IA014: Advanced Functional Programming

10. I/O and Concurrency

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Monadic I/O

Pure programs and I/O

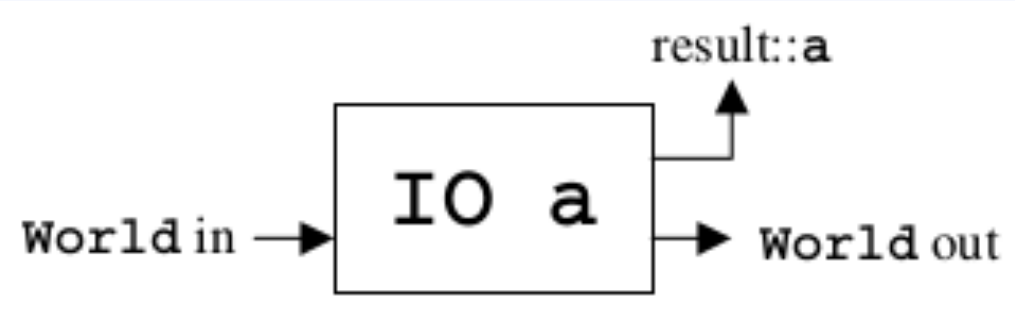
- *pure* functional program (e.g. in HASKELL) implements a *function*
- *no side effects*
- but the purpose of a program is to produce a side effect:
  - produce an output, send a message, modify a screen ...
- the side effects have to be *outside* of the program

Monadic I/O

A value of type `IO a` is an “action” (computation) that may do some input/output before producing a value of type `a`.

The meaning of “do some input/output”: *modify the outside world*

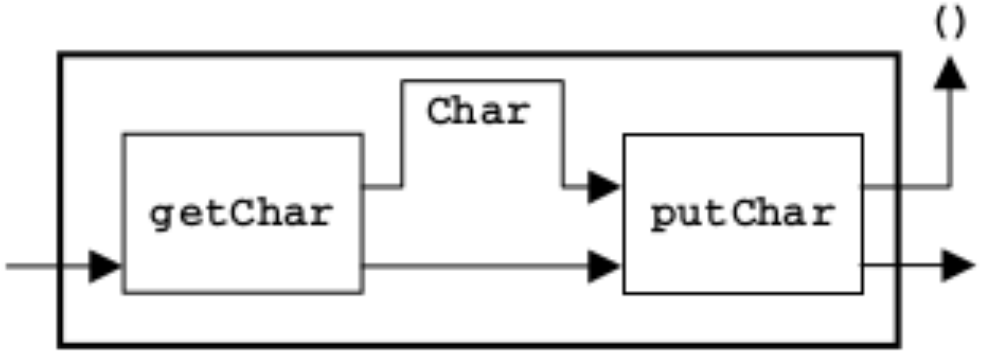
```
type IO a = World -> (a, World)
```



The IO monad

```
return ::  a -> IO a
```


```
(>>=) ::  IO a -> (a -> IO b) -> IO b
```



getChar >>= putChar

```
(>>) ::  IO a -> IO b -> IO b
(>>) a b = a >>= (\x -> b)
```

do notation can be used

Program as an I/O action

In HASKELL

- the whole program defines a single big I/O action.
- its type is `IO ()`, and
- the program is executed by performing the action.

Example:

```
main :: IO ()
main = do xs <- getLine
          putStrLn (reverse xs)
```

The world is treated in a *single-threaded* way:

- `>>=` is the only operation for combining I/O actions
- the world is never duplicated or thrown away
- (and the programmer cannot break this)

Control structures 1/2

Monadic I/O lets us do imperative programming, *including control structures*.

Examples:

- an infinite loop

```
forever :: IO () -> IO ()
forever a = a >> forever a
```

- n-times loop

```
repeatN :: Int -> IO a -> IO ()
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

Why we can do this: *actions as first class values*.

Control structures 2/2

Users are free to define other kinds of control structures.

Examples

Let us execute a sequence of actions:

```
sequence_ :: [IO a] -> IO ()
sequence_     [] = return ()
sequence_ [x:xs] = do x; sequence_ xs
```

Now we can define `for` as:

```
for :: [a] -> (a -> IO ()) -> IO ()
for ns f = sequence_ (map f ns)
```

Exercise:

- define `while :: IO Bool -> IO ()`

References

Goal: to create mutable variables

Idea: `IO` operations provide us with sequentialized input/output

```
data IORef a

newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Example:

```
import Data.IORef
main = do varA <- newIORef 0
          a0 <- readIORef varA
          writeIORef varA 1
          a1 <- readIORef varA
          print (a0, a1)
```

Arrays, hashtables etc. are implemented in the same way

Concurrency

Concurrency vs parallelism

Parallelism

- multiple processors/cores to gain performance
- no communication between processes
- no semantic impact (same result if executed sequentially and in parallel)
- deterministic

Concurrency

- concurrency as part of specification
- concurrent threads, each doing I/O independently
- behaviour is non-deterministic
- substantial semantic impact

We will use CONCURRENT HASKELL
(an extension of HASKELL 2010)

Threads

- *new threads* are created using `forkIO`

```
forkIO :: IO () -> IO ThreadID
```

- newly created thread runs concurrently with other threads
- side effects are *interleaved* with other threads
- in GHC threads are extremely *lightweight*
  (≤ 100 bytes plus stack)
- HASKELL threads are movable – eliminates fragmentation
- GHC automatically spread threads among cores

Threads example – interleaving

```
import Control.Concurrent
import Control.Monad
import System.IO
main = do hSetBuffering stdout NoBuffering
          forkIO (forever(putChar 'A'))
          forkIO (forever(putChar 'B'))
          threadDelay (10^3)
```

`threadDelay :: Int -> IO ()` suspends execution for a given number of microseconds.

One possible output:

```
AABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBABABABABABAB
```

Low level communication: `MVar`

`MVar` is very similar to `IORef`:

```
data MVar a

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
readMVar     :: MVar a -> IO a
```

- `MVar` can hold a value *or be empty*
- `takeMVar` *removes* a value and *blocks* if `MVar` is empty (until `MVar` becomes full)
- `putMVar` *inserts* a value and *blocks* if `MVar` is full (until `MVar` becomes empty)
- waiting threads are in a FIFO queue
- only one thread is woken up at a time

MVar example

```haskell
import Control.Concurrent
import System.Random

main :: IO ()
main = do
    var <- newMVar [] -- MVar to collect the results
    mapM_ (forkIO . task var) ["first", "second", "third"] -- 3 threa
    putStrLn "Press Return to show the results."
    _ <- getLine
    takeMVar var >>= mapM_ putStrLn -- print the results
    where
        task v s = do
            randomRIO (1,10) >>= \r -> threadDelay (r * 10000)
            val <- takeMVar v
            putMVar v (s : val)
```

Various uses of MVar

- *locks*
  - type `MVar ()` is sufficient
  - `takeMVar` acquires the lock, `putMVar` releases
    or the other way round – just be consistent
- one-place *channels*
  can be used for asynchronous communication
- containers for *shared mutable states*

Asynchronous I/O

```
import Control.Concurrent
import Data.ByteString as B
import GetURL

main = do
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar

  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Shovel"
    putMVar m1 r

  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Spade"
    putMVar m2 r

  r1 <- takeMVar m1
  r2 <- takeMVar m2
  print (B.length r1, B.length r2)
```

The async package

```
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
```

Our code can now be written as:

```
main = do
  a1 <- async (getURL "http://www.wikipedia.org/wiki/Shovel")
  a2 <- async (getURL "http://www.wikipedia.org/wiki/Spade")
  r1 <- wait a1
  r2 <- wait a2
  print (B.length r1, B.length r2)
```

Channels

`MVar`s can be used as a building block to construct larger abstractions.

Case study: Channels

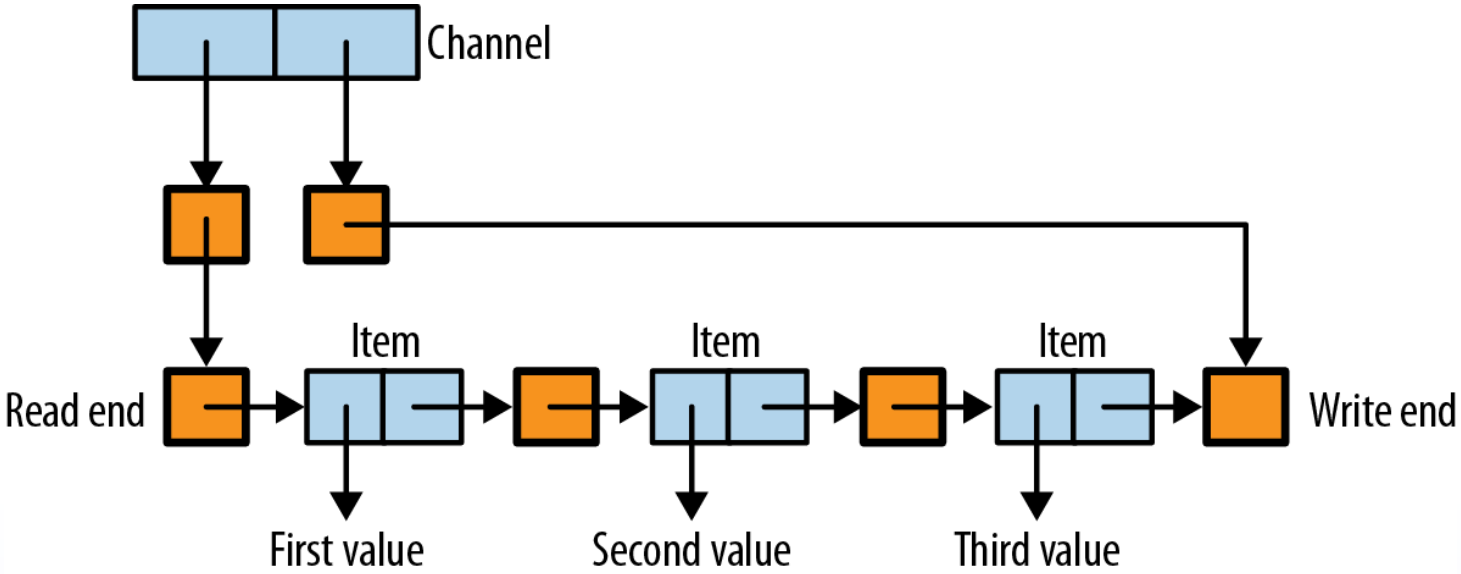
Channel – unlimited buffer for communicating between processes.

Interface:

```
data Chan a

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

Channel implementation



Channel contents:

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

Channel data type:

```
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```

(we have pointers to the first and last element)

Channel access

```
newChan :: IO (Chan a)
newChan = do
  hole  <- newEmptyMVar
  readVar  <- newMVar hole
  writeVar <- newMVar hole
  return (Chan readVar writeVar

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

readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
  stream <- takeMVar readVar
  Item val tail <- takeMVar stream
  putMVar readVar tail
  return val
```

Software Transactional Memory (STM)

Motivation

Problems with lock-based approaches

- races (missing some locks)
- deadlocks (wrong order of locks)
- too conservative use (inhibits concurrency)
- simple typos
- …
- locks do not *compose* well
  (no support for modular programming)

Atomic memory transactions

- alternative to locking
- inspiration in the database world (transactions, ACID)
- mark a block of code which should be atomic

```
atomically <code>
```

- whole block or nothing – **A**tomicity
- execute blocks independently – **I**solation
- no lock induced deadlocks (no locks!)
- easy error recovery

Such approaches are known as

Software Transactional Memory (STM)

How to implement STM?

One possible way:

- execute the atomic block without any locking
  (optimistic synchronization)
- record every memory read and write
- writes are simulated on side
- after the block execution is finished, *validate* the transaction
  - check, that every read variable has the same value as was recorded in the log
  - if valid, commit the changes to variables
  - if not, re-run the atomic block

Simplistic approach to `atomically`

"That's what `IO` is for . . . "

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

Sample use:

```
main = do r <- newIORef 0
          fork (atomically (incRef r))
          atomically (incRef r)
       where
         incRef var = do v <- readIORef var
                         writeIORef var (v+1)
```

Problems:

- What if we forget to add the `atomically` wrapper?
- I/O actions do not work well in this model:

```
atomically (if n>k then launch_missiles)
```

The STM monad

- introduces “tracked” imperative variables (`TVar`)
  (also known as “transactional variables”)
- the monad then “tracks” the transactions

```
data STM a :: * -> *

atomically :: STM a -> IO a

retry  :: STM a
orElse :: STM a -> STM a -> STM a
check  :: Bool -> STM ()

-----

data TVar a

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

Example: banking account

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount = do
    bal <- readTVar acc
    writeTVar acc (bal - amount)

deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)
```

Type system guarantees that `TVar`s cannot be used outside transactions:

```
bad :: Account -> IO ()
bad acc = do hPutStr stdout "Withdrawing..."
             withdraw acc 10
```

```
good :: Account -> IO ()
good acc = do hPutStr stdout "Withdrawing..."
              atomically (withdraw acc 10)
```

Blocking: `retry`

What if there are not sufficient funds?

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

`retry` semantics:

- current transaction is *abandoned* and *retried* at some later time (retrying immediately makes no sense)
- efficient implementation would wait for some write to `acc` (Why `acc`? Easily detected in the transaction log!)

Check & retry

The pattern we used on the previous slide is very common:

> Check a boolean condition, and retry if not satisfied.

```
check :: Bool -> STM ()
check True  = return ()
check False = retry
```

Now we can rewrite `limitedWithdrawal` in a more concise way:

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
    bal <- readTVar acc
    check (amount <= 0 || amount <= bal)
    writeTVar acc (bal - amount)
```

Trying alternative ways: `orElse`

What if we have two accounts, and, if the first one does not have sufficient funds, want to make withdrawal from the second?

The perfect job for `orElse :: STM a -> STM a -> STM a`!

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
limitedWithdraw2 acc1 acc2 amt =
        orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

The semantics of `orElse a1 a2`:

- tries to perform `a1`
- if `a1` retries, tries to perform `a2`
- if `a2` retries, the whole action retries

It’s Christmas!

The Santa Claus problem

Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

Trono, 1994

Reading

- S. Peyton Jones: *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*. Marktoberdorf 2001
- S. Peyton Jones: *Beautiful Concurrency*. Chapter in *Beautiful Code*, 2007.
- S. Marlow: *Parallel and Concurrent Programming in Haskell*. 2012.
- *IO inside.* Haskellwiki. `https://www.haskell.org/haskellwiki/IO_inside`

Original papers:

- T. Harris, S. Marlow, S. Peyton Jones, M. Herlihy: *Composable Memory Transactions.* PPoPP'05.