Lax monoidal functors with a different monoidal structure

Applicative functors are well-known and well-loved among Haskellers, for their ability to apply functions in an effectful context.

In category-theoretic terms, it can be shown that the methods of `Applicative`:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

are equivalent to having a `Functor f` with the operations:

```
unit :: f ()
(**) :: (f a, f b) -> f (a,b)
```

the idea being that to write `pure` you just replace the `()` in `unit` with the given value, and to write `(<*>)` you squish the function and argument into a tuple and then map a suitable application function over it.

Moreover, this correspondence turns the `Applicative` laws into natural monoidal-ish laws about `unit` and `(**)`, so in fact an applicative functor is precisely what a category theorist would call a lax monoidal functor (lax because `(**)` is merely a natural transformation and not an isomorphism).

Okay, fine, great. This much is well-known. But that's only one family of lax monoidal functors – those respecting the monoidal structure of the *product*. A lax monoidal functor involves two choices of monoidal structure, in the source and destination: here's what you get if you turn product into sum:

```
class PtS f where
  unit :: f Void
  (**) :: f a -> f b -> f (Either a b)

-- some example instances
instance PtS Maybe where
  unit = Nothing
  Nothing ** Nothing = Nothing
  Just a ** Nothing = Just (Left a)
  Nothing ** Just b = Just (Right b)
  Just a ** Just b = Just (Left a) -- ick, but it does satisfy the laws

instance PtS [] where
  unit = []
  xs ** ys = map Left xs ++ map Right ys
```

It seems like turning sum into other monoidal structures is made less interesting by `unit :: Void -> f Void` being uniquely determined, so you really have more of a semigroup going on. But still:

- Are other lax monoidal functors like the above studied or useful?
- Is there a neat alternative presentation for them like the `Applicative` one?

haskell functor applicative category-theory

edited May 26 '14 at 22:02

asked Apr 26 '14 at 20:38

Ben Millwood
4,723 ● 11 ● 37

When you say `Void` in the type of `PtS.unit`, don't you mean Empty, since it should be a unit for `Either`? – Dominique Devriese Apr 27 '14 at 7:04

Never mind, you probably intended `Void` to represent the empty type. I was confused because the name void in C-like languages corresponds to the unit type, which you write as `()`. – Dominique Devriese Apr 27 '14 at 7:20

Yep, sorry for the confusion, but there's reasonable precedent for it: hackage.haskell.org/package/void – Ben Millwood Apr 27 '14 at 11:10

2 Ack, it's arguably C's usage of void that is wrong, not yours ;) – Dominique Devriese Apr 27 '14 at 17:25

3 Answers

The "neat alternative presentation" for `Applicative` is based on the following two equivalencies

```
pure a = fmap (const a) unit
unit = pure ()

ff <*> fa = fmap (\(f,a) -> f a) $ ff ** fa
fa ** fb = pure (,) <*> fa <*> fb
```

The trick to get this "neat alternative presentation" for `Applicative` is the same as the trick for `zipWith` - replace explicit types and constructors in the interface with things that the type or constructor can be passed into to recover what the original interface was.

```
unit :: f ()
```

Is replaced with `pure` which we can substitute the type `()` and the constructor `() :: ()` into to recover `unit`.

```
pure :: a  -> f a
pure    () :: f ()
```

And similarly (though not as straightforward) for substituting the type `(a,b)` and the constructor `(,) :: a -> b -> (a,b)` into `liftA2` to recover `**`.

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2    (,)           :: f a -> f b -> f (a,b)
```

`Applicative` then gets the nice `<*>` operator by lifting function application `($) :: (a -> b) -> a -> b` into the functor.

```
(<*>) :: f (a -> b) -> f a -> f b
(<*>) = liftA2 ($)
```

To find a "neat alternative presentation" for `PtS` we need to find

- something we can substitute the type `Void` into to recover `unit`
- something we can substitute the type `Either a b` and the constructors `Left :: a -> Either a b` and `Right :: b -> Either a b` into to recover `**`

(If you notice that we already have something the constructors `Left` and `Right` can be passed to you can probably figure out what we can replace `**` with without following the steps I used; I didn't notice this until after I solved it)

unit

This immediately gets us an alternative to `unit` for sums:

```
empty :: f a
empty = fmap absurd unit

unit :: f Void
unit = empty
```

operator

We'd like to find an alternative to `(**)`. There is an alternative to sums like `Either` that allows them to be written as functions of products. It shows up as the visitor pattern in object oriented programming languages where sums don't exist.

```
data Either a b = Left a | Right b

{-# LANGUAGE RankNTypes #-}
type Sum a b = forall c. (a -> c) -> (b -> c) -> c
```

It's what you would get if you changed the order of `either`'s arguments and partially applied them.

```
either :: (a -> c) -> (b -> c) -> Either a b -> c

toSum :: Either a b -> Sum a b
toSum e = \forA forB -> either forA forB e

toEither :: Sum a b -> Either a b
toEither s = s Left Right
```

We can see that `Either a b ≅ Sum a b`. This allows us to rewrite the type for `(**)`

```
(**) :: f a -> f b -> f (Either a b)
(**) :: f a -> f b -> f (Sum a b)
(**) :: f a -> f b -> f ((a -> c) -> (b -> c) -> c)
```

Now it's clear what `**` does. It delays `fmap`ing something onto both of its arguments, and combines the results of those two mappings. If we introduce a new operator, `<||> :: f c -> f c -> f c` which simply assumes that the `fmap`ing was done already, then we can see that

```
fmap (\f -> f forA forB) (fa ** fb) = fmap forA fa <||> fmap forB fb
```

Or back in terms of `Either`:

```
fa ** fb = fmap Left fa <||> fmap Right fb
fa1 <||> fa2 = fmap (either id id) $ fa1 ** fa2
```

So we can express everything `PtS` can express with the following class, and everything that could implement `PtS` can implement the following class:

```
class Functor f => AlmostAlternative f where
    empty  :: f a
    (<||>) :: f a -> f a -> f a
```

This is almost certainly the same as the `Alternative` class, except we didn't require that the `Functor` be `Applicative`.

Conclusion

It's just a `Functor` that is a `Monoid` for all types. It'd be equivalent to the following:

```
class (Functor f, forall a. Monoid (f a)) => MonoidalFunctor f
```

The `forall a. Monoid (f a)` constraint is pseudo-code; I don't know a way to express constraints like this in Haskell.

edited Apr 29 '14 at 3:58

answered Apr 27 '14 at 7:00

Cirdec
16.1k ●1 ●24 ●62

+1 for actually analysing and argumenting your answer ;). – Dominique Devriese Apr 27 '14 at 7:17

Perfect! Now annoyed I didn't spot this myself :P (and also, curse you, you've given me a nontrivial choice of which answer to accept) – Ben Millwood Apr 27 '14 at 11:08

(Small point: you describe sums as analogous to "functions of products", do you mean "products of functions"?) – Ben Millwood Apr 27 '14 at 11:18

No, I mean't functions of products. A sum is a function that accepts the product of two functions. You get data out of the sum by passing in two functions (the product of two functions) - one for what to do for the first option, one for what to do with the second. If the sum is the first option, it takes the first function from the product, applies it to what the sum contains, and returns the result. If the sum is the second option, it takes the second function from the product, applies it to what the sum contains, and returns the result. A sum is a function that takes a product of functions. – Cirdec Apr 27 '14 at 16:12

Oh, I see what you mean, yes. – Ben Millwood Apr 27 '14 at 17:32

Before you can even talk about monoidal functors, you need to make sure you're in a monoidal category. It so happens that **Hask** is a monoidal category in the following way:

- `()` as identity
- `(,)` as bifunctor
- Identify isomorphic types, i.e. `(a,()) ≅ ((),a) ≅ a`, and `(a,(b,c)) ≅ ((a,b),c)`.

Like you observed, it's also a monoidal category when you exchange `()` for `Void` and `(,)` for `Either`.
However, monoidal doesn't get you very far – what makes **Hask** so powerful is that it's cartesian closed. That gives us currying and related techniques, without which applicative would be pretty much useless.

A monoidal category can be cartesian closed iff its identity is a terminal object, i.e. a type *onto* which there exists precisely one (of course, we disregard □ here) arrow. There is one function `A -> ()` for any type `A`, namely `const ()`. There is no function `A -> Void` however. Instead, `Void` is the *initial object*: there exists precisely one arrow *from* it, namely the `absurd :: Void -> a` method. Such a monoidal category can't be cartesian closed then.
Now, of course, you can switch between initial and terminal easily by turning around the arrow direction. That always places you in the dual structure, so we get a cocartesian closed category. But that means you also need to flip the arrows in your monoidal functors. Those are called decisive functors then (and generalise comonads). With Conor's ever-so-amazing naming scheme,

```
class (Functor f) => Decisive f where
  nogood :: f Void -> Void
  orwell :: f (Either s t) -> Either (f s) (f t)
```

edited Apr 26 '14 at 22:16

answered Apr 26 '14 at 21:44

leftaroundabout
42.6k ●3 ●68 ●142

2 I'm aware this doesn't really answer your question. A monoidal functor WRT coproducts might still be interesting in some way, but I suppose troubles like `unit` being trivial as you say largely hampers this. – leftaroundabout Apr 26 '14 at 21:53

I knew you must be the author of this post was by the time I finished reading the first sentence. – Cirdec

Apr 27 '14 at 4:51

My background in category theory is very limited, but FWIW, your PtS class reminds me of the `Alternative` class, which looks essentially like this:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

The only problem of course is that `Alternative` is an extension of `Applicative`. However, perhaps one can imagine it being presented separately, and the combination with `Applicative` is then quite reminiscent of a functor with a non-commutative ring-like structure, with the two monoid structures as the operations of the ring? There are also distributivity laws between `Applicative` and `Alternative` IIRC.

edited Apr 27 '14 at 7:17

answered Apr 27 '14 at 6:02

Dominique Devriese
2,044 ● 1 ● 5 ● 15

+1 for seeing straight through the problem. After working through the problem by hand, I arrived at the same conclusion for my answer. – Cirdec Apr 27 '14 at 7:04