FUNCTIONAL PEARL

Idioms: applicative programming with effects

CONOR MCBRIDE
University of Nottingham

ROSS PATERSON
City University, London

Abstract

In this paper, we introduce Idioms—an abstract characterisation of an applicative style of effectful programming, weaker than Monads and hence more widespread. Indeed, it is the ubiquity of this programming pattern which drew us to the abstraction. We shall take the same course in this paper, introducing the applicative pattern by diverse examples, then abstracting it to define the Idiom type class and associated laws. We compare this abstraction with monoids, monads and arrows, and identify the categorical structure of idioms.

1 Introduction

This is the story of a pattern which popped up time and again in our daily work, programming in Haskell (Peyton Jones, 2003), until the point where the temptation to abstract it became irresistable. Let us illustrate with some examples.

1.1 Sequencing commands

It is not unusual to execute a sequence of commands and collect the sequence of their responses:

```
sequence :: [IO x] → IO [x]
sequence    []     = return []
sequence (c : cs) = do
   x  ← c
   xs ← sequence cs
   return (x : xs)
```

In the $(c : cs)$ case, we collect the values of some effectful computations, which we then use as the arguments to a pure function (:). We could avoid the need for names to wire these values through to their point of usage if we had a kind of 'effectful application'. Fortunately, exactly such a thing lives in the standard Monad library. We may write

$$\begin{array}{l}
\mathsf{sequence} :: [\mathsf{IO}\ x] \to \mathsf{IO}\ [x] \\
\mathsf{sequence}\ \ []\ \ \ \ \ \ \ = \mathsf{return}\ [] \\
\mathsf{sequence}\ (c : cs) = \mathsf{return}\ (:)\ \text{`}\mathsf{ap}\text{`}\ c\ \text{`}\mathsf{ap}\text{`}\ \mathsf{sequence}\ cs
\end{array}$$

where the return operation, which every Monad must provide, lifts pure values to the effectful world, and the ap operation provides a kind of application within the the effectful world:

$$\begin{array}{l}
\mathsf{return} :: \mathsf{Monad}\ m \Rightarrow x \to m\ x \\
\\
\mathsf{ap} :: \mathsf{Monad}\ m \Rightarrow m\ (s \to t) \to m\ s \to m\ t \\
\mathsf{ap}\ mf\ ms = \mathbf{do} \\
\quad f \leftarrow mf \\
\quad s \leftarrow ms \\
\quad \mathsf{return}\ (f\ s)
\end{array}$$

If we could filter out the noise of the returns and aps, we could almost imagine that we are programming in a fairly standard applicative style, even though effects are present.

1.2 Transposing 'matrices'

Suppose we represent matrices (somewhat approximately) by lists of lists. It is not unusual to develop operations on matrices such as transposition.

$$\begin{array}{l}
\mathsf{transpose} :: [[x]] \to [[x]] \\
\mathsf{transpose}\ \ \ []\ \ \ \ \ \ \ \ = \mathsf{repeat}\ [] \\
\mathsf{transpose}\ (xs : xss) = \mathsf{zipWith}\ (:)\ xs\ (\mathsf{transpose}\ xss)
\end{array}$$

Now, the binary zipWith is one of a family of operations which 'vectorise' pure functions. As Daniel Fridlender and Mia Indrika (2000) point out, the entire family can be generated from repeat, which generates an infinite stream from its argument, and zapp, a kind of 'zippy' application.

$$\begin{array}{l}
\mathsf{repeat} :: x \to [x] \\
\mathsf{repeat}\ x = x : \mathsf{repeat}\ x \\
\\
\mathsf{zapp} :: [s \to t] \to [s] \to [t] \\
\mathsf{zapp}\ (f : fs)\ (s : ss) = f\ s : \mathsf{zapp}\ fs\ ss \\
\mathsf{zapp}\ _\ \ \ \ \ \ \ \ _\ \ \ \ \ \ \ \ = []
\end{array}$$

The general scheme is as follows:

$$\begin{array}{l}
\mathsf{zipWith}_n\ ::\ (s_1 \to \cdots \to s_n \to t) \to [s_1] \to \cdots \to [s_n] \to [t] \\
\mathsf{zipWith}_n\ f\ ss_1\ \ldots\ ss_n = \mathsf{repeat}\ f\ \text{`}\mathsf{zapp}\text{`}\ ss_1\ \text{`}\mathsf{zapp}\text{`}\ \ldots\ \text{`}\mathsf{zapp}\text{`}\ ss_n
\end{array}$$

In particular, transposition becomes

$$\begin{array}{l}
\mathsf{transpose} :: [[x]] \to [[x]] \\
\mathsf{transpose}\ \ \ []\ \ \ \ \ \ \ \ = \mathsf{repeat}\ [] \\
\mathsf{transpose}\ (xs : xss) = \mathsf{repeat}\ (:)\ \text{`}\mathsf{zapp}\text{`}\ xs\ \text{`}\mathsf{zapp}\text{`}\ \mathsf{transpose}\ xss
\end{array}$$

If we could filter out the noise of the repeats and zapps, we could almost imagine that we are programming in a fairly standard applicative style, even though we are working with vectors.

1.3 Evaluating expressions

When implementing an evaluator for a language of expressions, it is not unusual to pass around an environment, giving values to the free variables. Here is a very simple example

```
data Exp x = Var x
           | Val Int
           | Add (Exp x) (Exp x)

eval :: Exp x → Env x → Int
eval (Var x)     γ = fetch x γ
eval (Val i)     γ = i
eval (Add p q) γ = eval p γ + eval q γ
```

where Env x is some notion of environment and fetch x projects the value for the variable x.

Threading the environment explicitly clutters the code unnecessarily, but we can remedy the situation with a little help from some very old friends:

```
eval :: Exp x → Env x → Int
eval (Var x)     = fetch x
eval (Val i)     = 𝕂 i
eval (Add p q) = 𝕂 (+) ‘𝕊‘ eval p ‘𝕊‘ eval q
```

where

$$\mathbb{K} :: x \to env \to x$$
$$\mathbb{K}\ x\ \gamma = x$$
$$\mathbb{S} :: (env \to s \to t) \to (env \to s) \to (env \to t)$$
$$\mathbb{S}\ ef\ \ es\ \gamma = (ef\ \gamma)\ (es\ \gamma)$$

If we could filter out the noise of the $\mathbb{K}$ and $\mathbb{S}$ combinators[1], we could almost imagine that we are programming in a fairly standard applicative style, even though we are abstracting over an environment.

1.4 Parser combinators

This isn't an example of a transpose — perhaps it belongs later?

A similar interface was proposed for parsers by Röjemo (1995) and developed by Swierstra and colleagues (Swierstra & Duponcheel, 1996; Baars *et al.*, 2004). Two primitive components are used to implement each production of a grammar:

```
succeed :: x → Parser x
(⊛)        :: Parser (s → t) → Parser s → Parser t
```

[1] also known as the return and ap of the ‘reader’ Monad

These productions generally take the form

$$\mathsf{succeed}\ \mathit{semantics} \circledast \mathit{parser}_1 \circledast \ldots \circledast \mathit{parser}_n$$

where the parser_i are the parsers for successive components of the production and *semantics* is the pure function which delivers the value of the whole from the values of the parts.

If we could filter out the noise of the succeeds and $\circledast$s, we could almost imagine that we are programming in a fairly standard applicative style, even though we are combining parsers.

2 The Idiom class

We have seen four examples of this 'pure function applied to funny arguments' pattern in apparently quite diverse fields—let us now abstract out what they have in common. In each example, there is a type constructor i which embeds the usual notion of value, but supports its *own peculiar way* of giving meaning to the usual applicative language—its *idiom*. We correspondingly introduce the Idiom class:

```
infixl 4 ⊛
class Idiom i where
  ι   :: x → i x
  (⊛) :: i (s → t) → i s → i t
```

This class generalises $\mathbb{S}$ and $\mathbb{K}$ from threading an environment to threading an effect in general.

We shall require the following laws for idioms:

identity	$\iota\ \mathsf{id} \circledast u$	$=$	u
composition	$\iota\ (\cdot) \circledast u \circledast v \circledast w$	$=$	$u \circledast (v \circledast w)$
homomorphism	$\iota\ f \circledast \iota\ x$	$=$	$\iota\ (f\ x)$
interchange	$u \circledast \iota\ x$	$=$	$\iota\ (\lambda f \to f\ x) \circledast u$

These laws capture the intuition that ι embeds pure computations into the pure fragment of an effectful world—the resulting computations may thus be shunted around freely, as long as the order of the genuinely effectful computations is preserved.

Exercise 1 (Idiom*s functorial*)
Use these laws to show that the following function makes any idiom a functor:

```
lift₁ :: Idiom i ⇒ (a → b) → i a → i b
lift₁ f u = ι f ⊛ u
```

Using these laws, any expression built from the Idiom combinators can be transformed to a canonical form in which a single pure function is 'applied' to the effectful parts in depth-first order:

$$\iota\ f \circledast is_1 \circledast \ldots \circledast is_n$$

Exercise 2 (*canonical form*)

Show how this is done. You will need all four laws. *Hint:* devise a four-phase algorithm to perform the transformation, with one phase for each law.

This canonical form captures the essence of programming in an Idiom—computations have a fixed structure, given by the pure function, and a sequence of subcomputations, given by the effectful arguments. We therefore find it convenient, at least within this paper, to write this form using a special notation

$$[\![f\ is_1\ \ldots\ is_n]\!]$$

The brackets indicate a shift into an idiom where a pure function is applied to a sequence of computations. Our intention is to provide a sufficient indication that effects are present without compromising the readability of the code.

Exercise 3 (*Coding* $[\![\ldots]\!]$)
Given the functionality of Glasgow Haskell Compiler's `-fglasgow-exts` option, show how to replace '$[\![$' and '$]\!]$' by identifiers iI and Ii whose computational behaviour delivers the above expansion. *Hint:* define an overloaded function idiomatic such that

$$\mathsf{idiomatic}\ u\ is_1\ \ldots\ is_n\ \mathsf{Ii} = u \circledast is_1 \circledast \ldots \circledast is_n$$

Any Monad can be made an Idiom, taking

```
instance Idiom MyMonad where
  ι         = return
  mf ⊛ ms = do
    f ← mf
    s ← ms
    return (f s)
```

In fact, we could also choose to perform mf after ms, which would preserve the structure of the computation but reverse the effects. We shall work left-to-right in this paper. Taking the monadic Idiom for IO and $(\rightarrow)\ s$, we get what we expect for sequence and eval:

```
sequence :: [IO x] → IO [x]
sequence   []     = ⟦[]⟧
sequence (c : cs) = ⟦(:) c (sequence cs)⟧

eval :: Exp x → Env x → Int
eval (Var x)     = fetch x
eval (Val i)     = ⟦i⟧
eval (Add p q) = ⟦(+) (eval p) (eval q)⟧
```

If we want to do the same for our transpose example, we shall have to take an instance for Idiom [] which supports 'vectorisation', rather than the library's 'list of successes' (Wadler, 1985) monad:

```
instance Idiom [] where
  ι   = repeat
  (⊛) = zapp
```

$$
\begin{array}{l}
\mathsf{transpose} :: [[x]] \to [[x]] \\
\mathsf{transpose}\ \ [\,] \qquad = [\![\,[\,]\,]\!] \\
\mathsf{transpose}\ (xs : xss) = [\![\,(:)\ xs\ (\mathsf{transpose}\ xss)\,]\!]
\end{array}
$$

Exercise 4 (*the colist* Monad)
Although repeat and zapp are not the return and ap of the usual Monad [] instance, they are none the less the return and ap of an alternative monad, more suited to the coinductive interpretation of []. What is the join :: [[x]] → [x] of this monad? Comment on the relative efficiency of this monad's ap and our zapp.

3 Threading Idioms through IFunctors

Have you noticed that sequence and transpose now look rather alike? The details which distinguish the two programs are inferred by the compiler from their types. Both are instances of the *idiom distributor* for lists:

$$
\begin{array}{l}
\mathsf{idist} :: \mathsf{Idiom}\ i \Rightarrow [i\ x] \to i\ [x] \\
\mathsf{idist}\ \ [\,] \qquad = [\![\,[\,]\,]\!] \\
\mathsf{idist}\ (ix : ixs) = [\![\,(:)\ ix\ (\mathsf{idist}\ ixs)\,]\!]
\end{array}
$$

It is not unusual to combine distribution with 'map'. For example, we can map some failure-prone operation (a function in $s \to \mathsf{Maybe}\ t$) across a list of inputs in such a way that any individual failure causes failure overall.

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Idiom\ Maybe}\ \textbf{where} \quad \text{-- the usual return and ap} \\
\quad \iota\ x \qquad\qquad\quad = \mathsf{Just}\ x \\
\quad \mathsf{Just}\ f \circledast \mathsf{Just}\ s = \mathsf{Just}\ (f\ s) \\
\quad \_ \qquad \circledast \_ \qquad = \mathsf{Nothing} \\
\mathsf{flakyMap} :: (s \to \mathsf{Maybe}\ t) \to [s] \to \mathsf{Maybe}\ [t] \\
\mathsf{flakyMap}\ f\ ss = \mathsf{idist}\ (\mathsf{fmap}\ f\ ss)
\end{array}
$$

As you can see, flakyMap traverses *ss* twice—once to apply *f*, and again to collect the results. More generally, it is preferable to define this idiomatic mapping operation directly, with a single traversal:

$$
\begin{array}{l}
\mathsf{imap} :: \mathsf{Idiom}\ i \Rightarrow (s \to i\ t) \to [s] \to i\ [t] \\
\mathsf{imap}\ f\ \ [\,] \qquad = [\![\,[\,]\,]\!] \\
\mathsf{imap}\ f\ (x : xs) = [\![\,(:)\ (f\ x)\ (\mathsf{imap}\ f\ xs)\,]\!]
\end{array}
$$

This is just the way you would implement the ordinary fmap for lists, but with the right-hand sides wrapped in $[\![\cdots]\!]$, lifting them into the idiom. Just like fmap, imap is a useful gadget to have for many data structures, hence we introduce the type class IFunctor, capturing functorial data structures through which idioms thread:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor}\ f\ \textbf{where} \\
\quad \mathsf{imap} :: \mathsf{Idiom}\ i \Rightarrow (s \to i\ t) \to f\ s \qquad \to i\ (f\ t) \\
\quad \mathsf{idist}\ :: \mathsf{Idiom}\ i \Rightarrow \qquad\qquad\quad f\ (i\ x) \to i\ (f\ x) \\
\quad \mathsf{idist}\ = \mathsf{imap\ id}
\end{array}
$$

Of course, we can recover an ordinary 'map' operator by taking i to be the *identity* idiom—the usual applicative idiom in which all computations are pure:

$$\textbf{newtype}\ \mathsf{Id}\ x = \mathsf{An}\{\mathsf{an} :: x\}$$

Haskell's **newtype** declarations allow us to shunt the syntax of types around without changing the run-time notion of value or incurring any run-time cost. The 'labelled field' notation allows us to define the projection $\mathsf{an} :: \mathsf{Id}\ x \to x$ at the same time as the constructor $\mathsf{An} :: x \to \mathsf{Id}\ x$. The usual applicative idiom has the usual application:

```
instance Idiom Id where
  ι x            = An x
  An f ⊛ An s = An (f s)
```

So, with the **newtype** signalling which idiom to thread, we have

$$\mathsf{fmap}\ f = \mathsf{an} \cdot \mathsf{imap}\ (\mathsf{An} \cdot f)$$

The rule-of-thumb 'imap is like fmap but with $[\![\cdots]\!]$ on the right' is good for first-order type constructors, such as lists, trees,

```
data Tree x = Leaf | Node (Tree x) x (Tree x)
instance IFunctor Tree where
  imap f  Leaf          = ⟦Leaf⟧
  imap f (Node l x r) = ⟦Node (imap f l) (f x) (imap f r)⟧
```

and even 'nested' types, like the de Bruijn λ-terms (Bird & Paterson, 1999), parametrised by their type of free variables:

```
data Term x = TVar x
            | TApp (Term x) (Term x)
            | TLam (Term (Maybe x))
instance IFunctor Term where
  imap g (TVar x)    = ⟦TVar (g x)⟧
  imap g (TApp f s) = ⟦TApp (imap g f) (imap g s)⟧
  imap g (TLam t)    = ⟦TLam (imap (imap g) t)⟧
```

Exercise 5 (*distributing* Term *and* Maybe)
What does the specialised function $\mathsf{idist} :: \mathsf{Term}\ (\mathsf{Maybe}\ x) \to \mathsf{Maybe}\ (\mathsf{Term}\ x)$ tell you about a term with a free variable?

However, not every Functor is an IFunctor.

Exercise 6 (Functor *versus* IFunctor)
Find a functor whose imap, if it were well-defined, would solve the Halting Problem.

We are far from the first to consider this distribution of one functor through another in a general way. Paul Hoogendijk and Roland Backhouse (1997) construct 'half-zip' operations in a relational setting. These distribute one regular type constructor through another, requiring and preserving 'compatibility of shape'—matrix transpose is a key example.

In a functional setting, Lambert Meertens (1998) exhibits sufficient criteria on functors i to yield an idist-like operator, taking $f\ (i\ x) \to i\ (f\ x)$ for every regular functor f (that is, 'ordinary' uniform datatype constructors with one parameter, constructed by recursive sums of products). Idiom i certainly satisfy these criteria, hence Meertens confirms our intuition that at least the regular type constructors can all be made instances of IFunctor.

Curiously, monads are not Meertens' primary example of functors to thread through a data structure, although he does suggest that this might always work. Rather, he seeks to generalise accumulation or 'crush' operators, such as flattening trees and summing lists. We shall turn to these in the next section.

4 Monoids: the Phantom Idioms

The data which one may sensibly accumulate have the Monoid structure:

```
class Monoid o where
  ∅ :: o
  (⊕) :: o → o → o
```

These operations must satisfy the usual laws:

left identity	$\emptyset \oplus x$	$=$	x
right identity	$x \oplus \emptyset$	$=$	x
associativity	$(x \oplus y) \oplus z$	$=$	$x \oplus (y \oplus z)$

The functional programming world is full of monoids—numeric types (with respect to zero and plus, or one and times), lists with respect to [] and ++, and many others—so generic technology for working with them could well prove to be useful. Fortunately, every monoid induces an idiom, albeit in a slightly peculiar way:

```
newtype Accy o x = Acc{acc :: o}
```

Accy o x is a *phantom* type (Leijen & Meijer, 1999)—its values have nothing to do with x, but it does yield the idiom of accumulating computations:

```
instance Monoid o ⇒ Idiom (Accy o) where
  ι _               = Acc ∅
  Acc o₁ ⊛ Acc o₂ = Acc (o₁ ⊕ o₂)
```

Now 'crushing' is just traversing an IFunctor, in the same way as with any other idiom, just as Meertens suggested:

```
icrush :: (IFunctor f, Monoid o) ⇒ (x → o) → f x → o
icrush m = acc · imap (Acc · m)

isum :: (IFunctor f, Monoid o) ⇒ f o → o
isum = icrush id
```

Operations like flattening and concatenation become straightforward:

```
flatten :: Tree x → [x]
flatten = icrush (:[])

concat :: [[x]] → [x]
concat = isum
```

We can extract even more work from instance inference if we use the type system to distinguish different monoids available for a given datatype. Here, we use the disjunctive structure of Bool to test for the presence of an element satisfying a given predicate:

```
newtype Mighty = Might{ might :: Bool }
instance Monoid Mighty where
  ∅                  = Might False
  Might b ⊕ Might c = Might (b ∨ c)
any :: IFunctor f ⇒ (x → Bool) → f x → Bool
any p = might · icrush (Might · p)
elem :: (Eq x, IFunctor f) ⇒ x → f x → Bool
elem x = any (≡ x)
```

This elem function behaves as usual for lists, but it is just as effective at telling whether a variable in x occurs free in a Term x.

Meanwhile, Bool also has a conjunctive structure:

```
newtype Musty = Must{ must :: Bool }
instance Monoid Musty where
  ∅                = Must True
  Must b ⊕ Must c = Must (b ∧ c)
all :: IFunctor f ⇒ (x → Bool) → f x → Bool
all p = must · icrush (Must · p)
boundedBy :: (Ord x, IFunctor f) ⇒ x → f x → Bool
boundedBy x = all (⩽ x)
```

Exercise 7 (*every* Ord *induces a* Monoid)
Show how every instance of Ord induces an instance of Monoid. Use icrush to define the partial function

```
max :: (IFunctor f, Ord x) ⇒ f x → x
```

computing the greatest x contained in its input and undefined if there are none.

Exercise 8 (*adverbial programming*)
Define an overloaded operator ily such that

```
fmap   = An ‘ily‘ imap
icrush = Acc ‘ily‘ imap
any    = Might ‘ily‘ icrush
all    = Must ‘ily‘ icrush
```

and so on.

5 Idiom versus Monad?

We have seen that every Monad can be made an Idiom via return and ap. Indeed, our three introductory examples of idioms involved the IO monad, the ‘reader’ monad

$(\rightarrow)\ s$ and a (non-standard) monad for (coinductive) lists. However, the Accy o idioms are not monadic: return can deliver $\emptyset$, but if you try to define

$$(\ggg\!=) :: \mathsf{Accy}\ o\ s \rightarrow (s \rightarrow \mathsf{Accy}\ o\ t) \rightarrow \mathsf{Accy}\ o\ t$$

you'll find it tricky to extract an s from the first argument—all you get is an o. Correspondingly, there is no way to apply the second argument, and hence no way to accumulate its output. The $\circledast$ for Accy o is not the ap of a monad.

So now we know: there are strictly more Idioms than Monads. Should we just throw the Monad class away and use Idiom instead? Of course not! The reason there are fewer monads is just that the Monad structure is more powerful. Intuitively, the $(\ggg\!=) :: m\ s \rightarrow (s \rightarrow m\ t) \rightarrow m\ t$ of some Monad m allows the value returned by one computation to influence the choice of another, whereas $\circledast$ keeps the structure of a computation fixed, just sequencing the effects. For example, one may write

```
miffy :: Monad m ⇒ m Bool → m t → m t → m t
miffy mb mt me = do
  b ← mb
  if b then mt else me
```

so that the value of mb will choose between the *computations* mt and me, performing only one, whilst

```
iffy :: Idiom i ⇒ i Bool → i t → i t → i t
iffy ib it ie = ⟦ cond ib it ie ⟧ where
  cond b t e = if b then t else e
```

performs the effects of all three computations, using the value of ib to choose only between the *values* of it and ie. This can be a bad thing: for example,

$$\mathsf{iffy}\ [\![\,\mathsf{True}\,]\!]\ [\![\,t\,]\!]\ \mathsf{Nothing} = \mathsf{Nothing}$$

where

$$\mathsf{miffy}\ [\![\,\mathsf{True}\,]\!]\ [\![\,t\,]\!]\ \mathsf{Nothing} = [\![\,t\,]\!]$$

However, if you are working with miffy, it is probably because the condition is an expression with effectful components, so the idiom syntax provides quite a convenient extension to the monadic toolkit:

$$\mathsf{miffy}\ [\![\,(\leqslant)\ \mathit{getSpeed}\ \mathit{getSpeedLimit}\,]\!]\ \mathit{stepOnIt}\ \mathit{check4Cops}$$

The moral is this: if you've got an Idiom, that's good; if you've also got a Monad, that's even better! And the dual of the moral is this: if you want a Monad, that's good; if you only want an Idiom, that's even better! The weakness of idioms makes them easier to construct from components. In particular, although only certain pairs of monads are composable (Barr & Wells, 1984), the Idiom class is *closed under composition*,

```
newtype (i ∘ j) x = Comp{comp :: (i (j x))}
```

just by lifting the inner idiom operations to the outer idiom layer:

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{Idiom}\ i, \mathsf{Idiom}\ j) \Rightarrow \mathsf{Idiom}\ (i \circ j)\ \textbf{where} \\
\quad \iota\ x \qquad\qquad\qquad\qquad\quad = \mathsf{Comp}\ [\![(\iota\ x)]\!] \\
\quad \mathsf{Comp}\ \mathit{fij} \circledast \mathsf{Comp}\ \mathit{sij} = \mathsf{Comp}\ [\![(\circledast)\ \mathit{fij}\ \mathit{sij}]\!]
\end{array}
$$

As a consequence, the composition of two monads may not be a monad, but it is certainly an idiom. For example, IO ∘ Maybe is an idiom in which computations have a notion of 'failure' and 'prioritised choice', even if their 'real world' side-effects cannot be undone.

Exercise 9 (monads and accumulation)
We began this section by observing that Accy o is not a monad. Given Monoid o, define Accy o as the composition of two monadic idioms.

6 Idioms **lifting** Monoids

Idioms, IFunctors and Monoids give us the basic building blocks for a lot of routine programming. Every Idiom i can be used to lift monoids, as follows

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{Idiom}\ i, \mathsf{Monoid}\ o) \Rightarrow \mathsf{Monoid}\ (i\ o)\ \textbf{where} \\
\quad \emptyset \qquad\quad = [\![\emptyset]\!] \\
\quad xc \oplus yc = [\![(\oplus)\ xc\ yc]\!]
\end{array}
$$

although we shall have to choose individually the idioms i for which we apply this scheme. If we let IO lift monoids in this way, then we acquire the sequential composition of commands for the price of the trivial monoid:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Monoid}\ ()\ \textbf{where} \\
\quad \emptyset \qquad = () \\
\quad \_ \oplus \_ = ()
\end{array}
$$

Now one of the many specialised types of isum is $[\mathsf{IO}\ ()] \to \mathsf{IO}\ ()$. What does it do? Well, you know how to traverse a list, you know how to thread input/output, and you know how to combine (), so what do you think it does?

Which idioms should lift monoids? It would be disconcerting if the defaut $\oplus$ for $[x]$ were other than $+\!\!+$. Our rule of thumb is to prefer any natural monoid structure possessed by an idiom, but to lift monoids if no such structure presents itself. Correspondingly, we allow the environment-threading 'reader' to lift monoids pointwise: this is equivalent to taking

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Monoid}\ o \Rightarrow \mathsf{Monoid}\ (s \to o)\ \textbf{where} \\
\quad \emptyset \qquad = \lambda s \to \emptyset \\
\quad f \oplus g = \lambda s \to f\ s \oplus g\ s
\end{array}
$$

Pointwise lifting is quite powerful. For a start, it makes the well-known parser type from (Hutton & Meijer, 1998)

$$\mathsf{String} \to [(x, \mathsf{String})]$$

a monoid without further ado.

With a little more effort, we can use lifted Boolean monoids to do testing by the batch. For example,

$$\mathsf{elemInCommon} :: \mathsf{Eq}\ x \Rightarrow [x] \to [x] \to \mathsf{Bool}$$
$$\mathsf{elemInCommon}\ xs\ ys = \mathsf{might}\ ((\lambda x\ y \to \mathsf{Might}\ (x \equiv y))\ \text{`icrush`}\ xs\ \text{`icrush`}\ ys)$$

tests whether two lists have an element in common. The first icrush computes a batch of tests—one for each x in xs—and combines them disjunctively; the second icrush applies the combined test to each y in ys and takes the disjunction of the results. Note that we only structure we require of lists here is IFunctor [ ]. Correspondingly, we could type this function to operate on arbitrary traversible data structures without changing its code.

One casualty of our choice to let the $(\to)\ s$ idiom lift monoids is the library choice to make endomorphisms $s \to s$ a monoid with respect to id and $(\cdot)$. We think that wrapping endomorphisms in a **newtype** is a small price to pay for our iterable pointwise lifting, especially as id and flip $(\cdot)$ also make endomorphisms a monoid.

Exercise 10 (fast reverse)
Use the 'flip $(\cdot)$' monoid to define the 'fast reverse' function as an icrush on lists.

7 Lost Sheep

We can also take the product of idioms, taking a pair of notions of computation to a notion of pairs of computations.

$$\textbf{data}\ (i \boxtimes j)\ x = i\ x \boxdot j\ x$$
$$\textbf{instance}\ (\mathsf{Idiom}\ i, \mathsf{Idiom}\ j) \Rightarrow \mathsf{Idiom}\ (i \boxtimes j)\ \textbf{where}$$
$$\iota\ x = \iota\ x \boxdot \iota\ x$$
$$(fi \boxdot fj) \circledast (si \boxdot sj) = (fi \circledast si) \boxdot (fj \circledast sj)$$

7.1 *Combining* IFunctors

Just drop this? The IFunctor class is closed under the usual mechanisms for constructing first-order data structures. Firstly, identity and composition:

$$\textbf{instance}\ \mathsf{IFunctor}\ \mathsf{Id}\ \textbf{where}$$
$$\mathsf{imap}\ f\ (\mathsf{An}\ x) = [\![\mathsf{An}\ (f\ x)]\!]$$
$$\textbf{instance}\ (\mathsf{IFunctor}\ g, \mathsf{IFunctor}\ h) \Rightarrow \mathsf{IFunctor}\ (g \circ h)\ \textbf{where}$$
$$\mathsf{imap}\ f\ (\mathsf{Comp}\ ghx) = [\![\mathsf{Comp}\ (\mathsf{imap}\ (\mathsf{imap}\ f)\ ghx)]\!]$$

8 Idioms and Arrows

To handle situations where monads were inapplicable, Hughes (2000) defined an interface that he called *arrows*:

$$\textbf{class}\ \mathsf{Arrow}\ (\rightsquigarrow)\ \textbf{where}$$
$$\mathsf{arr} :: (a \to b) \to (a \rightsquigarrow b)$$
$$(\ggg) :: (a \rightsquigarrow b) \to (b \rightsquigarrow c) \to (a \rightsquigarrow c)$$
$$\mathsf{first} :: (a \rightsquigarrow b) \to ((a, c) \rightsquigarrow (b, c))$$

These structures include the ordinary function type, Kliesli arrows of monads and comonads, and much more. Equivalent structures called *Freyd-categories* had been independently developed as a device for structuring denotational semantics (Power & Robinson, 1997).

By fixing the first argument of an arrow type, we obtain an idiom, generalising the environment idiom we saw earlier:

$$
\begin{array}{l}
\textbf{newtype}\ \mathsf{FixArrow}\ (\leadsto)\ a\ b = \mathsf{Fix}\ (a \leadsto b) \\
\textbf{instance}\ \mathsf{Arrow}\ (\leadsto) \Rightarrow \mathsf{Idiom}\ (\mathsf{FixArrow}\ (\leadsto)\ a)\ \textbf{where} \\
\quad \iota\ x \qquad\qquad = \mathsf{Fix}\ (\mathsf{arr}\ (\mathsf{const}\ x)) \\
\quad \mathsf{Fix}\ u \circledast \mathsf{Fix}\ v \quad = \mathsf{Fix}\ (u \triangle v \ggg \mathsf{arr}\ (\lambda(f, x) \to f\ x)) \\
\qquad \textbf{where}\ u \triangle v \ = \mathsf{arr}\ \mathit{dup} \ggg \mathsf{first}\ u \ggg \mathsf{arr\ swap} \ggg \mathsf{first}\ v \ggg \mathsf{arr\ swap} \\
\qquad\qquad\quad \mathit{dup}\ x = (x, x)
\end{array}
$$

In the other direction, each idiom defines an arrow constructor that adds static information to an existing arrow:

$$
\begin{array}{l}
\textbf{newtype}\ \mathsf{StaticArrow}\ i\ (\leadsto)\ a\ b = \mathsf{Static}\ (i\ (a \leadsto b)) \\
\textbf{instance}\ (\mathsf{Idiom}\ i, \mathsf{Arrow}\ (\leadsto)) \Rightarrow \mathsf{Arrow}\ (\mathsf{StaticArrow}\ i\ (\leadsto))\ \textbf{where} \\
\quad \mathsf{arr}\ f \qquad\qquad\qquad = \mathsf{Static}\ [\![(\mathsf{arr}\ f)]\!] \\
\quad \mathsf{Static}\ f \ggg \mathsf{Static}\ g = \mathsf{Static}\ [\![(\ggg)\ f\ g]\!] \\
\quad \mathsf{first}\ (\mathsf{Static}\ f) \qquad = \mathsf{Static}\ [\![\mathsf{first}\ f]\!]
\end{array}
$$

To date, most applications of the extra generality provided by arrows over monads have been of two kinds: various forms of process, in which components may consume multiple inputs, and computing static properties of components. Indeed one of Hughes's motivations was the parsers of Swierstra and Duponcheel (1996). It may be that idioms will be a convenient replacement for arrows in the second class of applications.

9 Other definitions of Idioms

The Idiom class features the asymmetrical operation '⊛', but there are equivalent symmetrical definitions. For example we could assume the following two constants:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Liftable}\ i\ \textbf{where} \\
\quad \mathsf{unit} :: i\ () \\
\quad \mathsf{lift}_2 :: (a \to b \to c) \to i\ a \to i\ b \to i\ c
\end{array}
$$

We can define the combinators of Idiom in terms of those of Liftable:

$$
\begin{array}{l}
\iota \quad\ :: \mathsf{Liftable}\ i \Rightarrow x \to i\ x \\
\iota\ x\ \ = \mathsf{lift}_2\ (\mathsf{const}\ (\mathsf{const}\ x))\ \mathsf{unit\ unit} \\
(\circledast) :: \mathsf{Liftable}\ i \Rightarrow i\ (s \to t) \to i\ s \to i\ t
\end{array}
$$

$$
(\circledast) = \mathsf{lift}_2\ \mathsf{id}
$$

Conversely, we can define the Liftable interface in terms of Idiom:

$$\begin{array}{ll}
\mathsf{unit} & :: \mathsf{Idiom}\ i \Rightarrow i\ () \\
\mathsf{unit} & = \iota\ () \\
\mathsf{lift}_2 & :: \mathsf{Idiom}\ i \Rightarrow (a \to b \to c) \to i\ a \to i\ b \to i\ c \\
\mathsf{lift}_2\ f\ u\ v & = \iota\ f \circledast u \circledast v
\end{array}$$

The laws for this form are somewhat complicated, but we can use a more elementary form:

$$\begin{array}{l}
\mathbf{class}\ \mathsf{Functor}\ i \Rightarrow \mathsf{MFunctor}\ i\ \mathbf{where} \\
\quad \mathsf{unit} :: i\ () \\
\quad (\star)\ :: i\ a \to i\ b \to i\ (a, b)
\end{array}$$

The relationship between the Liftable and MFunctor classes is analogous to the relationship between zipWith and *zip*. The Liftable class may defined as

$$\begin{array}{ll}
\mathsf{lift}_2 & :: \mathsf{MFunctor}\ i \Rightarrow (a \to b \to c) \to i\ a \to i\ b \to i\ c \\
\mathsf{lift}_2\ f\ u\ v & = \mathsf{fmap}\ (\mathsf{uncurry}\ f)\ (u \star v)
\end{array}$$

and the Functor and MFunctor classes may be defined using Liftable:

$$\begin{array}{ll}
\mathsf{fmap} & :: \mathsf{Liftable}\ i \Rightarrow (a \to b) \to i\ a \to i\ b \\
\mathsf{fmap}\ f\ u & = \mathsf{lift}_2\ (\mathsf{const}\ f)\ \mathsf{unit}\ u \\
(\star) & :: \mathsf{Liftable}\ i \Rightarrow i\ a \to i\ b \to i\ (a, b) \\
u \star v & = \mathsf{lift}_2\ (,)\ u\ v
\end{array}$$

The laws of Idiom given in Section 2 are equivalent to the following laws of MFunctor:

functor identity	$\mathsf{fmap\ id}$	$=$	id
functor composition	$\mathsf{fmap}\ (f \cdot g)$	$=$	$\mathsf{fmap}\ f \cdot \mathsf{fmap}\ g$
naturality of $\star$	$\mathsf{fmap}\ (f \times g)\ (u \star v)$	$=$	$\mathsf{fmap}\ f\ u \star \mathsf{fmap}\ g\ v$
left identity	$\mathsf{fmap\ snd}\ (\mathsf{unit} \star v)$	$=$	v
right identity	$\mathsf{fmap\ fst}\ (u \star \mathsf{unit})$	$=$	u
associativity	$\mathsf{fmap\ assoc}\ (u \star (v \star w))$	$=$	$(u \star v) \star w$

for the functions

$$\begin{array}{l}
(\times) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d) \\
(f \times g)\ (x, y) = (f\ x, g\ y) \\
\mathsf{assoc} :: (a, (b, c)) \to ((a, b), c) \\
\mathsf{assoc}\ (a, (b, c)) = ((a, b), c)
\end{array}$$

Fans of category theory will recognise the above laws as the properties of a *lax monoidal functor* for the monoidal structure given by products. However the functor composition and naturality equations are actually stronger than their categorical counterparts. This is because we are working in a higher-order language, in which function expressions may include variables from the environment, as in the following definition:

$$\begin{array}{ll}
\iota & :: \mathsf{MFunctor}\ i \Rightarrow x \to i\ x \\
\iota\ x & = \mathsf{fmap}\ (\mathsf{const}\ x)\ \mathsf{unit}
\end{array}$$

Exercise 11 (MFunctor *and* swap)

Use the MFunctor laws and the above definition of ι to prove the equation

$$\mathsf{fmap\ swap}\ (\iota\ x \star u) = u \star \iota\ x$$

for the function

$$\begin{array}{ll} \mathsf{swap} & :: (a, b) \rightarrow (b, a) \\ \mathsf{swap}\ (a, b) & = (b, a) \end{array}$$

The proof makes essential use of higher-order functions.

9.1 Strong lax monoidal functors

In the first-order language of category theory, such data flow must be explicitly plumbed using *strong* functors, i.e. functors F equipped with a *tensorial strength*

$$t_{AB} : A \times F\,B \longrightarrow F\,(A \times B)$$

that makes the following diagrams commute.

$$\begin{array}{ccc} 1 \times FA & \cong & FA \\ {\scriptstyle t}\downarrow & & \| \\ F(1 \times A) & \cong & FA \end{array} \qquad \begin{array}{ccc} (A \times B) \times FC & \cong & A \times (B \times FC) \\ & & \downarrow{\scriptstyle A \times t} \\ {\scriptstyle t}\downarrow & & A \times F(B \times C) \\ & & \downarrow{\scriptstyle t} \\ F((A \times B) \times C) & \cong & F(A \times (B \times C)) \end{array}$$

The naturality axiom above then becomes *strong naturality*: the natural transformation m corresponding to '$\star$' must also respect the strength:

$$\begin{array}{ccc} (A \times B) \times (FC \times FD) & \cong & (A \times FC) \times (B \times FD) \\ {\scriptstyle (A \times B) \times m}\downarrow & & \downarrow{\scriptstyle t \times t} \\ (A \times B) \times F(C \times D) & & F(A \times C) \times F(B \times D) \\ {\scriptstyle t}\downarrow & & \downarrow{\scriptstyle m} \\ F((A \times B) \times (C \times D)) & \cong & F((A \times C) \times (B \times D)) \end{array}$$

Note that B and FC swap places in the above diagram: strong naturality implies commutativity with pure computations.

Thus in categorical terms idioms are strong lax monoidal functors. Every strong monad determines two such functors, as the definition is symmetrical.

10 Conclusions and further work

References

Baars, A.I., Löh, A., & Swierstra, S.D. (2004). Parsing permutation phrases. *Journal of functional programming*, **14**(6), 635–646.

Barr, Michael, & Wells, Charles. (1984). *Toposes, triples and theories*. Grundlehren der Mathematischen Wissenschaften, no. 278. New York: Springer. Chap. 9.

Bird, Richard, & Paterson, Ross. (1999). de Bruijn notation as a nested datatype. *Journal of functional programming*, **9**(1), 77–92.

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

Hoogendijk, Paul, & Backhouse, Roland. (1997). When do datatypes commute? *Pages 242–260 of:* Moggi, E., & Rosolini, G. (eds), *Category theory and computer science*. LNCS, vol. 1290. Springer.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(May), 67–111.

Hutton, Graham, & Meijer, Erik. (1998). Monadic parsing in Haskell. *Journal of functional programming*, **8**(4), 437–444.

Leijen, Daan, & Meijer, Erik. 1999 (Oct.). Domain specific embedded compilers. *2nd conference on domain-specific languages (DSL)*. USENIX, Austin TX, USA. Available from `http://www.cs.uu.nl/people/daan/papers/dsec.ps`.

McBride, Conor. (2002). Faking it (simulating dependent types in Haskell). *Journal of functional programming*, **12**(4& 5), 375–392. Special Issue on Haskell.

McBride, Conor. (2003). First-Order Unification by Structural Recursion. *Journal of functional programming*, **13**(6).

Meertens, Lambert. 1998 (June). Functor pulling. *Workshop on generic programming (WGP'98)*.

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

Power, John, & Robinson, Edmund. (1997). Premonoidal categories and notions of computation. *Mathematical structures in computer science*, **7**(5), 453–468.

Röjemo, Niklas. (1995). *Garbage collection and memory efficiency*. Ph.D. thesis, Chalmers.

Swierstra, S. Doaitse, & Duponcheel, Luc. (1996). Deterministic, error-correcting combinator parsers. *Pages 184–207 of:* Launchbury, John, Meijer, Erik, & Sheard, Tim (eds), *Advanced functional programming*. LNCS, vol. 1129. Springer.

Wadler, Philip. (1985). How to replace failure by a list of successes. *Pages 113–128 of:* Jouannaud, Jean-Pierre (ed), *Functional programming languages and computer architecture*. LNCS, vol. 201. Springer.

Solutions to Exercises

Solution 1 (Idiom*s functorial*)

We check that lift_1 respects id and $(\cdot)$ as follows:

- $$\begin{aligned} \mathsf{lift}_1\ \mathsf{id}\ u &= \iota\ \mathsf{id} \circledast u && \{\mathsf{lift}_1\} \\ &= u && \{\textbf{identity}\} \end{aligned}$$
- $$\begin{aligned} \mathsf{lift}_1\ (f \cdot g)\ u &= \iota\ (f \cdot g) \circledast u && \{\mathsf{lift}_1\} \\ &= \iota\ (\cdot) \circledast \iota\ f \circledast \iota\ g \circledast u && \{\textbf{homomorphism}\} \\ &= \iota\ f \circledast (\iota\ g \circledast u) && \{\textbf{composition}\} \\ &= \mathsf{lift}_1\ f\ (\mathsf{lift}_1\ g\ u) && \{\mathsf{lift}_1\} \end{aligned}$$

Solution 2 (*canonical form*)

We proceed in four phases—one for each law:

identity Ensure the expression has form $\iota\ f \circledast s$ by wrapping u as $\iota\ \mathsf{id} \circledast u$ if necessary.

composition Flatten the expression, replacing each right-nested $u \circledast (v \circledast w)$ with its left-nested counterpart $\iota\ (\cdot) \circledast u \circledast v \circledast w$. Note that this preserves the order of effects, merely inserting extra pure computations. The expression now has form

$$\iota\ f \circledast is_1 \circledast \ldots \circledast is_n$$

where some of the is's are of form $\iota\ s$ for pure s.

interchange Rewrite $u \circledast \iota\ s$ to $\iota\ (\lambda f \to f\ s) \circledast u$ wherever u is not some $\iota\ f$ itself. Now we have form

$$\iota\ f \circledast \iota\ p_1 \circledast \ldots \circledast \iota\ p_k \circledast e_1 \circledast \ldots \circledast e_m$$

where the e's are the effectful is's, in order.

homomorphism Collapse the initial segment of the expression, leaving

$$\iota\ (f\ p_1\ \ldots\ p_k) \circledast e_1 \circledast \ldots \circledast e_m$$

as required.

Solution 3 (*Coding* $[\![\ldots]\!]$)

Following a similar approach to the definition of the zipWith family in (McBride, 2002), we use a type class trick:

```
class Idiom i ⇒ Idiomatic i f g | g → f i where
  idiomatic :: i f → g
```

In general, f is a pure function type (possibly of arity 0) and g is somehow the corresponding version of f in the idiom i. As we process arguments from left to right, we accumulate a function in the idiom of type $i\ f$, from which to compute the remaining function g. Our 'open bracket' just initialises the accumulator with the pure function given.

```
iI :: Idiomatic i f g ⇒ f → g
iI f = idiomatic (ι f)
```

If must consume an argument in $i\ s$, the accumulator must be $\circledast$-able to it.

```
instance Idiomatic i f g ⇒ Idiomatic i (s → f) (i s → g) where
  idiomatic isf is = idiomatic (isf ⊛ is)
```

Meanwhile, our 'close bracket' is just the constructor of a datatype, introduced especially for this purpose:

```
data Ii = Ii
```

When we see Ii, we just unload the accumulator!

```
instance Idiom i ⇒ Idiomatic i x (Ii → i x) where
  idiomatic ix Ii = ix
```

This notation is quite extensible. Fans of parser combinators may like to add a symbol Ig meaning 'execute the following computation, but ignore its value'.

```
data Ig = Ig
```

```
instance Idiomatic i h g ⇒ Idiomatic i h (Ig→ i x → g) where
  idiomatic ih Ig ix = idiomatic (ι const ⊛ ih ⊛ ix)
```

A simple example of this in action might be:

```
exp :: Parser (Exp String)
exp = ⟦Val int⟧
    ⊕ ⟦Var ident⟧
    ⊕ ⟦Add Ig(tok "(") exp Ig(tok "+") exp Ig(tok ")")⟧
```

where int, ident and tok t are the parsers for integers, identifiers and the token t, respectively.

Solution 4 (*the colist* Monad)
The join of this list monad takes the diagonal of a 'matrix', however far it extends.

```
join :: [[x]] → [x]
join []              = []
join ( []
     : _         ) = []
join ( (x : _)
     : xss       ) = x
                     : join (fmap chop xss) where
  chop []       = []
  chop (_ : xs) = xs
```

Our alternative Monad thus has return = repeat and, as standard,

```
xs ≫= f = join (fmap f xs)
```

Correspondingly, the ap of this monad generates the matrix of pairwise applications from a list of functions and a list of arguments, solely for the purpose of taking its diagonal. Directly implementing zapp is clearly wiser.

Solution 5 (*distributing* Term *and* Maybe)
The specialised instance

```
idist = imap id :: Term (Maybe x) → Maybe (Term x)
```

is a kind of 'occur check' for the most local de Bruijn variable, Var Nothing, available for the input term. Either this term does not use its most local variable, in which case we get Just t for t a term with variables drawn only from x, or we get Nothing. That is, idist propagates an *effectful* renaming through its input; that renaming, id :: Maybe x → Maybe x is seen as the renaming from Maybe x to x which *fails* if its input is ever Nothing. The IFunctor behaviour of Maybe ensures that the effectful renaming is reindexed correctly under a Lda.

The rationalisation of the occur check as a failure-prone renaming is central to the first author's structurally recursive first-order unification algorithm (McBride, 2003). In this form, the occur check delivers the witness that a variable has been successfully eliminated.

Solution 6 (Functor *versus* IFunctor)

The idea behind imap is to traverse a data structure threading the effects produced by some operation on elements and combining the results into a single effectful computation. If the effect being threaded is inherently *strict*, for example the 'failure' effect coded by the Maybe idiom, then the data traversed must be *finite*. That is, if f fails for *any* element, then imap f should fail, hence imap f must visit every element before delivering a result. Correspondingly, we need only choose a Functor which acts as a container for infinitely many elements to make imap unavailable: $(\rightarrow)$ Integer will do very nicely.

Let

$$\mathsf{stateAfter} :: \mathsf{TuringMachine} \rightarrow \mathsf{Integer} \rightarrow \mathsf{Maybe\ State}$$

be such that stateAfter tm n returns Just s if Turing Machine tm is still running and in state s after n steps of execution, and Nothing if tm halts within n steps. Then

$$\mathsf{imap}\ (\mathsf{stateAfter}\ tm)\ \mathsf{id}$$

must return Nothing if tm halts for any number of steps in the range of id :: Integer $\rightarrow$ Integer.

Solution 7 (every Ord *induces a* Monoid*)*

Every instance of Ord induces a Monoid via its max operation, provided we have a bottom element to act as the $\emptyset$—we can always add such a thing:

```
data Pointed x = Bottom | Embedded x

instance Ord x ⇒ Ord (Pointed x) where
  compare  Bottom        Bottom       = EQ
  compare  Bottom        (Embedded y) = LT
  compare  (Embedded x)  Bottom       = GT
  compare  (Embedded x)  (Embedded y) = compare x y

instance Ord x ⇒ Monoid (Pointed x) where
  ∅   = Bottom
  (⊕) = max
```

Of course, we can also get the 'minimum' monoid by adding a top element.

Solution 8 (adverbial programming)

The adverbial style includes the constructor of a **newtype** but omits the projection. Accordingly, let us overload the latter

```
class Unpack p u | p → u where
  unpack :: p → u
```

For each of our **newtype**s, let us take unpack to be the projection:

```
instance Unpack (Accy o x) o where
  unpack = acc
```

and so on.

Now we may define

$\mathsf{ily} :: \mathsf{Unpack}\ p'\ u' \Rightarrow (u \to p) \to ((t \to p) \to t' \to p') \to$
$\quad (t \to u) \to t' \to u'$
$\mathsf{ily}\ \mathit{pack}\ \mathit{transform}\ f = \mathsf{unpack} \cdot \mathit{transform}\ (\mathit{pack} \cdot f)$

Solution 9 (*monads and accumulation*)
It is well established that the effect of 'writing' to a monoid is monadic, with return writing $\emptyset$ and join combining the effects with $\oplus$. For the standard Monad interface, this yields

$$\begin{aligned}
&\textbf{instance}\ \mathsf{Monoid}\ o \Rightarrow \mathsf{Monad}\ ((,)\ o)\ \textbf{where} \\
&\quad \mathsf{return}\ x = (\emptyset, x) \\
&\quad (o_1, s) \gg\!= f = (o_1 \oplus o_2, t)\ \textbf{where} \\
&\qquad (o_2, t) = f\ s
\end{aligned}$$

It is less well established—but nonetheless trivial—that the constant singleton

$$\textbf{data}\ \mathsf{Unit}\ x = \mathsf{Void}$$

is monadic.

We may take Accy o to be the type constructor which throws away its argument but delivers an o, as follows:

$$\textbf{type}\ \mathsf{Accy}\ o = ((,)\ o) \circ \mathsf{Unit}$$

The idiom resulting from this composition of monads behaves just like the one we defined directly.

Solution 10 (*fast reverse*)
Let us have

$$\textbf{newtype}\ \mathsf{EndoOp}\ x = \mathsf{OpEndo}\{\mathsf{opEndo} :: x \to x\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathsf{Monoida}\ \mathsf{EndoOp}\ \textbf{where} \\
&\quad \emptyset = \mathsf{OpEndo}\ \mathsf{id} \\
&\quad \mathsf{OpEndo}\ f \oplus \mathsf{OpEndo}\ g = g \cdot f
\end{aligned}$$

Now recall that (:) takes an element to an endofunction on lists!

$$\begin{aligned}
&\mathsf{rev} :: [x] \to [x] \\
&\mathsf{rev}\ xs = (\mathsf{OpEndo}\text{`}\mathsf{ily}\text{`})\ \mathsf{icrush}\ (:)\ xs\ [\,]
\end{aligned}$$

Solution 11 (MFunctor *and* swap)
We have that

$$\iota\ x = \mathsf{fmap}\ (\mathsf{const}\ x)\ \mathsf{unit}$$

Hence

$$\begin{aligned}
\mathsf{fmap}\ \mathsf{swap}\ (\iota\ x \star u) &= \mathsf{fmap}\ \mathsf{swap}\ (\mathsf{fmap}\ (\mathsf{const}\ x)\ \mathsf{unit} \star u) && \{iii\} \\
&= \mathsf{fmap}\ \mathsf{swap}\ (\mathsf{fmap}\ (\mathsf{const}\ x)\ \mathsf{unit} \star \mathsf{fmap}\ \mathsf{id}\ u) && \{\textbf{functor identity}\} \\
&= \mathsf{fmap}\ \mathsf{swap}\ (\mathsf{fmap}\ (\mathsf{const}\ x \times \mathsf{id})\ (\mathsf{unit} \star u)) && \{\textbf{naturality of}\ \star\} \\
&= \mathsf{fmap}\ (\mathsf{swap} \cdot (\mathsf{const}\ x \times \mathsf{id}))\ (\mathsf{unit} \star u) && \{\textbf{functor composition}\}
\end{aligned}$$