RTS

The run-time system runs its own event loop, in order to abstract you from low-level synchronization trouble.

- very efficient concurrent programming (“green threads”)
- you do not have to poll anything and there is no thread overhead
- you can share writeable objects just like in C

ForkIO

```
import Control.Concurrent
forkIO :: IO () → IO ThreadID
```

forkIO runs an IO action concurrently with the calling one.

Advantage: the green threads have almost zero overhead.

Concurrent programming

Communication among (green) threads can be done e.g. using MVar:

```
import Control.Concurrent.MVar

newEmptyMVar :: IO (MVar a)
newMVar :: a → IO (MVar a)

takeMVar :: MVar a → IO a
putMVar :: MVar a → a → IO ()
readMVar :: MVar a → IO a
swapMVar :: MVar a → a → IO a
```

MVar can be full or empty, put/take may block the thread.

Concurrent programming — example

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.MVar

main = do
  amount ← readLn :: IO Int
  com ← newEmptyMVar
  forkIO $ do
    forM_ [1..amount] $ λi → putMVar com i            ≫ threadDelay 33333
    forM_ [1..amount] $ λi → putMVar com (amount − i) ≫ threadDelay 33333
  let loop = do a ← takeMVar com
                when (a > 0) $ print a ≫ loop
  loop
```

```haskell
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad (forever, void)
import Network.Socket

main =
  withSocketsDo $ do
    counter ← newMVar 0
    bracket open close (serverLoop counter)
  where
    ...
```

```haskell
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad (forever, void)
import Network.Socket

main =
  withSocketsDo $ do
    counter ← newMVar 0
    bracket open close (serverLoop counter)
  where
    ...
```

```haskell
import Control.Concurrent
import Control.Exception (bracket)
import Control.Monad (forever, void)
import Network.Socket

main =
  withSocketsDo $ do
    counter ← newMVar 0
    bracket open close (serverLoop counter)
  where
    ...
```

```haskell
open = do
  sock ← socket AF_INET Stream defaultProtocol
  setSocketOption sock ReuseAddr 1
  bind sock $ SockAddrInet 8080 0
  setCloseOnExecIfNeeded $ fdSocket sock
  listen sock 16
  return sock
```

```haskell
open = do
  sock ← socket AF_INET Stream defaultProtocol
  setSocketOption sock ReuseAddr 1
  bind sock $ SockAddrInet 8080 0
  setCloseOnExecIfNeeded $ fdSocket sock
  listen sock 16
  return sock
```

```
open = do
  sock ← socket AF_INET Stream defaultProtocol
  setSocketOption sock ReuseAddr 1
  bind sock $ SockAddrInet 8080 0
  setCloseOnExecIfNeeded $ fdSocket sock
  listen sock 16
  return sock
```

```
serverLoop counter sock =
  forever $ do
    (conn, peer) <- accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (\_ -> close conn)
```

```
serverLoop counter sock =
  forever $ do
    (conn, peer) ← accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (λ_ → close conn)
```

```haskell
serverLoop counter sock =
  forever $ do
    (conn, peer) <- accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (\_ -> close conn)
```

Concurrent programming — TCP server

```
serverLoop counter sock =
  forever $ do
    (conn, peer) <- accept sock
    putStrLn $ "client: " ++ show peer
    void $ forkFinally (talk counter conn)
                       (\_ -> close conn)
```

Concurrent programming — TCP server

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    - String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET":_ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

Concurrent programming — TCP server

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    - String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET" : _ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

Concurrent programming — TCP server

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    -- String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET":_ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

Concurrent programming — TCP server

```
talk counter conn = do
  msg ← words ⟨$⟩ recv conn 1024    - String!
  putStrLn $ "Request: " ++ show msg
  case msg of
    "GET":_ → do
      ctr ← takeMVar counter
      putMVar counter (ctr + 1)
      send conn $
        "HTTP/1.0 200 OK\n" ++
        "Content-type: text/plain\n\n" ++
        "Hello, " ++ show ctr
    _ → send conn "HTTP 400 bad request\n\n"
```

IO extras

Overhead of MVar may be too big/unnecessary

- TVar (non-empty MVar with atomic operations over STM)
- IORef (non-empty "raw" reference variable)
- Ptr (ugly C-style reference to Plain Old Data without GC)
  ...the recommended way to crash it
- ForeignPtr (like Ptr but with a finalizer)
  ...auto-free
- STM (Software Transactional Memory monad) — gives you a serialized access to memory that can be organized among threads *without* IO.
- Chan, TChan, QSem, QSemN, ...

Foreign Function Interface

If you have to touch the hardware, you will have to call the system-level C code via FFI.

Typical uses:

- wrapping the system APIs
- wrapping the libraries from other languages ('bindings')
- number crunching

2 modes of FFI use

import

- files `.h` and `.c` are automatically compiled and linked from the program
- C functions are imported as IO actions or pure functions
- parameter types are mapped to primitive Haskell types

export

- export of a haskell function creates a `.h`
- the code for invoking the function is generated into an `.o`

(Typically you don't want to wrap whole libraries by hand; use `c2hs`.)

FFI — import

Haskell:

```haskell
{-# INCLUDE "rawread.h" #-}
{-# LANGUAGE ForeignFunctionInterface #-}
import Foreign.C
foreign import ccall "raw_read" rawRead :: Word → IO Word
main = rawRead 0 >>= print
```

`rawread.h`: `int64_t raw_read(int64_t);`

`rawread.c`: (BONUS: almost-valid use of reinterpret_cast)

```c
int64_t raw_read(int64_t addr) {
  int64_t* ptr=(int64_t*)addr;
  return *ptr;
}
```

FFI — import

Haskell:

```haskell
{-# INCLUDE "rawread.h" #-}
{-# LANGUAGE ForeignFunctionInterface #-}
import Foreign.C
foreign import ccall "raw_read" rawRead :: Word -> IO Word
main = rawRead 0 >>= print
```

`rawread.h`: `int64_t raw_read(int64_t);`

`rawread.c`: (BONUS: almost-valid use of reinterpret_cast)

```c
int64_t raw_read(int64_t addr) {
  int64_t* ptr=(int64_t*)addr;
  return *ptr;
}
```

FFI — export

`Factorial.hs`:

```haskell
import Foreign.C
foreign export ccall "fact" factorial :: Int → Int
factorial n = product [1..n]
```

`factorial.c`:

```c
#include "Factorial_stub.h"
void someF() {
  printf("%d\n", fact(5));
}
```

Parallel evaluation

Parallelization of Haskell is technically quite simple:

- Most of the stuff is effect-less
- RTS can do its own scheduling 'for free'
- GC is relatively modern and manages the parallelism well

  (as opposed to Java, C# and Python)

Implementation:

$$par :: a \longrightarrow b \longrightarrow b$$

"Indicates that it may be beneficial to evaluate the first argument in parallel with the second. Returns the value of the second argument.

a 'par' b is exactly semantically equivalent to b."

$$\begin{aligned}&\mathit{hardFunction} = \mathbf{let}\\&\quad \mathit{firstHalf} = \ldots\\&\quad \mathit{secondHalf} = \ldots\\&\quad \mathbf{in}\ \mathit{firstHalf}\ \text{`}\mathit{par}\text{`}\ \mathit{secondHalf}\ \text{`}\mathit{par}\text{`}\ \mathit{firstHalf} + \mathit{secondHalf}\end{aligned}$$

Compilation: `ghc -threaded -rtsopts hard.hs`

Running for 16 cores: `./hard +RTS -N16`

Disadvantage:

```haskell
hardFunction = let
   firstHalf = ...
   secondHalf = ...
   in firstHalf `par` secondHalf `par` firstHalf + secondHalf
```

Compilation: `ghc -threaded -rtsopts hard.hs`

Running for 16 cores: `./hard +RTS -N16`

Disadvantage: If we want to parallelize because of speed, it's usually better to write the program in C and call the (fast) building blocks via FFI (paralelly).

Better control over code execution:

```haskell
import Control.Parallel.Strategies
```

Parallel programming

Better control over code execution:

```
import Control.Parallel.Strategies
```

Acceleration via automated translation to parallel SIMD or OpenCL:

```
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Parallel programming

Better control over code execution:

```haskell
import Control.Parallel.Strategies
```

Acceleration via automated translation to parallel SIMD or OpenCL:

```haskell
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Parallel programming

Better control over code execution:

```
import Control.Parallel.Strategies
```

Acceleration via automated translation to parallel SIMD or OpenCL:

```
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Numeric computation with parallelization possibilities:

```
import Data.Array.Repa
```

Parallel programming

Better control over code execution:

```haskell
import Control.Parallel.Strategies
```

Acceleration via automated translation to parallel SIMD or OpenCL:

```haskell
import Data.Array.Accelerate
dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Numeric computation with parallelization possibilities:

```haskell
import Data.Array.Repa
```

✎Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming (Simon Marlow, 2013)

Streaming!

What is streaming?

- There's a data blob.
- it's big data™ and thus never fits in any memory,
- the processing involves IO and some stateful computation,
- each part of the pipeline does a different kind of IO,
- organizing this manually hurts.

We already have streaming at home!

Naive “lazy IO”:

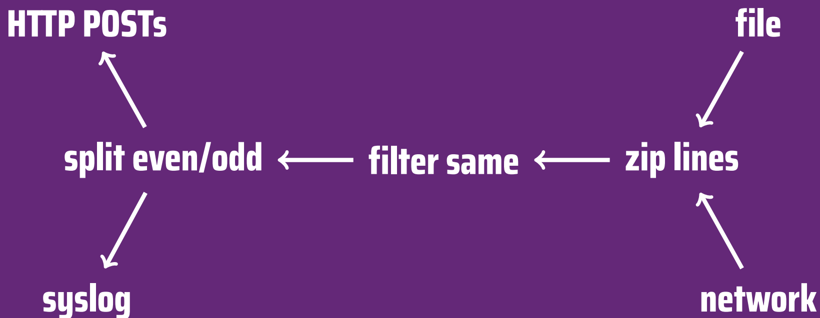
$$putContents \cdot unlines \cdot map\ (show \cdot length) \cdot lines \cdot getContents$$

... does not eat all memory (by a happy accident)

... contains a hidden *unsafePerformIO*

... will eat all memory if we try to save to 2 files at once

We need synchronization to achieve proper streaming (in a picture)



Streaming organizes side effects along data flow

Lists (no side effects):

$$uncons :: [a] \to \mathrm{Maybe}\ (a, [a])$$

Streams with effect, general idea, IO-only:

$$\begin{aligned}&\textbf{data}\ \mathrm{Stream}\ a\\ &\quad = \mathrm{Stream}\ \{ popStream :: \mathrm{IO}\ (\mathrm{Maybe}\ (a, \mathrm{Stream}\ a)) \}\end{aligned}$$

- This is a “pull-stream”.
- We could have a different monad than IO.

We aim to make the effects explicit (demo)

```
data Stream a
  = Stream { popStream :: IO (Maybe (a, Stream a)) }

numsFromTo :: Int → Int → Stream Int
numsFromTo i n
  | i > n       = Stream · pure $ Nothing
  | otherwise = Stream · pure $ Just (i, numsFromTo (succ i) n)

inputLines :: Stream String
inputLines = Stream $ do x ← getLine
                         pure $ case x of
                           "" → Nothing
                           _ → Just (x, inputLines)

eachTwice :: Stream a → Stream a
eachTwice s = do x' ← popStream s
                 pure $ case x' of
                   Nothing → Nothing
                   Just (x, s') → Just (x, Stream (pure $ Just (x, s')))
```

The fundamental issue of naive streaming

Pull-streams do not stream properly on splits.

Push-streams do not stream properly on merges.

Streaming libraries help with effect organization

- `conduit` — older, lots of ecosystem around
- `streamly` — very industrial
- `pipes` — cool types and bidirectional flows
- `streaming` — “free monad” approach, terse but powerful

Streaming libraries help with effect organization

- `conduit` — older, lots of ecosystem around
- `streamly` — very industrial
- `pipes` — cool types and bidirectional flows
- `streaming` — "free monad" approach, terse but powerful

What does `streaming` look like?

```haskell
import Streaming
import qualified Streaming.Prelude as S

S.stdinLn   :: Stream (Of String) IO ()
S.stdoutLn :: Stream (Of String) IO r → IO r

S.stdoutLn ∘ S.map (show ∘ length) ∘ S.stdinLn
```

... input and output actions are properly interleaved

... guaranteed not to explode

`streaming` describes data and effect generators

```
data Stream f m r
    = Step ! (f (Stream f m r))     – data stored in the wrap of f
    | Effect (m (Stream f m r))     – action stored in m
    | Return r                      – terminal value stored in r
```

Both Step and Effect "generate" more stream. Additionally, f is supposed to store an extra value:

```
type SimpleStream a r = Stream ((,) a)   IO r
type FasterStream  a r = Stream (Of  a)  IO r
type OnlyActions r     = Stream Identity IO r
```

Benefits:

- quite fast, data sources may omit unnecessary Effects!
- many cool uses for the return value:
    - stream continuation
    - "final" fold output

`streaming` contains some expectable tools

$$\begin{aligned}
&\mathit{yield} :: \text{Monad } m \Rightarrow a \rightarrow \text{Stream } (\text{Of } a)\ m\ ()\\
&\mathit{each} :: \text{Monad } m \Rightarrow [a] \rightarrow \text{Stream } (\text{Of } a)\ m\ ()
\end{aligned}$$

$$\begin{aligned}
&\mathit{stdinLn} :: \text{MonadIO } m \Rightarrow \text{Stream } (\text{Of String})\ m\ ()\\
&\mathit{fromHandle} :: \text{MonadIO } m \Rightarrow \text{Handle} \rightarrow \text{Stream } (\text{Of String})\ m\ ()\\
&\mathit{stdoutLn} :: \text{MonadIO } m \Rightarrow \text{Stream } (\text{Of String})\ m\ () \rightarrow m\ ()\\
&\mathit{toHandle} :: \text{MonadIO } m \Rightarrow \text{Handle} \rightarrow \text{Stream } (\text{Of String})\ m\ r \rightarrow m\ r
\end{aligned}$$

$$\begin{aligned}
&\mathit{mapM_} :: \text{Monad } m \Rightarrow (a \rightarrow m\ x) \rightarrow \text{Stream } (\text{Of } a)\ m\ r \rightarrow m\ r\\
&\mathit{effects} :: \text{Monad } m \Rightarrow \text{Stream } (\text{Of } a)\ m\ r \rightarrow m\ r
\end{aligned}$$

$$\begin{aligned}
&\mathit{map} :: \text{Monad } m \Rightarrow (a \rightarrow b) \rightarrow \text{Stream } (\text{Of } a)\ m\ r \rightarrow \text{Stream } (\text{Of } b)\ m\ r\\
&\mathit{filter} :: \text{Monad } m \Rightarrow (a \rightarrow \text{Bool}) \rightarrow \text{Stream } (\text{Of } a)\ m\ r \rightarrow \text{Stream } (\text{Of } a)\ m\ r
\end{aligned}$$

$$\mathit{fold} :: \text{Monad } m \Rightarrow (x \rightarrow a \rightarrow x) \rightarrow x \rightarrow (x \rightarrow b) \rightarrow \text{Stream } (\text{Of } a)\ m\ r \rightarrow m\ (\text{Of } b\ r)$$

...

Streaming functions interpret&create the structure

```
filterS pred s = case s of
  Return r → return r
  Effect m → Effect (filtersS pred ⟨$⟩ m)
  Step (x :⟩ rest)
    | pred x      → Step (x :⟩ filterS pred rest)
    | otherwise → filterS pred rest

eachTwice s = case s of
  Return r → return r
  Effect m → Effect (eachTwice ⟨$⟩ m)
  Step (x :⟩ rest) = do
    S.yield x   -- the monad behaves much like a Writer
    S.yield x
    eachTwice rest
```

How do we solve the stream splitting issue?

```
>>> S.print $ each ["one", "two"]
"one"
"two"
>>> S.stdoutLn $ each ["one", "two"]
one
two
>>> S.print $ S.stdoutLn $ S.copy $ each ["one", "two"]
"one"
one
"two"
two
```

The above actually streams properly!

`streaming`: Split streams never really fork off

$$copy :: \text{Monad}\ m \Rightarrow \text{Stream}\ (\text{Of}\ a)\ m\ r \rightarrow \text{Stream}\ (\text{Of}\ a)\ (\text{Stream}\ (\text{Of}\ a)\ m)\ r$$
$$stdoutLn :: \text{MonadIO}\ m \Rightarrow \text{Stream}\ (\text{Of String})\ m\ () \rightarrow m\ ()$$

$$printTwice = S.stdoutLn\ \$\ S.stdoutLn\ \$\ S.copy\ \$\ each\ [\texttt{"one"}, \texttt{"two"}]$$

- Stream is a Monad, Stream over IO is a MonadIO
- *copy* streams the elements interleaved by side effects that copy the input
- The outer stream can run the IO effects just like plain IO!
  - Actions get “recorded” in the inner stream as side effects.
- The inner stream gets consumed by the other fold.
  - The outer fold must execute the interleaved actions in the correct order as it takes data out.
  - It can’t cheat: the data is hidden behind running the Effects.

`streaming`: Split streams never really fork off

$$copy :: \text{Monad } m \Rightarrow \text{Stream } (\text{Of } a)\ m\ r \rightarrow \text{Stream } (\text{Of } a)\ (\text{Stream } (\text{Of } a)\ m)\ r$$
$$stdoutLn :: \text{MonadIO } m \Rightarrow \text{Stream } (\text{Of String})\ m\ () \rightarrow m\ ()$$

$$printTwice = S.stdoutLn \ \$\ S.stdoutLn \ \$\ S.copy \ \$\ each\ [\texttt{"one"}, \texttt{"two"}]$$

- Stream is a Monad, Stream over IO is a MonadIO
- *copy* streams the elements interleaved by side effects that copy the input
- The outer stream can run the IO effects just like plain IO!
  - Actions get “recorded” in the inner stream as side effects.
- The inner stream gets consumed by the other fold.
  - The outer fold must execute the interleaved actions in the correct order as it takes data out.
  - It can’t cheat: the data is hidden behind running the Effects.

`streaming`: Split streams never really fork off

$$copy :: \text{Monad } m \Rightarrow \text{Stream (Of } a) \; m \; r \rightarrow \text{Stream (Of } a) \; (\text{Stream (Of } a) \; m) \; r$$
$$stdoutLn :: \text{MonadIO } m \Rightarrow \text{Stream (Of String) } m \; () \rightarrow m \; ()$$

```
printTwice = S.stdoutLn $ S.stdoutLn $ S.copy $ each ["one", "two"]
```

- Stream is a Monad, Stream over IO is a MonadIO
- *copy* streams the elements interleaved by side effects that copy the input
- The outer stream can run the IO effects just like plain IO!
  - Actions get “recorded” in the inner stream as side effects.
- The inner stream gets consumed by the other fold.
  - The outer fold must execute the interleaved actions in the correct order as it takes data out.
  - It can’t cheat: the data is hidden behind running the Effects.

`streaming`: Split streams never really fork off

$$copy :: \text{Monad } m \Rightarrow \text{Stream (Of } a\text{) } m\ r \rightarrow \text{Stream (Of } a\text{) (Stream (Of } a\text{) } m\text{) } r$$

$$stdoutLn :: \text{MonadIO } m \Rightarrow \text{Stream (Of String) } m\ () \rightarrow m\ ()$$

```
printTwice = S.stdoutLn $ S.stdoutLn $ S.copy $ each ["one", "two"]
```

- Stream is a Monad, Stream over IO is a MonadIO
- *copy* streams the elements interleaved by side effects that copy the input
- The outer stream can run the IO effects just like plain IO!
    - Actions get “recorded” in the inner stream as side effects.
- The inner stream gets consumed by the other fold.
    - The outer fold must execute the interleaved actions in the correct order as it takes data out.
    - It can’t cheat: the data is hidden behind running the Effects.

```
copy s =
  case s of
    Return r → Return r
    Effect m → Effect (copy <$> lift m)
    Step (a :> rest) →
      Effect (Step (a :> Return (Step (a :> copy rest))))
copy2 s =
  case s of
    Return r → Return r
    Effect m → Effect (copy2 <$> lift m)
    Step (a :> rest) →
      Step (a :> Effect (Step (a :> Return (copy2 rest))))
```