ embedding of monads into idioms referred to in Section 3.1.

Such an adapter can be parcelled up existentially:

$$\mathbf{data}\ \mathit{IAdapter}\ m = \forall g.\ \mathit{Idiom}\ (g\ m) \Rightarrow \mathit{IAdapter}\ (\forall a.\ m\ a \to g\ m\ a)\ (\forall a.\ g\ m\ a \to m\ a)$$

$$\begin{array}{l} \mathit{backwards} :: \mathit{Idiom}\ m \Rightarrow \mathit{IAdapter}\ m \\ \mathit{backwards} = \mathit{IAdapter\ Backwards\ runBackwards} \end{array}$$

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

$$\begin{array}{l} \mathit{ptraverse} :: (\mathit{Idiom}\ m, \mathit{Traversable}\ t) \Rightarrow \mathit{IAdapter}\ m \to (a \to m\ b) \to t\ a \to m\ (t\ b) \\ \mathit{ptraverse}\ (\mathit{IAdapter\ wrap\ unwrap})\ f = \mathit{unwrap} \circ \mathit{traverse}\ (\mathit{wrap} \circ f) \end{array}$$

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

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

5. LAWS OF TRAVERSE

The *traverse* operator is *ad-hoc datatype-generic*: the type class *Traversable* determines its signature, but one must provide a definition of *dist* or *traverse* independently for each datatype that implements *Traversable*. In line with other instances of ad-hoc datatype-genericity such as *Functor* and *Monad*, we should consider also what properties the definition ought to enjoy.

5.1. Free theorems

The free theorem [28] arising from the type of *dist* is

$$\mathit{dist} \circ \mathit{fmap}\ (\mathit{fmap}\ k) = \mathit{fmap}\ (\mathit{fmap}\ k) \circ \mathit{dist}$$

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

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

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

5.2. Composition

We have seen that idioms compose: there is an identity idiom *Id* and, for any two idioms *m* and *n*, a composite idiom *Comp m n*. We impose on implementations of *dist* the constraint of respecting this compositional structure. Specifically, the distributor *dist* respects the identity idiom:

$$\mathit{dist} \circ \mathit{fmap}\ \mathit{Id} = \mathit{Id}$$

and the composition of idioms:

$$\mathit{dist} \circ \mathit{fmap}\ \mathit{Comp} = \mathit{Comp} \circ \mathit{fmap}\ \mathit{dist} \circ \mathit{dist}$$

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 idiom is essentially just *fmap*, as mentioned in Section 3.5. The second provides a fusion rule for traversals, whereby two consecutive traversals can be fused into one.

5.3. Naturality

We also impose the constraint that the distributor *dist* is *natural in the idiom*, as follows. An *idiom transformation* $\phi :: m\ a \rightarrow n\ a$ from idiom *m* to idiom *n* is a polymorphic function (natural transformation) that respects the idiom structure:

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

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

Then *dist* must satisfy the following naturality property: for idiom 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 idiom transformation from idiom *Id* to idiom *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:

$$\begin{array}{l}\textbf{instance}\ \mathit{Traversable\ Tree}\ \textbf{where}\\ \quad \mathit{traverse}\ f\ (\mathit{Leaf}\ a) = \mathit{pure\ Leaf} \circledast f\ a\\ \quad \mathit{traverse}\ f\ (\mathit{Bin}\ t\ u) = \mathit{pure\ Bin} \circledast \mathit{traverse}\ f\ u \circledast \mathit{traverse}\ f\ t\end{array}$$

With this definition, $\mathit{traverse\ pure} = \mathit{pure} \circ \mathit{mirror}$, 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 idiom. Similarly, a definition with two copies of $\mathit{traverse}\ f\ t$ and none of $\mathit{traverse}\ f\ u$ 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 idiom; the effect of the reversal is that elements of the tree are traversed ‘from right to left’.

$$\begin{array}{l}\textbf{instance}\ \mathit{Traversable\ Tree}\ \textbf{where}\\ \quad \mathit{traverse}\ f\ (\mathit{Leaf}\ a) = \mathit{pure\ Leaf} \circledast f\ a\\ \quad \mathit{traverse}\ f\ (\mathit{Bin}\ t\ u) = \mathit{pure}\ (\mathit{flip\ Bin}) \circledast \mathit{traverse}\ f\ u \circledast \mathit{traverse}\ f\ t\end{array}$$

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

5.4. Composition of monadic traversals

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

$$\begin{array}{l}(\bullet) :: \mathit{Monad}\ m \Rightarrow (b \to m\ c) \to (a \to m\ b) \to (a \to m\ c)\\ (f \bullet g)\ x = \textbf{do}\ \{y \leftarrow g\ x; z \leftarrow f\ y; \mathit{return}\ z\}\end{array}$$

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

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

When interpreted in the idiom of a commutative monad m, traversals with $f :: b \to m\ c$ and $g :: a \to m\ b$ fuse:

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

This follows from the fact that $\mu \circ \mathit{unComp}$ forms an idiom transformation from $\mathit{Comp}\ m\ m$ 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 idioms in Section 5.2 allows two different idioms 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 idiom, 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.

5.5. No duplication

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 = 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. EXAMPLE

As a small example of fusion of traversals, we will consider the familiar *repmin* problem [2]. The problem here is to take a binary tree of integers, compute the minimum element, then replace every element of the tree by that minimum — but to do so in a single traversal rather than the obvious two. Our point here is not the circularity for which the problem was originally introduced, but simply an illustration of the two kinds of traversal (mapping and accumulating) and their fusion.

Flushed with our success at capturing different kinds of traversal idiomatically, we might try computing the minimum in a monoidal idiom,

```
newtype Min a = Min{unMin :: a}
instance (Ord a, Bounded a) ⇒ Monoid (Min a) where
  ∅     = Min maxBound
  x ⊕ y = Min (unMin x ‘min‘ unMin y)
tmin1 :: (Ord a, Bounded a) ⇒ a → K (Min a) a
tmin1 = K ∘ Min
```

and replacing in the idiom of the environment monad.

```
trep1 :: a → Env b b
trep1 = λa → Env id
```

These two combine elegantly (modulo the type isomorphisms):

```
trepmin1 :: (Ord a, Bounded a) ⇒ Tree a → Tree a
trepmin1 t = unEnv (traverse trep1 t) (unMin $ unK $ traverse tmin1 t)
```

However, the two traversals do not fuse: the first traversal computes the minimum and discards the tree, which then needs to be reintroduced for the second traversal.

(Notice that this program is generic in the data structure traversed; the only constraint is that it should be *Traversable*.

```
grepmin1 :: (Ord a, Bounded a, Traversable t) ⇒ t a → t a
grepmin1 t = unEnv (traverse trep1 t) (unMin $ unK $ traverse tmin1 t)
```

The same observation will apply to all versions of the program in this section; but to avoid carrying the *Traversable t* context around, we will specialize to *Tree*.)

Apparently the traversal that computes the minimum ought to retain and return the tree as well; this suggests using the idiom of the *State* monad. The state records the minimum element; the first traversal updates this state, and the second traversal reads from it.

$$\begin{array}{l}
\mathit{tmin}_2 :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{State}\ a\ a \\
\mathit{tmin}_2\ a = \mathbf{do}\ \{b \leftarrow \mathit{get}; \mathit{put}\ (\mathit{min}\ a\ b); \mathit{return}\ a\} \\
\mathit{trep}_2\ :: a \rightarrow \mathit{State}\ a\ a \\
\mathit{trep}_2\ a\ = \mathit{get}
\end{array}$$

Again, these compose.

$$\begin{array}{l}
\mathit{trepmin}_2 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{trepmin}_2\ t = \mathit{fst}\ \$\ \mathit{runState}\ ((\mathit{traverse}\ \mathit{trep}_2 \bullet \mathit{traverse}\ \mathit{tmin}_2)\ t)\ \mathit{maxBound}
\end{array}$$

But when we try to apply the fusion law for monadic traversals, we are forced to admit that the *State* monad is the epitome of a non-commutative monad, and in particular that the two stateful operations tmin_2 and trep_2 do not commute; therefore, the two traversals do not fuse.

There is a simple way to make the two stateful operations commute, and that is by giving them separate parts of the state on which to act. The following implementation uses a pair as the state; the first component is where the minimum element is accumulated, and the second component holds what is copied across the tree.

$$\begin{array}{l}
\mathit{tmin}_3 :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{State}\ (a,b)\ a \\
\mathit{tmin}_3\ a = \mathbf{do}\ \{(a',b) \leftarrow \mathit{get}; \mathit{put}\ (\mathit{min}\ a\ a', b); \mathit{return}\ a\} \\
\mathit{trep}_3\ :: a \rightarrow \mathit{State}\ (a,b)\ b \\
\mathit{trep}_3\ a\ = \mathbf{do}\ \{(a',b) \leftarrow \mathit{get}; \mathit{return}\ b\}
\end{array}$$

Of course, the whole point of the exercise is that the two parts of the state *should* interact; but with lazy evaluation we can use the standard circular programming trick [2] to tie the two together, outside the traversal.

$$\begin{array}{l}
\mathit{trepmin}_3 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{trepmin}_3\ t = \mathbf{let}\ (u,(m,\_)) = \mathit{runState}\ \mathit{iteration}\ (\mathit{maxBound}, m)\ \mathbf{in}\ u \\
\quad \mathbf{where}\ \mathit{iteration} = (\mathit{traverse}\ \mathit{trep}_3 \bullet \mathit{traverse}\ \mathit{tmin}_3)\ t
\end{array}$$

Now, although the *State* monad is not commutative, the two stateful operations tmin_3 and trep_3 commute (because they do not interfere), and the two traversals may be fused into one.

$$\begin{array}{l}
\mathit{trepmin}'_3 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{trepmin}'_3\ t = \mathbf{let}\ (u,(m,\_)) = \mathit{runState}\ \mathit{iteration}\ (\mathit{maxBound}, m)\ \mathbf{in}\ u \\
\quad \mathbf{where}\ \mathit{iteration} = \mathit{traverse}\ (\mathit{trep}_3 \bullet \mathit{tmin}_3)\ t
\end{array}$$

Modifying the stateful operations in this way to keep them from interfering is not scalable, and it is not clear whether this trick is possible in general anyway. Fortunately, idioms provide a much simpler means of fusion. Using the same single-component stateful operations tmin_2 and trep_2 as above, but dispensing with Kleisli composition, we get the following composition of traversals.

$$\begin{array}{l}
\mathit{trepmin}_4 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{trepmin}_4\ t = \mathbf{let}\ (\mathit{sf}, m) = \mathit{runState}\ \mathit{iteration}\ \mathit{maxBound}\ \mathbf{in}\ \mathit{fst}\ (\mathit{runState}\ \mathit{sf}\ m) \\
\quad \mathbf{where}\ \mathit{iteration} = \mathit{fmap}\ (\mathit{traverse}\ \mathit{trep}_2)\ (\mathit{traverse}\ \mathit{tmin}_2\ t)
\end{array}$$

Kleisli composition has the effect of flattening two levels into one; here we have to deal with both levels separately, hence the two occurrences of *runState*. The payback is that fusion of idiomatic traversals applies without side conditions!

$$\begin{array}{l}
\mathit{trepmin}'_4 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{trepmin}'_4\ t = \mathbf{let}\ (\mathit{sf}, m) = \mathit{runState}\ \mathit{iteration}\ \mathit{maxBound}\ \mathbf{in}\ \mathit{fst}\ (\mathit{runState}\ \mathit{sf}\ m) \\
\quad \mathbf{where}\ \mathit{iteration} = \mathit{unComp}\ \$\ \mathit{traverse}\ (\mathit{Comp} \circ \mathit{fmap}\ \mathit{trep}_2 \circ \mathit{tmin}_2)\ t
\end{array}$$

Note that the Kleisli composition of two monadic computations imposes the constraint that both computations are in the same monad; in our example above, both computing the minimum and distributing the result use the *State* monad. However, these two monadic computations are actually rather different in structure, and use different aspects of the *State* monad: the first writes, whereas the second reads. We could capture this observation directly by using two different monads, each tailored for its particular use.

$$\begin{aligned}
&\mathit{tmin}_5 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow a \rightarrow \mathit{Writer}\ (\mathit{Min}\ a)\ a\\
&\mathit{tmin}_5\ a = \mathbf{do}\ \{\mathit{tell}\ (\mathit{Min}\ a); \mathit{return}\ a\}\\
&\mathit{trep}_5\ :: a \rightarrow \mathit{Reader}\ a\ a\\
&\mathit{trep}_5\ a\ = \mathit{ask}
\end{aligned}$$

The use of two different monads like this rules out Kleisli composition. However, idiomatic composition handles two different idioms (and hence two different monads) with aplomb.

$$\begin{aligned}
&\mathit{trepmin}_5 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a\\
&\mathit{trepmin}_5\ t = \mathbf{let}\ (r, m) = \mathit{runWriter}\ \mathit{iteration}\ \mathbf{in}\ \mathit{runReader}\ r\ (\mathit{unMin}\ m)\\
&\quad \mathbf{where}\ \mathit{iteration} = \mathit{fmap}\ (\mathit{traverse}\ \mathit{trep}_5)\ (\mathit{traverse}\ \mathit{tmin}_5\ t)
\end{aligned}$$

These two traversals fuse in exactly the same way as before.

$$\begin{aligned}
&\mathit{trepmin}'_5 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a\\
&\mathit{trepmin}'_5\ t = \mathbf{let}\ (r, m) = \mathit{runWriter}\ \mathit{iteration}\ \mathbf{in}\ \mathit{runReader}\ r\ (\mathit{unMin}\ m)\\
&\quad \mathbf{where}\ \mathit{iteration} = \mathit{unComp}\ \$\ \mathit{traverse}\ (\mathit{Comp} \circ \mathit{fmap}\ \mathit{trep}_5 \circ \mathit{tmin}_5)\ t
\end{aligned}$$

7. CONCLUSIONS

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 [8]. 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 also hope to investigate the categorical structure of *dist* further: naturality in the idiom appears to be related to Beck's distributive laws [1], 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 idioms this would never have happened. As well as the clear debt we owe to [20], 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 idiom.

REFERENCES

[1] J. Beck. Distributive laws. In B. Eckmann, editor, *LNM 80: Seminar on Triples and Categorical Homology Theory*, pages 119–140, 1969.

[2] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[3] M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Department INF, Universiteit Twente, June 1994.

[4] M. M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4):81–82, June 1990.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] J. Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *LNCS 2297: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 148–203. 2002.

[7] J. Gibbons. Origami programming. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave, 2003.

[8] J. Gibbons. Design patterns as higher-order datatype-generic programs. Submitted for publication, June 2006.

[9] J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1), Apr. 2001. Coalgebraic Methods in Computer Science.

[10] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In R. Backhouse and J. Gibbons, editors, *LNCS 2793: Summer School on Generic Programming*, pages 1–56, 2003.

[11] R. Hinze and S. Peyton Jones. Derivable type classes. In *International Conference on Functional Programming*, 2000.

[12] P. Jansson and J. Jeuring. PolyP – a polytypic programming language extension. In *Principles of Programming Languages*, pages 470–482, 1997.

[13] P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[14] B. Jay and P. Steckler. The functional imperative: Shape! In C. Hankin, editor, *LNCS 1381: European Symposium on Programming*, pages 139–53, Lisbon, Portugal, 1998.

[15] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[16] M. P. Jones and L. Duponcheel. Composing monads. Technical Report RR-1004, Department of Computer Science, Yale, Dec. 1993.

[17] D. J. King and P. Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*. Springer, 1993.

[18] O. Kiselyov and R. Lämmel. Haskell's Overlooked Object System. Draft; submitted for publication, 2005.

[19] T. Kühne. Internal iteration externalized. In R. Guerraoui, editor, *LNCS 1628: ECOOP*, pages 329–350, 1999.

[20] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, To appear.

[21] L. Meertens. Calculate polytypically! In H. Kuchen and S. D. Swierstra, editors, *LNCS 1140: Programming Language Implementation and Logic Programming*, pages 1–16, 1996.

[22] L. Meertens. Functor pulling. In R. Backhouse and T. Sheard, editors, *Workshop on Generic Programming*, Marstrand, Sweden, June 1998.

[23] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *LNCS 925: Advanced Functional Programming*, 1995.

[24] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[25] E. Moggi, G. Bellè, and C. B. Jay. Monads, shapely functors and traversals. In M. Hoffman, D. Pavlovic, and P. Rosolini, editors, *Category Theory in Computer Science*, 1999.

[26] A. Pardo. Combining datatypes and effects. In *Advanced Functional Programming*, 2005.

[27] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages*, pages 71–84, 1993.

[28] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[29] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the Marktoberdorf Summer School*, 1992.