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

Control.Monad.Reader

Contents

MonadReader class
The Reader monad
The ReaderT monad transformer
Example 1: Simple Reader Usage
Example 2: Modifying Reader Content With `local`
Example 3: `ReaderT` Monad Transformer

Computation type:
Computations which read values from a shared environment.
Binding strategy:
Monad values are functions from the environment to a value. The bound function is applied to the bound value, and both have access to the shared environment.
Useful for:
Maintaining variable bindings, or other shared environment.
Zero and plus:
None.
Example type:
`Reader [(String,Value)] a`

The `Reader` monad (also called the Environment monad). Represents a computation, which can read values from a shared environment, pass values from function to function, and execute sub-computations in a modified environment. Using `Reader` monad for such computations is often clearer and easier than using the `State` monad.

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.

MonadReader class

```
class Monad m => MonadReader r m | m -> r where
```
 # Source

See examples in Control.Monad.Reader. Note, the partially applied function type `(->) r` is a simple reader monad. See the `instance` declaration below.

Minimal complete definition

`(ask | reader), local`

Methods

ask :: m r **# Source**

Retrieves the monad environment.

local **# Source**

:: (r -> r) The function to modify the environment.
-> m a Reader to run in the modified environment.
-> m a

Executes a computation in a modified environment.

reader **# Source**

:: (r -> a) The selector function to apply to the environment.
-> m a

Retrieves a function of the current environment.

▽ Instances

▽ MonadReader r m => MonadReader r (MaybeT m) **# Source**

Defined in Control.Monad.Reader.Class

Methods

ask :: MaybeT m r **# Source**

local :: (r -> r) -> MaybeT m a -> MaybeT m a **# Source**

reader :: (r -> a) -> MaybeT m a **# Source**

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

Defined in Control.Monad.Reader.Class

Methods

ask :: AccumT w m r **# Source**

local :: (r -> r) -> AccumT w m a -> AccumT w m a **# Source**

reader :: (r -> a) -> AccumT w m a **# Source**

▿ MonadReader r m => MonadReader r (ExceptT e m) # Source *Since: 2.2*

Defined in Control.Monad.Reader.Class

Methods

ask :: ExceptT e m r # Source

local :: (r -> r) -> ExceptT e m a -> ExceptT e m a # Source

reader :: (r -> a) -> ExceptT e m a # Source

▿ MonadReader r m => MonadReader r (IdentityT m) # Source

Defined in Control.Monad.Reader.Class

Methods

ask :: IdentityT m r # Source

local :: (r -> r) -> IdentityT m a -> IdentityT m a # Source

reader :: (r -> a) -> IdentityT m a # Source

▿ Monad m => MonadReader r (ReaderT r m) # Source

Defined in Control.Monad.Reader.Class

Methods

ask :: ReaderT r m r # Source

local :: (r -> r) -> ReaderT r m a -> ReaderT r m a # Source

reader :: (r -> a) -> ReaderT r 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

▿ 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

▿ (Monoid w, MonadReader r m) => MonadReader r (WriterT w m) # Source *Since: 2.3*

Defined in Control.Monad.Reader.Class

Methods

ask :: WriterT w m r # Source

local :: (r -> r) -> WriterT w m a -> WriterT w m a # Source

reader :: (r -> a) -> WriterT w m a # Source

▿ (Monoid w, MonadReader r m) => MonadReader r (WriterT w m) # Source

Defined in Control.Monad.Reader.Class

Methods

ask :: WriterT w m r # Source

local :: (r -> r) -> WriterT w m a -> WriterT w m a # Source

reader :: (r -> a) -> WriterT w m a # Source

▿ (Monoid w, MonadReader r m) => MonadReader r (WriterT w m) # Source

Defined in Control.Monad.Reader.Class

Methods

ask :: WriterT w m r # Source

local :: (r -> r) -> WriterT w m a -> WriterT w m a # Source

reader :: (r -> a) -> WriterT w m a # Source

▿ MonadReader r' m => MonadReader r' (SelectT r m) # Source *Since: 2.3*

Defined in Control.Monad.Reader.Class

Methods

ask :: SelectT r m r' # Source

local :: (r' -> r') -> SelectT r m a -> SelectT r m a # Source

reader :: (r' -> a) -> SelectT r m a # Source

▿ MonadReader r ((->) r) # Source

Defined in Control.Monad.Reader.Class

Methods

ask :: r -> r # Source

local :: (r -> r) -> (r -> a) -> r -> a # Source

reader :: (r -> a) -> r -> a # Source

▿ MonadReader r' m => MonadReader r' (ContT r m) # Source

Defined in Control.Monad.Reader.Class

Methods

ask :: ContT r m r' # Source

local :: (r' -> r') -> ContT r m a -> ContT r m a # Source

reader :: (r' -> a) -> ContT r m a # Source

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

Defined in Control.Monad.Reader.Class

Methods

ask :: RWST r w s m r # Source

local :: (r -> r) -> RWST r w s m a -> RWST r w s m a # Source

reader :: (r -> a) -> RWST r w s m a # Source

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

Defined in Control.Monad.Reader.Class

Methods

ask :: RWST r w s m r # Source

local :: (r -> r) -> RWST r w s m a -> RWST r w s m a # Source

reader :: (r -> a) -> RWST r w s m a # Source

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

Defined in Control.Monad.Reader.Class

Methods

ask :: RWST r w s m r # Source

local :: (r -> r) -> RWST r w s m a -> RWST r w s m a # Source

reader :: (r -> a) -> RWST r w s m a # Source

asks # Source

```
:: MonadReader r m
=> (r -> a)          The selector function to apply to the environment.
-> m a
```

Retrieves a function of the current environment.

The Reader monad

type Reader r = ReaderT r Identity

The parameterizable reader monad.

Computations are functions of a shared environment.

The return function ignores the environment, while >>= passes the inherited environment to both subcomputations.

runReader

```
:: Reader r a   A Reader to run.
```

```
-> r          An initial environment.
-> a
```

Runs a `Reader` and extracts the final value from it. (The inverse of `reader`.)

`mapReader :: (a -> b) -> Reader r a -> Reader r b` #

Transform the value returned by a `Reader`.

- `runReader (mapReader f m) = f . runReader m`

`withReader` #

```
:: (r' -> r)      The function to modify the environment.
-> Reader r a     Computation to run in the modified environment.
-> Reader r' a
```

Execute a computation in a modified environment (a specialization of `withReaderT`).

- `runReader (withReader f m) = runReader m . f`

The ReaderT monad transformer

`newtype` **`ReaderT`** `r (m :: Type -> Type) a` #

The reader monad transformer, which adds a read-only environment to the given monad.

The `return` function ignores the environment, while >>= passes the inherited environment to both subcomputations.

Constructors

`ReaderT` `(r -> m a)`

▽ **Instances**

▿ `MonadAccum w m => MonadAccum w (ReaderT r m)` *Since: 2.3*

 # Source

Defined in Control.Monad.Accum

Methods

`look :: ReaderT r m w` **# Source**

```
add :: w -> ReaderT r m ()
```
Source

```
accum :: (w -> (a, w)) -> ReaderT r m a
```
Source

▿ `MonadError e m => MonadError e (ReaderT r m)`
Source

Defined in Control.Monad.Error.Class

Methods

```
throwError :: e -> ReaderT r m a
```
Source

```
catchError :: ReaderT r m a -> (e -> ReaderT r m a) -> ReaderT r m a
```
Source #

▿ `Monad m => MonadReader r (ReaderT r m)` # Source

Defined in Control.Monad.Reader.Class

Methods

```
ask :: ReaderT r m r
```
Source

```
local :: (r -> r) -> ReaderT r m a -> ReaderT r m a
```
Source

```
reader :: (r -> a) -> ReaderT r m a
```
Source

▿ `MonadSelect r' m => MonadSelect r' (ReaderT r m)`
Source

Provides a read-only environment of type `r` to the 'strategy' function. However, the 'ranking' function (or more accurately, representation) has no access to `r`. Put another way, you can influence what values get chosen by changing `r`, but not how solutions are ranked.

Since: 2.3

Defined in Control.Monad.Select

Methods

```
select :: ((a -> r') -> a) -> ReaderT r 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
```

▿ MonadWriter w m => MonadWriter w (ReaderT r m)
Source

Defined in Control.Monad.Writer.Class

Methods

```
writer :: (a, w) -> ReaderT r m a                       # Source

tell :: w -> ReaderT r m ()                             # Source

listen :: ReaderT r m a -> ReaderT r m (a, w)           # Source

pass :: ReaderT r m (a, w -> w) -> ReaderT r m a        # Source
```

▿ MonadTrans (ReaderT r)

Defined in Control.Monad.Trans.Reader

Methods

```
lift :: Monad m => m a -> ReaderT r m a                 #
```

▿ MonadFail m => MonadFail (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

```
fail :: String -> ReaderT r m a                         #
```

▿ MonadFix m => MonadFix (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

```
mfix :: (a -> ReaderT r m a) -> ReaderT r m a           #
```

▿ MonadIO m => MonadIO (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

```
liftIO :: IO a -> ReaderT r m a                                    #
```

▿ MonadZip m => MonadZip (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

```
mzip :: ReaderT r m a -> ReaderT r m b -> ReaderT r m (a, b)       #

mzipWith :: (a -> b -> c) -> ReaderT r m a -> ReaderT r m b -> ReaderT r m c
                                                                   #
munzip :: ReaderT r m (a, b) -> (ReaderT r m a, ReaderT r m b)     #
```

▿ Contravariant m => Contravariant (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

```
contramap :: (a' -> a) -> ReaderT r m a -> ReaderT r m a'          #

(>$) :: b -> ReaderT r m b -> ReaderT r m a                        #
```

▿ Alternative m => Alternative (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

```
empty :: ReaderT r m a                                             #

(<|>) :: ReaderT r m a -> ReaderT r m a -> ReaderT r m a           #

some :: ReaderT r m a -> ReaderT r m [a]                           #

many :: ReaderT r m a -> ReaderT r m [a]                           #
```

▿ Applicative m => Applicative (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

```
pure :: a -> ReaderT r m a                                         #

(<*>) :: ReaderT r m (a -> b) -> ReaderT r m a -> ReaderT r m b    #

liftA2 :: (a -> b -> c) -> ReaderT r m a -> ReaderT r m b -> ReaderT r m c
```

(*>) :: ReaderT r m a -> ReaderT r m b -> ReaderT r m b #

(<*) :: ReaderT r m a -> ReaderT r m b -> ReaderT r m a #

▿ Functor m => Functor (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

fmap :: (a -> b) -> ReaderT r m a -> ReaderT r m b #

(<$) :: a -> ReaderT r m b -> ReaderT r m a #

▿ Monad m => Monad (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

(>>=) :: ReaderT r m a -> (a -> ReaderT r m b) -> ReaderT r m b #

(>>) :: ReaderT r m a -> ReaderT r m b -> ReaderT r m b #

return :: a -> ReaderT r m a #

▿ MonadPlus m => MonadPlus (ReaderT r m)

Defined in Control.Monad.Trans.Reader

Methods

mzero :: ReaderT r m a #

mplus :: ReaderT r m a -> ReaderT r m a -> ReaderT r m a #

▿ MonadCont m => MonadCont (ReaderT r m) **# Source**

Defined in Control.Monad.Cont.Class

Methods

callCC :: ((a -> ReaderT r m b) -> ReaderT r m a) -> ReaderT r m a **# Source**

runReaderT :: ReaderT r m a -> r -> m a #

mapReaderT :: (m a -> n b) -> ReaderT r m a -> ReaderT r n b #

Transform the computation inside a ReaderT.

- `runReaderT (mapReaderT f m) = f . runReaderT m`

withReaderT #

```
:: forall r' r (m :: Type -> Type) a. (r' -> r)
-> ReaderT r m a
-> ReaderT r' m a
```

The function to modify the environment.

Computation to run in the modified environment.

Execute a computation in a modified environment (a more general version of `local`).

- `runReaderT (withReaderT f m) = runReaderT m . f`

`module Control.Monad.Trans`

Example 1: Simple Reader Usage

In this example the Reader monad provides access to variable bindings. Bindings are a Map of integer variables. The variable `count` contains number of variables in the bindings. You can see how to run a Reader monad and retrieve data from it with `runReader`, how to access the Reader data with `ask` and `asks`.

```
import           Control.Monad.Reader
import           Data.Map (Map)
import qualified Data.Map as Map

type Bindings = Map String Int

-- Returns True if the "count" variable contains correct bindings size.
isCountCorrect :: Bindings -> Bool
isCountCorrect bindings = runReader calc_isCountCorrect bindings

-- The Reader monad, which implements this complicated check.
calc_isCountCorrect :: Reader Bindings Bool
calc_isCountCorrect = do
    count <- asks (lookupVar "count")
    bindings <- ask
    return (count == (Map.size bindings))

-- The selector function to use with 'asks'.
-- Returns value of the variable with specified name.
lookupVar :: String -> Bindings -> Int
lookupVar name bindings = maybe 0 id (Map.lookup name bindings)

sampleBindings :: Bindings
sampleBindings = Map.fromList [("count", 3), ("1", 1), ("b", 2)]
```

```
main :: IO ()
main = do
    putStr $ "Count is correct for bindings " ++ (show sampleBindings) ++ ": "
    putStrLn $ show (isCountCorrect sampleBindings)
```

Example 2: Modifying Reader Content With `local`

Shows how to modify Reader content with `local`.

```
import Control.Monad.Reader

calculateContentLen :: Reader String Int
calculateContentLen = do
    content <- ask
    return (length content);

-- Calls calculateContentLen after adding a prefix to the Reader content.
calculateModifiedContentLen :: Reader String Int
calculateModifiedContentLen = local ("Prefix " ++) calculateContentLen

main :: IO ()
main = do
    let s = "12345";
    let modifiedLen = runReader calculateModifiedContentLen s
    let len = runReader calculateContentLen s
    putStrLn $ "Modified 's' length: " ++ (show modifiedLen)
    putStrLn $ "Original 's' length: " ++ (show len)
```

Example 3: `ReaderT` Monad Transformer

Now you are thinking: 'Wow, what a great monad! I wish I could use Reader functionality in MyFavoriteComplexMonad!'. Don't worry. This can be easily done with the `ReaderT` monad transformer. This example shows how to combine `ReaderT` with the IO monad.

```
import Control.Monad.Reader

-- The Reader/IO combined monad, where Reader stores a string.
printReaderContent :: ReaderT String IO ()
printReaderContent = do
    content <- ask
    liftIO $ putStrLn ("The Reader Content: " ++ content)

main :: IO ()
main = runReaderT printReaderContent "Some Content"
```