memo-sqlite

Claude Heiland-Allen

2011–2018

Contents

1 Data/Memo/Sqlite.hs

```
{- |
Module      :  Data.Memo.Sqlite
Copyright   :  (c) 2011 Claude Heiland-Allen
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  portable

Memoize functions in a SQLite3 database.

The functions memoized while having type @f :: k -> IO v@ must result in
the same output given the same input, otherwise all kinds of wrongness
will result.

The @cleanup@ action returned by the memoizers must not be called if you
are going to use the memoized function again; beware.

An example program (included in the examples directory) might look like:

> -- fib.hs
> import Data.Memo.Sqlite (memoRec', readShow, table)
>
> import Control.Monad (liftM2)
> import System.Environment (getArgs)
>
> fib :: (Integer -> IO Integer) -> Integer -> IO Integer
> fib _fib' n@0 = print n >> return 0
> fib _fib' n@1 = print n >> return 1
> fib  fib' n   = print n >> liftM2 (+) (fib' (n - 1)) (fib' (n - 2))
>
> main :: IO ()
> main = do
>   [file, ts, ns] <- getArgs
>   let Just t = table ts
```

```
  >       n = read ns
  >   (cleanup, fib') <- memoRec' readShow file t fib
  >   fib' n >>= \nth -> putStrLn $ "fib(" ++ show n ++ ") = " ++ show nth
  >   cleanup

Example usage:

  > ghc --make fib.hs
  > ./fib fibs.sqlite3 fibs 10
  > ./fib fibs.sqlite3 fibs 10
  > ./fib fibs.sqlite3 fibs 100
  > ./fib fibs.sqlite3 fibs 100

See also:

  * the @data-memocombinators@ package for pure in-memory memoization.

-}

{-# LANGUAGE OverloadedStrings #-}

module Data.Memo.Sqlite
  (
  -- * Database table identifier.
    Table()
  , table
  -- * Database (de)serialization.
  , Sqlite(..)
  -- ** Wrapper types for control over (de)serialization.
  , Wrap
  , Unwrap
  , Wrapper
  -- ** Read/Show (de)serialization.
  , ReadShow()
  , readShow
  -- * Memoizers.
  -- ** Memoizer types.
  , Memo
  , MemoRec
  , MkMemo
  , MkMemoRec
  -- ** Memoizer functions.
  , memo
  , memoRec
  , memo'
  , memoRec'
  -- * SQLite3 data (re-exported from the direct-sqlite package).
  , SQLData(..)
  ) where

import Prelude hiding (lookup, all)

import System.IO (fixIO)

import Data.Text as T

import Database.SQLite3 -- hackage: direct-sqlite
```

```
-- | A valid SQLite3 table name.
newtype Table = Table Text deriving (Eq, Ord, Show)

-- | Construct a table name.
table :: Text -> Maybe Table
table s
  | T.null s = Nothing
  | T.head s `elem` letters &&
    all (`elem` alphaNum) s &&
    not ("sqlite_" `isPrefixOf` s) = Just (Table $ "\"" <> s <> "\"")
  | otherwise = Nothing
  where
    letters  = ['a'..'z'] ++ ['A'..'Z']
    digits   = ['0'..'9']
    alphaNum = letters ++ digits ++ "_"

-- | Database (de)serialization
class Sqlite t where
  -- | Serialize to SQLite3 data.
  toSqlite   :: t -> SQLData
  -- | Deserialize from SQLite3 data.
  fromSqlite :: SQLData -> t

type Wrap    s t k v = (k -> IO v) -> s k -> IO (t v)
type Unwrap  s t k v = (s k -> IO (t v)) -> k -> IO v
type Wrapper s t k v = (Wrap s t k v, Unwrap s t k v)

-- | Use Read and Show for database (de)serialization.
newtype ReadShow t = ReadShow{ unReadShow :: t }

instance (Read t, Show t) => Sqlite (ReadShow t) where
  toSqlite   (ReadShow t) = SQLText (T.pack $ show t)
  fromSqlite (SQLText  s) = ReadShow $ read (T.unpack s)

toReadShow :: Wrap ReadShow ReadShow k v
toReadShow f k = ReadShow `fmap` f (unReadShow k)

fromReadShow :: Unwrap ReadShow ReadShow k v
fromReadShow f k = unReadShow `fmap` f (ReadShow k)

-- | Wrapper using Read and Show for (de)serialization of both keys and values.
readShow :: Wrapper ReadShow ReadShow k v
readShow = (toReadShow, fromReadShow)

type Memo k v = (k -> IO v) -> IO (IO (), k -> IO v)
type MemoRec k v = ((k -> IO v) -> k -> IO v) -> IO (IO (), k -> IO v)
type MkMemo k v = Text -> Table -> Memo k v
type MkMemoRec k v = Text -> Table -> MemoRec k v

-- | Memoize a function using an SQLite3 database.
memo :: (Sqlite k, Sqlite v) => MkMemo k v
memo file (Table tab) f = do
  db <- open file
  -- create database
  create <- prepare db $ "CREATE TABLE IF NOT EXISTS " <> tab <> " ( k TEXT ↙
      ↳ PRIMARY KEY, v TEXT )"
```

```
    _ <- step create
    finalize create
    -- prepare statements
    lookup <- prepare db $ "SELECT v FROM " <> tab <> " WHERE k = ? LIMIT 1"
    insert <- prepare db $ "INSERT INTO " <> tab <> " ( k , v ) VALUES ( ? , ? )"
    let -- clean up database
        cleanup = do
          finalize lookup
          finalize insert
          close db
        -- memoize a value in the database
        remember k = do
          -- lookup key
          let ks = toSqlite k
          reset lookup
          bind lookup [ks]
          r <- step lookup
          case r of
            Row -> do -- found: return value
              vs <- column lookup 0
              return (fromSqlite vs)
            Done -> do -- not found: calculate and insert
              v <- f k
              let vs = toSqlite v
              reset insert
              bind insert [ks, vs]
              _ <- step insert
              return v
    return ( cleanup , remember )

-- | Memoize a recursive function using an SQLite3 database.
memoRec :: (Sqlite k, Sqlite v) => MkMemoRec k v
memoRec file tab f = fixIO $ \rf -> memo file tab $ f (snd rf)

-- | Memoize a function using an SQLite3 database, using the supplied wrapper ↙
    ↳ for control of (de)serialization.
memo' :: (Sqlite (s k), Sqlite (t v)) => Wrapper s t k v -> MkMemo k v
memo' (wrap, unwrap) file tab = via wrap unwrap (memo file tab)

via    i o a f = fmap o `fmap` a (i f)

-- | Memoize a recursive function using an SQLite3 database, using the supplied ↙
    ↳ wrapper for control of (de)serialization.
memoRec' :: (Sqlite (s k), Sqlite (t v)) => Wrapper s t k v -> MkMemoRec k v
memoRec' (wrap, unwrap) file tab = viaRec wrap unwrap (memoRec file tab)

viaRec i o a f = fmap o `fmap` a (i . f . o)
```

2 examples/fib.hs

```
import Control.Monad (liftM2)
import System.Environment (getArgs)
import qualified Data.Text as T
import Data.Memo.Sqlite (memoRec', readShow, table)

-- | Compute Fibonacci numbers.
fib :: (Integer -> IO Integer) -> Integer -> IO Integer
```

```
fib _fib' n@0 = print n >> return 0
fib _fib' n@1 = print n >> return 1
fib  fib' n   = print n >> liftM2 (+) (fib' (n - 1)) (fib' (n - 2))

-- | Example program.
main :: IO ()
main = do
  [file, ts, ns] <- getArgs
  let Just t = table (T.pack ts)
      n = read ns
  (cleanup, fib') <- memoRec' readShow (T.pack file) t fib
  fib' n >>= \fn -> putStrLn $ "fib(" ++ show n ++ ") = " ++ show fn
  cleanup

{-
  Example usage:

  $ ghc -O2 -Wall --make fib.hs
  $ ./fib fibs.sqlite3 fibs 10
  $ ./fib fibs.sqlite3 fibs 10
  $ ./fib fibs.sqlite3 fibs 100
  $ ./fib fibs.sqlite3 fibs 100
-}
```

3 .gitignore

```
.cabal-sandbox
cabal.sandbox.config
dist
```

4 LICENSE

LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

5 memo-sqlite.cabal

```
Name:                memo-sqlite
Version:             0.2
Synopsis:            memoize functions using SQLite3 database
Homepage:            https://code.mathr.co.uk/memo-sqlite
License:             BSD3
License-file:        LICENSE
Author:              Claude Heiland-Allen
Maintainer:          claude@mathr.co.uk
Category:            Database
Build-type:          Simple
Extra-source-files:  examples/fib.hs
Cabal-version:       >=1.10

Library
  Exposed-modules:     Data.Memo.Sqlite
  Build-depends:       base < 5,
                       direct-sqlite >= 2.3 && < 2.4,
                       text >= 1.2 && < 1.3
  default-language:    Haskell2010
  other-extensions:    OverloadedStrings

source-repository head
  type: git
  location: https://code.mathr.co.uk/memo-sqlite.git

source-repository this
  type: git
  location: https://code.mathr.co.uk/memo-sqlite.git
  tag: memo-sqlite-0.2
```

6 Setup.hs

```
import Distribution.Simple
main = defaultMain
```