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. Abstract 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:

\[
\text{infixl 9 } \langle \rangle \\
\text{class } \text{Idiom} \ i \ \text{where}
\]

\[
\text{idi} \quad :: \quad x \rightarrow i \ x
\]

\[
\langle \langle \rangle \rangle \quad :: \quad i \ (s \rightarrow t) \rightarrow i \ s \rightarrow i \ t \quad \text{— pronounced ‘apply’}
\]

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

\[
\text{instance } \text{Monad } m \Rightarrow \text{Idiom} \ m \ \text{where}
\]

\[
\text{idi} = \text{return}
\]

\[
m \langle \langle \rangle \rangle \ n s = \text{do}
\]

\[
f \leftarrow mf
\]

\[
s \leftarrow ns
\]

\[
\text{return } (f \ s)
\]

However, the reverse is plainly not the case. The \langle \rangle operation keeps idiomatic programming on the \text{i-level}. There is no way to define a general operation \text{join} :: \text{i (i x)} \rightarrow \text{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 (+) and an atomic character-checker, eat :: Char → Parser ()

We might define a data structure, say,

\[
data\ Tree = \text{Leaf} | \text{Node} \; \text{Tree}
\]

and equip it with a simple parser, with * for leaves and (t₁t₂) for nodes.

\[
p\Tree :: \text{Parser} \; \text{Tree} \\
p\Tree = \text{do} \; \text{eat} \; '*' \\
\quad \text{return} \; \text{Leaf} \\
\quad (+) \quad \text{do} \; \text{eat} \; '(!' \\
\quad \quad \quad \quad t₁ ← p\Tree \\
\quad \quad \quad \quad \text{eat} \; 'V' \\
\quad \quad \quad \quad t₂ ← p\Tree \\
\quad \quad \quad \quad \text{eat} \; ')’ \\
\quad \text{return} \; (\text{Node} \; t₁ \; t₂)
\]

If you’ll allow me the idiomatic generalization of the monadic ≫,

\[
\text{infixl 9 } ⟨/⟩ \\

⟨⟨⟩⟩ :: i \; s → i \; t → i \; s \\
\text{si } ⟨/⟩ \; ti = \text{id} \; \text{const} \; ⟨%⟩ \; \text{si } ⟨%⟩ \; ti
\]

I’ll write the parser the idiomatic way:

\[
p\Tree :: \text{Parser} \; \text{Tree} \\
p\Tree = \text{id} \; \text{Leaf } ⟨/⟩ \; \text{eat} \; '*' \\
\quad (+) \; \text{id} \; \text{Node } ⟨/⟩ \; \text{eat} \; '(!' \; ⟨%⟩ \; p\Tree \; ⟨/⟩ \; \text{eat} \; 'V' \; ⟨%⟩ \; p\Tree \; ⟨/⟩ \; \text{eat} \; ')’
\]

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 ⟨%⟩ and ⟨/⟩ 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) → 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 t₁ and t₂—to reconstitute the
parts of the computation which are functional.

3 Idiomatic mapping

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

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

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:

\[ \text{imap} :: \text{Idiom} \ i \Rightarrow (s \rightarrow i \ t) \rightarrow [s] \rightarrow i \ [t] \]
\[ \text{imap} \ f \ [\ ] = \text{id} \ [\ ] \]
\[ \text{imap} \ f \ (x : xs) = \text{id} \ (:) \ (\%) \ f \ x \ (\%) \ \text{imap} \ f \ xs \]

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

```haskell
class IFunctor f where
  imap :: Idiom i ⇒ (s → i t) → f s → i (f t)
```

Note that

\[ \text{imap} \ \text{id} :: f \ (i \ x) \rightarrow i \ (f \ x) \]

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 id
and (\%), just as we did with lists. This construction does not extend to higher-order
types: (s →) 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:

```haskell
all :: IFunctor f ⇒ (x → Bool) → f x → Bool
all p = isJust ∘ imap (λx → idi x (\/) guard (p x))
```

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 →) 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

\[
\text{mempty} :: x \\
\text{mappend} :: x \rightarrow x \rightarrow x
\]

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 $\gg$. 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

\[
\text{zero} :: x \\
(\text{+t}) :: x \rightarrow x \rightarrow x \\
\text{infixr 6 (t)}
\]

I define the type of accumulations as follows:

\[
\text{newtype Monoid x ⇒ Acc x t = Acc \{ accumulated :: x\}}
\]

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

\[
\text{instance Monoid x ⇒ Idiom (Acc x) where} \\
\text{id} \_ = \text{Acc zero} \\
\text{Acc fx \langle%\rangle Acc sx = Acc (fx \langle\rangle sx)}
\]

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:

\[
\text{icrush :: (IFunctor f, Monoid y) ⇒ (x → y) → f x → y} \\
\text{icrush g = accumulated \cdot imap (Acc \cdot g)}
\]

Let’s revisit our all example. We’ll need the monoid of conjunctive Booleans:\n
\[
\text{2 also known as ‘skip’ and ‘sequential composition, respectively} \\
\text{3 I make Might the monoid of disjunctive Booleans, dually defined}
\]
newtype Must = Must {must :: Bool}

instance Monoid Must where
  zero = Must True
  Must x (⁺) Must y = Must (x && y)

Now we can define all like so:

\[
\text{all} :: \text{IFunctor} \ f \Rightarrow (x \rightarrow \text{Bool}) \rightarrow f \ x \rightarrow \text{Bool}
\]
\[
\text{all} \ p = \text{must} \cdot \text{icrush} (\text{Must} \cdot \ p)
\]

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 (⁺) them together:

\[
\text{trail} :: (\text{Idiom} \ i, \text{Monoid} (i \ x), \text{IFunctor} \ f) \Rightarrow f \ x \rightarrow i \ x
\]
\[
\text{trail} = \text{icrush} \ \text{idi}
\]

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

\[
\| \ f \ t_1 \ldots t_n \|
\]

for

\[
\text{idi} \ f \ 〈%〉 \ t_1 \ 〈%〉 \ldots \ 〈%〉 \ t_n
\]

In this notation, the IFunctor [] instance becomes just
instance \textbf{IFunctor} [] where

\[
\begin{align*}
\text{imap} \ f \ [] & = [] [] \\
\text{imap} \ f \ (x : \text{xs}) & = \bot (:) (f \ x) (\text{imap} \ f \ \text{xs}) \top
\end{align*}
\]

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 \texttt{⟨f\rangle} rather than \texttt{(%)|}—by prefixing them with \texttt{†}. Our little parser becomes

\[
p\text{Tree} \ :: \ \text{Parser Tree} \\
p\text{Tree} = \bot \ \text{Leaf} \ † (\text{eat} ’\ast\') \top \\
\quad \quad \quad \bot \ \text{Node} \ † (\text{eat} ’\{’) \ p\text{Tree} \ † (\text{eat} ‘\}’) \ p\text{Tree} \ † (\text{eat} ’\)’) \top
\]

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.

\[
\begin{align*}
data \bot & = \bot \\
data † & = †
\end{align*}
\]

class \textbf{Idiom} \ i ⇒ \textbf{Idiomatic} \ i \ f \ g \ | \ g \ → \ i \ f \ \text{where} \\
\bot : : f \ → \ g \\
\text{idiomatic} : : i \ f \ → \ g
\]

instance \textbf{Idiom} \ i ⇒ \textbf{Idiomatic} \ i \ x \ (\bot \ → \ i \ x) \ \text{where} \\
\bot \ x \ \bot = \text{idi} \ x \\
\text{idiomatic} \ xi \ \bot = xi
\]

instance \textbf{Idiomatic} \ i \ f \ g ⇒ \textbf{Idiomatic} \ i \ (s \ → \ f) \ (i \ s \ → \ g) \ \text{where} \\
\bot \ sf = \text{idiomatic} \ (\text{idi} \ sf) \\
\text{idiomatic} \ sfi \ si = \text{idiomatic} \ (sfi \ (\%) \ si)
\]

instance \textbf{Idiomatic} \ i \ f \ g ⇒ \textbf{Idiomatic} \ i \ f \ († \ → \ i \ x \ → \ g) \ \text{where} \\
\bot \ f = \text{idiomatic} \ (\text{idi} \ f) \\
\text{idiomatic} \ fi \ † \ xi = \text{idiomatic} \ (fi \ (/) \ xi)
\]

Fig. 1. Implementing the \texttt{⟨} † \ ⟩ notation

I typically type \texttt{⟨} as \texttt{idI}, \texttt{⟩} as \texttt{Idi}, and \texttt{†} as \texttt{Ig}, in compliance with Haskell’s conventions for function and constructor names.

\section{6 Idiom is short for...}

\textit{weakly symmetric lax monoidal functor!}