AccumT's MonadAccum instance

Transformers provides a monad transformer AccumT. Mtl provides a type class MonadAccum. There should be an `instance (Monoid w, Monad m) ⇒ MonadAccum w (AccumT w m)`, but this is not the case.

AccumT and MonadAccum

AccumT is a monad transformer that's similar to StateT and WriterT.

AccumT *definition*

```
newtype AccumT w m a = AccumT { runAccumT :: w -> m (a, w) }
```

> **NOTE** AccumT is actually isomorphic to StateT.

MonadState allows you to modify a state (s) similar to a variable in imperative languages. MonadAccum only appends to that variable (w) via (<>) from Monoid. MonadAccum is similar to MonadWriter in that way, but also allows you to look at what has already been accumulated.

Minimal MonadAccum *definition*

```
class (Monoid w, Monad m) => MonadAccum w m | m -> w where
  add :: w -> m () -- Append to variable.
  look :: m w -- Look at current state of variable.
```

The current instance

AccumT's MonadAccum *instance currently*

```
instance Monoid w => MonadAccum w (AccumT w Identity) where
  add w = AccumT $ \ _ -> return ((), w)
  look = AccumT $ \ w -> return (w, mempty)
```

> **NOTE** The Functor, Applicative and Monad instances of AccumT are fine. Applicative and Monad use a Monoid w constraint and they look like a combination of StateT and WriterT, but nothing really special is going on.

Here comes the part that annoys me...

Why is the MonadAccum instance restricted to AccumT w Identity???

We can easily implement this instance using the exact same methods, but without restricting the

base monad. Therefore I propose to change this instance in the following way.

`AccumT`'s `MonadAccum` *instance proposed by me*

```
instance (Monoid w, Monad m) => MonadAccum w (AccumT w m) where
  add w = AccumT $ \ _ -> return ((), w)
  look = AccumT $ \ w -> return (w, mempty)
```

Almost every other transformer instance uses this pattern.

```
instance Monad m => MonadExample (ExampleT m)
```

We should also use that pattern here.

NOTE	There are two exceptions in transformers, that don't follow this rule. `ContT` doesn't need the `Monad m` constraint and `SelectT` doesn't work with any base monad that carries monadic state.

The laws of `MonadAccum`

I'm going to check the laws of `MonadAccum` to make sure I don't break anything.

IMPORTANT	If you find any mistakes, please let me know and I'll fix them.

`MonadAccum` *laws to prove*

1. `look *> look = look`
2. `add mempty = pure ()`
3. `add x *> add y = add (x <> y)`
4. `add x *> look = look >>= w → add x $> w <> x`

To help with the proof we can use the laws of `Monoid` and `Monad`, because that's how we want to constrain the `MonadAccum` instance.

`Monoid` *laws*

Left identity	`mempty <> x = x`
Right identity	`x <> mempty = x`
Associativity	`x <> (y <> z) = (x <> y) <> z`
Concatenation	`mconcat = foldr (<>) mempty`

`Monad` *laws*

Left identity	`return a >>= k = k a`

> **Right identity** `m >>= return = m`
>
> **Associativity** `m >>= (\x → k x >>= h) = (m >>= k) >>= h`

Aside from these laws, we will only need definitions.

Proving law 1: `look *> look = look`

Substituting the `(>>=)` definition makes the terms grow quite a bit, but we can use a direct proof.

`look *> look`	
`look >> look`	`(*>) = (>>)`
`look >>= \ _ -> look`	Definition of `(>>)`.

`AccumT $ \ w1 -> do`
`  (a, w2) <- runAccumT look w1`
`  (b, w3) <- runAccumT ((\ _ -> look) a) (w1 <> w2)`
`  return (b, w2 <> w3)`
	Definition of `(>>=)` from `Monad (AccumT w m)`.
	NOTE `do`-block in `m`.

`AccumT $ \ w1 -> do`
`  (_, w2) <- return (w1, mempty)`
`  (b, w3) <- return (w1 <> w2, mempty)`
`  return (b, w2 <> w3)`
	Definition of `look` and `runAccumT`. Simplification using function application.

`AccumT $ \ w1 ->` `  return (w1 <> mempty, mempty <> mempty)`	*Simplification of* `do`*-block using* `Monad`*'s "left identity".* `return a >>= k = k a`
`AccumT $ \ w1 -> return (w1, mempty)`	`Monoid`*'s "right identity".* `x <> mempty = x`
`look`	Definition of `look`.

Proving law 2: `add mempty = pure ()`

This is a simple direct proof.

`add mempty`	

`AccumT $ \ _ -> return ((), mempty)`	Definition of `add`.
`return ()`	Definition of `return` from `Monad (AccumT w m)`.
`pure ()`	`return = pure`

Proving law 3: `add x *> add y = add (x <> y)`

I guess you can probably figure out the approach by now.

> **TIP** It's a direct proof.

Unfortunately we will have to substitute `(>>=)` again. Overall the proof has the same structure as the proof of the first law.

`add x *> add y`	
`add x >> add y`	`(*>) = (>>)`
`add x >>= \ _ -> add y`	Definition of `(>>)`.

`AccumT $ \ w1 -> do`
`  (a, w2) <- runAccumT (add x) w1`
`  (b, w3) <- runAccumT ((\ _ -> add y) a) (w1 <> w2)`
`  return (b, w2 <> w3)`
	Definition of `(>>=)` from `Monad (AccumT w m)`.
	> **NOTE** `do`-block in `m`.

`AccumT $ \ w1 -> do`
`  (_, w2) <- return ((), x)`
`  (b, w3) <- return ((), y)`
`  return (b, w2 <> w3)`
	Definition of `add` and `runAccumT`. Simplification using function application.

`AccumT $ \ w1 -> return ((), x <> y)`	*Simplification of* `do`*-block using* `Monad`*'s "left identity".* `return a >>= k = k a`
`add (x <> y)`	Definition of `add`.

Proving law 4: `add x *> look = look >>= \ w → add x $> w <> x`

This time we will transform both sides of the equation and we will reach terms that are obviously equivalent.

We are starting with the left side.

`add x *> look`	
`add x >> look`	`(*>) = (>>)`
`add x >>= \ _ -> look`	Definition of `(>>)`.

`AccumT $ \ w1 -> do`
`  (a, w2) <- runAccumT (add x) w1`
`  (b, w3) <- runAccumT ((\ _ -> look) a) (w1 <> w2)`
`  return (b, w2 <> w3)`
	Definition of `(>>=)` from `Monad (AccumT w m)`.
	NOTE `do`-block in `m`.

`AccumT $ \ w1 -> do`
`  (_, w2) <- return ((), x)`
`  (b, w3) <- return (w1 <> w2, mempty)`
`  return (b, w2 <> w3)`
	Definition of `add`, `look` and `runAccumT`. Simplification using function application.

`AccumT $ \ w1 -> return (w1 <> x, x <> mempty)`	*Simplification of* `do`*-block using* `Monad`*'s "left identity".* `return a >>= k = k a`
`AccumT $ \ w1 -> return (w1 <> x, x)`	`Monoid`*'s "right identity".* `x <> mempty = x`

NOTE *Now we have to check that the right side is equivalent to this.*

```
AccumT $ \ w1 -> return (w1 <> x, x)
```

`look >>= \ w -> add x $> w <> x`	

```
AccumT $ \ w1 -> do
  (a, w2) <- runAccumT look w1
  (b, w3) <- runAccumT (add x $> a <> x) (w1 <> w2)
  return (b, w2 <> w3)
```

Definition of `(>>=)` from `Monad (AccumT w m)`.

> **NOTE** `do`-block in `m`.

Simplification using function application.

```
AccumT $ \ w1 -> do
  (a, w2) <- return (w1, mempty)
  (b, w3) <- runAccumT (add x $> a <> x) (w1 <> w2)
  return (b, w2 <> w3)
```

Definition of `add` and `runAccumT`.

```
AccumT $ \ w1 -> do
  (b, w3) <- runAccumT
               (add x $> w1 <> x)
               (w1 <> mempty)
  return (b, mempty <> w3)
```

Simplification of `do`*-block using* `Monad`*'s "left identity".*

```
return a >>= k = k a
```

```
AccumT $ \ w1 -> do
  (b, w3) <- runAccumT
               (add x >>= \ _ -> return (w1 <> x))
               (w1 <> mempty)
  return (b, mempty <> w3)
```

Substituting `Functor`*'s* `($>)` *using* `Monad`.

```
  a $> b
= a >>= \ _ -> return b
```

```
AccumT $ \ w1 -> do
  (b, w3) <- do
    (_, v2) <- runAccumT
                 (add x)
                 (w1 <> mempty)
    (q, v3) <- runAccumT
                 (return (w1 <> x))
                 ((w1 <> mempty) <> v2)
    return (q, v2 <> v3)
  return (b, mempty <> w3)
```

Definition of `(>>=)` from `Monad (AccumT w m)`.

> **NOTE** `do`-block in `m`.

Simplification using function application.

```
AccumT $ \ w1 -> do
  (b, w3) <- do
    (_, v2) <- return ((), x))
    (q, v3) <- return (w1 <> x, mempty)
    return (q, v2 <> v3)
  return (b, mempty <> w3)
```

Definition of `add` and `runAccumT`. Simplification using function application.

`AccumT $ \ w1 -> do`
`  (b, w3) <- return (w1 <> x, x <> mempty)`
`  return (b, mempty <> w3)`
	Simplification of `do`*-block using* `Monad`*'s "left identity".*
	`return a >>= k = k a`

`AccumT $ \ w1 ->` `  return (w1 <> x, mempty <> (x <> mempty))`	*Simplification of* `do`*-block using* `Monad`*'s "left identity".* `return a >>= k = k a`

`AccumT $ \ w1 -> return (w1 <> x, x)`	`Monoid`*'s "right identity".*
	`x <> mempty = x`
	`Monoid`*'s "left identity".*
	`mempty <> x = x`

And thus we have reached our goal. Both sides of the equation are actually equivalent.