Vícevláknové programování

Jenom GHC a pořádně jenom při kompilování s parametrem -threaded

Modul IO

```
type IOError = IOException
userError :: String -> IOError
isAlreadyExistsError:: IOError -> Bool
isDoesNotExistError :: IOError -> Bool
isAlreadyInUseError :: IOError -> Bool
isFullError         :: IOError -> Bool
isEOFError          :: IOError -> Bool
isIllegalOperation  :: IOError -> Bool
isPermissionError   :: IOError -> Bool
isUserError         :: IOError -> Bool

ioError :: IOError -> IO a
catch   :: IO a -> (IOError -> IO a) -> IO a
try     :: IO a -> IO (Either IOError a)
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket_:: IO a -> (a -> IO b) -> IO c -> IO c

do = c<-catch getLine (\e -> if IO.isEOFError e then return [] else ioError e)
```

Modul Data.Dynamic

```
module Data.Typeable

toDyn      ::Typeable a => a -> Dynamic
fromDyn    ::Typeable a => Dynamic -> a -> a
fromDynamic::Typeable a => Dynamic -> Maybe a
```

Modul Control.Exception

```
data Exception = ArithException|IOException|... deriving (Eq,Show,Typeable)
ioErrors        :: Exception -> Maybe IOError
arithExceptions:: Exception -> Maybe ArithException
errorCalls      :: Exception -> Maybe String
dynExceptions   :: Exception -> Maybe Dynamic

throwIO :: Exception -> IO a
throw   :: Exception -> a        I mimo IO monádu
throwTo :: ThreadId -> Exception -> IO ()
assert  :: Bool -> a -> a
catch   :: IO a -> (Exception -> IO a)-> IO a
handle  :: (Exception -> IO a) -> IO a -> IO a
try     :: IO a -> IO (Either Exception a)

throwDyn  :: Typeable exception => exception -> b
throwDynTo:: Typeable exception => ThreadId -> exception -> IO ()
catchDyn  :: Typeable exception => IO a -> (exception -> IO a) -> IO a

block  :: IO a -> IO a           Dědí se skrz forkIO
unblock:: IO a -> IO a           K použití uvnitř blocku

bracket:: IO a-> (a -> IO b)-> (a -> IO c)-> IO c
bracket before after thing = block (do a <- before
                                       r <- catch (unblock (thing a))
                                                  (\e->do { after a; throw e })
                                       after a
                                       return r)
finally:: IO a-> IO b-> IO a
a `finally` sequel = block (do r <- catch (unblock a)
                                          (\e -> do { sequel; throw e })
                               sequel
                               return r)
```

```
                              Modul Control.Parallel
                              ----------------------
par :: a -> b -> b
pseq:: a-> b-> b

                             Modul Control.Concurrency
                             -------------------------
data ThreadId deriving (Eq, Ord, Show)
myThreadId :: IO ThreadId
numCapabilities :: Int                           +RTS -N
forkIO      :: IO () -> IO ThreadId
forkOnIO    :: Int -> IO () -> IO ThreadId          {-je v @GHC.Conc@-}
killThread :: ThreadId -> IO ()

yield       :: IO ()
threadDelay      :: Int -> IO ()
threadWaitRead   :: Fd -> IO ()
threadWaitWrite :: Fd -> IO ()

                              Control.Concurrent.MVar
                              -----------------------
data MVar a
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
tryTakeMVar :: MVar a -> IO (Maybe a)
tryPutMVar :: MVar a -> a -> IO Bool
isEmptyMVar :: MVar a -> IO Bool

withMVar :: MVar a -> (a -> IO b) -> IO b
withMVar m io = block $ do a <- takeMVar m
                           b <- Exception.catch (unblock (io a))
                                (\e -> do putMVar m a; throw e)
                           putMVar m a
                           return b
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()      Stejně jako withMVar exception safe
modifyMVar  :: MVar a -> (a -> IO (a, b)) -> IO b

test1 m = do threadDelay 3000000; putStrLn "b"; putMVar m ()
test2 m = do threadDelay 1000000; putStrLn "a"; putMVar m ()
test3 m = do threadDelay 5000000; putStrLn "c"; putMVar m ()

w1 = do m1 <-newEmptyMVar; m2 <-newEmptyMVar; m3 <-newEmptyMVar
        forkIO (test1 m1); forkIO (test2 m2); forkIO (test3 m3)
        takeMVar m1;       takeMVar m2;       takeMVar m3

                              Control.Concurrent.QSem
                              -----------------------
data QSem
newQSem   :: Int -> IO QSem
waitQSem  :: QSem -> IO ()
signalQSem:: QSem -> IO ()

Střeva QSemu
newtype QSem = QSem (MVar (Int, [MVar ()]))
newQSem init = do sem <- newMVar (init,[])
                  return (QSem sem)
waitQSem (QSem sem) = do (avail,blocked) <- takeMVar sem  -- gain ex. access
                         if avail > 0 then putMVar sem (avail-1,[])
                          else do block <- newEmptyMVar
                                  putMVar sem (0, blocked++[block])
                                  takeMVar block
signalQSem (QSem sem) = do (avail,blocked) <- takeMVar sem
                           case blocked of [] -> putMVar sem (avail+1,[])
                                           (block:blocked') -> do
                                             putMVar sem (0,blocked')
                                             putMVar block ()
```

Control.Concurrent.Chan

```
data Chan a
newChan    :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
isEmptyChan :: Chan a -> IO Bool
```

Další střeva *QSem*u

```
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
type Stream a = MVar (ChItem a)
data ChItem a = ChItem a (Stream a)
newChan = do hole  <- newEmptyMVar
             read  <- newMVar hole
             write <- newMVar hole
             return (Chan read write)
writeChan (Chan _read write) val = do new_hole <- newEmptyMVar
                                      modifyMVar_ write $ \old_hole -> do
                                        putMVar old_hole (ChItem val new_hole)
                                        return new_hole
readChan (Chan read _write) = do modifyMVar read $ \read_end -> do
                                   (ChItem val new_read_end) <-readMVar read_end
                                   return (new_read_end, val)
```

Ale co synchronizace?

```
inc::IORef Int->QSem->IO ()
inc x s = do v <- readIORef x
             writeIORef x (v + 1)
             signalQSem s
doInc::Int -> IORef Int -> IO ()
doInc n r = do s <- newQSem
               sequence_ (replicate n $ forkIO (inc r s))
               sequence_ (replicate n $ waitQSem s)
w2 = do r <- newIORef 0
        doInc 100000 r
        x <- readIORef r
        putStrLn (show x)
```

Control.Monad.STM

```
data STM a

atomically:: STM a -> IO a
retry     :: STM a
orElse    :: STM a -> STM a -> STM a
catchSTM  :: STM a -> (Exception -> STM a) -> STM a

data TVar a
newTVar  :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar:: TVar a -> a -> STM ()
```

Synchronizace

```
incS::TVar Int->QSem->IO ()
incS x s = do atomically $ do v <- readTVar x
                              writeTVar x (v + 1)
                              signalQSem s 1
doIncS::Int -> TVar Int -> IO ()
doIncS n r = do s <- newQSem 0
                sequence_ (replicate n $ forkIO (incS r s))
                sequence_ (replicate n $ waitQSem s)
w3 = do r <- atomically (newTVar 0)
        doIncS 100000 r
        x <- atomically (readTVar r)
        putStrLn (show x)
```

Čekání na událost

```
produce::TVar [Int]->Int->IO ()
produce q n = do atomically $ do s <- readTVar q
                                 writeTVar q (n : s)
consume::TVar Int->TVar [Int]->IO ()
consume e q = do s <- atomically $ do l <- readTVar q
                                      if length l < 100 then retry
                                          else do writeTVar q (drop 100 l)
                                                  return (take 100 l)
                 putStrLn (show $ sum s)
                 atomically $ do k <- readTVar e
                                 writeTVar e (k-1)
w4 = do s <- atomically (newTVar [])
        e <- atomically (newTVar 10)
        sequence_ (replicate 10 $ (forkIO $ consume e s))
        mapM_ (\n -> forkIO (produce s n)) [1..1000]
        atomically $ do ne <- readTVar e
                        if ne /= 0 then retry else return ()
```

Ukončování programu –– jakmile skončí main, skončí všechno
Možnosti úpravy

```
myForkIO :: IO () -> IO (MVar ())
myForkIO io = do mvar <- newEmptyMVar
                 forkIO (io `finally` putMVar mvar ())
                 return mvar
```

Nebo ještě pohodlněji

```
children :: MVar [MVar ()]
children = unsafePerformIO (newMVar [])

waitForChildren = do cs <- takeMVar children
                     case cs of
                       []   -> return ()
                       m:ms -> do putMVar children ms
                                  takeMVar m
                                  waitForChildren

forkChild :: IO () -> IO ThreadId
forkChild io = do mvar <- newEmptyMVar
                  childs <- takeMVar children
                  putMVar children (mvar:childs)
                  forkIO (io `finally` putMVar mvar ())
main = ... waitForChildren
```