Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Advanced Functional Programming

2012-2013, periode 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

December 17, 2012

12. Exceptions, Networking, Concurrency

12.1 Handle-based IO

Handles

Most basic IO functions such as getLine, putStr have a cousin that works on an arbitrary **handle**.

System.IO

```
data Handle   -- abstract

-- opening files
openFile       :: FilePath → IOMode → IO Handle
openBinaryFile :: FilePath → IOMode → IO Handle
```

Using handles

System.IO

```
-- operating on handles
hClose              :: Handle → IO ()
hIsTerminalDevice   :: Handle → IO Bool
hSetBuffering       :: Handle → BufferMode → IO ()
hFlush              :: Handle → IO ()

-- input/output on handles
hPutStr             :: Handle → String → IO ()
hGetLine            :: Handle → IO String
hGetContents        :: Handle → IO String

-- standard handles
stdout              :: Handle
stdin               :: Handle
stderr              :: Handle
```

Terminal IO

Most terminal IO functions are special cases of the handle IO functions:

```
putStr :: String → IO ()
putStr s = hPutStr stdout s

getLine :: IO String
getLine = hGetLine stdin

getContents :: IO String
getContents = hGetContents stdin
```

Buffering

Buffering is often a source of unexpected behaviour.

```
data BufferMode = NoBuffering
                | LineBuffering
                | BlockBuffering (Maybe Int)
```

- No buffering: everything is directly read or written.
- Line-buffering: everything up to the next newline is buffered and then read or written at once.
- Block-buffering: blocks of a certain size are buffered and then read or written at once.

Terminals should typically be line-buffered, files are typically block-buffered. The stderr output is typically not buffered.

Lazy IO

Another pitfall is the use of lazy IO:

```
hGetContents :: Handle → IO String

getContents :: IO String
getContents = hGetContents stdin

readFile :: FilePath → IO String
readFile name = openFile name ReadMode ≫= hGetContents
```

The function hGetContents internally makes use of

```
System.IO.Unsafe.unsafeInterleaveIO :: IO a → IO a
```

This function delays an IO computation of type IO a and returns it as a thunk of type a.

The dangers of lazy IO

- The whole file is returned as a string, but it is only read when the string is evaluated.
- Lazy IO is only safe if the underlying files are essentially static and do not change.
- If the file changes in the meantime, unexpected effects can occur.
- The use of lazy IO in Haskell is often considered a mistake.

12.2 (Extensible) Exceptions

Exceptions

- IO operations are a common source of exceptions in Haskell.
- GHC has a very general exception handling mechanism.
- Exceptions have been changed for GHC-6.10.1 so that the exception mechanism is more extensible.

The Exception class

```
class (Typeable e, Show e) ⇒ Exception e where
  toException   :: e → SomeException
  fromException :: SomeException → Maybe e

  -- default implementations
  toException                          = SomeException
  fromException (SomeException e) = cast e

data SomeException where
  SomeException :: ∀e.(Exception e) ⇒ e → SomeException
cast :: (Typeable a, Typeable b) ⇒ a → Maybe b
```

Note that SomeException is an existential type defined using GADT syntax.

Excursion: GADT syntax

Existential typescome for free using GADT syntax, as in:

```
data Stream :: * where
  Stream :: (s → Step s) → s → Int → Stream
data Step :: * → * where
  Done ::                  Step s
  Yield :: Word8 → s →     Step s
  Skip  ::          s →    Step s
```

Excursion: Typeable – dynamic typing in Haskell

```
class Typeable a where
  typeOf :: a → TypeRep

cast :: (Typeable a, Typeable b) ⇒ a → Maybe b
cast x = r
  where
    r = if typeOf x == typeOf (fromJust r)
        then Just (unsafeCoerce x)
        else Nothing

unsafeCoerce :: a → b
```

- The type TypeRep holds run-time type information.
- The function cast is safe if no two types get the same TypeRep and the equality function on TypeRep is sane.
- Unfortunately, this cannot yet be enforced in Haskell's class system.

Excursion: Typeable – deriving instances

- Defining good instances of Typeable is critical for the safety of a program.
- Therefore, Typeable instances should best not be defined by hand.

Excursion: Typeable – deriving instances

- Defining good instances of Typeable is critical for the safety of a program.
- Therefore, Typeable instances should best not be defined by hand.
- In GHC, you can **derive** instances of Typeable when enabling the `DeriveDataTypeable` language extension.

Excursion: Typeable – deriving instances

- Defining good instances of Typeable is critical for the safety of a program.
- Therefore, Typeable instances should best not be defined by hand.
- In GHC, you can **derive** instances of Typeable when enabling the `DeriveDataTypeable` language extension.
- The class Typeable shows how we can integrate dynamic typing into a statically type language (available via Data.Dynamic):

```
data Dynamic    -- abstract, contains a TypeRep
toDyn   :: (Typeable a) ⇒ a → Dynamic
fromDyn :: (Typeable a) ⇒ Dynamic → Maybe a -- uses cast
```

Working with exceptions

Control.Exception.Extensible

```
-- throwing exceptions
throw    :: (Exception e) ⇒ e → a
throwIO  :: (Exception e) ⇒ e → IO a
-- handling exceptions
try      :: (Exception e) ⇒ IO a → IO (Either e a)
catch    :: (Exception e) ⇒ IO a → (e → IO a) → IO a
handle   :: (Exception e) ⇒ (e → IO a) → IO a → IO a
...
```

While exceptions can be thrown everywhere, they can only be caught in an IO context.

Implementation of catch

- Given a particular thrown exception of type e, use toException to turn it into SomeException.
- Next, use fromException to see if we we can turn it into type e' of the handler.
- If the type of the handler matches, then call it.
- Otherwise, re-throw the exception.

Example: catching an exception

```
import Prelude hiding (catch)
import Control.Monad
import Control.Exception.Base
import Control.Exception.Extensible

safeRead :: FilePath → IO (Maybe String)
safeRead name =
  catch (liftM Just (readFile name))
        (λ(e :: IOException) →
           putStrLn "Error" ≫ return Nothing)
```

- Omitting the type annotation IOException triggers a type error – why?
- The type annotation requires enabling a language extension: PatternSignatures for GHC-6.8, and ScopedTypeVariables for GHC-6.10.

12.3 Networking

Sockets

- Sockets allow processes to communicate via the network.
- A server listens on a particular network port for incoming connections.
- A client can connect to a host on a particular port.
- Once a connection has been established, two-way communication via the socket becomes possible.
- In Haskell, a high-level socket library is offered by the Network module.
- A much more detailed network socket library is available via the Network.Socket module.

Communicating via sockets

Network

```
-- initialization
withSocketsDo :: IO a → IO a

-- server-side
listenOn    :: PortID → IO Socket
accept      :: Socket → IO (Handle, HostName, PortNumber)
sClose      :: Socket → IO ()

-- client-side
connectTo :: HostName → PortID → IO Handle
```

More on sockets

- Use withSocketsDo as an initializer:

```
main = withSocketsDo $ realMain
```

- The function listenOn opens a socket. It does not block.
- Both connectTo on the client side and accept on the server side block until a connection has been established.
- A socket gives both the client and the server a handle to communicate over.
- Both server and client can terminate an individual connection by closing the handle.
- The server can also close the entire socket.
- It is usually a good idea to let the server fork a new thread upon accepting a connection and continue to listen for more connections on the original thread.

12.4 Threads

Working with threads

Control.Concurrent

```
-- creating a thread
forkIO        :: IO () → IO ThreadId

-- managing current thread
threadDelay :: Int → IO ()
yield         :: IO ()
myThreadId :: IO ThreadId

-- managing other threads
killThread    :: ThreadId → IO ()
throwTo       :: (Exception e) ⇒ ThreadId → e → IO ()
```

Forking a new thread

```
forkIO :: IO () → IO ThreadId
```

- Note that IO () is also the type of main. We can thus run a whole program in its own thread.
- Any thread can create new threads.
- Haskell threads are very lightweight. They are created and scheduled by the Haskell runtime system and do not require creation of an OS thread.
- If the main program ends, all its threads are stopped too.
- You can explicitly control other threads by killing them or sending them exceptions via their thread ids.

Threads and shared data

How to communicate between threads?

- Using IORefs to share data between threads is unsafe.
- Generally, when working with threads, you have to watch out that they don't interfere with each other (while writing files, for example).

Unsafe shared data demonstration

```
test n       = do
                  x ← newIORef 0
                  mapM_ (forkIO ∘ writer x) [1 .. n]
                  writer x 0

writer x m = do
                  writeIORef x m
                  n ← readIORef x
                  when (m ≢ n) (putStrLn (show m))
                  writer x m
```

MVars

Control.Concurrent.MVar

```
data MVar    -- abstract
newMVar      :: a → IO (MVar a)
newEmptyMVar :: IO (MVar a)
readMVar     :: MVar a → IO a
putMVar      :: MVar a → a → IO ()
takeMVar     :: MVar a → IO a
```

- Unlike an IORef, an MVar can be **empty**.
- Using putMVar works only if the MVar is empty before. If the MVar is not yet empty, the call blocks.
- Using takeMVar leaves the MVar empty in the process. If the MVar is empty, the call blocks.
- Using MVars, we can implement other concurrency abstractions such as semaphores or channels.

Using MVars to implement semaphores

- A semaphore controls that no more than a particular number of clients access a particular resource.

```
newtype QSem = QSem (MVar (Int, [MVar ()]))
```

- The first component of the pair indicates how many clients can still access the resource; the list implements a queue of waiting threads.
- Note the nested use of MVars.

Creating a new semaphore

```
newQSem :: Int → IO QSem
newQSem initial =
  do
    sem ← newMVar (initial, [])
    return (QSem sem)
```

Waiting for a semaphore

```
waitQSem :: QSem → IO ()
waitQSem (QSem sem) =
  do
    (avail, blocked) ← takeMVar sem
    if avail > 0 then
      putMVar sem (avail − 1, [])
    else
      do
        block ← newEmptyMVar
        putMVar sem (0, blocked ++ [block])
        takeMVar block
```

The last call waits for the empty MVar to be filled.

Signalling a semaphore

```
signalQSem :: QSem → IO ()
signalQSem (QSem sem) =
  do
    (avail, blocked) ← takeMVar sem
    case blocked of
      []                  → putMVar sem (avail + 1, [])
      (block : blocked′) → do
                             putMVar sem (0, blocked′)
                             putMVar block ()
```

The last call signals the blocked thread that it can continue.

Channels

FIFO channels to communicate data safely between threads. Channels are a very useful and much more high-level abstraction than MVars or semaphores.

Control.Concurrent.Chan

```
data Chan   -- abstract

newChan   :: IO (Chan a)
dupChan   :: Chan a → IO (Chan a)
unGetChan :: Chan a → a → IO ()
readChan  :: Chan a → IO a
writeChan :: Chan a → a → IO ()
```

12.5 Software Transactional Memory

Software Transactional Memory (STM)

- An alternative implementation of concurrency abstractions in GHC.
- Use ideas from database systems.
- Threads can start transactions.
- Transactions are guaranteed to be run atomically.
- The implementation keeps a log of all the memory accesses during a transaction, but does not actually perform any writes yet.
- At the end of a transaction, the log is checked against the memory. If the memory is still consistent, the transaction is committed. Otherwise, it is restarted.

STM is lock-free

- STM does not use locking.
- Deadlocks cannot occur.
- However, large transactions can take a huge number of retries, so STM works best if transactions are kept as small as possible.

STM and Haskell

- Generally, it is difficult to log all side effects during a transaction.
- In Haskell, we can easily use the type system to keep track of such effects.
- The Haskell STM implementation introduces an STM monad, which is (again) a restricted form of the IO monad.
- Only TVars can be accessed within the STM monad, which are a logged version of IORefs.
- It is thus easy to guarantee the safety of STM in Haskell.

STM interface

STM in Haskell is provided by the stm package.

Control.Concurrent.STM

```
data STM   -- abstract
instance Monad STM

-- running a transaction
atomically   :: STM a → IO a

-- TVars
newTVar    :: a → STM (TVar a)
newTVarIO  :: a → IO (TVar a)
readTVar   :: TVar a → STM a
writeTVar  :: TVar a → a → STM ()
```

Thread example using STM

```
test n       = do
                  x ← newTVarIO 0
                  mapM_ (forkIO ∘ writer x) [1 . . n]
                  writer x 0
writer x m = do
                  n ← atomically $ do
                     writeTVar x m
                     readTVar x
                  when (m ≠ n) $ putStrLn $ show m
                  writer x m
```

Implementing MVar using STM

```
type MVar a = TVar (Maybe a)

newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing
```

Implementing MVar using STM

```
type MVar a = TVar (Maybe a)

newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing
```

```
takeMVar :: MVar a → STM a
takeMVar m = do
                x ← readTVar m
                case x of
                  Nothing → retry
                  Just v  → do
                              writeTVar m Nothing
                              return v
```

The function putMVar is similar.

More STM combinators

Control.Concurrent.STM

> retry :: STM a

Request to retry the current transaction. Will usually block the thread until one of the TVars read from is updated by another thread.

> orElse :: STM a → STM a → STM a

If the first action retries, the second action is tried next.

In addition, STM supports checking for invariants, catching exceptions in the STM monad, and it has an implementation of MVars and Chans that can be used in the STM monad – TMVars and TChans.

Interesting papers

- "Tackling the Awkward Squad" by Simon Peyton-Jones – on IO, concurrency, exceptions and foreign function calls
- "An Extensible Dynamically-Typed Hierarchy of Exceptions" by Simon Marlow – the paper where extensible exceptions for Haskell were introduced
- "Haskell Session Types with (Almost) No Class – session types give you advanced typed communication between different threads.
- "Composable memory transactions" by Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy – the paper where STM for Haskell was introduced