Haskell4Life: Parallelism and Concurrency

(Practical Example)

Sergiu Ivanov

sergiu.ivanov@lacl.fr

Slides and code examples available online:

http://lacl.fr/~sivanov/doku.php?id=en:haskell_for_life

Outline

1. Get Warm

2. Parallel Programming

3. Concurrent Programming

Outline

1. Get Warm
2. Parallel Programming
3. Concurrent Programming

Parallel and Concurrent Programming

Parallel and Concurrent Programming

Parallelism: solve different parts of the same problem in parallel

- is generally about splitting the problem into subproblems and subsequently combining the results.

Concurrency: solve different interdependent problems at the same time

- is generally about ensuring proper utilisation of shared resources.

The IO Monad

Provides an interface to the outer world.

```
getLine :: IO String
putStrLn :: String -> IO ()
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Lazy Evaluation Strategy

In Haskell, a value is only evaluated when needed.

Lazy Evaluation Strategy

In Haskell, a value is only evaluated when needed.

Define a as the product of all elements of a big list:

```
λ> let a = product [1..10000000]
```

The actual value of a will be computed when we need it:

```
λ> a
```

... (working)

Lazy Evaluation Strategy

In Haskell, a value is only evaluated when needed.

Define a as the product of all elements of a big list:

```
λ> let a = product [1..10000000]
```

The actual value of a will be computed when we need it:

```
λ> a
```

... (working)

This makes infinite lists possible.

```
λ> let xs = [1..]
λ> take 5 xs
[1,2,3,4,5]
```

List Comprehensions

The mathematical definition

$$\{x \mid x \in \{1, \ldots, 10\}, x \text{ is odd}\}$$

can be written is Haskell as

```
[x | x <- [1..10], odd x]
```

Outline

1. Get Warm

2. Parallel Programming

3. Concurrent Programming

Annotations for Parallel Programming

``x `pseq` y``	evaluate `x`, then return `y`.
``x `par` y``	evaluate `x` in parallel with returning `y`.
`force x`	completely evaluate `x` (don't be lazy).

The annotation 'par' is not enforcing: the runtime will carry out the evaluation in parallel only if it finds the idea "reasonable".

Check out `sorting.hs`!

Outline

1. Get Warm
2. Parallel Programming
3. Concurrent Programming

Haskell Runtime Threads

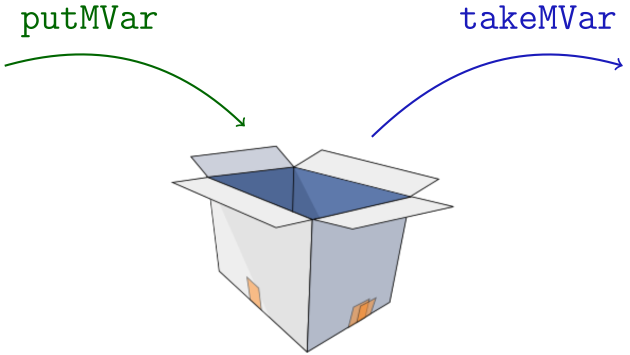
```
forkIO :: IO () -> IO ThreadId
```

Perform the IO action in a different thread and return the identifier of the new thread.

Defined in `Control.Concurrent`.

Haskell thread have less overhead than OS threads.

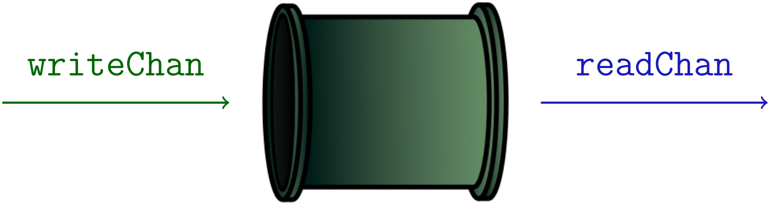
Mutable Locations: MVar




```
newMVar  :: a -> IO (MVar a)
putMVar  :: MVar a -> a -> IO ()      (blocks if full)
takeMVar :: MVar a -> IO a            (blocks if empty)
```

https://openclipart.org/

Channels: Chan




```
newChan   :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
```
(**blocks** if empty)

https://openclipart.org/

Software Transactional Memory (STM) (in Haskell)

STM = using transactions to handle concurrent memory accesses (like in databases)

Software Transactional Memory (STM) (in Haskell)

STM = using transactions to handle concurrent memory accesses (like in databases)

We will use STM versions of `Chan` and `MVar` — `TChan` and `TMVar` — which are more flexible.

Software Transactional Memory (STM) (in Haskell)

STM = using transactions to handle concurrent memory accesses (like in databases)

We will use STM versions of `Chan` and `MVar` — `TChan` and `TMVar` — which are more flexible.

Usually (i.e. in the `IO` monad), calls to STM functions need to be wrapped in `atomically $`:

```
atomically $ writeChan chan newVal
```

Check out `scanner.hs`!