mtl-2.3.1: Monad classes for transformers, using functional dependencies

Quick Jump · Instances · Source · Contents · Index

Control.Monad.State.Lazy

Contents

Lazy state monads.

This module is inspired by the paper *Functional Programming with Overloading and Higher-Order Polymorphism*, Mark P Jones (http://web.cecs.pdx.edu/~mpj/) Advanced School of Functional Programming, 1995.

MonadState class

```
class Monad m => MonadState s m | m -> s where
```

 # Source

Minimal definition is either both of `get` and `put` or just `state`

Minimal complete definition

```
state | get, put
```

Methods

get :: m s **# Source**

Return the state from the internals of the monad.

put :: s -> m () # Source

Replace the state inside the monad.

state :: (s -> (a, s)) -> m a # Source

Embed a simple state action into the monad.

▽ **Instances**

▽ MonadState s m => MonadState s (MaybeT m) # Source

Defined in Control.Monad.State.Class

Methods

get :: MaybeT m s # Source

put :: s -> MaybeT m () # Source

state :: (s -> (a, s)) -> MaybeT m a # Source

▽ (Monoid w, MonadState s m) => MonadState s (AccumT w m) # Source *Since: 2.3*

Defined in Control.Monad.State.Class

Methods

get :: AccumT w m s # Source

put :: s -> AccumT w m () # Source

state :: (s -> (a, s)) -> AccumT w m a # Source

▽ MonadState s m => MonadState s (ExceptT e m) # Source *Since: 2.2*

Defined in Control.Monad.State.Class

Methods

get :: ExceptT e m s # Source

put :: s -> ExceptT e m () # Source

state :: (s -> (a, s)) -> ExceptT e m a # Source

▽ MonadState s m => MonadState s (IdentityT m) # Source

Defined in Control.Monad.State.Class

Methods

```
get :: IdentityT m s                                    # Source
put :: s -> IdentityT m ()                              # Source
state :: (s -> (a, s)) -> IdentityT m a                 # Source
```

▿ `MonadState s m => MonadState s (ReaderT r m)` **# Source**

Defined in Control.Monad.State.Class

Methods

```
get :: ReaderT r m s                                    # Source
put :: s -> ReaderT r m ()                              # Source
state :: (s -> (a, s)) -> ReaderT r m a                 # Source
```

▿ `MonadState s m => MonadState s (SelectT r m)` **# Source** *Since: 2.3*

Defined in Control.Monad.State.Class

Methods

```
get :: SelectT r m s                                    # Source
put :: s -> SelectT r m ()                              # Source
state :: (s -> (a, s)) -> SelectT r m a                 # Source
```

▿ `Monad m => MonadState s (StateT s m)` **# Source**

Defined in Control.Monad.State.Class

Methods

```
get :: StateT s m s                                     # Source
put :: s -> StateT s m ()                               # Source
state :: (s -> (a, s)) -> StateT s m a                  # Source
```

▿ `Monad m => MonadState s (StateT s m)` **# Source**

Defined in Control.Monad.State.Class

Methods

```
get :: StateT s m s                                    # Source

put :: s -> StateT s m ()                              # Source

state :: (s -> (a, s)) -> StateT s m a                 # Source
```

▿ `(Monoid w, MonadState s m) => MonadState s (WriterT w m)` **# Source** *Since: 2.3*

Defined in Control.Monad.State.Class

Methods

```
get :: WriterT w m s                                   # Source

put :: s -> WriterT w m ()                             # Source

state :: (s -> (a, s)) -> WriterT w m a                # Source
```

▿ `(Monoid w, MonadState s m) => MonadState s (WriterT w m)` **# Source**

Defined in Control.Monad.State.Class

Methods

```
get :: WriterT w m s                                   # Source

put :: s -> WriterT w m ()                             # Source

state :: (s -> (a, s)) -> WriterT w m a                # Source
```

▿ `(Monoid w, MonadState s m) => MonadState s (WriterT w m)` **# Source**

Defined in Control.Monad.State.Class

Methods

```
get :: WriterT w m s                                   # Source

put :: s -> WriterT w m ()                             # Source

state :: (s -> (a, s)) -> WriterT w m a                # Source
```

▿ `MonadState s m => MonadState s (ContT r m)` **# Source**

Defined in Control.Monad.State.Class

Methods

`get :: ContT r m s` # Source

`put :: s -> ContT r m ()` # Source

`state :: (s -> (a, s)) -> ContT r m a` # Source

▿ `(Monad m, Monoid w) => MonadState s (RWST r w s m)` # Source *Since: 2.3*

Defined in Control.Monad.State.Class

Methods

`get :: RWST r w s m s` # Source

`put :: s -> RWST r w s m ()` # Source

`state :: (s -> (a, s)) -> RWST r w s m a` # Source

▿ `(Monad m, Monoid w) => MonadState s (RWST r w s m)` # Source

Defined in Control.Monad.State.Class

Methods

`get :: RWST r w s m s` # Source

`put :: s -> RWST r w s m ()` # Source

`state :: (s -> (a, s)) -> RWST r w s m a` # Source

▿ `(Monad m, Monoid w) => MonadState s (RWST r w s m)` # Source

Defined in Control.Monad.State.Class

Methods

`get :: RWST r w s m s` # Source

`put :: s -> RWST r w s m ()` # Source

`state :: (s -> (a, s)) -> RWST r w s m a` # Source

`modify :: MonadState s m => (s -> s) -> m ()` # Source

Monadic state transformer.

Maps an old state to a new state inside a state monad. The old state is thrown away.

```
Main> :t modify ((+1) :: Int -> Int)
modify (...) :: (MonadState Int a) => a ()
```

This says that `modify (+1)` acts over any Monad that is a member of the `MonadState` class, with an `Int` state.

`modify' :: MonadState s m => (s -> s) -> m ()` **# Source**

A variant of `modify` in which the computation is strict in the new state.

Since: 2.2

`gets :: MonadState s m => (s -> a) -> m a` **# Source**

Gets specific component of the state, using a projection function supplied.

The State monad

`type State s = StateT s Identity` #

A state monad parameterized by the type `s` of the state to carry.

The `return` function leaves the state unchanged, while >>= uses the final state of the first computation as the initial state of the second.

`runState` #

`:: State s a`	state-passing computation to execute
`-> s`	initial state
`-> (a, s)`	return value and final state

Unwrap a state monad computation as a function. (The inverse of `state`.)

`evalState` #

`:: State s a`	state-passing computation to execute
`-> s`	initial value
`-> a`	return value of the state computation

Evaluate a state computation with the given initial state and return the final value, discarding the final state.

- ```
  evalState m s = fst (runState m s)
  ```
  ```

execState #

```
:: State s a    state-passing computation to execute
-> s            initial value
-> s            final state
```

Evaluate a state computation with the given initial state and return the final state, discarding the final value.

- `execState m s = snd (runState m s)`

mapState `:: ((a, s) -> (b, s)) -> State s a -> State s b` #

Map both the return value and final state of a computation using the given function.

- `runState (mapState f m) = f . runState m`

withState `:: (s -> s) -> State s a -> State s a` #

`withState f m` executes action `m` on a state modified by applying `f`.

- `withState f m = modify f >> m`

The StateT monad transformer

`newtype` **StateT** `s (m :: Type -> Type) a` #

A state transformer monad parameterized by:

- `s` - The state.
- `m` - The inner monad.

The `return` function leaves the state unchanged, while >>= uses the final state of the first computation as the initial state of the second.

Constructors

StateT `(s -> m (a, s))`

▽ Instances

▽ `MonadAccum w m => MonadAccum w (StateT s m)` **Source** # *Since: 2.3*

Defined in Control.Monad.Accum

Methods

```
look :: StateT s m w
```
Source

```
add :: w -> StateT s m ()
```
Source

```
accum :: (w -> (a, w)) -> StateT s m a
```
Source

▿ `MonadError e m => MonadError e (StateT s m)` Source #

Defined in Control.Monad.Error.Class

Methods

```
throwError :: e -> StateT s m a
```
Source

```
catchError :: StateT s m a -> (e -> StateT s m a) -> StateT s m a
```
Source

▿ `MonadReader r m => MonadReader r (StateT s m)` Source #

Defined in Control.Monad.Reader.Class

Methods

```
ask :: StateT s m r
```
Source

```
local :: (r -> r) -> StateT s m a -> StateT s m a
```
Source

```
reader :: (r -> a) -> StateT s m a
```
Source

▿ `MonadSelect w m => MonadSelect w (StateT s m)` Source #

'Readerizes' the state: the 'ranking' function can *see* a value of type `s`, but not modify it. Effectively, can be thought of as 'extending' the 'ranking' by all values in `s`, but *which* `s` gets given to any rank calls is predetermined by the 'outer state' (and cannot change).

Since: 2.3

Defined in Control.Monad.Select

Methods

```
select :: ((a -> w) -> a) -> StateT s m a
```
Source

▿ Monad m => MonadState s (StateT s m) # Source

Defined in Control.Monad.State.Class

Methods

get :: StateT s m s # Source

put :: s -> StateT s m () # Source

state :: (s -> (a, s)) -> StateT s m a # Source

▿ MonadWriter w m => MonadWriter w (StateT s m) Source #

Defined in Control.Monad.Writer.Class

Methods

writer :: (a, w) -> StateT s m a # Source

tell :: w -> StateT s m () # Source

listen :: StateT s m a -> StateT s m (a, w) # Source

pass :: StateT s m (a, w -> w) -> StateT s m a # Source

▿ MonadTrans (StateT s)

Defined in Control.Monad.Trans.State.Lazy

Methods

lift :: Monad m => m a -> StateT s m a #

▿ MonadFail m => MonadFail (StateT s m)

Defined in Control.Monad.Trans.State.Lazy

Methods

fail :: String -> StateT s m a #

▿ MonadFix m => MonadFix (StateT s m)

Defined in Control.Monad.Trans.State.Lazy

Methods

mfix :: (a -> StateT s m a) -> StateT s m a #

▿ `MonadIO m => MonadIO (StateT s m)`

Defined in Control.Monad.Trans.State.Lazy

Methods

`liftIO :: IO a -> StateT s m a` #

▿ `Contravariant m => Contravariant (StateT s m)`

Defined in Control.Monad.Trans.State.Lazy

Methods

`contramap :: (a' -> a) -> StateT s m a -> StateT s m a'` #

`(>$) :: b -> StateT s m b -> StateT s m a` #

▿ `(Functor m, MonadPlus m) => Alternative (StateT s m)`

Defined in Control.Monad.Trans.State.Lazy

Methods

`empty :: StateT s m a` #

`(<|>) :: StateT s m a -> StateT s m a -> StateT s m a` #

`some :: StateT s m a -> StateT s m [a]` #

`many :: StateT s m a -> StateT s m [a]` #

▿ `(Functor m, Monad m) => Applicative (StateT s m)`

Defined in Control.Monad.Trans.State.Lazy

Methods

`pure :: a -> StateT s m a` #

`(<*>) :: StateT s m (a -> b) -> StateT s m a -> StateT s m b` #

`liftA2 :: (a -> b -> c) -> StateT s m a -> StateT s m b -> StateT s m c` #

`(*>) :: StateT s m a -> StateT s m b -> StateT s m b` #

`(<*) :: StateT s m a -> StateT s m b -> StateT s m a` #

▿ `Functor m => Functor (StateT s m)`

Defined in Control.Monad.Trans.State.Lazy

Methods

`fmap :: (a -> b) -> StateT s m a -> StateT s m b` #

`(<$) :: a -> StateT s m b -> StateT s m a` #

▿ `Monad m => Monad (StateT s m)`

Defined in Control.Monad.Trans.State.Lazy

Methods

`(>>=) :: StateT s m a -> (a -> StateT s m b) -> StateT s m b` #

`(>>) :: StateT s m a -> StateT s m b -> StateT s m b` #

`return :: a -> StateT s m a` #

▿ `MonadPlus m => MonadPlus (StateT s m)`

Defined in Control.Monad.Trans.State.Lazy

Methods

`mzero :: StateT s m a` #

`mplus :: StateT s m a -> StateT s m a -> StateT s m a` #

▿ `MonadCont m => MonadCont (StateT s m)` **# Source**

Defined in Control.Monad.Cont.Class

Methods

`callCC :: ((a -> StateT s m b) -> StateT s m a) -> StateT s m a` **# Source**

`runStateT` `:: StateT s m a -> s -> m (a, s)` #

`evalStateT` `:: Monad m => StateT s m a -> s -> m a` #

Evaluate a state computation with the given initial state and return the final value, discarding the final state.

- `evalStateT m s = liftM fst (runStateT m s)`

execStateT :: Monad m => StateT s m a -> s -> m s #

Evaluate a state computation with the given initial state and return the final state, discarding the final value.

- execStateT m s = liftM snd (runStateT m s)

mapStateT :: (m (a, s) -> n (b, s)) -> StateT s m a -> StateT s n b #

Map both the return value and final state of a computation using the given function.

- runStateT (mapStateT f m) = f . runStateT m

withStateT :: forall s (m :: Type -> Type) a. (s -> s) -> StateT s m a -> StateT s m a #

withStateT f m executes action m on a state modified by applying f.

- withStateT f m = modify f >> m

module Control.Monad.Trans

Examples

A function to increment a counter. Taken from the paper *Generalising Monads to Arrows*, John Hughes (http://www.cse.chalmers.se/~rjmh/Papers/arrows.pdf), November 1998:

```
tick :: State Int Int
tick = do n <- get
          put (n+1)
          return n
```

Add one to the given number using the state monad:

```
plusOne :: Int -> Int
plusOne n = execState tick n
```

A contrived addition example. Works only with positive numbers:

```
plus :: Int -> Int -> Int
plus n x = execState (sequence $ replicate n tick) x
```

An example from *The Craft of Functional Programming*, Simon Thompson (http://www.cs.kent.ac.uk/people/staff/sjt/), Addison-Wesley 1999: "Given an arbitrary tree, transform it to a tree of integers in which the original elements are replaced by natural numbers, starting from 0. The same element has to be replaced by the same number at every

occurrence, and when we meet an as-yet-unvisited element we have to find a 'new' number to match it with:"

```
data Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show, Eq)
type Table a = [a]
```

```
numberTree :: Eq a => Tree a -> State (Table a) (Tree Int)
numberTree Nil = return Nil
numberTree (Node x t1 t2)
       =  do num <- numberNode x
             nt1 <- numberTree t1
             nt2 <- numberTree t2
             return (Node num nt1 nt2)
    where
    numberNode :: Eq a => a -> State (Table a) Int
    numberNode x
       = do table <- get
            (newTable, newPos) <- return (nNode x table)
            put newTable
            return newPos
    nNode::  (Eq a) => a -> Table a -> (Table a, Int)
    nNode x table
       = case (findIndexInList (== x) table) of
         Nothing -> (table ++ [x], length table)
         Just i  -> (table, i)
    findIndexInList :: (a -> Bool) -> [a] -> Maybe Int
    findIndexInList = findIndexInListHelp 0
    findIndexInListHelp _ _ [] = Nothing
    findIndexInListHelp count f (h:t)
       = if (f h)
         then Just count
         else findIndexInListHelp (count+1) f t
```

numTree applies numberTree with an initial state:

```
numTree :: (Eq a) => Tree a -> Tree Int
numTree t = evalState (numberTree t) []
```

```
testTree = Node "Zero" (Node "One" (Node "Two" Nil Nil) (Node "One" (Node "Zero"
numTree testTree => Node 0 (Node 1 (Node 2 Nil Nil) (Node 1 (Node 0 Nil Nil) Nil)
```

sumTree is a little helper function that does not use the State monad:

```
sumTree :: (Num a) => Tree a -> a
sumTree Nil = 0
sumTree (Node e t1 t2) = e + (sumTree t1) + (sumTree t2)
```