beyond Typeclassopedia

There are quite a few type classes, that are completely fine the way they are defined in the base library. These are mostly part of the famous Typeclassopedia.

- `Semigroup` and `Monoid`
- `Functor`, `Applicative` and `Monad`
- `Foldable` and `Traversable`

This article is mainly an overview and doesn't try to explain each type class. I also want to point out some opinionated critique.

Typeclassopedia

The Typeclassopedia explains a few more type classes.

`Pointed` is a superclass of `Applicative` with just `pure`. I am not against this separation, but can't think of an example of `Pointed`, that isn't also `Applicative`.

`MonadFix` is not as well known as some other type classes, but is actually quite useful. The `RecursiveDo` extension adds some syntactic sugar to `MonadFix`es. This allows recursive bindings in *do*-notation.

CAUTION	I am only covering the "left" side of the Typeclassopedia. The `Category` and `Arrow` type classes are not discussed here. I am also ignoring `Comonad` for now.

To conclude the Typeclassopedia, let's look at `Alternative`.

`Alternative` is a monoid

NOTE	`MonadPlus m` is equivalent to `(Alternative m, Monad m)`. It's an unnecessary alias, but it doesn't do any harm.

`Alternative` can actually be expressed with the other already introduced type classes.

CAUTION	The `QuantifiedConstraints` language extension is required.

```
class (Applicative f, forall x. Monoid (f x)) => Alternative f where

  empty :: f a
  empty = mempty

  (<|>) :: f a -> f a -> f a
```

```
(<|>) = (<>)
```

NOTE The `Applicative f` superclass is not necessary at all, but otherwise it would be an alias for `forall x. Monoid (f x)`.

The only problem is, that base introduces some `Monoid` instances, that don't work with this definition. `Maybe` for example uses a superclass constraint.

```
class Semigroup a => Monoid (Maybe a) where
  -- ...
```

The proposed `Alternative` doesn't work, because `forall a. Semigroup a` (from `Maybe`) is a stronger constraint than `forall x.` (from `Alternative`). I am not quite sure, whether this change can actually break `Alternative`s laws. The `Maybe` example will be caught by the compiler at least.

TIP The solution would be to use compatible instances. The currently used instances can still be made available with `newtype`s.

The `newtype Alt` already exists for the other direction `Alternative f ⇒ Monoid (f x)`.

The current situation isn't bad though, considering that `newtype`s are annoying to use. Maybe idris made the right choice with named implementations (multiple named instances in Haskell terms).

Another interesting change would be the separation of `(<|>)` and `empty`, similar to the separation of `Semigroup` and `Monoid` (or `Pointed` and `Applicative`).

TIP The structure of `Applicative` and `Alternative` is similar to a semiring.

`Applicative` / `Alternative`	**natural numbers**	**boolean algebra**	**algebra of types**
`empty`	0	*false*	`Void`
`pure ()`	1	*true*	`()`
`<|>`	+	*or*	`Either`
`<*>`	×	*and*	`(,)`

The Typeclassopedia is actually quite old by now and there are more additions in base.

base library

Let's start with `MonadFail`. This type class was a stupid idea to allow pattern matching in *do*-notation. Whenever writing actual code you should use a *case*-block. Patterns that *always* match are fine, but other patterns should just give a warning like "patterns are non-exhaustive".

`Bifunctor`, `Bifoldable` and `Bitraversable` are nice to have. Generalizations for `Trifunctor`, `Quadrofunctor`, etc. are missing though. I'm not sure how those type classes would be implemented

without boilerplate.

`MonadIO` is a type class for all monads, that use `IO` as the base monad. This is a useful type class, but it can be completely replaced with `MonadBase` from transformers-base. Additionaly `MonadBaseControl` from monad-control could be added to base.

NOTE	I am in favor of completely removing `MonadIO`, which some people might find a bit harsh.

`MonadTrans` from transformers and `MonadTransControl` from monad-control also fit into the same category, but for monad transformers.

Base introduces `Contravariant` functors. Analogously to `Applicative` and `Alternative`, the contravariant library defines `Divisible` and `Decidable`. These two type class can be quite useful, but I haven't explored this direction any further.

NOTE	This didn't cover *all* type classes from base, but only the ones similar to the Typeclassopedia.

Takeaways

- Classes like `Monoid` will often have multiple lawful instances, but Haskell requires us to use a `newtype` for each implementation. Instances should be chosen wisely.
- `Alternative` and `Applicative` can be thought of as monoids.
- `MonadFail` is a *fail*.
- `MonadBase` makes `MonadIO` unnecessary.
- `MonadBaseControl` and `MonadTransControl` would be a nice addition to base, including `MonadTrans` and `MonadBase`.
- Other type classes like `Num`, `IsString`, `IsList` or other stock classes deserve their own discussion.