



Advanced Functional Programming

Week 11 – Concurrent Channels, Asynchronous Actions, Cancellations and Timeouts

René Thiemann

Department of Computer Science

Last Week

- parallelism
 - use multiple cores to speed up computation
 - high-level interface via strategies
 - separate what is computed from how it is computed
 - `expr `using` rpar` – evaluate `expr` in parallel to WHNF
 - `expr `using` parList rseq` – evaluate each list element in parallel to WHNF
 - `expr `using` parList rdeepseq` – evaluate each list element in parallel to normal form
 - underlying mechanism: `runEval` and `Eval`-monad
 - example: parallel quicksort
- concurrency
 - separate threads for different tasks
 - thread creation via `forkIO`
 - low-level communication via `MVars`
 - blocking operations `takeMVar` and `getMVar`
 - if main thread ends, then all other threads will be stopped
 - example: logger thread with one-message buffer

Higher Level Interfaces for Concurrency – Channels

Channels

- design of `MVar a`: store at most **one** value of type **a**
- aim: design a channel, i.e., an arbitrary length FIFO buffer
- advantage: in logger application, sending some log-message is not blocking, even if there are pending log-messages
- data structure design
 - single linked list
 - all references in the list will be `MVars`
 - references to both ends of the list
- data structure in Haskell

```
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
```

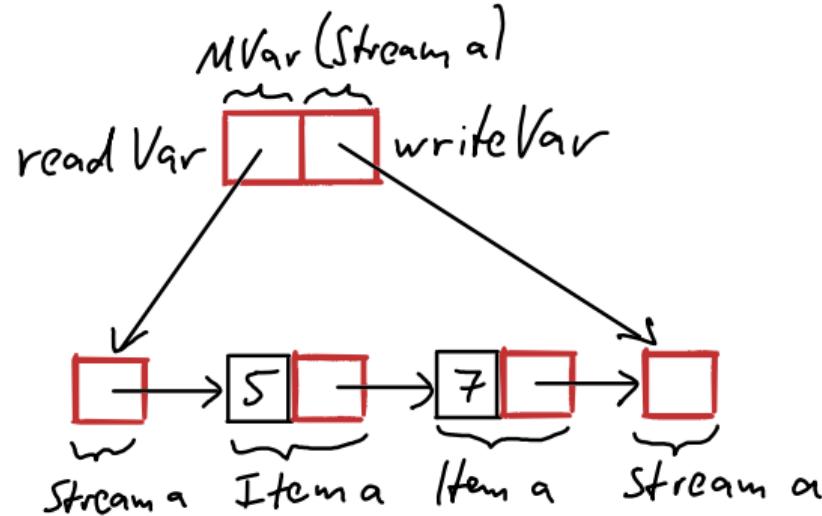
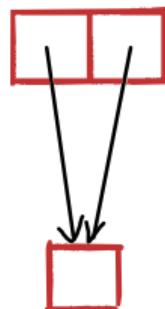
```
data Chan a = Chan {
    readVar :: MVar (Stream a),
    writeVar :: MVar (Stream a)
}
```

Channels Illustrated

- data structure

```
type Stream a = MVar (Item a)  
data Item a = Item a (Stream a)  
data Chan a = Chan { readVar, writeVar :: MVar (Stream a) }
```

- left: empty channel; right: channel with elements 5 and 7;
black: normal values; red: MVars



Channel Operations

```
newChan :: IO (Chan a)
newChan = do
  hole <- newEmptyMVar
  rVar <- newMVar hole
  wVar <- newMVar hole
  return $ Chan { readVar = rVar, writeVar = wVar }

writeChan :: Chan a -> a -> IO ()
writeChan c val = do
  newHole <- newEmptyMVar
  oldHole <- takeMVar (writeVar c)
  putMVar oldHole $ Item val newHole
  putMVar (writeVar c) newHole

readChan :: Chan a -> IO a
readChan c = do
  rVar <- takeMVar (readVar c)
  Item x next <- takeMVar rVar
  putMVar (readVar c) next
  return x
```

Example Application: Improved Logger

- adjusting the logger to use a channel is trivial: use `Chan`-operations instead of `MVar`-operations

- old code

```
newtype Logger =  
  Logger (MVar LogCommand)  
initLogger = do  
  m <- newEmptyMVar  
  ...  
  loop = do  
    cmd <- takeMVar m  
    ...  
    logMessage (Logger m) s  
    = putMVar m (Message s)  
  logStop (Logger m) = do  
    s <- newEmptyMVar  
    putMVar m (Stop s)  
    takeMVar s
```

- new code

```
newtype Logger =  
  Logger (Chan LogCommand)  
initLogger = do  
  c <- newChan  
  ...  
  loop = do  
    cmd <- readChan c  
    ...  
    logMessage (Logger c) s  
    = writeChan c (Message s)  
  logStop (Logger c) = do  
    s <- newEmptyMVar  
    writeChan c (Stop s)  
    takeMVar s
```

Testing the Logger (cabal run Demo11 -- logger)

- code for testing the modified logger

```
message s i = "message " ++ show i ++ " of " ++ s
announceLogMessage l m = do
  putStrLn $ "sending message to logger: " ++ m
  logMessage l m

mainLogger = do
  l <- initLogger
  forkIO $ mapM_ (announceLogMessage l . message "fork 1") [1..100]
  forkIO $ mapM_ (announceLogMessage l . message "fork 2") [1..100]
  mapM_ (announceLogMessage l . message "main thread") [1..100]
  logStop l
```

- `announceLogMessage` immediately prints a message, before it is send to logger
- in total, three threads send 100 messages each
- logger starts its main loop with 2 seconds delay (delay inserted into Logger-code)
- result: all "sending message..." outputs are immediately done, no blocking

Extending the Channel-Code – Multicasts

- channel code also supports multicast-operations, i.e., one writer and several readers
- preparation: `readMVar` in order to read, but not consume some content in an `MVar`

```
readMVar :: MVar a -> IO a
readMVar m = do
    x <- takeMVar m
    putMVar m x
    return x
```

- duplication of channel for multicasts

- both channels will read all upcoming write operations of either channel
- duplicated channel will initially be empty

```
dupChan :: Chan a -> IO (Chan a)
dupChan c = do
    hole <- readMVar (writeVar c)
    newRVar <- newMVar hole
    return $ Chan { readVar = newRVar, writeVar = writeVar c }
```

- in implementation of `readChan`, operation `takeMVar` has to be replaced by `readMVar`

Testing Channel Duplication (cabal run Demo11 -- channel)

- testing code

```
mainChannel = do
  c <- newChan
  mapM_ (writeChan c) ['a' .. 'k']
  d <- dupChan c
  forkIO $ do
    mapM_ (writeChan c) ['l' .. 's']
    forever (readChan c >>= \ a -> putStrLn $ "read from c: " ++ [a])
  forkIO $ do
    mapM_ (writeChan d) ['t' .. 'z']
    forever (readChan d >>= \ a -> putStrLn $ "    read from d: " ++ [a])
  threadDelay $ 1000
```

- letters a..k are only in channel **c**, they will not be copied to **d**
- letters l..s are send to **c** and will become visible in both channels
- letters t..z are send to **d** and will become visible in both channels
- main thread stops execution after 1ms and kills both forked threads
- result: a..z are received via **c**, l..z via **d**, but order of l..z is not fixed, might be ltmunv...

Final Remarks on MVars and Channels

- operation `readMVar` is already predefined
 - predefined version differs from presented implementation: it is ensured that `takeMVar` and `putMVar` operation are performed atomically
 - consequence: no possibility that thread is interrupted between these two operations in the predefined version
- Chan `a` is also predefined
 - `https://hackage.haskell.org/package/base/docs/Control-Concurrent-Chan.html`
 - package offers one further primitive for getting full channel content as lazy list (similar to `readFile` and `hGetContents`)
`getChanContents :: Chan a -> IO [a]`

Higher Level Interfaces for Concurrency – ASync

Aim: Asynchronous I/O

- task: perform asynchronous I/O
 - I/O is performed in background while main thread is doing other tasks
 - running example: download some websites in the background
 - utilized interface based on `Network.HTTP.Conduit` (requires some cabal packages)
`getURL :: String -> IO ByteString`
- first implementation is based on `forkIO` and `MVar`

Asynchronous I/O via forkIO and MVar: cabal run Demo11 -- url1

- source code

```
mainGetURL1 = do
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Red"
    putMVar m1 r
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Green"
    putMVar m2 r
  r1 <- takeMVar m1
  r2 <- takeMVar m2
  print (B.length r1, B.length r2)      -- B = ByteString
```

- code is rather verbose
- try to abstract pattern for **asynchronous action** execution

An Interface for Asynchronous Actions

- interface should provide a way to turn I/O-actions into asynchronous actions
- also waiting on results should be possible
- implementation works by synchronization on some `MVar`

```
data Async a = Async (MVar a)
```

```
async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyMVar
  forkIO (do r <- action; putMVar var r)
  return (Async var)
```

```
wait :: Async a -> IO a
wait (Async var) = readMVar var
```

- `readMVar` instead of `takeMVar`, so that multiple waits are supported

Change of Application (cabal run Demo11 -- url2)

- application code becomes much cleaner

```
mainGetURL2 = do
  a1 <- async $ getURL "http://www.wikipedia.org/wiki/Red"
  a2 <- async $ getURL "http://www.wikipedia.org/wiki/Green"
  -- do something in between
  r1 <- wait a1
  r2 <- wait a2
  print (B.length r1, B.length r2)
```

Combined with Other Monadic Combinators (cabal run Demo11 -- url3)

- process list of websites, include time information

```
timeit a = do start <- getCurrentTime; x <- a; end <- getCurrentTime;
               return (x, end `diffUTCTime` start)

timeDownload url = do
  (page, time) <- timeit $ getURL url
  putStrLn $ "downloaded " ++ url
  ++ " (" ++ show (B.length page) ++ " bytes, " ++ show time ++ ")"

sites = ["http://www.bing.com", ..., "http://www.duckduckgo.com"]

mainGetURL3 = do
  as <- mapM (async . timeDownload) sites -- start concurrent download
  mapM_ wait as -- and wait on completion
```

Error Handling with Async (cabal run Demo11 -- url3bad)

- let us modify the list of websites, so that some website is not existing (or disable internet connection, or cause some other problem leading to an exception)

```
sitesBad = ["http://www.bing.com",
            "http://someurlThatDoesNotExist",
            "http://www.metager.de",
            "http://www.duckduckgo.com"]
```

```
mainGetURL3bad = do
  as <- mapM (async . timeDownload) sitesBad
  mapM_ wait as
```

- execution results in deadlock

```
downloaded http://www.bing.com (52477 bytes, 0.201074s)
```

```
... exception error message: ConnectionFailure ...
```

```
Demo11: thread blocked indefinitely in an MVar operation
```

- reason: because of exception during download action, `putMVar` is not executed in `async`

Error Handling with Async – Extend Async (1/1)

- aims
 - forward exceptions in asynchronous actions to thread that invokes `wait`
 - ensure that exceptions do not lead to deadlock, by always filling `MVars` of `async`
- solution: modify and extend `Async`

```
data Async a = Async (MVar (Either SomeException a))
```

```
async :: IO a -> IO (Async a)
```

```
async action = do
  var <- newEmptyMVar :: MVar (Either SomeException a)
  forkIO $ do { r <- try action; putMVar var r }
  return $ Async var
```

```
waitCatch :: Async a -> IO (Either SomeException a)
```

```
waitCatch (Async var) = readMVar var
```

Error Handling with Async – Extend Async (2/2)

- we also modify `wait` in a way that exceptions from the forked thread are re-thrown in the thread that invokes `wait`

```
waitCatch :: Async a -> IO (Either SomeException a) -- previous slide
```

```
wait :: Async a -> IO a
wait a = do
  r <- waitCatch a
  case r of
    Left e  -> throwIO e
    Right a -> return a
```

Merging of Asyncs

- situation
 - assume there are multiple asynchronous actions
 - aim: wait until the **first one** is completed
 - task: integration into **Async**-framework
- solution via one more **MVar**
- for each asynchronous action, a new thread is created that tries to write into this **MVar**
- implementation in Haskell

```
waitAny :: [Async a] -> IO a
waitAny as = do
  m <- newEmptyMVar
  let forkWait a = forkIO $ do r <- try (wait a); putMVar m r
  mapM_ forkWait as
  wait (Async m)
```

Application for Merging of Asyncs (cabal run Demo11 -- url5)

- application stays on high level

```
mainGetURL5 = do
  let download url = (,) url <$> getURL url
  as <- mapM (async . download) sites
  (url, r) <- waitAny as
  putStrLn $ url ++ " was first (" ++ show (B.length r) ++ " size)"
```

- remarks

- `waitForAny` really just **waits** on any asynchronous action to complete
- the other actions are not aborted, but will continue to run in the background
- if `main = mainGetURL5` then this effect will not be visible, since the main thread stops soonish after invoking `waitForAny` and then the runtime system stops all other threads

Cancellation and Timeouts

Cancellation of Tasks

- cancellations of tasks may be desirable for several reasons
 - user of web browser clicks “stop”-button, e.g., to stop downloads
 - prover spawns several alternative search algorithms to find a successful proof; as soon as first search algorithm is successful, the other searches should be stopped
- two parties
 - (C) a controller thread that wants to cancel some other thread
 - (W) a worker thread, that should be cancelled
- two cancellation policies
 - (P) polling: (W) regularly asks (C) whether it should stop
 - (A) asynchronous cancellation: (W) is interrupted by (C) and will be stopped
- tradeoff
 - danger of (P): if (W) does not query regularly, then system becomes unresponsive
 - danger of (A): if (W) is interrupted and immediately killed, then it cannot release locks, close files, kill external spawned processes, etc.
- imperative languages usually take (P) as default: danger of inconsistent state of (A)
- Haskell takes (A) as default: pure computations cannot poll

Asynchronous Exceptions

- exception handling has been handled before
- however, there are two kinds of exceptions
 - synchronous exceptions
 - occurrence is anticipated
 - example: if code performs `readFile`, it is clear that this might lead to an I/O-exception
 - asynchronous exceptions
 - these are raised by a different thread and are not anticipated
 - example: code that just computes some complex function and then prints the result does not expect any exception
- in Haskell, asynchronous exceptions can be thrown via

```
throwTo :: Exception e => ThreadId -> e -> IO ()
```

- `ThreadId` is obtained from `forkIO`
- `throwTo tid` has no effect, if thread `tid` is already finished

Extending Async Again for Cancellations

- aim: implement `cancel :: Async a -> IO ()`
- solution: extend datatype `Async` by `ThreadId`

```
data Async a = Async ThreadId (MVar (Either SomeException a))
```

```
cancel (Async t var) = throwTo t ThreadKilled
```

```
async :: IO a -> IO (Async a)
```

```
async action = do
  var <- newEmptyMVar
  t <- forkIO $ do { r <- try action; putMVar var r }
  return $ Async t var
```

- `ThreadKilled` exception is usually used for cancelling threads
- note: this version of `Async` is available in module `Control.Concurrent.Async`
- also available: `waitAnyCancel :: [Async a] -> IO (Async a, a)`, like `waitAny`, but with cancellation of remaining asynchronous actions

Asynchronous Exceptions for Timeouts

- aim: run some IO action within a given time limit

```
timeout :: Int -> IO a -> IO (Maybe a)
```

- implementation available in module System.Timeout
- semantics

- `timeout t m` is `Just <$> m`, provided the result is computed within `t` microseconds (approximately)
- `timeout t m` is `Nothing`, if timeout occurs

- implementation is based on asynchronous exceptions
 - a separate thread is spawned, which throws a timeout exception after delay `t`
 - this exception is caught and turned into a `Nothing` result

Catching Asynchronous Exceptions

- module `Control.Exception` provides high-level functions that take care of releasing some resource, even in case of (asynchronous) exceptions
- we illustrate `bracket` in more detail
- `bracket :: IO a` (require resource)
 `-> (a -> IO b)` (finally release resource)
 `-> (a -> IO c)` (compute in-between)
 `-> IO c` (result of in-between computation)
- if an exception occurs, the release code is executed and then the exception is re-thrown
- example
`bracket (openFile "filename" ReadMode) hClose`
 `(\ handle -> do { ... })`
- further high-level exception handling functions
 - `bracketOnError` is like `bracket`, but release only happens if exception occurs
 - `finally`, `onException`, ... are specialized versions of `bracket(onError)`

Application

- with functions like `bracket` and `timeout` and `waitAnyCancel` it is now possible to implement sophisticated search-strategies, e.g., in termination proof search
- example
 - search in parallel for some LPO and some other termination order (for at most 5 seconds)
 - with 2 seconds delay, try tree-automata based termination techniques (for at most 10 seconds)
 - take the first successful result of any of the above techniques
 - iterate this process until either a full termination proof has been established, or all techniques fail
- `bracket` and similar functions should be used to reliably kill externally spawned processes if the own thread is cancelled

Literature

- Simon Marlow, Parallel and Concurrent Programming in Haskell, 2013, O'Reilly, Chapters 7 – 9
- <https://hackage.haskell.org/package/base/docs/Control-Concurrent-Chan.html>
- <https://hackage.haskell.org/package/async/docs/Control-Concurrent-Async.html>
- <https://hackage.haskell.org/package/base/docs/Control-Exception.html>