

Haskell in the Real World

Stefan Wehr (`wehr@cp-med.com`) and Emin Karayel
(`karayel@cp-med.com`)
factis research GmbH

4. September 2014

- ▶ Presentation
 - ▶ Introduction of a messaging system writing in Haskell
 - ▶ Introduction to important Haskell technologies
- ▶ Hacking
 - ▶ Write your own extension of the messaging system

Who am I? What are we doing?

- ▶ Haskell user since 2013
- ▶ Started in academia
- ▶ Since 2010: using Haskell in industry
- ▶ *factis research*, Freiburg, Germany
 - ▶ Software for the healthcare market
 - ▶ Server-side software written in Haskell
 - ▶ Vast experience with complex mobile applications

Why functional? Why Haskell?

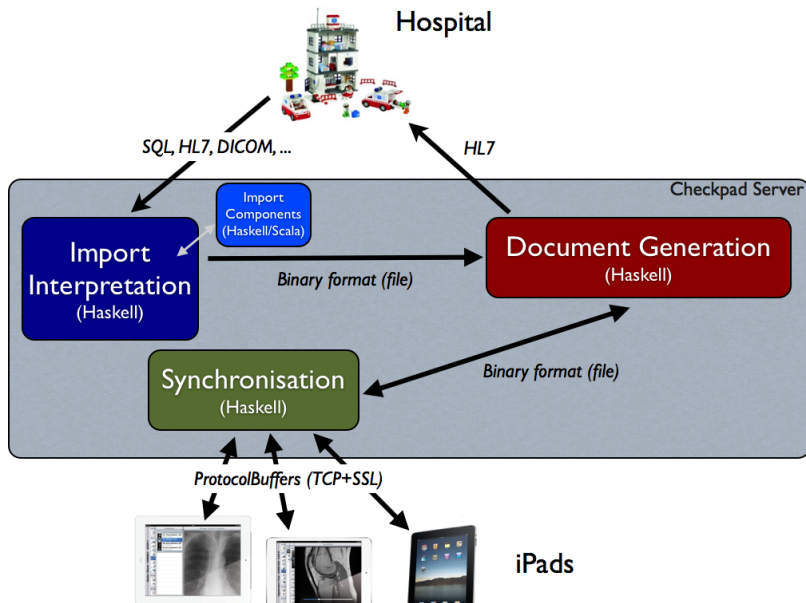
- ▶ Controllable side-effects
- ▶ Modularity
- ▶ Testability
- ▶ Short but readable code
- ▶ Abstraction made easy
- ▶ Reusability
- ▶ Expressive type-system: if the program compiles it works ;-)
- ▶ “World’s finest imperative programming language”
- ▶ Great support for parallel and concurrent programming
- ▶ “Smart people”

What to expect today?

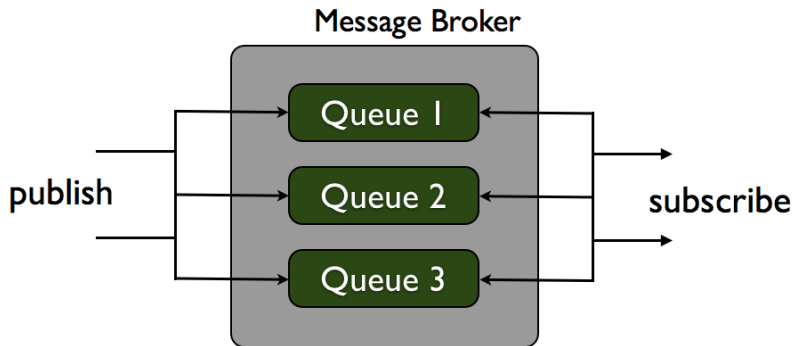
- ▶ Have a look at a commercial software product written in Haskell
- ▶ Experience from more than 4 years of commercial software development in Haskell
- ▶ Example: Messaging System
 - ▶ Serialization and persistence
 - ▶ Network programming
 - ▶ Concurrent programming
 - ▶ Testing
 - ▶ Logging
 - ▶ Build system

- ▶ *Checkpad MED*: electronic health record for the iPad/iPhone
 - ▶ Give doctors instant access to all medical data
 - ▶ Support collaboration
 - ▶ Independent from the Hospital Information System
 - ▶ Demo
- ▶ Under development since 2010
- ▶ Today: several paying customers, several proof-of-concept installations

Architecture



Messaging System



- ▶ Code at github: <https://github.com/factisresearch/mq-demo>

Key data type: MessageBroker

```
data MessageBroker q
  = MessageBroker
    { mb_subscribeToQueue ::
        q -> Subscriber -> STM SubscriberId
    , mb_unsubscribeFromQueue ::
        q -> SubscriberId -> STM ()
    , mb_publishMessage ::
        q -> Message -> IO ()
    , mb_lookupQueue ::
        QueueName -> STM (Maybe q)
    , mb_knownQueues :: STM [QueueName]
    }
```

- ▶ Demo

- ▶ Code sample:

server/src/lib/Mgw/MessageQueue/Types.hs

Concurrency with STM

- ▶ Simple example: bank accounts
 - ▶ `transfer acc1 acc2 amount`: transfer amount from account acc1 to account acc2
 - ▶ transfer should be thread-safe
 - ▶ In our example: bank accounts live only in memory
- ▶ Problems with threads
 - ▶ Deadlocks, Race conditions
 - ▶ Break modularity
- ▶ The idea behind STM
 - ▶ Declare which parts of your code are “atomic”
 - ▶ The runtime system ensures atomicity (similar to database transactions)
- ▶ Advantages of Haskell:
 - ▶ Immutability
 - ▶ Lazyness

Bank account transfer with STM

```
transfer :: Account -> Account -> Int -> IO ()
transfer acc1 acc2 amount =
    atomically (do deposit acc2 amount
                  withdraw acc1 amount)
```

- ▶ `atomically :: STM a -> IO a`
 - ▶ Executes the given action atomically
 - ▶ Argument to `atomically` is often called “transaction”
 - ▶ `STM a`: type of a transaction with result type `a`
 - ▶ `STM` is a monad

Transaction variables

- ▶ STM-actions communicate via transaction variables
 - ▶ `TVar a`: transaction variable for values of type `a`
- ▶ `readTVar :: TVar a -> STM a`
- ▶ `writeTVar :: TVar a -> a -> STM ()`

```
type Account = TVar Int
```

```
deposit :: Account -> Int -> STM ()
```

```
deposit acc amount =
```

```
    do bal <- readTVar acc
```

```
       let !newBal = bal + amount
```

```
       writeTVar acc newBal
```

```
withdraw :: Account -> Int -> STM ()
```

```
withdraw acc amount = deposit acc (- amount)
```

Blocking with STM

- ▶ Concurrent programming often requires waiting for a certain condition
- ▶ Example: `limitedWithdraw` should block until there is enough money on the account

```
limitedWithdraw :: Account -> Int -> STM ()  
limitedWithdraw acc amount =  
    do bal <- readTVar acc  
       if amount > 0 && amount > bal  
       then retry  
       else writeTVar acc (bal - amount)
```

- ▶ `retry :: STM a`
 - ▶ Aborts the current transaction
 - ▶ Retries it later (if there's a chance it could succeed)

Combining transactions

- ▶ Example: withdraw money from the first account with enough money on it

```
limitedWithdrawMany :: [Account] -> Int -> STM ()  
limitedWithdrawMany [] _ = retry  
limitedWithdrawMany (acc:rest) amount =  
    limitedWithdraw acc amount 'orElse'  
    limitedWithdrawMany rest amount
```

- ▶ `orElse :: STM a -> STM a -> STM a`
 - ▶ `orElse t1 t2` first executes `t1`
 - ▶ If `t1` is successful, so is `orElse t1 t2`
 - ▶ If `t1` aborts (by calling `retry`), then `t2` is executed
 - ▶ If `t2` retries, so does `orElse t1 t2`

Summary of the STM API

```
atomically :: STM a -> IO a
```

```
retry      :: STM a
```

```
orElse     :: STM a -> STM a -> STM a
```

```
newTVar    :: a -> STM (TVar a)
```

```
readTVar   :: TVar a -> STM a
```

```
writeTVar  :: TVar a -> a -> STM ()
```

Type system prevents disaster (1)

- ▶ Cannot re-execute IO actions
- ▶ Type system prevents you from executing IO actions inside a STM transaction

```
launchMissiles :: IO ()  
launchMissiles = -- ...
```

```
bad xv yv =  
    atomically (do x <- readTVar xv  
                  y <- readTVar yv  
                  when (x > y) launchMissiles)
```

```
$ ghc Bad.hs
```

```
Bad.hs:12:25:
```

```
    Couldn't match type 'IO' with 'STM'
```

```
    Expected type: STM ()
```

```
    Actual type: IO ()
```


Type system prevents disaster (2)

- ▶ Accessing transaction variables outside of transactions easily leads to race conditions
- ▶ Type system allows access to transaction variables only from within a transaction

```
doSomethingBad :: TVar Int -> IO ()  
doSomethingBad v =  
    do x <- readTVar v  
       writeTVar v (x + 1)
```

Bad2.hs:5:13:

```
Couldn't match type 'STM' with 'IO'  
Expected type: IO Int  
Actual type: STM Int
```

Bad2.hs:6:8:

```
Couldn't match type 'STM' with 'IO'
```

- ▶ Avoid storing thunks in TVars or make sure to force the thunks directly after the transaction
 - ▶ Otherwise, you get memory leaks
- ▶ Use `runTx` instead of `atomically`
 - ▶ Emits a warning if a transaction retries very often
 - ▶ It's on hackage: `stm-stats` (`runTx` is called `trackSTM` there)

Example: Executing STM



Thread A: `transfer acc1 acc2 50`

Example: Executing STM



Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
```

Log A

acc2: read 50

Example: Executing STM



Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50))
```

Log A

acc2: read 50

acc2: write 100

Example: Executing STM



Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
```

Log A

acc2: read 50

acc2: write 100

acc1: read 100

Example: Executing STM



Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log A

acc2: read 50

acc2: write 100

acc1: read 100

acc1: write 50

Example: Executing STM



Thread A: transfer acc1 acc2 50

Thread B: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log A

acc2: read 50

acc2: write 100

acc1: read 100

acc1: write 50

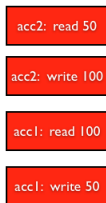
Example: Executing STM



Thread A: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

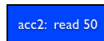
Log A



Thread B: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
```

Log B



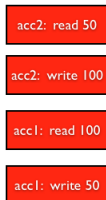
Example: Executing STM



Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

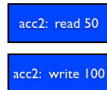
Log A



Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50))
```

Log B



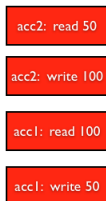
Example: Executing STM



Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

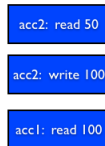
Log A



Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1)
```

Log B



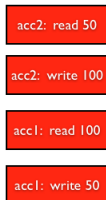
Example: Executing STM



Thread A: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log A

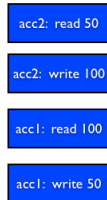


Validation

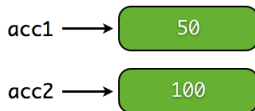
Thread B: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log B



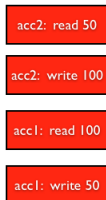
Example: Executing STM



Thread A: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log A

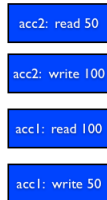


Commit

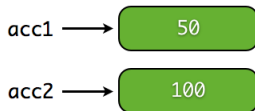
Thread B: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log B



Example: Executing STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log B

acc2: read 50

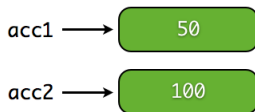
acc2: write 100

acc1: read 100

acc1: write 50

**Validation
schlägt
feh!**

Example: Executing STM



Thread A

Thread B: `transfer acc1 acc2 50`

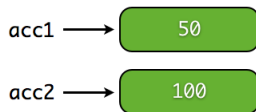
`atomically (do bal2 <- readTVar acc2`

Rollback

Log B

`acc2: read 100`

Example: Executing STM



Thread A

Thread B: `transfer acc1 acc2 50`

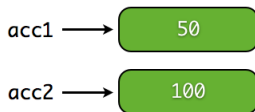
```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (100 + 50))
```

Log B

acc2: read 100

acc2: write 150

Example: Executing STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (100 + 50)
               bal1 <- readTVar acc1
```

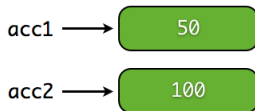
Log B

acc2: read 100

acc2: write 150

acc1: read 50

Example: Executing STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (100 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (50 - 50))
```

Log B

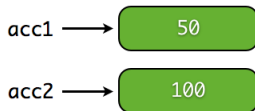
acc2: read 100

acc2: write 150

acc1: read 50

acc1: write 0

Example: Executing STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (100 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (50 - 50))
```

Log B

acc2: read 100

acc2: write 150

acc1: read 50

acc1: write 0

Validation

Example: Executing STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (100 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (50 - 50))
```

Log B

acc2: read 100

acc2: write 150

acc1: read 50

acc1: write 0

Commit

Executing STM, Summary

- ▶ Execute atomically `t` optimistically, without locks
- ▶ `writeTVar v x` writes to the log, not into memory
- ▶ `readTVar v` reads first from the log, and if `v` is not in the log from the memory
- ▶ If `readTVar v` reads a value from memory, it records the value read in the log.
- ▶ Validation at the end of a transaction
 - ▶ Must be atomic (typically uses locks)
 - ▶ Validation is successful if all values read during the transaction are still consistent with the main memory.
- ▶ On successful validation: write values to main memory
- ▶ `retry` uses the log to check whether it should re-run the transaction

Local MessageBroker

```
createLocalBroker :: LogId
                  -> Maybe FilePath
                  -> [(QueueName, QueueOpts)]
                  -> IO (MessageBroker Queue)

data QueuePersistence
    = PersistentQueue | TransientQueue
    deriving (Eq, Show)

data QueueOpts
    = QueueOpts { go_persistence :: QueuePersistence }
    deriving (Eq, Show)
```

► Code:

server/src/lib/Mgw/MessageQueue/LocalBroker.hs

Writing tests with HTF

- ▶ Package HTF on Hackage
- ▶ Automatically collects your unit tests and QuickCheck properties
- ▶ Error messages contain file name and line number
- ▶ Diff for failing equality assertions
- ▶ Replay of failing QuickCheck properties
- ▶ Parallel execution of tests
- ▶ Machine-readable output (if desired)

Serialization with SafeCopy

```
{-# LANGUAGE TemplateHaskell #-}

deriveSafeCopy 1 'base ''MessageId
deriveSafeCopy 1 'base ''Message

safeDecode :: (Monad m, SafeCopy a)
            => BS.ByteString -> m a
safeEncode :: SafeCopy a => a -> BS.ByteString
```


Migration with SafeCopy

```
data MessageV1 = MessageV1
  { msgV1_id :: !MessageId
  , msgV1_payload :: !BS.ByteString }

data Message = Message
  { msg_id :: !MessageId
  , msg_time :: !(Option ClockTime)
  , msg_payload :: !BS.ByteString }

deriveSafeCopy 1 'base ''MessageId
deriveSafeCopy 1 'base ''MessageV1
deriveSafeCopy 2 'extension ''Message
instance Migrate Message where
  type MigrateFrom Message = MessageV1
  migrate msg =
    Message { msg_id = msgV1_id msg
            , msg_payload = msgV1_payload msg
            , msg_time = None }
```

Lazyness and memory leaks

- ▶ Thunks (expression not yet evaluated) may lead to memory leaks
- ▶ Difficult to find
- ▶ Quiz: Does the following code have a memory leak?
 - ▶ Assumption: the map of checksums is long-living

```
import qualified Data.ByteString as BS
```

```
md5 :: BS.ByteString -> MD5
```

```
md5 bs = MD5 (md5' bs)
```

```
  where
```

```
    md5' :: BS.ByteString -> BS.ByteString
```

```
    md5' = ...
```

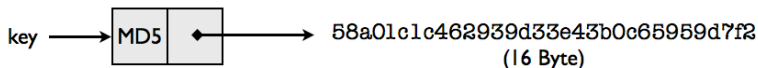
```
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5  
          -> Map.Map Key MD5
```

```
storeMD5 key bs = Map.insert key (md5 bs)
```

No memory leak!

```
import qualified Data.Map.Strict as Map

data MD5 = MD5 { unMD5 :: !BS.ByteString }
```

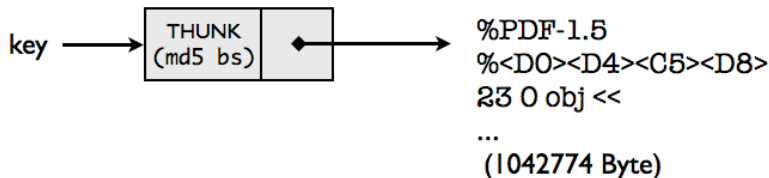


```
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5
          -> Map.Map Key MD5
storeMD5 key bs = Map.insert key (md5 bs)
```

Yes, there is a memory leak!

```
import qualified Data.Map.Lazy as Map

data MD5 = MD5 { unMD5 :: !BS.ByteString }
```

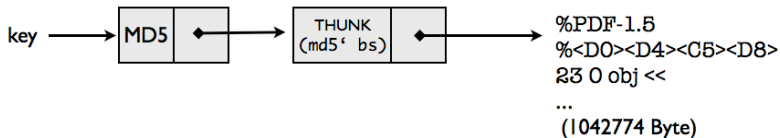


```
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5
          -> Map.Map Key MD5
storeMD5 key bs = Map.insert key (md5 bs)
```

Another memory leak!

```
import qualified Data.Map.Strict as Map

data MD5 = MD5 { unMD5 :: BS.ByteString }
```

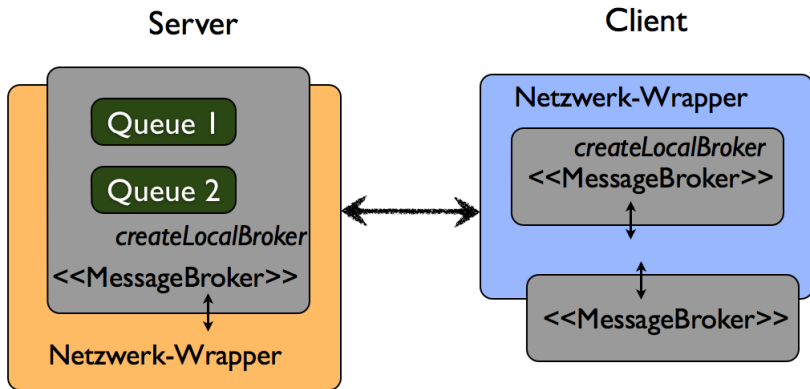


```
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5
          -> Map.Map Key MD5
storeMD5 key bs = Map.insert key (md5 bs)
```

Our convention

- ▶ Long-living data structures must be strict (more precise: deep strict)
- ▶ Requires discipline but works quite good

Server for our MessageBroker



Messages between Client and Server

```
data ServerMessage
  = ServerQueues !(V.Vector QueueName)
  | ServerPublishMessage !QueueName !Message
  deriving (Show, Eq)
```

```
data ClientMessage
  = ClientSubscribe !QueueName
  | ClientPublishMessage !QueueName !Message
  deriving (Show, Eq)
```


Network programming with io-streams

- ▶ `InputStream c`: read-only stream of values of type `c`
 - ▶ `read :: InputStream c -> IO (Maybe c)`
 - ▶ `unRead :: c -> InputStream c -> IO ()`
- ▶ `OutputStream c`: write-only stream of values of type `c`
 - ▶ `write :: Maybe c -> OutputStream c -> IO ()`
 - ▶ Two things to note:
 - ▶ Passing `Nothing` to `write` does not close the underlying resource
 - ▶ Just values supplied after the first `Nothing` are usually discarded, but that's up to the implementation of the `OutputStream`
- ▶ Support for converting `Handle`, `Socket`, parse functions, ... into streams

Simple io-streams example

```
import qualified System.IO.Streams as S
import Data.Maybe

copy :: S.InputStream c -> S.OutputStream c -> IO ()
copy input output = loop
  where
    loop = do mx <- S.read input
              S.write mx output
              if isJust mx then loop else return ()

copyFile :: FilePath -> FilePath -> IO ()
copyFile inputFile outputFile =
  S.withFileAsInput inputFile $ \input ->
  S.withFileAsOutput outputFile $ \output ->
    copy input output
```

Creating output streams

```
makeOutputStream :: (Maybe c -> IO ())  
                  -> IO (OutputStream c)
```

```
import qualified System.IO.Streams as S  
import qualified Data.ByteString as BS  
import System.IO
```

```
handleToOutputStream :: Handle  
                     -> IO (S.OutputStream BS.ByteString)
```

```
handleToOutputStream h = S.makeOutputStream write
```

```
  where
```

```
    write mx = case mx of
```

```
        Nothing -> hFlush h
```

```
        Just x -> BS.hPut h x
```

Creating input streams

```
makeInputStream :: IO (Maybe a)
                -> IO (InputStream a)

import qualified System.IO.Streams as S
import qualified Data.ByteString as BS
import System.IO

handleToInputStream :: Handle
                    -> IO (S.InputStream BS.ByteString)

handleToInputStream h = S.makeInputStream produce
  where
    produce =
      do x <- BS.hGetSome h 32752
      return $! if BS.null x then Nothing
                else Just x
```

The Generator monad

```
yield :: c -> Generator c () -- generates a single output  
fromGenerator :: Generator c a -> IO (InputStream c)  
  
import qualified System.IO.Streams as S  
-- Already available  
fromList :: [c] -> IO (S.InputStream c)  
fromList l = S.fromGenerator (mapM_ S.yield l)
```

Transforming streams, utilities

```
map :: (a -> b) -> InputStream a -> IO (InputStream b)
mapM :: (a -> IO b) -> InputStream a -> IO (InputStream b)
```

```
contramap :: (a -> b) -> OutputStream b
           -> IO (OutputStream a)
contramapM :: (a -> IO b) -> OutputStream b
           -> IO (OutputStream a)
```

```
connect :: InputStream a -> OutputStream a -> IO ()
supply :: InputStream a -> OutputStream a -> IO ()
```

```
atEndOfInput :: IO b -> InputStream a
              -> IO (InputStream a)
atEndOfOutput :: IO b -> OutputStream a
              -> IO (OutputStream a)
```

-- and many more...

Server for MessageBroker: abstracting over the network

- ▶ Function handling a single client in the messaging server

```
runClientHandler ::
```

```
    LogId  
-> MessageBroker q  
-> S.InputStream ClientMessage  
-> S.OutputStream ServerMessage  
-> IO ()
```

- ▶ Code:

```
server/src/lib/Mgw/MessageQueue/BrokerServer.hs
```

Client for MessageBroker

```
createBrokerStub ::  
    LogId  
    -> (TBMChan ServerMessage, TBMChan ClientMessage)  
    -> IO (MessageBroker Queue)
```

- ▶ TBMChan: bounded, closable Channels
- ▶ Channels are connected to `InputStream ServerMessage` and `OutputStream ClientMessage`
- ▶ The resulting `MessageBroker` can handle connection aborts
- ▶ Code:
`server/src/lib/Mgw/MessageQueue/BrokerStub.hs`

Exchanging data with ProtocolBuffers

Google ProtocolBuffers

- ▶ Efficient serialization format
- ▶ Language-neutral, platform-neutral
- ▶ Allows for forward- and backwards-compatibility
- ▶ Packages on Hackage: `hprotoc-fork`,
`protocol-buffers-fork`,
`protocol-buffers-descriptor-fork`

In our code

- ▶ Definitions: `protocols/protos/MessageQueue.proto`
- ▶ Conversions: `server/src/lib/Mgw/Message/Protocol.hs`
- ▶ Generated code:
`server/build/gen-hs/Com/Factisresearch/Checkpad/Protos`

Building build-systems with shake

- ▶ Shake is a Haskell library for building build-systems
 - ▶ No predefined build rules
 - ▶ Rules are given as Haskell code
- ▶ Dynamic dependencies
 - ▶ All dependencies arise during a build
 - ▶ Different from (say) `make`, where all dependencies are fixed at the beginning
- ▶ More powerful but also more complex than `Cabal`
- ▶ See http://community.haskell.org/~ndm/downloads/paper-shake_before_building-10_sep_2012.pdf

shake in action (1)

- ▶ Action-Monad: tracking of dependencies
 - ▶ Example: `readFile' f` introduces a dependency on file `f`

```
cIncludes :: FilePath -> Action [FilePath]
cIncludes x =
    do s <- readFile' x
       return $ mapMaybe parseInclude (lines s)
    where
        parseInclude line =
            do rest <- List.stripPrefix "#include \"" line
               return $ takeWhile (/= "'") rest
```

shake in action (2)

- Rules-Monad: definition of build rules

```
(*>) :: FilePattern -> (FilePath -> Action ()) -> Rules ()
```

```
rules :: Rules ()
```

```
rules =
```

```
    "*.o" *> \out ->
```

```
        do let c = replaceExtension out "c"
```

```
           need (cIncludes c)
```

```
           system' "gcc" ["-o", out, "-c", c]
```

Now it's your turn!

- ▶ Timestamp for messages
 - ▶ Messages should carry a timestamp, indicating their creation time
- ▶ Deregistering subscribers from the server
 - ▶ With the code shown so far, a client never deregisters a subscriber from the server, even if there are no more local subscribers at the client. This is correct but wastes network traffic.
 - ▶ Improve the existing code so that clients send a derigistration message to the server as soon as the last local subscriber deregisters itself.

Now it's your turn!

- ▶ Proxy message server
 - ▶ Write a proxy message server. A proxy message server acts itself as a message server. It connects to several other message server and forwards calls from its client to those servers.
 - ▶ If a client sends a subscription for a queue Q to the proxy server, the proxy server forwards the subscription to all servers supporting queue Q .
 - ▶ Similarly, if a client sends a message for queue Q to the proxy server, the proxy server forwards the message to all servers support queue Q .

Now it's your turn!

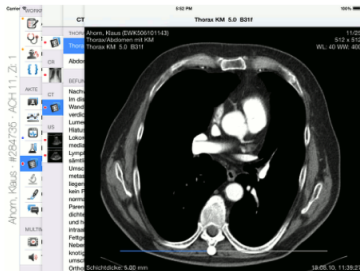
- ▶ Synchronouos client API
 - ▶ With the code shown so far, the API for a message client is asynchronous: a call of `mb_subscribe` or `mb_publish` returns directly after the request has been sent out to the network.
 - ▶ In some situations, it's better to wait until the server has performed the corresponding action (synchronous client API). For example, with a synchronous API, we could get rid of the the calls to `sleepTimeSpan` in the tests.
 - ▶ Implement a synchronous client API.

Summary

- ▶ Haskell is a great tool for developing commercial software
 - ▶ High productivity
 - ▶ Correctness
 - ▶ Security
 - ▶ Reusability
- ▶ Haskell is the “World’s finest imperative programming language”
- ▶ Haskell is efficient



- ▶ Haskell Hackathon in Berlin
- ▶ 26 Sep - 28 Sep 2014
- ▶ Highlights:
 - ▶ Haskell hacking
 - ▶ Discussions with enthusiastic haskell programmers
 - ▶ Talks by Andres Löh and José Pedro Magalhães
 - ▶ Barbecue
- ▶ Details: <http://www.haskell.org/haskellwiki/HacBerlin2014>
- ▶ Please register!



- ✓ Software Engineer (Haskell)
- ✓ Integration Engineer (Haskell)
- ✓ Support Engineer
- ✓ Software QA Engineer
- ✓ Software Engineer (iOS)
- ✓ Software Engineer (Android)
- ✓ DevOps Engineer
- ✓ Unix-Admin
- ✓ Praktikum