Distinction between typeclasses MonadPlus, Alternative, and Monoid?

The standard-library Haskell typeclasses `MonadPlus`, `Alternative`, and `Monoid` each provide two methods with essentially the same semantics:

- An empty value: `mzero`, `empty`, or `mempty`.
- An operator `a -> a -> a` that joins values in the typeclass together: `mplus`, `<|>`, or `mappend`.

All three specify these laws to which instances should adhere:

```
mempty `mappend` x = x
x `mappend` mempty = x
```

Thus, it seems the three typeclasses are all providing the *same* methods.

(`Alternative` also provides `some` and `many`, but their default definitions are usually sufficient, and so they're not too important in terms of this question.)

So, my query is: why have these three extremely similar classes? Is there any real difference between them, besides their differing superclass constraints?

haskell functional-programming typeclass applicative monoids

edited Apr 16 '12 at 2:06
Xenon
2,415 ●9 ●30

asked Apr 16 '12 at 1:53
00Davo
622 ●8 ●13

That's a good question. In particular, `Applicative` and `MonadPlus` seem to be *exactly* the same (modulo superclass constraints). – Peter Apr 16 '12 at 2:22

1 There's also `ArrowZero` and `ArrowPlus` for arrows. My bet: to make type signatures cleaner (which makes differing superclass constraints *the* real difference). – Cat Plus Plus Apr 16 '12 at 2:28

1 @CatPlusPlus: well, `ArrowZero` and `ArrowPlus` have kind `* -> * -> *`, which means you can pass them in for the arrow type once for a function that needs to use them for a multitude of types, to use a `Monoid` you'd have to require an instance of `Monoid` for each particular instantiation, and you'd have no guarantee they were handled in a similar way, the instances could be unrelated! – Edward KMETT Apr 16 '12 at 2:52 ✎

1 Answer

`MonadPlus` and `Monoid` serve different purposes.

A `Monoid` is parameterized over a type of kind `*`.

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

and so it can be instantiated for almost any type for which there is an obvious operator that is associative and which has a unit.

However, `MonadPlus` not only specifies that you have a monoidal structure, but also that that structure is related to how the `Monad` works, *and* that that structure doesn't care about the value contained in the monad, this is (in part) indicated by the fact that `MonadPlus` takes an argument of kind `* -> *`.

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

In addition to the monoid laws, we have two potential sets of laws we can apply to `MonadPlus`.

Sadly, the community disagrees as to what they should be.

At the least we know

```
mzero >>= k = mzero
```

but there are two other competing extensions, the left (sic) distribution law

```
mplus a b >>= k = mplus (a >>= k) (b >>= k)
```

and the left catch law

```
mplus (return a) b = return a
```

So any instance of `MonadPlus` should satisfy one or both of these additional laws.

So what about `Alternative`?

`Applicative` was defined after `Monad`, and logically belongs as a superclass of `Monad`, but largely due to the different pressures on the designers back in Haskell 98, even `Functor` wasn't a superclass of `Monad` until 2015. Now we finally have `Applicative` as a superclass of `Monad` in GHC (if not yet in a language standard.)

Effectively, `Alternative` is to `Applicative` what `MonadPlus` is to `Monad`.

For these we'd get

```
empty <*> m = empty
```

analogously to what we have with `MonadPlus` and there exist similar distributive and catch properties, at least one of which you should satisfy.

Unfortunately, even `empty <*> m = empty` law is too strong a claim. It doesn't hold for Backwards, for instance!

When we look at MonadPlus, the empty >>= f = empty law is nearly forced on us. The empty construction can't have any 'a's in it to call the function `f` with anyways.

However, since `Applicative` is *not* a superclass of `Monad` and `Alternative` is *not* a superclass of `MonadPlus`, we wind up defining both instances separately.

Moreover, even if `Applicative` was a superclass of `Monad`, you'd wind up needing the `MonadPlus` class anyways, because even if we did obey

```
empty <*> m = empty
```

that isn't strictly enough to prove that

```
empty >>= f = empty
```

So claiming that something is a `MonadPlus` is stronger than claiming it is `Alternative`.

Now, by convention, the `MonadPlus` and `Alternative` for a given type should agree, but the `Monoid` may be *completely* different.

For instance the `MonadPlus` and `Alternative` for `Maybe` do the obvious thing:

```
instance MonadPlus Maybe where
    mzero = Nothing
    mplus (Just a) _  = Just a
    mplus _        mb = mb
```

but the `Monoid` instance lifts a semigroup into a `Monoid`. Sadly because there did not exist a `Semigroup` class at the time in Haskell 98, it does so by requring a `Monoid`, but not using its unit. ಠ_ಠ

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    mappend (Just a) (Just b) = Just (mappend a b)
    mappend Nothing x = x
    mappend x Nothing = x
    mappend Nothing Nothing = Nothing
```

TL;DR `MonadPlus` is a stronger claim than `Alternative`, which in turn is a stronger claim than `Monoid`, and while the `MonadPlus` and `Alternative` instances for a type should be related, the `Monoid` may be (and sometimes is) something completely different.

edited Jul 3 '15 at 7:07

answered Apr 16 '12 at 2:36
Edward KMETT
22.5k ●2 ●65 ●94

2 Excellent answer, however the last definition seems to be wrong, it doesn't satisfy `` mempty `mappend` x ≡ x ``. – Vitus Apr 16 '12 at 6:44

2 Great answer. Does anyone know of a (commonly used) type that has *different* `MonadPlus` and `Alternative` implementations? – Peter Apr 16 '12 at 11:05

6 @EdwardKmett: This answer seems to imply that there could be a `Monad` which is an `Alternative` but not a `MonadPlus`. I asked a question about finding a specific example of this; if you know of one, I'd love

to see it. – Antal Spector-Zabusky Oct 29 '12 at 13:36

2 Can you explain the left catch law for monadplus? It is apparently violated by []; should [] really ignore its second argument if its first is non-empty? – ben w Feb 13 '13 at 2:18

4 @benw left distribution is arguably the more sensible law, but it doesn't hold for some instances. left catch is an alternate law that those other instances tend to support, but which aren't supported by most of the others. Consequently, we really have 2 largely unrelated sets of laws being implemented by different instances, so `MonadPlus` is really two classes disguised as one because most people don't care. – Edward KMETT Feb 18 '13 at 22:01