FUNCTIONAL PEARLS
[ABORTED] A trail told by an idiom

Conor McBride

1 Introduction

Nobody likes their programs to be full of sound and fury, signifying nothing. Abstraction is the weapon of choice in the war on wanton waffle. This paper is about an abstraction which I find rather handy. It's a weaker variation on the theme of a monad, but it has a more *functional* feel. I call it an *idiom*:

```
infixl 9 ⟨%⟩
class Idiom i where
  idi   :: x → i x
  (⟨%⟩) :: i (s → t) → i s → i t      — pronounced 'apply'
```

The **idi** operation shifts values into an idiom, just like the **return** of a monad. The ⟨%⟩ operation lets you *program* in the idiom, applying *i*-functions to *i*-arguments, yielding *i*-results. Of course, every monad gives us an idiom—the **idi** operation is just **return**, and the ⟨%⟩ operation is also found in the library as **ap**.

```
instance Monad m ⇒ Idiom m where
  idi = return
  mf ⟨%⟩ ms = do
    f ← mf
    s ← ms
    return (f s)
```

However, the reverse is plainly not the case. The ⟨%⟩ operation keeps idiomatic programming on the *i*-level. There is no way to define a general operation **join** :: $i\ (i\ x) \rightarrow x$ which collapses structure—that's what makes monads a strictly stronger notion. There are three key motivations for working with idioms

1. There are useful idioms which are not monads. As we shall see, the composition of two idioms is an idiom—not necessarily so for monads. Moreover, every monoid induces a non-monadic idiom.
2. Idioms—*by construction*—naturally support programming in a more functional style, where the 'do' notation is more imperative in flavour.
3. There are many operations which are commonly defined for monads, but which require only idiomatic structure—we can widen their applicability by using idioms instead, without losing their behaviour for monads.

Of course, I'm not saying that we should abandon monads: far from it! There are many situations, from IO to syntax-with-substitution, where the full strength of monads is absolutely necessary. However, we've just seen that every monad is an idiom, so the technology I exhibit in this paper provides a convenient functional interface for programming with monads where only the idiomatic power is in use. It's also very easy to mix the two styles.

2 Idioms in action and vice versa

Let's write an old monadic favourite in the idiomatic style. Let Parser be a monad for string parsing, equipped with a prioritized choice operator $\langle + \rangle$ and an atomic character-checker, **eat** :: Char $\rightarrow$ Parser ().

We might define a data structure, say,

$$\text{data } \mathsf{Tree} \;=\; \mathsf{Leaf} \mid \mathsf{Node}\ \mathsf{Tree}\ \mathsf{Tree}$$

and equip it with a simple parser, with `*` for leaves and `(`t_1`V`t_2`)` for nodes.

$$\begin{array}{l}
\mathbf{pTree} :: \mathsf{Parser\ Tree} \\
\mathbf{pTree} \;=\; \text{do } \begin{array}[t]{l} \mathbf{eat}\ \text{`}\mathtt{*}\text{'} \\ \mathbf{return}\ \mathsf{Leaf} \end{array} \\
\qquad\qquad \langle + \rangle \\
\qquad\qquad \text{do } \begin{array}[t]{l} \mathbf{eat}\ \text{`}\mathtt{(}\text{'} \\ t_1 \leftarrow \mathbf{pTree} \\ \mathbf{eat}\ \text{`}\mathtt{V}\text{'} \\ t_2 \leftarrow \mathbf{pTree} \\ \mathbf{eat}\ \text{`}\mathtt{)}\text{'} \\ \mathbf{return}\ (\mathsf{Node}\ t_1\ t_2) \end{array}
\end{array}$$

If you'll allow me the idiomatic generalization of the monadic $\gg$,

$$\begin{array}{l}
\text{infixl } 9\ \langle / \rangle \\
\\
(\langle / \rangle) :: i\ s \rightarrow i\ t \rightarrow i\ s \\
si\ \langle / \rangle\ ti = \mathbf{idi\ const}\ \langle \% \rangle\ si\ \langle \% \rangle\ ti
\end{array}$$

I'll write the parser the idiomatic way:

$$\begin{array}{l}
\mathbf{pTree} :: \mathsf{Parser\ Tree} \\
\mathbf{pTree} = \mathbf{idi}\ \mathsf{Leaf}\ \langle / \rangle\ \mathbf{eat}\ \text{`}\mathtt{*}\text{'} \\
\qquad \langle + \rangle\ \mathbf{idi}\ \mathsf{Node}\ \langle / \rangle\ \mathbf{eat}\ \text{`}\mathtt{(}\text{'}\ \langle \% \rangle\ \mathbf{pTree}\ \langle / \rangle\ \mathbf{eat}\ \text{`}\mathtt{V}\text{'}\ \langle \% \rangle\ \mathbf{pTree}\ \langle / \rangle\ \mathbf{eat}\ \text{`}\mathtt{)}\text{'}
\end{array}$$

We've got a much neater presentation of the parser, much closer to the form of the grammar we might write down. Of course, this will come as no surprise to those of you who are familiar with parser combinators. The idiomatic operators $\langle \% \rangle$ and $\langle / \rangle$ show up time and again in that setting—it's the applicative behaviour of the monad which parsers tend to exploit. What makes monads special is that they also support the structure-collapsing 'join' behaviour $m\ (m\ x) \rightarrow m\ x$, but we don't so often need to compute a parser from a parser parser.

The nice thing is that we haven't lost the functional structure of the parser, just because we've lost its purity. We've just switched to an impure functional idiom in which some computations have effects—consuming characters from the input. We don't need to introduce the extra plumbing—binding t_1 and t_2—to reconsitute the parts of the computation which *are* functional.

3 Idiomatic mapping

The standard Haskell Prelude defines a monadically lifted **mapM** operator for lists

$$\mathbf{mapM} :: \mathbf{Monad}\ m \Rightarrow (s \rightarrow m\ t) \rightarrow [s] \rightarrow m\ [t]$$

which maps an effectful operation across a list, returning an effectful list of results. For the Maybe monad, you get the unreliable mapping of an unreliable function across a list—if f fails for any element, then **mapM** f fails for the whole list. This operation, too, exploits only the idiomatic behaviour of m. We can redefine it thus:

$$\begin{aligned}
&\mathbf{imap} :: \mathbf{Idiom}\ i \Rightarrow (s \rightarrow i\ t) \rightarrow [s] \rightarrow i\ [t] \\
&\mathbf{imap}\ f\ []\qquad\quad = \mathbf{idi}\ [] \\
&\mathbf{imap}\ f\ (x : xs) = \mathbf{idi}\ (:)\ \langle\%\rangle\ f\ x\ \langle\%\rangle\ \mathbf{imap}\ f\ xs
\end{aligned}$$

As we shall shortly see, this is such a useful operation that it's worth making a class for type constructors which support it:

$$\begin{aligned}
&\text{class}\ \mathbf{IFunctor}\ f\ \text{where} \\
&\quad \mathbf{imap} :: \mathbf{Idiom}\ i \Rightarrow (s \rightarrow i\ t) \rightarrow f\ s \rightarrow i\ (f\ t)
\end{aligned}$$

Note that

$$\mathbf{imap\ idi} :: f\ (i\ x) \rightarrow i\ (f\ x)$$

generalizing the standard Prelude's **sequence** operator.

You can define instances of **IFunctor** for first-order polymorphic types much the way you do for **Functor**, except that you need to lift the right-hand sides with **idi** and $\langle\%\rangle$, just as we did with lists. This construction does not extend to *higher-order* types: $(s \rightarrow)$ is a **Functor** and indeed an **Idiom**,[1] but not an **IFunctor**.

We can exploit the idiomatic behaviour to generalize well-known operators, such as the predicate transformer **all** which jacks a predicate on elements conjunctively up to a predicate on lists. Here's a way we might do it:

$$\begin{aligned}
&\mathbf{all} :: \mathbf{IFunctor}\ f \Rightarrow (x \rightarrow \mathsf{Bool}) \rightarrow f\ x \rightarrow \mathit{Bool} \\
&\mathbf{all}\ p = \mathbf{isJust} \cdot \mathbf{imap}\ (\lambda x \rightarrow \mathbf{idi}\ x\ \langle/\rangle\ \mathbf{guard}\ (p\ x))
\end{aligned}$$

The idea is that we **imap** in the Maybe world, using a function which is Just exactly when p holds—we can only get a Just answer if p holds everywhere. This example may help explain why we should be glad that $(s \rightarrow)$ isn't an **IFunctor**. If it were, then a machine could check if all possible outputs from a function (eg, the state of a Turing machine after a given number of steps) satisfied a given predicate (eg, halting), and lots of us would be unemployed.

[1] Fans of combinatory logic will recognize this idiom as an old friend.

But there is something annoying about this definition of **all**—it involves copying its input, when we should just be accumulating the results of the p's with &&, initialized by True. Fortunately, accumulation is an idiom!

4 Monoids and idiomatic accumulation

The GHC library module **Data**:**Monoid** introduces a class **Monoid** x with methods including

$$\begin{aligned}&\mathbf{mempty} :: x\\&\mathbf{mappend} :: x \to x \to x\end{aligned}$$

which This is intended to describe types supporting an associative binary operation which absorbs a neutral element. These structures are ubiquitous—lists with [] and $+\!\!+$, endofunctions with **id** and $\cdot$, or IO () with **return** () and $\gg$[2]. I'm too fond of monoids to use a prefix binary operator with a long name, so please indulge me and pretend that we actually have

$$\begin{aligned}&\mathbf{zero} :: x\\&(\langle + \rangle) :: x \to x \to x\\&\text{infixr } 6\ \langle + \rangle\end{aligned}$$

I define the type of *accumulations* as follows:

$$\text{newtype } \mathbf{Monoid}\ x \Rightarrow \mathsf{Acc}\ x\ t = \mathsf{Acc}\ \{\mathbf{accumulated} :: \mathbf{x}\}$$

This is a *phantom* type: the t plays no part in describing the data the type contains. Rather, it is used to describe the *source* type from which the x has been accumulated. Perhaps you've guessed that

$$\begin{aligned}&\text{instance } \mathbf{Monoid}\ x \Rightarrow \mathbf{Idiom}\ (\mathsf{Acc}\ x) \text{ where}\\&\quad \mathbf{idi}\ \_ = \mathsf{Acc}\ \mathbf{zero}\\&\quad \mathsf{Acc}\ \mathit{fx}\ \langle\%\rangle\ \mathsf{Acc}\ \mathit{sx} = \mathsf{Acc}\ (\mathit{fx}\ \langle + \rangle\ \mathit{sx})\end{aligned}$$

The $\langle\%\rangle$ for this idiom just combines the value accumulated from the function with that accumulated from the argument. Meanwhile, the default contribution to the accumulation is nothing. We can now write a *crushing* function from any **IFunctor** to any **Monoid**:

$$\begin{aligned}&\mathbf{icrush} :: (\mathbf{IFunctor}\ f, \mathbf{Monoid}\ y) \Rightarrow (x \to y) \to f\ x \to y\\&\mathbf{icrush}\ g = \mathbf{accumulated} \cdot \mathbf{imap}\ (\mathbf{Acc} \cdot g)\end{aligned}$$

Let's revisit our **all** example. We'll need the monoid of *conjunctive* Booleans[3]:

[2] also known as 'skip' and 'sequential composition, respectively

[3] I make Might the monoid of *disjunctive* Booleans, dually defined

$$\text{newtype } \mathsf{Must} = \mathsf{Must}\ \{\mathbf{must} :: \mathsf{Bool}\}$$

$$\begin{array}{l} \text{instance } \mathbf{Monoid}\ \mathsf{Must} \text{ where} \\ \quad \mathbf{zero} = \mathsf{Must\ True} \\ \quad \mathsf{Must}\ x\ \langle + \rangle\ \mathsf{Must}\ y = \mathsf{Must}\ (x\ \&\&\ y) \end{array}$$

Now we can define **all** like so:

$$\begin{array}{l} \mathbf{all} :: \mathbf{IFunctor}\ f \Rightarrow (x \rightarrow \mathsf{Bool}) \rightarrow f\ x \rightarrow Bool \\ \mathbf{all}\ p = \mathbf{must} \cdot \mathbf{icrush}\ (\mathsf{Must} \cdot p) \end{array}$$

We can also use **icrush** to compute the trail of elements from an **IFunctor** data structure in depth-first traversal order, eg, flattening a binary search tree into a list. In fact, any idiom which is monoidal for the element type can be used to accumulate the 'trail'. We **idi** the elements into the idiom (for lists, **idi** $x = [x]$) then we $\langle + \rangle$ them together:

$$\begin{array}{l} \mathbf{trail} :: (\mathbf{Idiom}\ i, \mathbf{Monoid}\ (i\ x), \mathbf{IFunctor}\ f) \Rightarrow f\ x \rightarrow i\ x \\ \mathbf{trail} = \mathbf{icrush}\ \mathbf{idi} \end{array}$$

So **trail** flattens if you want a list. Also, its Maybe-returning instance computes the leftmost element if there is one. Moreover, iff the elements themselves inhabit a monoid, then **trail** for the identity idiom (which is also the identity monoid-transformer) just computes their 'sum'. Such a lot of work we can extract for the price of lifting 'map' to an arbitrary idiom!

Now, there's an obvious program comprehension problem if you take this approach too far. When you see **trail** in the middle of a pile of code, how on earth can you tell what it's doing? You may also have noticed that the programs are becoming shorter than their types. That's because the *type is most of the program*, or rather, it's the type which drives Haskell to write your program for you! It's best to use these operators only where the actual usage type is obvious. I don't like doing type inference in my head, so I prefer to give idiomatic operators a bunch of less polymorphic aliases with sensible names and explicit type signatures.

5 A tidier notation for idioms

I propose a notation for idiomatic programming which is just noisy enough to let you know that it's happening, but quiet enough to preserve the basic functional appearance of your program. I write

$$\langle\!| f\ t_1\ \ldots\ t_n |\!\rangle$$

for

$$\mathbf{idi}\ f\ \langle\%\rangle\ t_1\ \langle\%\rangle\ \ldots\ \langle\%\rangle\ t_n$$

In this notation, the **IFunctor** [] instance becomes just

```
instance IFunctor [] where
  imap f []       = ⦇ [] ⦈
  imap f (x : xs) = ⦇ (:) (f x) (imap f xs) ⦈
```

That is, modulo making infix into prefix, just the idiomatic bracketing of the ordinary map operator. Moreover, I can add 'ignored' arguments inside the bracket—those where we use ⟨/⟩ rather than ⟨%⟩—by prefixing them with †. Our little parser becomes

```
pTree :: Parser Tree
pTree = ⦇ Leaf  † (eat '*') ⦈
      ⟨+⟩ ⦇ Node  † (eat '(') pTree  † (eat 'V') pTree  † (eat ')') ⦈
```

As it happens, you can teach a Haskell compiler to understand this notation, suitably rendered into ASCII. It's done by just the kind of type class Prolog you might expect from a crook like me. I am suitably ashamed of myself.

```
data ⦈ = ⦈
data † = †

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

instance Idiom i ⇒ Idiomatic i x (⦈ → i x) where
  ⦇ x ⦈ = idi x
  idiomatic xi ⦈ = xi

instance Idiomatic i f g ⇒ Idiomatic i (s → f) (i s → g) where
  ⦇ sf = idiomatic (idi sf)
  idiomatic sfi si = idiomatic (sfi ⟨%⟩ si)

instance Idiomatic i f g ⇒ Idiomatic i f († → i x → g) where
  ⦇ f = idiomatic (idi f)
  idiomatic fi † xi = idiomatic (fi ⟨/⟩ xi)
```

Fig. 1. Implementing the ⦇ † ⦈ notation

I typically type ⦇ as `idI`, ⦈ as `Idi`, and † as `Ig`, in compliance with Haskell's conventions for function and constructor names.

6 Idiom is short for...

weakly symmetric lax monoidal functor!