Confused by the meaning of the 'Alternative' type class and its relationship to other type classes

I've been going through the Typeclassopedia to learn the type classes. I'm stuck understanding `Alternative` (and `MonadPlus`, for that matter).

The problems I'm having:

> the 'pedia says that "the Alternative type class is for Applicative functors which also have a monoid structure." I don't get this -- doesn't Alternative mean something totally different from Monoid? i.e. I understood the point of the Alternative type class as picking between two things, whereas I understood Monoids as being about combining things.
>
> why does Alternative need an `empty` method/member? I may be wrong, but it seems to not be used at all ... at least in the code I could find. And it seems not to fit with the theme of the class -- if I have two things, and need to pick one, what do I need an 'empty' for?
>
> why does the Alternative type class need an Applicative constraint, and why does it need a kind of `* -> *`? Why not just have `<|> :: a -> a -> a`? All of the instances could still be implemented in the same way ... I think (not sure). What value does it provide that Monoid doesn't?
>
> what's the point of the `MonadPlus` type class? Can't I unlock all of its goodness by just using something as both a `Monad` and `Alternative`? Why not just ditch it? (I'm sure I'm wrong, but I don't have any counterexamples)

Hopefully all those questions are coherent ... !

Bounty update: @Antal's answer is a great start, but Q3 is still open: what does Alternative provide that Monoid doesn't? I find this answer unsatisfactory since it lacks concrete examples, and a specific discussion of how the higher-kindedness of Alternative distinguishes it from Monoid.

If it's to combine applicative's effects with Monoid's behavior, why not just:

```
liftA2 mappend
```

This is even more confusing for me because many Monoid instances are exactly the same as the Alternative instances.

That's why I'm looking for **specific examples** that show why Alternative is necessary, and how it's different -- or means something different -- from Monoid.

haskell typeclass

edited Nov 3 '13 at 17:42
PeeHaa
38.5k • 28 • 123 • 203

asked Oct 26 '12 at 4:11
Matt Fenwick
22k • 7 • 72 • 133

2 Check out this question and the two questions linked within. – Rafael Caetano Oct 26 '12 at 4:44

Also see this answer. – Matt Fenwick Dec 3 '12 at 21:07

5 Answers

To begin with, let me offer short answers to each of these questions. I will then expand each into a longer detailed answer, but these short ones will hopefully help in navigating those.

1. No, `Alternative` and `Monoid` don't mean different things; `Alternative` is for types which have the structure both of `Applicative` and of `Monoid`. "Picking" and "combining" are two different intuitions for the same broader concept.
2. `Alternative` contains `empty` as well as `<|>` because the designers thought this would be useful, and because this gives rise to a monoid. In terms of picking, `empty` corresponds to making an impossible choice.
3. We need both `Alternative` and `Monoid` because the former obeys (or should) *more* laws than the latter; these laws relate the monoidal and applicative structure of the type constructor. Additionally, `Alternative` can't depend on the inner type, while `Monoid` can.
4. `MonadPlus` is slightly stronger than `Alternative`, as it must obey more laws; these laws relate the monoidal structure to the monadic structure in addition to the applicative structure. If you have instances of both, they should coincide.

> Doesn't `Alternative` mean something totally different from `Monoid`?

Not really! Part of the reason for your confusion is that the Haskell `Monoid` class uses some pretty bad (well, insufficiently general) names. This is how a mathematician would define a monoid (being very explicit about it):

Definition. A *monoid* is a set M equipped with a distinguished element $\varepsilon \in M$ and a binary operator $\cdot : M \times M \to M$, denoted by juxtaposition, such that the following two conditions hold:

1. ε is the identity: for all $m \in M$, $m\varepsilon = \varepsilon m = m$.

2. $\cdot$ is associative: for all $m_1,m_2,m_3 \in M$, $(m_1m_2)m_3 = m_1(m_2m_3)$.

That's it. In Haskell, ε is spelled `mempty` and · is spelled `mappend` (or, these days, `<>`), and the set *M* is the type `M` in `instance Monoid M where ...`.

Looking at this definition, we see that it says nothing about "combining" (or about "picking," for that matter). It says things about · and about *ε*, but that's it. Now, it's certainly true that combining things works well with this structure: *ε* corresponds to having no things, and m_1m_2 says that if I glom m_1 and m_2's stuff together, I can get a new thing containing all their stuff. But here's an alternative intuition: *ε* corresponds to no choices at all, and m_1m_2 corresponds to a choice between m_1 and m_2. This is the "picking" intuition. Note that both obey the monoid laws:

1. Having nothing at all and having no choice are both the identity.
   If I have no stuff and glom it together with some stuff, I end up with that same stuff again.

   If I have a choice between no choice at all (something impossible) and some other choice, I have to pick the other (possible) choice.
2. Glomming collections together and making a choice are both associative.
   If I have three collections of things, it doesn't matter if I glom the first two together and then the third, or the last two together and then the first; either way, I end up with the same total glommed collection.

   If I have a choice between three things, it doesn't matter if I (a) first choose between first-or-second and third and then, if I need to, between first and second, or (b) first choose between first and second-or-third and then, if I need to, between second and third. Either way, I can pick what I want.

(Note: I'm playing fast and loose here; that's why it's intuition. For instance, it's important to remember that · need not be commutative, which the above glosses over: it's perfectly possible that $m_1m_2 \neq m_2m_1$.)

Behold: both these sorts of things (and many others—is multiplying numbers really either "combining" *or* "picking"?) obey the same rules. Having an intuition is important to develop understanding, but it's the rules and definitions that determine what's *actually* going on.

And the best part is that these both of these intuitions can be interpreted by the same carrier! Let *M* be some set of sets (not a set of *all* sets!) containing the empty set, let *ε* be the empty set ∅, and let · be set union ∪. It is easy to see that ∅ is an identity for ∪, and that ∪ is associative, so we can conclude that $(M,\emptyset,\cup)$ is a monoid. Now:

1. If we think about sets as being collections of things, then ∪ corresponds to glomming them together to get more things—the "combining" intuition.
2. If we think about sets as representing possible actions, then ∪ corresponds to increasing your pool of possible actions to pick from—the "picking" intuition.

And this is exactly what's going on with `[]` in Haskell: `[a]` is a `Monoid` for all `a`, and `[]` as an applicative functor (and monad) is used to represent nondeterminism. Both the combining and the picking intuitions coincide at the same type: `mempty = empty = []` and `mappend = (<|>) = (++)`.

So the `Alternative` class is just there to represent objects which (a) are applicative functors, and (b) when instantiated at a type, have a value and a binary function on them which follow some rules. Which rules? The monoid rules. Why? Because it turns out to be useful :-)

> Why does `Alternative` need an empty method/member?

Well, the snarky answer is "because `Alternative` represents a monoid structure." But the real question is: *why* a monoid structure? Why not just a semigroup, a monoid without *ε*? One answer is to claim that monoids are just more useful. I think many people (but perhaps not Edward Kmett) would agree with this; almost all of the time, if you have a sensible `(<|>)` / `mappend` /·, you'll be able to define a sensible `empty` / `mempty` /*ε*. On the other hand, having the extra generality is nice, since it lets you place more things under the umbrella.

You also want to know how this meshes with the "picking" intuition. Keeping in mind that, in some sense, the right answer is "know when to abandon the 'picking' intuition," I think you *can* unify the two. Consider `[]`, the applicative functor for nondeterminism. If I combine two values of type `[a]` with `(<|>)`, that corresponds to nondeterministically picking either an action from the left or an action from the right. But sometimes, you're going to have no possible actions on one side—and that's fine. Similarly, if we consider parsers, `(<|>)` represents a parser which parses either what's on the left or what's on the right (it "picks"). And if you have a parser which always fails, that ends up being an identity: if you pick it, you immediately reject that pick and try the other one.

All this said, remember that it *would* be entirely possible to have a class almost like `Alternative`, but lacking `empty`. That would be perfectly valid—it could even be a superclass of `Alternative` —but happens not to be what Haskell did. Presumably this is out of a guess as to what's useful.

> Why does the `Alternative` type class need an `Applicative` constraint, and why does it need a kind of `* -> *`? … Why not just [use] `liftA2 mappend`?

Well, let's consider each of these three proposed changes: getting rid of the `Applicative` constraint for `Alternative`; changing the kind of `Alternative` 's argument; and using `liftA2 mappend` instead of `<|>` and `pure mempty` instead of `empty`. We'll look at this third change first, since it's the most different. Suppose we got rid of `Alternative` entirely, and replaced the class with two plain functions:

```
fempty :: (Applicative f, Monoid a) => f a
fempty = pure mempty

(>|<) :: (Applicative f, Monoid a) => f a -> f a -> f a
(>|<) = liftA2 mappend
```

We could even keep the definitions of `some` and `many`. And this *does* give us a monoid structure, it's true. But it seems like it gives us the wrong one . Should `Just fst >|< Just snd` fail, since `(a,a) -> a` isn't an instance of `Monoid`? No, but that's what the above code would result in. The monoid instance we *want* is one that's inner-type agnostic (to borrow terminology from Matthew Farkas-Dyck in a very related haskell-cafe discussion which asks some very similar questions); the `Alternative` structure is about a monoid determined by `f`'s structure, not the structure of `f`'s argument.

Now that we think we want to leave `Alternative` as some sort of type class, let's look at the two proposed ways to change it. If we change the kind, we *have* to get rid of the `Applicative` constraint; `Applicative` only talks about things of kind `* -> *`, and so there's no way to refer to it. That leaves two possible changes; the first, more minor, change is to get rid of the `Applicative` constraint but leave the kind alone:

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

The other, larger, change is to get rid of the `Applicative` constraint and change the kind:

```
class Alternative'' a where
  empty'' :: a
  (<|||>) :: a -> a -> a
```

In both cases, we have to get rid of `some` / `many`, but that's OK; we can define them as standalone functions with the type `(Applicative f, Alternative' f) => f a -> f [a]` or `(Applicative f, Alternative'' (f [a])) => f a -> f [a]`.

Now, in the second case, where we change the kind of the type variable, we see that our class is exactly the same as `Monoid` (or, if you still want to remove `empty''`, `Semigroup`), so there's no advantage to having a separate class. And in fact, even if we leave the kind variable alone but remove the `Applicative` constraint, `Alternative` just becomes `forall a. Monoid (f a)`, although we can't write these quantified constraints in Haskell, not even with all the fancy GHC extensions. (Note that this expresses the inner-type–agnosticism mentioned above.) Thus, if we can make either of these changes, then we have no reason to keep `Alternative` (except for being able to express that quantified constraint, but that hardly seems compelling).

So the question boils down to "is there a relationship between the `Alternative` parts and the `Applicative` parts of an `f` which is an instance of both?" And while there's nothing in the documentation, I'm going to take a stand and say *yes*—or at the very least, there *ought* to be. I think that `Alternative` is supposed to obey some laws relating to `Applicative` (in addition to the monoid laws); in particular, I think those laws are something like

1. **Right distributivity (of `<*>`):** `(f <|> g) <*> a = (f <*> a) <|> (g <*> a)`
2. **Right absorption (for `<*>`):** `empty <*> a = empty`
3. **Left distributivity (of `fmap`):** `f <$> (a <|> b) = (f <$> a) <|> (f <$> b)`
4. **Left absorption (for `fmap`):** `f <$> empty = empty`

These laws appear to be true for `[]` and `Maybe`, and (pretending its `MonadPlus` instance is an `Alternative` instance) `IO`, but I haven't done any proofs or exhaustive testing. (For instance, I originally thought that *left* distributivity held for `<*>`, but this "performs the effects" in the wrong order for `[]`.) By way of analogy, though, it is true that `MonadPlus` is expected to obey similar laws (although there is apparently some ambiguity about which). I had originally wanted to claim a third law, which seems natural:

> **Left absorption (for `<*>`):** `a <*> empty = empty`

However, although I believe `[]` and `Maybe` obey this law, `IO` doesn't, and I think (for reasons that will become apparent in the next couple of paragraphs) it's best not to require it.

And indeed, it appears that Edward Kmett has some slides where he espouses a similar view; to get into that, we'll need to take brief digression involving some more mathematical jargon. The final slide, "I Want More Structure," says that "A Monoid is to an Applicative as a Right Seminearring is to an Alternative," and "If you throw away the argument of an Applicative, you get a Monoid, if you throw away the argument of an Alternative you get a RightSemiNearRing."

Right seminearrings? "How did right seminearrings get into it?" I hear you cry. Well,

Definition. A *right near-semiring* (also *right seminearring*, but the former seems to be used more on Google) is a quadruple $(R,+,\cdot,0)$ where $(R,+,0)$ is a monoid, $(R,\cdot)$ is a semigroup, and the following two conditions hold:

1. $\cdot$ is right-distributive over $+$: for all $r,s,t \in R$, $(s + t)r = sr + tr$.
2. 0 is right-absorbing for $\cdot$: for all $r \in R$, $0r = 0$.

A *left near-semiring* is defined analogously.

Now, this doesn't quite work, because `<*>` is not truly associative or a binary operator—the types don't match. I think this is what Edward Kmett is getting at when he talks about "throw[ing] away the argument." Another option might be to say (I'm unsure if this is right) that we actually want (`f a` , `<|>` , `<*>` , `empty`) to form a *right near-semiringoid*, where the "-oid" suffix indicates that the binary operators can only be applied to specific pairs of elements (à la *groupoids*). And we'd also want to say that (`f a` , `<|>` , `<$>` , `empty`) was a left near-semiringoid, although this could conceivably follow from the combination of the `Applicative` laws and the right near-semiringoid structure. But now I'm getting in over my head, and this isn't deeply relevant anyway.

At any rate, these laws, being *stronger* than the monoid laws, mean that perfectly valid `Monoid` instances would become invalid `Alternative` instances. There are (at least) two examples of this in the standard library: `Monoid a => (a,)` and `Maybe` . Let's look at each of them quickly.

Given any two monoids, their product is a monoid; consequently, tuples can be made an instance of `Monoid` in the obvious way (reformatting the base package's source):

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  (a1,b1) `mappend` (a2,b2) = (a1 `mappend` a2, b1 `mappend` b2)
```

Similarly, we can make tuples whose first component is an element of a monoid into an instance of `Applicative` by accumulating the monoid elements (reformatting the base package's source):

```
instance Monoid a => Applicative ((,) a) where
  pure x = (mempty, x)
  (u, f) <*> (v, x) = (u `mappend` v, f x)
```

However, tuples aren't an instance of `Alternative` , because they can't be—the monoidal structure over `Monoid a => (a,b)` isn't present for all types `b` , and `Alternative` 's monoidal structure must be inner-type agnostic. Not only must `b` be a monad, to be able to express `(f <> g) <*> a` , we need to use the `Monoid` instance for functions, which is for functions of the form `Monoid b => a -> b` . And even in the case where we have all the necessary monoidal structure, it violates *all four* of the `Alternative` laws. To see this, let `ssf n = (Sum n, (<> Sum n))` and let `ssn = (Sum n, Sum n)` . Then, writing `(<>)` for `mappend` , we get the following results (which can be checked in GHCi, with the occasional type annotation):

1. Right distributivity:
   - `(ssf 1 <> ssf 1) <*> ssn 1 = (Sum 3, Sum 4)`
   - `(ssf 1 <*> ssn 1) <> (ssf 1 <*> ssn 1) = (Sum 4, Sum 4)`
2. Right absorption:
   - `mempty <*> ssn 1 = (Sum 1, Sum 0)`
   - `mempty = (Sum 0, Sum 0)`
3. Left distributivity:
   - `(<> Sum 1) <$> (ssn 1 <> ssn 1) = (Sum 2, Sum 3)`
   - `((<> Sum 1) <$> ssn 1) <> ((<> Sum 1) <$> ssn 1) = (Sum 2, Sum 4)`
4. Left absorption:
   - `(<> Sum 1) <$> mempty = (Sum 0, Sum 1)`
   - `mempty = (Sum 1, Sum 1)`

Next, consider `Maybe` . As it stands, `Maybe` 's `Monoid` and `Alternative` instances *disagree*. (Although the haskell-cafe discussion I mention at the beginning of this section proposes changing this, there's an `Option` newtype from the semigroups package which would produce the same effect.) As a `Monoid` , `Maybe` lifts semigroups into monoids by using `Nothing` as the identity; since the base package doesn't have a semigroup class, it just lifts monoids, and so we get (reformatting the base package's source):

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m       = m
  m       `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

On the other hand, as an `Alternative` , `Maybe` represents prioritized choice with failure, and so we get (again reformatting the base package's source):

```
instance Alternative Maybe where
  empty = Nothing
  Nothing <|> r = r
  l       <|> _ = l
```

And it turns out that only the latter satisfies the `Alternative` laws. The `Monoid` instance fails less badly than `(,)` 's; it *does* obey the laws with respect to `<*>` , although almost by accident—it comes form the behavior of the only instance of `Monoid` for functions, which (as mentioned above), lifts functions that return monoids into the reader applicative functor. If you work it out (it's all very mechanical), you'll find that right distributivity and right absorption for `<*>` all hold for both the `Alternative` and `Monoid` instances, as does left absorption for `fmap` . And left distributivity for `fmap` does hold for the `Alternative` instance, as follows:

```
f <$> (Nothing <|> b)
  = f <$> b                          by the definition of (<|>)
  = Nothing <|> (f <$> b)            by the definition of (<|>)
  = (f <$> Nothing) <|> (f <$> b)    by the definition of (<$>)

f <$> (Just a <|> b)
  = f <$> Just a                     by the definition of (<|>)
  = Just (f a)                       by the definition of (<$>)
```

```
  = Just (f a) <|> (f <$> b)         by the definition of (<|>)
  = (f <$> Just a) <|> (f <$> b)     by the definition of (<$>)
```

However, it fails for the `Monoid` instance; writing `(<>)` for `mappend`, we have:

`(<> Sum 1) <$> (Just (Sum 0) <> Just (Sum 0)) = Just (Sum 1)`

`((<> Sum 1) <$> Just (Sum 0)) <> ((<> Sum 1) <$> Just (Sum 0)) = Just (Sum 2)`

Now, there is one caveat to this example. If you only require that `Alternative`s be compatibility with `<*>`, and not with `<$>`, then `Maybe` is fine. Edward Kmett's slides, mentioned above, don't make reference to `<$>`, but I think it seems reasonable to require laws with respect to it as well; nevertheless, I can't find anything to back me up on this.

Thus, we can conclude that being an `Alternative` is a *stronger* requirement than being a `Monoid`, and so it requires a different class. The purest example of this would be a type with an inner-type agnostic `Monoid` instance and an `Applicative` instance which were incompatible with each other; however, there aren't any such types in the base package, and I can't think of any. (It's possible none exist, although I'd be surprised.) Nevertheless, these inner-type gnostic examples demonstrate why the two type classes must be different.

> What's the point of the `MonadPlus` type class?

`MonadPlus`, like `Alternative`, is a strengthening of `Monoid`, but with respect to `Monad` instead of `Applicative`. According to Edward Kmett in his answer to the question "Distinction between typeclasses `MonadPlus`, `Alternative`, and `Monoid`?", `MonadPlus` is *also* stronger than `Alternative`: the law `empty <*> a`, for instance, doesn't imply that `empty >>= f`. AndrewC provides two examples of this: `Maybe` and its dual. The issue is complicated by the fact that there are two potential sets of laws for `MonadPlus`. It is universally agreed that `MonadPlus` is supposed to form a monoid with `mplus` and `mempty`, and it's supposed to satisfy the *left zero* law, `mempty >>= f = mempty`. However, some `MonadPlus`ses satisfy *left distribution*, `mplus a b >>= f = mplus (a >>= f) (b >>= f)`; and others satisfy *left catch*, `mplus (return a) b = return a`. (Note that left zero/distribution for `MonadPlus` are analogous to right distributivity/absorption for `Alternative`; `(<*>)` is more analogous to `(=<<)` than `(>>=)`.) Left distribution is probably "better," so any `MonadPlus` instance which satisfies left catch, such as `Maybe`, is an `Alternative` but not the first kind of `MonadPlus`. And since left catch relies on ordering, you can imagine a newtype wrapper for `Maybe` whose `Alternative` instance is *right*-biased instead of *left*-biased: `a <|> Just b = Just b`. This will satisfy neither left distribution nor left catch, but will be a perfectly valid `Alternative`.

However, since any type which *is* a `MonadPlus` ought to have its instance coincide with its `Alternative` instance (I believe this is required in the same way that it is required that `ap` and `(<*>)` are equal for `Monad`s that are `Applicative`s), you could imagine defining the `MonadPlus` class instead as

```
class (Monad m, Alternative m) => MonadPlus' m
```

The class doesn't need to declare new functions; it's just a promise about the laws obeyed by `empty` and `(<|>)` for the given type. This design technique isn't used in the Haskell standard libraries, but is used in some more mathematically-minded packages for similar purposes; for instance, the lattices package uses it to express the idea that a lattice is just a join semilattice and a meet semilattice over the same type which are linked by absorption laws.

The reason you can't do the same for `Alternative`, even if you wanted to guarantee that `Alternative` and `Monoid` always coincided, is because of the kind mismatch. The desired class declaration would have the form

```
class (Applicative f, forall a. Monoid (f a)) => Alternative''' f
```

but (as mentioned far above) not even GHC Haskell supports quantified constraints.

Also, note that having `Alternative` as be a superclass of `MonadPlus` would require `Applicative` being a superclass of `Monad`, so good luck getting that to happen. If you run into that problem, there's always the `WrappedMonad` newtype, which turns any `Monad` into an `Applicative` in the obvious way; there's an `instance MonadPlus m => Alternative (WrappedMonad m) where ...` which does exactly what you'd expect.

edited Oct 29 '12 at 23:26 answered Oct 26 '12 at 6:03
Antal Spector-Zabusky
22.9k ●4 ●45 ●103

Thank you, this helps a ton. I'm still stuck on point 3 though -- what value Alternative has over Monoid ... will have to think more about that one. – Matt Fenwick Oct 26 '12 at 12:42

1 @MattFenwick I'm in the same boat. `Monoid` I understand, but I'm not sure why we need a seperate and basically identical `Alternative` / `MonadPlus`. My big guess is hitory, but sometimes they have different semantics (which can be useful, but isn't a great argument for having both, IMHO). – singpolyma Oct 26 '12 at 16:16

I love hysterical raisins. – AndrewC Oct 29 '12 at 3:12

1 For the record - I was the one who fought the windmill about leaving `<>` only in `Data.Semigroup`. Edward folded early. – Yitz Nov 3 '13 at 19:10

2 from 2015: Alternative is a superclass of MonadPlus hackage.haskell.org/package/base-4.8.0.0/docs/… – sam boosalis May 14 '15 at 21:37

```
import Data.Monoid
import Control.Applicative
```

Let's trace through an example of how Monoid and Alternative interact with the `Maybe` functor and the `ZipList` functor, but let's start from scratch, partly to get all the definitions fresh in our minds, partly to stop from switching tabs to bits of hackage all the time, but mainly so I can run this past ghci to correct my typos!

```
(<>) :: Monoid a => a -> a -> a
(<>) = mappend -- I'll be using <> freely instead of `mappend`.
```

Here's the Maybe clone:

```
data Perhaps a = Yes a | No  deriving (Eq, Show)

instance Functor Perhaps where
   fmap f (Yes a) = Yes (f a)
   fmap f No      = No

instance Applicative Perhaps where
   pure a = Yes a
   No    <*> _     = No
   _     <*> No    = No
   Yes f <*> Yes x = Yes (f x)
```

and now ZipList:

```
data Zip a = Zip [a]  deriving (Eq,Show)

instance Functor Zip where
   fmap f (Zip xs) = Zip (map f xs)

instance Applicative Zip where
   Zip fs <*> Zip xs = Zip (zipWith id fs xs)   -- zip them up, applying the fs to the xs
   pure a = Zip (repeat a)   -- infinite so that when you zip with something, lengths
don't change
```

Structure 1: combining elements: Monoid

Maybe clone

First let's look at `Perhaps String`. There are two ways of combining them. Firstly concatenation

```
(<++>) :: Perhaps String -> Perhaps String -> Perhaps String
Yes xs <++> Yes ys = Yes (xs ++ ys)
Yes xs <++> No     = Yes xs
No     <++> Yes ys = Yes ys
No     <++> No     = No
```

Concatenation works inherently at the String level, not really the Perhaps level, by treating `No` as if it were `Yes []`. It's equal to `liftA2 (++)`. It's sensible and useful, but maybe we could generalise from just using `++` to using any way of combining - any Monoid then!

```
(<++>) :: Monoid a => Perhaps a -> Perhaps a -> Perhaps a
Yes xs <++> Yes ys = Yes (xs `mappend` ys)
Yes xs <++> No     = Yes xs
No     <++> Yes ys = Yes ys
No     <++> No     = No
```

This monoid structure for `Perhaps` tries to work as much as possible at the `a` level. Notice the `Monoid a` constraint, telling us we're using structure from the `a` level. This isn't an Alternative structure, it's a derived (lifted) Monoid structure.

```
instance Monoid a => Monoid (Perhaps a) where
   mappend = (<++>)
   mempty = No
```

Here I used the structure of the data a to add structure to the whole thing. If I were combining `Set`s, I'd be able to add an `Ord a` context instead.

ZipList clone

So how should we combine *elements* with a zipList? What should these zip to if we're combining them?

```
   Zip ["HELLO","MUM","HOW","ARE","YOU?"]
<> Zip ["this", "is", "fun"]
=  Zip ["HELLO" ? "this",   "MUM" ? "is",   "HOW" ? "fun"]

mempty = ["","","","",..]   -- sensible zero element for zipping with ?
```

But what should we use for `?`. I say the only sensible choice here is `++`. Actually, for lists, `(<>) = (++)`

```
   Zip [Just 1,  Nothing, Just 3, Just 4]
<> Zip [Just 40, Just 70, Nothing]
 =  Zip [Just 1 ? Just 40,    Nothing ? Just 70,    Just 3 ? Nothing]

mempty = [Nothing, Nothing, Nothing, .....]  -- sensible zero element
```

But what can we use for `?` I say that we're meant to be combining elements, so we should use the element-combining operator from Monoid again: `<>`.

```
instance Monoid a => Monoid (Zip a) where
   Zip as `mappend` Zip bs = Zip (zipWith (<>) as bs) -- zipWith the internal mappend
   mempty = Zip (repeat mempty)  -- repeat the internal mempty
```

This is the only sensible way of combining the elements using a zip - so it's the only sensible monoid instance.

Interestingly, that doesn't work for the Maybe example above, because Haskell doesn't know how to combine `Int`s - should it use `+` or `*`? To get a Monoid instance on numerical data, you wrap them in `Sum` or `Product` to tell it which monoid to use.

```
Zip [Just (Sum 1),   Nothing,       Just (Sum 3), Just (Sum 4)] <>
Zip [Just (Sum 40),  Just (Sum 70), Nothing]
= Zip [Just (Sum 41),Just (Sum 70), Just (Sum 3)]

   Zip [Product 5,Product 10,Product 15]
<> Zip [Product 3, Product 4]
 =  Zip [Product 15,Product 40]
```

Key point

Notice the fact that the type in a Monoid has kind `*` is exactly what allows us to put the `Monoid a` context here - we could also add `Eq a` or `Ord a`. In a Monoid, the raw elements matter. A Monoid instance is *designed* to let you manipulate and combine the data inside the structure.

Structure 2: higher-level choice: Alternative

A choice operator is similar, but also different.

Maybe clone

```
(<||>) :: Perhaps String -> Perhaps String -> Perhaps String
Yes xs <||> Yes ys = Yes xs   -- if we can have both, choose the left one
Yes xs <||> No     = Yes xs
No     <||> Yes ys = Yes ys
No     <||> No     = No
```

Here there's *no concatenation* - we didn't use `++` at all - this combination works purely at the `Perhaps` level, so let's change the type signature to

```
(<||>) :: Perhaps a -> Perhaps a -> Perhaps a
Yes xs <||> Yes ys = Yes xs   -- if we can have both, choose the left one
Yes xs <||> No     = Yes xs
No     <||> Yes ys = Yes ys
No     <||> No     = No
```

Notice there's no constraint - we're not using the structure from the `a` level, just structure at the `Perhaps` level. This is an Alternative structure.

```
instance Alternative Perhaps where
   (<|>) = (<||>)
   empty = No
```

ZipList clone

How should we choose between two ziplists?

```
Zip [1,3,4] <|> Zip [10,20,30,40] = ????
```

It would be very tempting to use `<|>` on the elements, but we can't because the type of the elements isn't available to us. Let's start with the `empty`. It can't use an element because we don't know the type of the elements when defining an Alternative, so it has to be `Zip []`. We need it to be a left (and preferably right) identity for `<|>`, so

```
Zip [] <|> Zip ys = Zip ys
Zip xs <|> Zip [] = Zip xs
```

There are two sensible choices for `Zip [1,3,4] <|> Zip [10,20,30,40]`:

1. `Zip [1,3,4]` because it's first - consistent with Maybe
2. `Zip [10,20,30,40]` because it's longest - consistent with `Zip []` being discarded

Well that's easy to decide: since `pure x = Zip (repeat x)`, both lists might be infinite, so comparing them for length might never terminate, so it has to be pick the first one. Thus the only sensible Alternative instance is:

```
instance Alternative Zip where
   empty = Zip []
   Zip [] <|> x = x
   Zip xs <|> _ = Zip xs
```

This is the only sensible Alternative we could have defined. Notice how different it is from the Monoid instance, because we couldn't mess with the elements, we couldn't even look at them.

Key Point

Notice that because `Alternative` takes a constructor of kind `* -> *` there is *no possible way* to add an `Ord a` or `Eq a` or `Monoid a` context. An Alternative is **not allowed** to use any information about the data inside the structure. You cannot, no matter how much you would like to, *do* anything to the data, except possibly throw it away.

Key point: What's the difference between Alternative and Monoid?

Not a lot - they're both monoids, but to summarise the last two sections:

`Monoid *` instances make it possible to combine internal data. `Alternative (* -> *)` instances make it impossible. Monoid provides flexibility, Alternative provides guarantees. The kinds `*` and `(* -> *)` are the main drivers of this difference. Having them both allows you to use both sorts of operations.

This is the right thing, and our two flavours are both appropriate. The Monoid instance for `Perhaps String` represents putting together all characters, the Alternative instance represents a choice between Strings.

There is nothing wrong with the Monoid instance for Maybe - it's doing its job, *combining* data. There's nothing wrong with the Alternative instance for Maybe - it's doing its job, *choosing* between things.

The Monoid instance for Zip combines its elements. The Alternative instance for Zip is forced to choose one of the lists - the first non-empty one.

It's good to be able to do both.

What's the Applicative context any use for?

There's some interaction between choosing and applying. See Antal S-Z's laws stated in his question or in the middle of his answer here.

From a practical point of view, it's useful because Alternative is something that is used for some Applicative Functors to choose. The functionality was being used for Applicatives, and so a general interface class was invented. Applicative Functors are good for representing computations that produce values (IO, Parser, Input UI element,...) and some of them have to handle failure - Alternative is needed.

Why does Alternative have `empty`?

> why does Alternative need an empty method/member? I may be wrong, but it seems to not be used at all ... at least in the code I could find. And it seems not to fit with the theme of the class -- if I have two things, and need to pick one, what do I need an 'empty' for?

That's like asking why addition needs a 0 - if you want to add stuff, what's the point in having something that doesn't add anything? The answer is that 0 is the crucual pivotal number around which everything revolves in addition, just like 1 is crucial for multiplication, `[]` is crucial for lists (and `y=e^x` is crucial for calculus). In practical terms, you use these do-nothing elements to start your building:

```
sum = foldr (+) 0
concat = foldr (++) []
msum = foldr (`mappend`) mempty          -- any Monoid
whichEverWorksFirst = foldr (<|>) empty  -- any Alternative
```

Can't we replace MonadPlus with Monad+Alternative?

> what's the point of the MonadPlus type class? Can't I unlock all of its goodness by just using something as both a Monad and Alternative? Why not just ditch it? (I'm sure I'm wrong, but I don't have any counterexamples)

You're not wrong, there aren't any counterexamples!

Your interesting question has got Antal S-Z, Petr Pudlák and I delved into what the relationship between MonadPlus and Applicative really is. The answer, here and here is that anything that's a `MonadPlus` (in the left distribution sense - follow links for details) is also an `Alternative`, but not the other way around.

This means that if you make an instance of Monad and MonadPlus, it satisfies the conditions for Applicative and Alternative anyway. This means if you follow the rules for MonadPlus (with left dist), you may as well have made your Monad an Applicative and used Alternative.

If we remove the MonadPlus class, though, we remove a sensible place for the rules to be documented, and you lose the ability to specify that something's Alternative without being MonadPlus (which technically we ought to have done for Maybe). These are theoretical reasons. The practical reason is that it would break existing code. (Which is also why neither Applicative nor Functor are superclasses of Monad.)

Aren't Alternative and Monoid the same? Aren't Alternative and Monoid completely different?

> the 'pedia says that "the Alternative type class is for Applicative functors which also have a monoid structure." I don't get this -- doesn't Alternative mean something totally different from Monoid? i.e. I understood the point of the Alternative type class as picking between two

> things, whereas I understood Monoids as being about combining things.

Monoid and Alternative are two ways of getting one object from two in a sensible way. Maths doesn't care whether you're choosing, combining, mixing or blowing up your data, which is why Alternative was referred to as a Monoid for Applicative. You seem to be at home with that concept now, but you now say

> for types that have both an Alternative and a Monoid instance, the instances are intended to be the same

I disagree with this, and I think my Maybe and ZipList examples are carefully explained as to why they're different. If anything, I think it should be rare that they're the same. I can only think of one example, plain lists, where this is appropriate. That's because lists are a fundamental example of a monoid with `++`, but also lists are used in some contexts as an indeterminate choice of elements, so `<|>` should also be `++`.

answered Nov 1 '12 at 10:03
AndrewC
24.8k ●4 ●54 ●95

Summary

> We need to define (instances that provide the same operations as) Monoid instances for some applicative functors, that genuinely combine at the applicative functor level, and not just lifting lower level monoids. The example error below from `litvar = liftA2 mappend literal variable` shows that `<|>` cannot in general be defined as `liftA2 mappend`; `<|>` works in this case by combining parsers, not their data.
>
> If we used Monoid directly, we'd need language extensions to define the instances. `Alternative` is higher kinded so you can make these instances without requiring language extensions.

Example: Parsers

Let's imagine we're parsing some declarations, so we import everything we're going to need

```
import Text.Parsec
import Text.Parsec.String
import Control.Applicative ((<$>),(<*>),liftA2,empty)
import Data.Monoid
import Data.Char
```

and think about how we'll parse a type. We choose simplistic:

```
data Type = Literal String | Variable String  deriving Show
examples = [Literal "Int",Variable "a"]
```

Now let's write a parser for literal types:

```
literal :: Parser Type
literal = fmap Literal $ (:) <$> upper <*> many alphaNum
```

Meaning: parse an `upper` case character, then `many alphaNum` eric characters, combine the results into a single String with the pure function `(:)`. Afterwards, apply the pure function `Literal` to turn those `String` s into `Type` s. We'll parse variable types exactly the same way, except for starting with a `lower` case letter:

```
variable :: Parser Type
variable = fmap Variable $ (:) <$> lower <*> many alphaNum
```

That's great, and `parseTest literal "Bool" == Literal "Bool"` exactly as we'd hoped.

Question 3a: If it's to combine applicative's effects with Monoid's behavior, why not just `liftA2 mappend`

Edit:Oops - forgot to actually use `<|>`!
Now let's combine these two parsers using Alternative:

```
types :: Parser Type
types = literal <|> variable
```

This can parse any Type: `parseTest types "Int" == Literal "Bool"` and `parseTest types "a" == Variable "a"`. This combines the two *parsers*, not the two *values*. That's the sense in which it works at the Applicative Functor level rather than the data level.

However, if we try:

```
litvar = liftA2 mappend literal variable
```

that would be asking the compiler to combine the two *values* that they generate, at the data level. We get

```
No instance for (Monoid Type)
  arising from a use of `mappend'
Possible fix: add an instance declaration for (Monoid Type)
```

```
In the first argument of `liftA2', namely `mappend'
In the expression: liftA2 mappend literal variable
In an equation for `litvar':
    litvar = liftA2 mappend literal variable
```

So we found out the first thing; the Alternative class does something genuinely different to `liftA2 mappend`, becuase it combines objects at a different level - it combines the parsers, not the parsed data. If you like to think of it this way, it's combination at the genuinely higher-kind level, not merely a lift. I don't like saying it that way, because `Parser Type` has kind `*`, but it is true to say we're combining the `Parser` s, not the `Type` s.

(Even for types with a Monoid instance, `liftA2 mappend` won't give you the same parser as `<|>`. If you try it on `Parser String` you'll get `liftA2 mappend` which parses one after the other then concatenates, versus `<|>` which will try the first parser and default to the second if it failed.)

Question 3b: In what way does Alternative's `<|> :: f a -> f a -> f a` differ from Monoid's `mappend :: b -> b -> b`?

Firstly, you're right to note that it doesn't provide new functionality over a Monoid instance.

Secondly, however, there's an issue with using Monoid directly: Let's try to use `mappend` on parsers, at the same time as showing it's the same structure as `Alternative`:

```
instance Monoid (Parser a) where
    mempty = empty
    mappend = (<|>)
```

Oops! We get

```
Illegal instance declaration for `Monoid (Parser a)'
  (All instance types must be of the form (T t1 ... tn)
   where T is not a synonym.
   Use -XTypeSynonymInstances if you want to disable this.)
In the instance declaration for `Monoid (Parser a)'
```

So if you have an applicative functor `f`, the `Alternative` instance shows that `f a` is a monoid, but you could only declare that as a `Monoid` with a language extension.

Once we add `{-# LANGUAGE TypeSynonymInstances #-}` at the top of the file, we're fine and can define

```
typeParser = literal `mappend` variable
```

and to our delight, it works: `parseTest typeParser "Yes" == Literal "Yes"` and `parseTest typeParser "a" == Literal "a"`.

Even if you don't have any synonyms (`Parser` and `String` are synonyms, so they're out), you'll still need `{-# LANGUAGE FlexibleInstances #-}` to define an instance like this one:

```
data MyMaybe a = MyJust a | MyNothing deriving Show
instance Monoid (MyMaybe Int) where
   mempty = MyNothing
   mappend MyNothing x = x
   mappend x MyNothing = x
   mappend (MyJust a) (MyJust b) = MyJust (a + b)
```

(The monoid instance for Maybe gets around this by lifting the underlying monoid.)

Making a standard library unnecessarily dependent on language extensions is clearly undesirable.

So there you have it. Alternative is just Monoid for Applicative Functors (and isn't just a lift of a Monoid). It needs the higher-kinded type `f a -> f a -> f a` so you can define one without language extensions.

Your other Questions, for completeness:

1. Why does Alternative need an empty method/member?
   Because having an identity for an operation is sometimes useful. For example, you can define `anyA = foldr (<|>) empty` without using tedious edge cases.

2. what's the point of the MonadPlus type class? Can't I unlock all of its goodness by just using something as both a Monad and Alternative? No. I refer you back to the question you linked to:

> Moreover, even if Applicative was a superclass of Monad, you'd wind up needing the MonadPlus class anyways, because obeying `empty <*> m = empty` isn't strictly enough to prove that `empty >>= f = empty`.

....and I've come up with an example: Maybe. I explain in detail, with proof in this answer to Antal's question. For the purposes of this answer, it's worth noting that I was able to use >>= to make the MonadPlus instance that broke the Alternative laws.

Monoid structure is useful. Alternative is the best way of providing it for Applicative Functors.

edited Oct 29 '12 at 22:48 answered Oct 29 '12 at 2:30

AndrewC
24.8k ●4 ●54 ●95

1 @MattFenwick These aren't silly questions. Alternative *is* the same as a monoid instance for Parser, yes. I show that (a) `(<|>)` is not equal to `liftA2 mappend`, addressing your question why we don't just do that, and (b) that you'd need a language extension to define that monoid instance, which is why there's a separate class, addressing your main question. – AndrewC Oct 29 '12 at 14:27

@MattFenwick So sorry - I realise now I never actually used <|> so there really weren't any <|> examples to contrast with how using Monoid didn't work! I've changed mainly the start of section 3a but also a bit of 3b. – AndrewC Oct 29 '12 at 20:47

@MattFenwick hopefully now I actually *included* the examples, it should make more sense! – AndrewC Oct 29 '12 at 20:51

I won't cover MonadPlus because there is disagreement about its laws.

After trying and failing to find any meaningful examples in which the structure of an Applicative leads naturally to an Alternative instance that disagrees with its Monoid instance*, I finally came up with this:

Alternative's laws are more strict than Monoid's, because the result *cannot* depend on the inner type. This excludes a large number of Monoid instances from being Alternatives. These datatypes allow partial (meaning that they only work for some inner types) Monoid instances which are forbidden by the extra 'structure' of the `* -> *` kind. Examples:

> the standard Maybe instance for Monoid assumes that the inner type is Monoid => not an Alternative
>
> ZipLists, tuples, and functions can all be made Monoids, *if* their inner types are Monoids => not Alternatives
>
> sequences that have at least one element -- cannot be Alternatives because there's no `empty`:
>
> ```
> data Seq a
>     = End a
>     | Cons a (Seq a)
>   deriving (Show, Eq, Ord)
> ```

On the other hand, some data types cannot be made Alternatives because they're `*`-kinded:

> unit -- `()`
>
> `Ordering`
>
> numbers, booleans

My inferred conclusion: **for types that have both an Alternative and a Monoid instance, the instances are intended to be the same.** See also this answer.

---

excluding Maybe, which I argue doesn't count because its standard instance should not require Monoid for the inner type, in which case it would be identical to Alternative

edited Dec 3 '12 at 21:08

community wiki
4 revs
Matt Fenwick

> I understood the point of the Alternative type class as picking between two things, whereas I understood Monoids as being about combining things.

If you think about this for a moment, they are the same.

The `+` combines things (usually numbers), and it's type signature is `Int -> Int -> Int` (or whatever).

The `<|>` operator selects between alternatives, and it's type signature is also the same: take two matching things and return a combined thing.

answered Oct 26 '12 at 12:00
MathematicalOrchid
28.5k ●9 ●64 ●132

<div><img src="/posts/13080606/ivc/82c9" class="dno" alt="" width="0" height="0"></div>

<div id="noscript-warning">Stack Overflow works best with JavaScript enabled<img src="http://pixel.quantserve.com/pixel/p-c1rF4kxgLUzNc.gif" alt="" class="dno"></div>