



Cartesian Closed Comic: <http://ro-che.info/ccc/04.html>

Lazy Evaluation, IO, Modules, Cabal

LECTURE 4

Based on slides of Graham Hutton/Erik Meijer

LAZY EVALUATION

Different Orders of Evaluation

```
inc :: Int → Int  
inc n = n + 1
```

```
inc (2 * 3)
```

We can
evaluate
2 * 3

```
inc (2 * 3)
```

We can
evaluate
the call
to inc

Different Orders of Evaluation

```
inc :: Int → Int  
inc n = n + 1
```

```
inc (2 * 3)
```

Innermost
redex

```
inc (2 * 3)
```

Outermost
redex

Innermost Redex

```
inc (2 * 3)
```

=

```
inc 6
```

=

```
6 + 1
```

=

```
7
```

An innermost
redex does not
contain any
other redexes!

```
inc :: Int → Int  
inc n = n + 1
```

Outermost Redex

```
inc (2 * 3)
```

=

```
(2 * 3) + 1
```

=

```
6 + 1
```

=

```
7
```

An outermost
redex is not
contained in any
other redexes!

```
inc :: Int → Int  
inc n = n + 1
```

Innermost/Outermost

Call-by-value

`mult(1+2, 2+3)`

=

`mult(3, 2+3)`

=

`mult(3, 5)`

=

`3*5`

=

15

Call-by-name

`mult(1+2, 2+3)`

=

`(1+2)* (2+3)`

=

`3*(2+3)`

=

`3*5`

=

15

`mult (x, y)
= x*y`

Church-Rosser

If a can be reduced to both b and c there must be a term d to which both b and c can be reduced.

Proved in 1936 by Alonzo Church and J. Barkley Rosser.

Termination

```
inf = 1 + inf
```

Efficiency

square $n = n * n$

Facts

- Outermost reduction may give a result when innermost reduction fail to terminate.
- For a given expression, if there exists any reduction sequence that terminates, then outermost reduction also terminates, with the same result.
- Outermost reduction may require more steps than innermost reduction.

Best of Both Worlds

Can we get the low number of evaluation steps of call-by-value and the higher number of terminating programs of call-by-name in one system?

- Yes, we can do call-by-name and share computations!

Best of Both Worlds

```
square (1+2)
```

=

```
let n = (1+2) in n*n
```

=

```
let n = 3 in 3*3
```

=

```
9
```

Lazy evaluation =

Call-by-name + sharing

- Terminates more often than call-by-value
- Never uses more steps than call-by-value

More Facts

- Lazy evaluation never requires more reduction steps than innermost reduction.
- Haskell uses lazy evaluation

Infinite Structures

ones = 1 : ones

Take

```
take :: Int -> [a] -> [a]
take 0 xs      = []
take n []     = []
take n (x:xs) = x : take n xs
```

Can be applied to finite and infinite lists!
(as can map, foldr, head, tail)

Warning

```
> take 3 ones  
[1,1,1]
```

```
>filter (<= 5) [1..]  
1:(2:(3:(4:(5:⊥))))
```

```
>takeWhile(<= 5) [1..]  
1:(2:(3:(4:(5:[]))))
```

Prime Numbers

A simple procedure for generating the infinite list of all prime numbers is as follows:

1. Write down the list [2,3,4,...]
2. Mark the first value p in the list as prime
3. Delete all multiples of p from the list
4. Return to step 2

The name of this procedure: seive of Eratosthenes

Prime Numbers

The first steps look as follows:

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...

[2, 3, 5, 7, 9, 11, 13, 15, ...

[2, 3, 5, 7, 11, 13, ...

[2, 3, 5, 7, 11, 13, 15, ...

[2, 3, 5, 7, 11, 13, ...

[2, 3, 5, 7, 11, 13, ...

Prime Numbers

We can easily translate this into Haskell:

```
primes :: [Int]
primes = seive [2..]

seive :: [Int] -> [Int]
seive (p:xs) = p : seive [x | x <- xs
                             , x `mod` p /= 0]
```

```
> primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
,59,61,67,71,73,79,83,89,97,101,103,107,109,
113,127,131,137,139,149,151,157,163,167,173,
179,181,191,193,197,199,211,223,227,229,233,
239,241,251,257,263,269,271,277,281,283,293,
307,311,313,317,331,337,347,349,353,359,367,
373,379,383,389,397,401,409,419,431,433,439,443,449,457,461,467,479,487,491,499,503,509,521,523,541,547,557,563,569,577,587,593,601,607,613,617,641,643,647,653,659,661,673,677,683,689,697,701,709,713,727,733,739,743,751,757,761,769,773,787,797,809,811,823,827,833,839,853,857,863,877,881,887,893,907,911,919,923,929,937,941,947,953,967,971,977,983,989,997]
```

Prime Numbers

Without the constraint of finiteness we get a modular definition on which different boundary conditions can be imposed.

```
>take 10 primes  
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

```
>takeWhile (<15) primes  
[2, 3, 5, 7, 11, 13]
```

Lazy evaluation is a powerful programming tool!

Quiz

- Define a program `fibs :: [Integer]` that generates the infinite Fibonacci sequence: `[0,1,1,2,3,5,8,13,21,34,...]` using the following simple procedure:
 1. The first two numbers are 0 and 1
 2. The next is the sum of the previous two
 3. Return to step 2
- Define a function `fib :: Int -> Integer` that calculates the `n`th Fibonacci number.

Based on slides of Graham Hutton

INTERACTIVE PROGRAMS

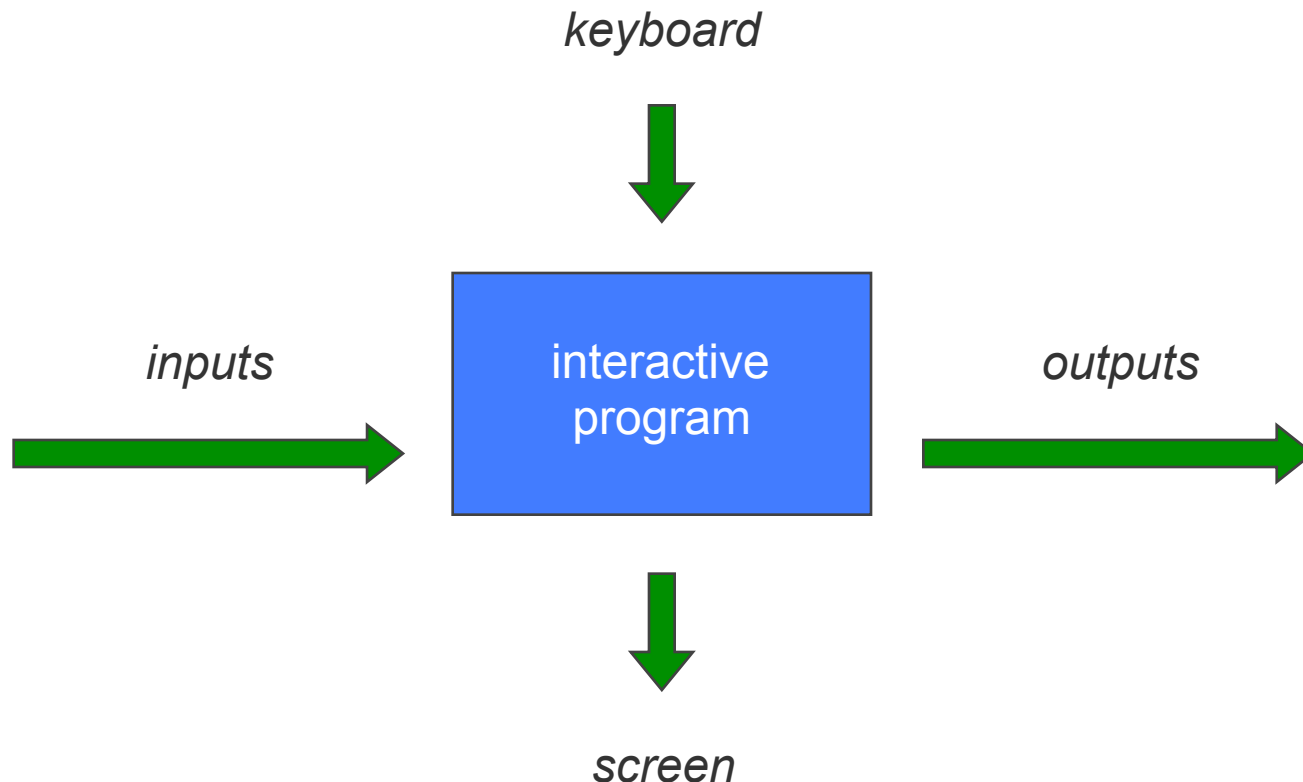
Interactive Programs

So far, we have seen how we can write pure functions. We could give these functions some inputs and they returned an output.



Interactive Programs

However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.



A Problem

Haskell programs are pure mathematical functions:

- Haskell programs have no side effects.

However, reading from the keyboard and writing to the screen are side effects:

- Interactive programs have side effects.

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

`IO a`

The type of actions that return a value of type `a`.

Examples

IO Char

The type of actions that return a character.

IO ()

The type of actions that return no result value.

Note:

- () is the type of tuples with no components

Basic Actions

The standard library provides a number of actions, including the following primitives:

```
getChar :: IO Char
```

```
putChar :: Char → IO ()
```

```
putStrLn :: String → IO ()
```

```
return :: a → IO a
```

Sequencing IO Actions

A sequence of actions can be combined as a single composite action using the keyword do.

For example:

```
a :: IO (Char,Char)
a = do x ← getChar
      getChar
      y ← getChar
      return (x,y)
```

More Actions

A whole lot of actions are defined in the standard Prelude. And in libraries such as System.IO.

You can find them using Hoogle and Hayoo, but also in the documentation.

Combining IO and Pure Functions

- A pure functions can not call an IO function.
- An IO function can call a pure function.

```
strlen :: IO ()
strlen = do putStrLn "Enter a string: "
            xs ← getLine
            putStrLn "The string has "
            putStrLn (show (length xs))
            putStrLnLn " characters"
```

Hangman

Consider the following version of hangman:

- One player secretly types in a word.
- The other player tries to deduce the word, by entering a sequence of guesses.
- For each guess, the computer indicates which letters in the secret word occur in the guess.
- The game ends when the guess is correct.

Keep IO to a Minimum

- Once you are in IO you cannot get out!
- If a function calls an IO function it will itself be an IO function.
- A pure function called from IO is still pure!
 - Pure functions are generally easier to work with

A Few More Things

- “return” is just another function, it merely lifts pure values into IO.
- Binding pure values to a variable is preferably done using let.

```
a :: Num a => IO (Int,a)
a = do
    x ← someIOInt
    getChar
    let y = 4 + 5
    z ← return $ 4 + 5
    return (x,y)
```

Random

In `System.Random` everything for generating random numbers can be found.

```
getStdGen :: IO StdGen
```

There are a lot of functions in this library for generating infinite streams of random numbers, a single number, numbers within a certain range etc.

```
Random :: RandomGen g => g -> (a, g)
```

MODULES

Importing Modules

So far we have already imported a few modules using:

```
import Data.List
```

This brings all exported definitions of Data.List into scope.

We can restrict this by specifying what we want:

```
import Data.List(member, nub)
```

Importing Modules

We can also hide certain functions from an imported module (because we already have a function with that name).

```
import Data.List hiding (foldr)
```

Importing Modules

We can also add a prefix to all definitions of this module:

```
import qualified Data.List
```

```
Data.List.foldr (+) 0 [1,2,3]
```

We can also define a shorter prefix:

```
import qualified Data.List as L
```

```
L.foldr (+) 0 [1,2,3]
```

Importing Modules

Some modules are designed to be imported qualified:

- Data.List
- Data.Map
- Data.Either
- Data.Maybe

Importing Modules

Sometimes we only want class instances from a module:

```
import Data.List ()
```

It is not possible to hide class instances.

Importing in GHCi

We can import modules within GHCi:

```
>import Data.List  
Data.List >
```

Defining a Module

We can also define our own modules:

```
module Hangman where

import qualified Data.List
...

hangman :: IO ()
```

The filename now must be Hangman.hs

Defining a Module

We have fine control over what we export (except for instances):

```
module Hangman (hangman,  
sgetLine) where  
  
import qualified Data.List  
...  
  
hangman :: IO ()
```

Structuring a Set of Modules

If we have a large amount of modules we might want to structure them a bit by adding some descriptive names.

For example:

```
module Main where ...  
module Game.Logic where ...  
module Game.Board where ...  
module Game.AI.Normal where ...
```

Structuring a Set of Modules

These files must be structured on your hard drive as:

```
./Main.hs  
./Game/Logic.hs  
./Game/Board.hs  
./Game/AI/Normal.hs
```

The Main Module

The module Main should contain a function main:

```
module Main where

main :: IO ()
main = undefined
```

When we compile the program main will be the function that get's executed!

The module Main may have a different filename!

Compiling a Program

The GHCi comes with a compiler called GHC

```
> ghc -make Main.hs  
[1 of 1] Compiling Main  
( Main.hs, Main.o )  
Linking Main ...  
> ./Main  
Main: Prelude.undefined
```

Reading Input Parameters

The main function doesn't get any arguments. But it is in IO, so it can have side effects!

The module System.Environment provides a means to collect arguments passed to your program!



CABAL

A Package Manager

The Cabal package manager comes with Haskell Platform.

- It provides a way to track dependencies
- It provides access to a big package database

HackageDB

<http://hackage.haskell.org>

Installing a Package

- If it is on Hackage:

```
> cabal update
Downloading the latest package list from
hackage.haskell.org
> cabal install hinvaders
...
Installing executable(s) in
~/Library/Haskell/ghc-7.0.3/lib/hinvaders-0.1/
bin
>
```

- Otherwise: Download the package, unpack, go into package dir, type “cabal install”

Making Your Own Package

TableAlgebra2.cabal:

```
cabal-version: >=1.8
Name:          TableAlgebra2
synopsis:      Ferry Table Algebra
Category:     Database
Version:      0.0.0.1
Description:   The DSH Table Algebra
               library
License:      BSD3
License-file: LICENSE
Author:       Jeroen Weijers
               <jeroen.weijers@uni-tuebingen.de>
Maintainer:   Jeroen Weijers
               <jeroen.weijers@uni-tuebingen.de>
Build-Type:   Simple
```

Making Your Own Package

Library

```
buildable:           True
build-depends:       base >= 4.2 && < 5,
                     HaXml >= 1.22,
                     mtl >= 2.0.1.0
exposed-modules:     Database.Algebra.PF
                     Database.Algebra.XML
                     ...
hs-source-dirs:      src
GHC-Options:         -Wall -fno-warn-orphans
other-modules:       Database.Algebra.Monadic
                     Database.Algebra.Algebra
```

Making Your Own Package

```
executable x100shgen
  Main-is: Database/Tools/SharingCodeGen.hs
  GHC-Options: -Wall -fno-warn-orphans
  hs-source-dirs: src
  build-depends: base >= 4.2 && < 5,
                  HaXml >= 1.22,
                  mt1 >= 2.0.1.0
  other-modules: Database.Algebra.Monadics
                  Database.Algebra
```

Making Your Own Package

You also need a file Setup.hs

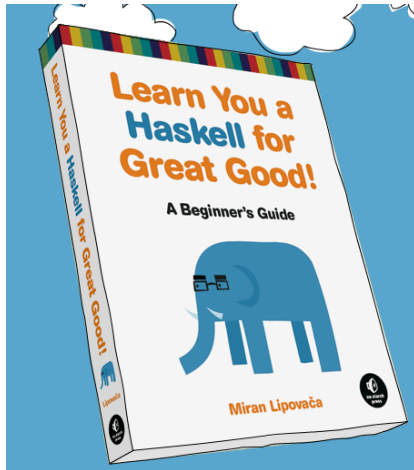
```
import Distribution.Simple  
  
main = defaultMain
```

And a file LICENSE (may be empty)

Cabal

- Cabal supports plugins from external tools
- You can customise nearly everything!
- More information: <http://haskell.org/cabal>

Reading material



- Learn you a Haskell:
 - Chapter 7 and 9

- Why Functional Programming Matters
 - John Huges