

A NEIGHBORHOOD OF INFINITY

SATURDAY, JANUARY 17, 2009

Haskell Monoids and their Uses

Haskell is a great language for constructing code modularly from small but orthogonal building blocks. One of these small blocks is the monoid. Although monoids come from mathematics (algebra in particular) they are found everywhere in computing. You probably use one or two monoids implicitly with every line of code you write, whatever the language, but you might not know it yet. By making them explicit we find interesting new ways of constructing those lines of code. In particular, ways that are often easier to both read and write. So the following is an intro to monoids in Haskell. I'm assuming familiarity with type classes, because Haskell monoids form a type class. I also assume some familiarity with monads, though nothing too complex.

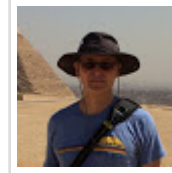
This post is literate Haskell so you can play with the examples directly.

Defining Monoids

In Haskell, a monoid is a type with a rule for how two elements of that type can be combined to make another element of the same type. To be a monoid there also needs to be an element that you can think of as representing 'nothing' in the sense that when it's combined with other elements it leaves the other element unchanged.

A great example is lists. Given two lists, say `[1,2]` and `[3,4]`, you can join them together using `++` to get `[1,2,3,4]`. There's also the empty list `[]`. Using `++` to combine `[]` with any list gives you back the same list, for example `[]++[1,2,3,4]==[1,2,3,4]`.

ABOUT ME



DAN PIPONI

Blog: [A Neighborhood of Infinity](#)

Code: [Github](#)

Twitter: [sigfpe](#)

Home page: [www.sigfpe.com](#)

[VIEW MY COMPLETE PROFILE](#)

PREVIOUS POSTS

[Rewriting Monadic Expressions with Template Haskell...](#)

[The Mother of all Monads](#)

[An Approach to Algorithm Parallelisation](#)

[Some thoughts on reasoning and monads](#)

[From Monoids to Monads](#)

[Operads and their Monads](#)

[What's the use of a transfinite ordinal?](#)

[Untangling with Continued Fractions: part 5](#)

[On writing Python one-liners.](#)

[Untangling with Continued Fractions: Part 4](#)

Another example is the type of integers, `Integer`. Given two elements, say 3 and 4, we can combine them with `+` to get 7. We also have the element 0 which when added to any other integer leaves it unchanged.

So here is a possible definition for the monoid type class:

```
class Monoid m where
  mappend :: m -> m -> m
  mempty  :: m
```

The function `mappend` is the function we use to combine pairs of elements, and `mempty` is the 'nothing' element. We can make lists an instance like this:

```
instance Monoid [a] where
  mappend = (++)
  mempty  = []
```

Because we want `mempty` to do nothing when combined with other elements we also require monoids to obey these two rules

```
a `mappend` mempty = a
```

and

```
mempty `mappend` a = a.
```

Notice how there are two ways to combine `a` and `b` using `mappend`. We can write `a `mappend` b` or `b `mappend` a`. There is no requirement on a monoid that these be equal to each other. (But see below.) But there is another property that monoids are required to have. Suppose we start with the list `[3,4]`. And now suppose we want to concatenate it with `[1,2]` on the left and `[5,6]` on the right. We could do the left concatenation first to get `[1,2]++[3,4]` and then

form `([1,2]++[3,4])++[5,6]`. But we could do the right one first and get `[1,2]++([3,4]++[5,6])`. Because we're concatenating at opposite ends the two operations don't interfere and it doesn't matter which we do first. This gives rise to the third and last requirement we have of monoids:

```
(a `mappend` b) `mappend` c == a `mappend` (b `mappend` c)
```

and you can summarise it with the slogan 'combining on the left doesn't interfere with combining on the right'. Notice how the integers, combined with `+`, also have this property. It's such a useful property it has a name: associativity.

That's a complete specification of what a monoid is. Haskell doesn't enforce the three laws I've given, but anyone reading code using a monoid will expect these laws to hold.

Some Uses of Monoids

But given that we already have individual functions like `++` and `+`, why would we ever want to use `mappend` instead?

One reason is that with a monoid we get another function called `mconcat` for free. `mconcat` takes a list of values in a monoid and combines them all together. For example `mconcat [a,b,c]` is equal to `a `mappend` (b `mappend` c)`. Any time you have a monoid you have this quick and easy way to combine a whole list together. But note that there is some ambiguity in the idea behind `mconcat`. To compute `mconcat [a,b,...,c,d]` which order should we work in? Should we work from left to right and compute `a `mappend` b` first? Or should we start with `c `mappend` d`. That's one place where the associativity law comes in: it makes no difference.

Another place where you might want to use a monoid is in code that is agnostic about how you want to combine elements. Just as `mconcat` works with any monoid, you might want to write your own code that works with any monoid.

Explicitly using the Monoid type class for a function also tells the

reader of your code what your intentions are. If a function has signature `[a] -> b` you know it takes a list and constructs an object of type `b` from it. But it has considerable freedom in what it can do with your list. But if you see a function of type `(Monoid a) => a -> b`, even if it is only used with lists, we know what kind of things the function will do with the list. For example, we know that the function might add things to your list, but it's never going to pull any elements out of your list.

The same type can give rise to a monoid in different ways. For example, I've already mentioned that the integers form a monoid. So we could define:

```
instance Monoid Integer where
    mappend = (+)
    mempty = 0
```

But there's a good reason not to do that: there's another natural way to make integers into a monoid:

```
instance Monoid Integer where
    mappend = (*)
    mempty = 1
```

We can't have both of these definitions at the same time. So the `Data.Monoid` library doesn't make `Integer` into a `Monoid` directly. Instead, it wraps them with `Sum` and `Product`. It also does so more generally so that you can make any `Num` type into a monoid in two different ways. We have both

```
Num a => Monoid (Sum a)
```

and

```
Num a => Monoid (Product a)
```

To use these we wrap our values in the appropriate wrapper and we

can then use the monoid functions. For example `mconcat [Sum 2,Sum 3,Sum 4]` is `Sum 9`, but `mconcat [Product 2,Product 3,Product 4]` is `[Product 24]`.

Using `Sum` and `Product` looks like a complicated way to do ordinary addition and multiplication. Why do things that way?

The Writer Monad

You can think of monoids as being accumulators. Given a running total, `n`, we can add in a new value `a` to get a new running total `n' = n `mappend` a`. Accumulating totals is a very common design pattern in real code so it's useful to abstract this idea. This is exactly what the Writer monad allows. We can write monadic code that accumulates values as a "side effect". The function to perform the accumulation is (somewhat confusingly) called `tell`. Here's an example where we're logging a trace of what we're doing.

```
> import Data.Monoid
> import Data.Foldable
> import Control.Monad.Writer
> import Control.Monad.State

> fact1 :: Integer -> Writer String Integer
> fact1 0 = return 1
> fact1 n = do
>   let n' = n-1
>   tell $ "We've taken one away from " ++ show n ++ "\n"
>   m <- fact1 n'
>   tell $ "We've called f " ++ show m ++ "\n"
>   let r = n*m
>   tell $ "We've multiplied " ++ show n ++ " and " ++ show m ++ "\n"
>   return r
```

This is an implementation of the factorial function that tells us what it did. Each time we call `tell` we combine its argument with the running log of all of the strings that we've 'told' so far. We use

`runWriter` to extract the results back out. If we run

```
> ex1 = runWriter (fact1 10)
```

we get back both $10!$ and a list of what it took to compute this.

But `Writer` allows us to accumulate more than just strings. We can use it with any monoid. For example, we can use it to count how many multiplications and subtractions were required to compute a given factorial. To do this we simply tell a value of the appropriate type. In this case we want to add values, and the monoid for addition is `Sum`. So instead we could implement:

```
> fact2 :: Integer -> Writer (Sum Integer) Integer
> fact2 0 = return 1
> fact2 n = do
>   let n' = n-1
>   tell $ Sum 1
>   m <- fact2 n'
>   let r = n*m
>   tell $ Sum 1
>   return r
```

```
> ex2 = runWriter (fact2 10)
```

There's another way we could have written this, using the state monad:

```
> fact3 :: Integer -> State Integer Integer
> fact3 0 = return 1
> fact3 n = do
```

```

> let n' = n-1
> modify (+1)
> m <- fact3 n'
> let r = n*m
> modify (+1)
> return r

> ex3 = runState (fact3 10) 0

```

It works just as well, but there is a big advantage to using the `Writer` version. It has type signature `f :: Integer -> Writer (Sum Integer) Integer`. We can immediately read from this that our function has a side effect that involves accumulating a number in a purely additive way. It's never going to, for example, multiply the accumulated value. The type information tells us a lot about what is going on inside the function without us having to read a single line of the implementation. The version written with `State` is free to do whatever it likes with the accumulated value and so it's harder to discern its purpose.

`Data.Monoid` also provides an `Any` monoid. This is the `Bool` type with the disjunction operator, better known as `||`. The idea behind the name is that if you combine together any collection of elements of type `Any` then the result is `Any True` precisely when at least any one of the original elements is `Any True`. If we think of these values as accumulators then they provide a kind of one way switch. We start accumulating with mempty, ie. `Any False`, and we can think of this as being the switch being off. Any time we accumulate `Any True` into our running 'total' the switch is turned on. This switch can never be switched off again by accumulating any more values. This models a pattern we often see in code: a flag that we want to switch on, as a side effect, if a certain condition is met at any point.

```

> fact4 :: Integer -> Writer Any Integer
> fact4 0 = return 1
> fact4 n = do
>   let n' = n-1
>   m <- fact4 n'
>   let r = n*m

```

```

> tell (Any (r==120))
> return r

> ex4 = runWriter (fact4 10)

```

At the end of our calculation we get $n!$, but we are also told if at any stage in the calculation two numbers were multiplied to give 120. We can almost read the `tell` line as if it were English: "tell my caller if any value of `r` is ever 120". Not only do we get the plumbing for this flag with a minimal amount of code. If we look at the type for this version of `f` it tells us exactly what's going on. We can read off immediately that this function, as a "side effect", computes a flag that can be turned on, but never turned off. That's a lot of useful information from just a type signature. In many other programming languages we might expect to see a boolean in the type signature, but we'd be forced to read the code to get any idea of how it will be used.

Commutative Monoids, Non-Commutative Monoids and Dual Monoids

Two elements of a monoid, x and y , are said to commute if $x \text{ `mappend` } y == y \text{ `mappend` } x$. The monoid itself is said to be commutative if all of its elements commute with each other. A good example of a commutative monoid is the type of integers. For any pair of integers, $a+b==b+a$.

If a monoid isn't commutative, it's said to be non-commutative. If it's non-commutative it means that for some x and y , $x \text{ `mappend` } y$ isn't the same as $y \text{ `mappend` } x$, so `mappend` and `flip mappend` are not the same function. For example `[1,2] ++ [3,4]` is different from `[3,4] ++ [1,2]`. This has the interesting consequence that we can make another monoid in which the combination function is `flip mappend`. We can still use the same `mempty` element, so the first two monoid laws hold. Additionally, it's a nice exercise to prove that the third monoid law still holds. This flipped monoid is called the dual monoid and `Data.Monoid` provides the `Dual` type constructor to build the dual of a monoid. We can use this to reverse the order in which the writer

monad accumulates values. For example the following code collects the execution trace in reverse order:

```
> fact5 :: Integer -> Writer (Dual String) Integer
> fact5 0 = return 1
> fact5 n = do
>   let n' = n-1
>   tell $ Dual $ "We've taken one away from " ++ show n ++ "\n"
>   m <- fact5 n'
>   tell $ Dual $ "We've called f " ++ show m ++ "\n"
>   let r = n*m
>   tell $ Dual $ "We've multiplied " ++ show n ++ " and " ++ show m ++ "\n"
>   return r

> ex5 = runWriter (fact5 10)
```

The Product Monoid

Suppose we want to accumulate two side effects at the same time. For example, maybe we want to both count instructions and leave a readable trace of our computation. We could use monad transformers to combine two writer monads. But there is a slightly easier way - we can combine two monoids into one 'product' monoid. It's defined like this:

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  mappend (u,v) (w,x) = (u `mappend` w, v `mappend` x)
```

Each time we use mappend on the product we actually perform a pair of mappends on each of the elements of the pair. With these small helper functions:

```
> tellFst a = tell $ (a,empty)
> tellSnd b = tell $ (empty,b)
```

we can now use two monoids simultaneously:

```
> fact6 :: Integer -> Writer (String,Sum Integer) Integer
> fact6 0 = return 1
> fact6 n = do
>   let n' = n-1
>   tellSnd (Sum 1)
>   tellFst $ "We've taken one away from " ++ show n ++ "\n"
>   m <- fact6 n'
>   let r = n*m
>   tellSnd (Sum 1)
>   tellFst $ "We've multiplied " ++ show n ++ " and " ++ show m ++ "\n"
>   return r

> ex6 = runWriter (fact6 5)
```

If we had simply implemented our code using one specific monoid, like lists, our code would be very limited in its application. But by using the general `Monoid` type class we ensure that users of our code can use not just any individual monoid, but even multiple monoids. This can make for more efficient code because it means we can perform multiple accumulations while traversing a data structure once. And yet we still ensure readability because our code is written using the interface to a single monoid making our algorithms simpler to read.

Foldable Data

One last application to mention is the `Data.Foldable` library. This provides a generic approach to walking through a datastructure, accumulating values as we go. The `foldMap` function applies a function to each element of our structure and then accumulates the

return values of each of these applications. An implementation of `foldMap` for a tree structure might be:

```
> data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)

> instance Foldable Tree where
>   foldMap f Empty = mempty
>   foldMap f (Leaf x) = f x
>   foldMap f (Node l k r) = foldMap f l `mappend` f k `mappend` foldMap f r
```

We can now use any of the monoids discussed above to compute properties of our trees. For example, we can use the function `(== 1)` to test whether each element is equal to 1 and then use the `Any` monoid to find out if any element of the tree is equal to 1. Here are a pair of examples: one to compute whether or not an element is equal to 1, and another to test if every element is greater than 5:

```
> tree = Node (Leaf 1) 7 (Leaf 2)

> ex7 = foldMap (Any . (== 1)) tree
> ex8 = foldMap (All . (> 5)) tree
```

Note, of course, that these expressions can be used, unmodified, with any foldable type, not just trees.

I hope you agree that this expresses our intentions in a way that is easy to read.

That suggests another exercise: write something similar to find the minimum or maximum element in a tree. You may need to construct a new monoid along the lines of `Any` and `All`. Try finding both in one traversal of the tree using the product monoid.

The foldable example also illustrates another point. The implementor of `foldMap` for the tree doesn't need to worry about whether the left tree should be combined with the central element before the right

tree. Associativity means it can be implemented either way and give the same results.

Recap

Monoids provide a general approach to combining and accumulating values. They allow us to write code that is agnostic about the method we will use to combine values, and that makes our code more reusable. By using named monoids we can write type signatures that express our intentions to people reading our code: for example by using `Any` instead of `Bool` we make it clear just how our boolean value is to be used. And we can combine the monoid-based building blocks provided by Haskell libraries to build useful and readable algorithms with a minimum of effort.

Some final notes: mathematicians often refer to `mappend` as a 'binary operator' and often it's called 'multiplication'. Just like in ordinary algebra, it's often also written with `ab` and `a*b` might both represent `a `mappend` b`. You can read more about monoids at [Wikipedia](#). And I wish I had time to talk about monoid morphisms, and why the list monoid is free (and what consequences that might have for how you can write code), and how compositing gives you [monoids](#) and a whole lot more.

POSTED BY DAN PIPONI AT SATURDAY, JANUARY 17, 2009



37 COMMENTS:

 jag said...

"Suppose we start with the list [3,4]. And now suppose we want to concatenate it with [1,2] on the left and [3,4] on the right."

Did you mean [5,6] on the right?

SATURDAY, 17 JANUARY, 2009

 sigfpe said...

jag,

Yes. When I have a moment I'll fix it.

SATURDAY, 17 JANUARY, 2009

 Vagif Verdi said...

Great article. Very well written, clear and accessible to haskell newbees.

I'm sure you know about haskell wikibook, which is a collection of haskell tutorials. Currently Monoids article is missing from wikibooks: <http://en.wikibooks.org/w/index.php?title=Haskell/Monoids&action=edit&redlink=1>

Would you mind to incorporate your article to wikibooks ? It would be a great addition and will enjoy much higher visibility there.

SATURDAY, 17 JANUARY, 2009

 jag said...

Indeed, great article, very well written. I hope you can find the time to tell us about all those things you're teasing us with in the last paragraph.

SATURDAY, 17 JANUARY, 2009

 Nicolas said...

Great article! I usually try to read your posts, but they are too complex for me most of the time. This one was very well written and simple enough.

I think I noticed a mistake:

mconcat [Product 2,Product 3,Product 4] is [Product 24].

Shouldn't that be Product 24 instead of the list [Product 24]?

Thanks for taking the time to write on this!

SUNDAY, 18 JANUARY, 2009

 Drew P. Vogel said...

This post seems as if it sprung from the Monoid vs Appendable naming discussion on haskell-cafe. This explanation of the Monoid type class is **exactly** what the documentation needs. Not only did I

fully understand it, I now also understand what the Writer Monad does for me.


SUNDAY, 18 JANUARY, 2009

 sigfpe said...

Drew,

I was, of course, following the little flamewar about monoids!

SUNDAY, 18 JANUARY, 2009

 Anonymous said...

Thanks, a very clear explanation of not just what monoids are, but also (and much more importantly) why they are an interesting abstraction for programming. I hope you get time to talk about the interesting-sounding stuff you mentioned at the end of your post.

A quick question: for the Writer monad, is it necessary for mappend to be associative?

SUNDAY, 18 JANUARY, 2009

 單中杰 said...

Great article! Surely you mean to cite [your earlier article on algorithm parallelization](#). (:

SUNDAY, 18 JANUARY, 2009

 Shin no Noir said...

I've come up with the following solution to the Min/Max monoid exercise, but it only works for instances of Bounded:

<http://hpaste.org/14067>

Does a nice solution exists for non-bounded types? The only possible solution I thought of so far is introducing a type that's isomorphic to Maybe.

SUNDAY, 18 JANUARY, 2009

 sigfpe said...

This comment has been removed by the author.

SUNDAY, 18 JANUARY, 2009

 Shin no Noir said...

Ah, right... deriving to the rescue. :)

SUNDAY, 18 JANUARY, 2009

 sigfpe said...

Here's a possible solution to the min/max problem:

This part ought to be in a library:

```
> data Smallest a = Smallest a | PlusInfinity deriving (Show,Eq,Ord)
```

```
> data Largest a = MinusInfinity | Largest a deriving (Show,Eq,Ord)
```

```
> instance (Ord a) => Monoid (Smallest a) where
```

```
> mempty = PlusInfinity
```

```
> mappend = min
```

```
> instance (Ord a) => Monoid (Largest a) where
```

```
> mempty = MinusInfinity
```

```
> mappend = max
```

```
> ex10 = foldMap (Smallest &&& Largest) tree :: (Smallest  
Int,Largest Int)
```

And the implementation would then be just:

```
> ex10 = foldMap (Smallest &&& Largest) tree :: (Smallest  
Int,Largest Int)
```

I got the idea for using &&& from <http://hpaste.org/14067>

This is an updated version. My first attempt was wrong :-(

SUNDAY, 18 JANUARY, 2009

 leithaus said...

Dan,

And when we have a 'notion of composition' (monoid ~ monad) plus
a 'notion of collection' (essentially presented as a monad) we can
autogenerate a 'relationally complete' logic.

$$m,n ::= e \mid g1 \mid \dots \mid gN \mid m*n$$

That freely generates our monoid terms (for a monoid with N generators), but needs to be whacked down by $m * e = m = e * m$, $m_1 * (m_2 * m_3) = (m_1 * m_2) * m_3$ (plus whatever other identities a specific monoid supports). If our 'notion of collection is set', then we get the following logic 'for free'.

```
k,l ::= true | ~k | k&l
e | g1 | ... | gN | k*l
for x in k.l | rec x.k | x
```

The boolean connectives at the top come from the fact that we're using set. The 'structural' connectives in the middle come from the monoid structure. The other structure comes from calculations of fixpoints.

See [algebraic databases](#) for more discussion. Or, [indexed compositions](#) for applications.

MONDAY, 19 JANUARY, 2009



lodi said...

This was an amazingly straightforward yet enlightening article; the section on the 'Product' monoid was particularly novel for me. I would love to see you expand on the concepts you mentioned towards the end, grounded with real-world haskell examples (as you did in this post.) Thanks for writing sigfpe!

P.S. In my opinion you just single-handedly resolved the 'Appendable' debate on haskell-cafe with your tree folding examples.

WEDNESDAY, 21 JANUARY, 2009



Anonymous said...

Multiplying monoids gives a product monoid. This is associative and has a neutral element. So monoids form a monoid :) (up to isomorphism)

THURSDAY, 22 JANUARY, 2009



sigfpe said...

Anonymous,

Mooids form a [monoidal category](#) in fact.

THURSDAY, 22 JANUARY, 2009

 leithaus said...

And monads are a monoid object in the category of endofunctors.

In the language of the times monoid is a minimalist notion of composition before categorification. Monad is a minimalist notion of composition post categorification.

THURSDAY, 22 JANUARY, 2009

 André Pang said...

Thanks for an excellent article, Dan. I normally get about halfway through your blog posts before my brain explodes; eliminating all the jargon in your post and explaining all the maths terms helped tremendously. (For example, I know what associativity is, but I always confuse it with commutativity, and it's great that you cater for the dumb folk like me without having to look up all the terms to clarify the terms :). It helps a ton with the flow of the article.)

Please keep up writing blog posts targeted at the poor folks like me, who have some Haskell skills but don't really know all the maths jargon!

SATURDAY, 24 JANUARY, 2009

 Peter Berry said...

With the comment about "Any" being clearer than "Bool" about the intentions of the code, it seems like it's actually a Good Thing that Haskell requires newtype wrappers to coax different monoids out of the same type.

As for Writer and commutativity, I think that to be a monad it has to satisfy the monad "commutativity" law, for which it relies on the monoid's commutativity. Is that right?

SATURDAY, 24 JANUARY, 2009

 Anonymous said...

Wow, I for one would love to see a book full of material like this ! A book exposing Haskell mathematical underpinnings and techniques to harness this power in day to day programming...

Do you feel up to it?

TUESDAY, 27 JANUARY, 2009

 sigfpe said...


Peter,

Any monoid makes Writer a monad. Any commutative monoid makes Writer a commutative monad.

Anynonymous,

I'd love to collect together a bunch of algebraic programming stuff in a book. But I also know it's a vast amount of work to get a book written. I'd need several months off work.

TUESDAY, 27 JANUARY, 2009

 Anonymous said...

Well, you can certainly rescue some of your blog posts to that end. This one on monoid is perfect really, theory meets practice !

Maybe you could team up with [Good Math](#) ?

The two of you have already produced quite a bit along those lines.

The book would obviously need a concerted approach with respect to the topics presented...

I am pretty sure, a lot of newcomers to Haskell would love to read more about the Mathematics lurking behind its powerful features.

Maybe it could be self-published, something like Lulu ?

THURSDAY, 05 FEBRUARY, 2009

 Carl Masak said...

Thank you for an informative and approachable post. The things about Writer was especially helpful for a problem in Haskell that I've been mulling over.


SATURDAY, 21 FEBRUARY, 2009

 johnbender said...

As much as everyone else has said this: great article. I'm relatively new to Haskell and this was completely readable.

Thank you for sharing!

SATURDAY, 11 APRIL, 2009

 Anonymous said...

Great post. One minor note:

(Monoid a) => a -> b

should be

(Monoid a) => a -> a

otherwise there is no way the monoid could be changed and a the eval unsafe cast avoided.

TUESDAY, 14 APRIL, 2009

 Qrilka said...

Dan, I want to translate your post to russian and publish (linking to the original text). Do I need make any additional steps from copyright point of view?

SUNDAY, 03 MAY, 2009

 sigfpe said...

Qrilka,

Feel free to make a translation. Link back here and maybe put a link here pointing to the translation.

SUNDAY, 03 MAY, 2009

 Qrilka said...

BTW "to combine [] with any list leaves gives you back" maybe "leaves" is unnecessary here?

TUESDAY, 05 MAY, 2009

 Qrilka said...

And also why are you talking about signatures and adding/pulling out elements? The lists are immutable here I think. Am I wrong?

SATURDAY, 09 MAY, 2009

 Qrilka said...

So we finished 1st ussue of our journal and your article is in it - <http://fprog.ru/2009/issue1/>

(Not sure if you can read Russian though)

MONDAY, 20 JULY, 2009

 Loup Vaillant said...

@ Qrilka


> And also why are you talking about signatures and adding/pulling out elements? The lists are immutable here I think. Am I wrong?

That was just an abuse of language. But saying "producing a result which is the addition/removal of elements of the original list" would sound a bit cumbersome.

Note that the exact same abuse of language is made when we are talking about mutable variables: when you "change" a variable, you actually replace the value it contains by another value. The variable itself, as a container, didn't really change. And the value that was previously in it certainly didn't change at all. You just lost one reference to it.

Anyway, [assingment](#) is evil, so...

TUESDAY, 24 NOVEMBER, 2009

 Anonymous said...

Reading this plain-language excellent clearly-spoken clear article makes me real mad about how poorly written is your given Haskell package documentation. Take Data.Monoid for example. This just does NOT parse for anyone coming to read it for the first time.

Too Much Elitism among Haskell documentation writers.

Also, the stuff about Any being clear on the semantics, I don't agree. It's just a suggestive name; it could have semantics of anything really. We still need to read into the code, and now it is FAR far away, hidden somewhere in an instance declaration. And Haddock isn't really at all good at producing instance documentation, is it, being that it does NOT produce any.

Instead of writing all this bits and pieces in myriad of classes, why not just use a functional approach, and make e.g. foldMap function to accept an explicit functional argument for a combining function?

That's what's done implicitly anyway, selecting the specific mappend version thru the type dictionary behind the curtain.

Is all this typeclass stuff just a distraction really?

WEDNESDAY, 06 JULY, 2011

 sigfpe said...

@Anonymous Hey! It's not **my** Data.Monoid package.

I don't believe the sparsity of the documentation has anything to do with elitism. It's missing documentation because Haskell code is largely written by volunteers who have limited time on their hands. I wrote this article to help remedy this fact. I hope it's been of use. Even better might be for me to fold some of what I've written here back into the documentation.


I agree that seeing 'Any' on its own means little. But with documentation it makes a great mnemonic.

WEDNESDAY, 06 JULY, 2011

 dstacruz said...

Just wanted to add my +1 to this article. Thanks for taking the time to write it!

FRIDAY, 15 JULY, 2011

 Anonymous said...

"Also, the stuff about Any being clear on the semantics, I don't agree. It's just a suggestive name; it could have semantics of anything really. We still need to read into the code, and now it is FAR far away, hidden somewhere in an instance declaration."

Not true, you don't need to read any instance anything (obviously you'll have to know what Monoids are, then you can intuit what Any does). Once you see Any in the signature you know *something* is getting flagged. You will have to look at the code for that specific function to find out exactly *what*.

"Instead of writing all this bits and pieces in myriad of classes, why not just use a functional approach, and make e.g. foldMap function to accept an explicit functional argument for a combining function?"

Haskell already has this if that's all you want (e.g. foldr/foldl). Things like foldMap allow you to get even tighter with your type definitions. The more of your **logic** you can turn into a type the more the type system can help you find bugs.

TUESDAY, 04 OCTOBER, 2011



Anonymous said...

Great article, as we say in french : ce qui se conçoit bien s'énonce clairement. Thanks.

SATURDAY, 26 JANUARY, 2013

[POST A COMMENT](#)

LINKS TO THIS POST:

[CREATE A LINK](#)

[<< Home](#)