Fachhochschule
Nordwestschweiz

Functional Programming

http://www.pngall.com/colored-smoke-png/download/15792

Input/Output

Daniel Kröni
University of Applied Sciences Northwestern Switzerland

Learning Targets

You understand the problem of impure functions

You appreciate the simplicity of pure functions

You can perform console, file and network I/O

You understand Haskell's I/O system and how it supports pure functional programming

Content

- **Example**
- **Terminal I/O**
  - Read type class
- **Concept**
  - Evaluate vs Execute
  - Pure vs Impure
- **File I/O**
- **Network I/O**

IO by Example

- **Program (FirstIO.hs)**

```
main = do putStrLn "Please enter your name:"
          name <- getLine
          let msg = "Welcome to the real world " ++ name
          putStrLn msg
```

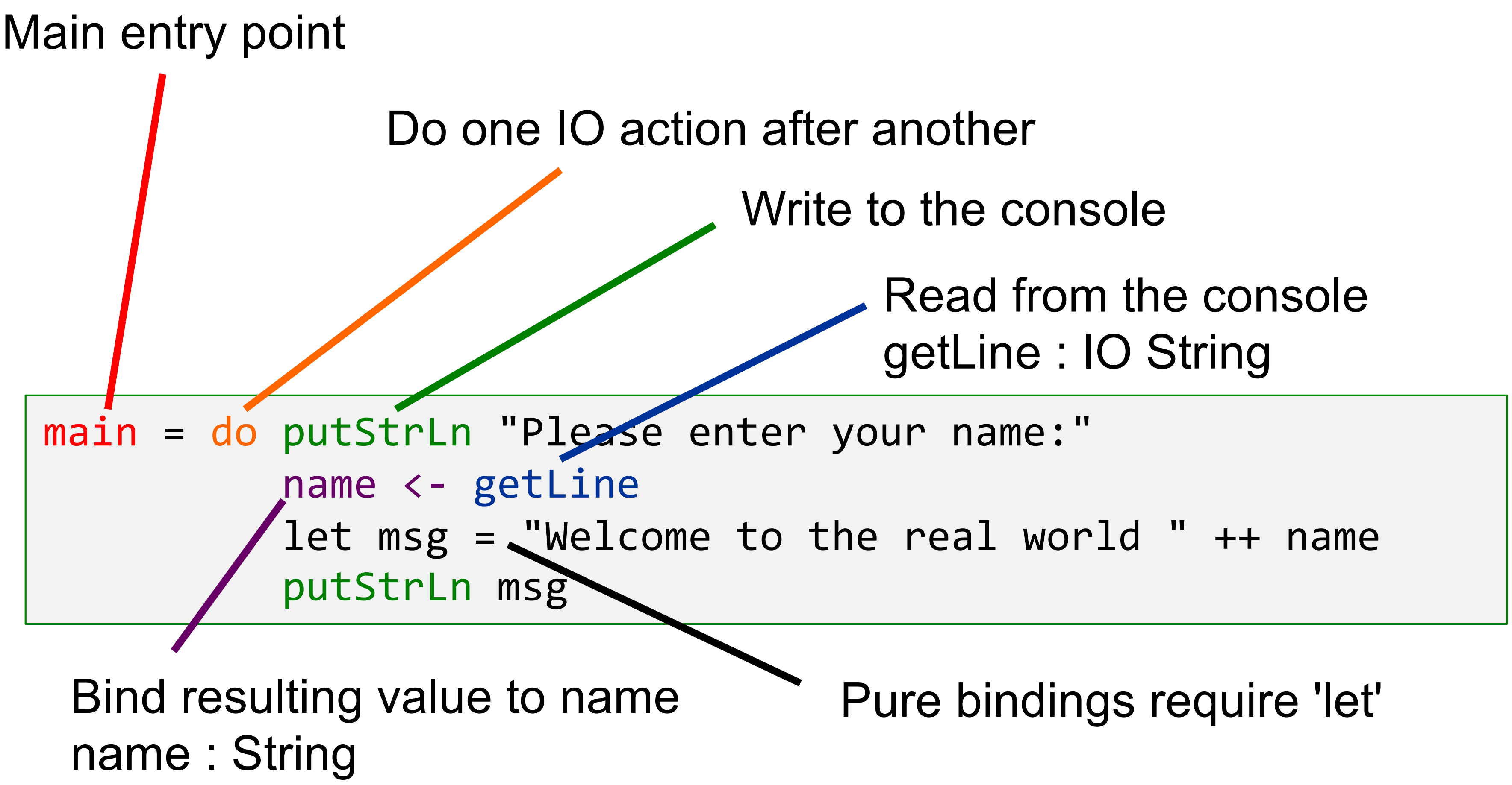
- **Compile**

```
> ghc FirstIO.hs
```

- **Run**

```
> ./FirstIO
Please enter your name:
Daniel
Welcome to the real world Daniel
```

IO by Example



Main entry point

Do one IO action after another

Write to the console

Read from the console
getLine : IO String

```
main = do putStrLn "Please enter your name:"
          name <- getLine
          let msg = "Welcome to the real world " ++ name
          putStrLn msg
```

Bind resulting value to name
name : String

Pure bindings require 'let'

Excursion: Read Typeclass

- **read is used to convert a String into a value of a member of Read**

```
> :t read
read :: Read a => String -> a
```

- **Example**

```
> read "5"
<interactive>:2:1:
    No instance for (Read a0) arising from a use of `read'
    The type variable `a0' is ambiguous
    Possible fix: add a type signature that fixes these type
variable(s)
```

```
> (read "5") :: Int
5
```

Worksheet: MiniCalc

```
$ ./MiniCalc
Welcome to MiniCalc!
Please enter a first number:
12
Please enter a second number:
34
12 + 34 = 46
```

What is side effect free?

- **Haskell is a pure functional language**

  => Every function returns the same result if applied to the same parameters

```
func :: Int -> Int
func i = i + 1

constant :: Int
constant :: 42
```

now consider this use of the above functions:

```
let a = (func 3) - (func 3)
    b = constant - constant
```

  - What are the values of a and b?
  - In order to answer this question: What do you need to now about the functions definition?
  - Does it matter if the first or the second occurrence of func and constant respectively is evaluated first?

The Problem with IO

- **Side effect free means:**
  - Every function returns the same result if applied to the same parameters
  - But getLine which always returns the same string is meaningless!
- **Possible approach**
  - Provide *impure* functions like `inputInt :: Int` which read and return the users input
- **Problem**

```
inputDiff = inputInt - inputInt
```

  - What is the value of inputDiff?
  - In order to answer this question: What do you need to know about the definition of inputInt?
  - Does it help to know how inputInt is implemented?
  - Does the evaluation order of the two occurrences of inputInt matter?

The Problem with Side Effects cont.

- **Reasoning about the program's behavior becomes substantially more difficult with such a model.**
  - We can't understand the meaning of an expression anymore just by looking at the meaning of its parts. The environment of an expression becomes relevant.
- **This is because any function may be affected by IO:**

```
lookingPure :: Int -> Int
lookingPure i = inputInt + i
```

- **This is like programming in Java & Co. where every method call could potentially delete your disk or even launch a missile!**

NOT GOOD

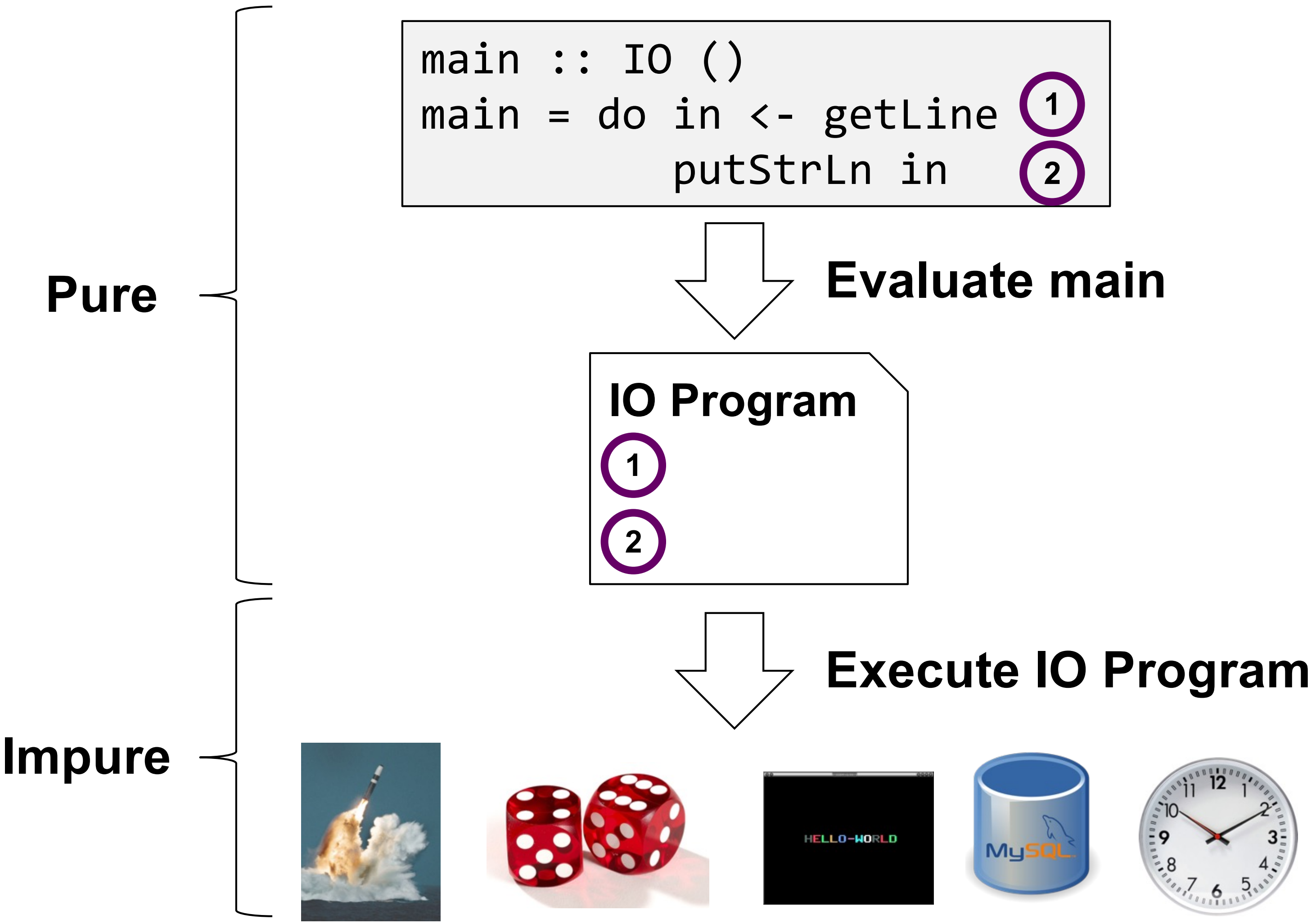
Haskell's Solution

- **I/O is best described as actions happening in sequence**
  - Example: Read an input then write to a file
- **Haskell provides the type** `IO a` **which is an I/O action of type** `a` **or an I/O program of type** `a`**.**
- **A value belonging to** `IO a` **is a program which could do some I/O and then returns a value of type** `a`**.**
- **Haskell provides some primitive I/O programs as well as a mechanism to sequence these I/O operations.**
- **I/O is not performed until the I/O programs are executed:**

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

main returns an I/O program which is then executed by the Haskell runtime system.

Haskell IO Illustrated



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

Evaluate main

IO Program

Execute IO Program

Basic IO: Reading Input, Writing Output

- **Reading a line from the standard input**

```
getLine :: IO String
```

  - A line is terminated by hitting Enter in a shell

- **Reading a single character**

```
getChar :: IO Char
```

- **Writing a string**

```
putStr :: String -> IO ()
```

- **Writing a string followed by a newline**

```
putStrLn :: String -> IO ()
```

Unit – The One-Element Type

- **All IO actions must return a value:**

```
getChar :: IO Char
```

returns a Char

```
getLine :: IO String
```

returns a String

- **What should we do if you have nothing useful to return?**

```
putStrLn :: String -> IO ()
```

  - The IO action created by putStrLn does not return an interesting value
    It is performed for its I/O-effect only
- **Haskell defines the type `()` called Unit which has only one single value which is also written `()` like the empty tuple**
- **Not often used since it can't transport any information**
- **We use it only in IO to denote that the returned value is not of interest but only the effect of the action**

Return

- **return creates an IO action which returns its argument when executed without any other effect**

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

- **Example**

```
main :: IO ()
main = do input    <- getLine
          putStrLn (reverse input)
          continue <- getChar
          if continue == 'y' then main else return ()
```

- **Return has nothing to do with Java's return statement!**

Simple File IO

- **Identifying a file by a path**

```
type FilePath = String
```

- **Reading content from a file**

```
readFile :: FilePath -> IO String
```

- **Writing content to a file**

```
writeFile :: FilePath -> String -> IO ()
```

- **Example: Copy a text file**

```
main = do content <- readFile "in.txt"
          writeFile "out.txt" content
```

Command Line Arguments

- **Most command line tools take arguments**

```
$ copy file1.txt file2.txt
```

- **getArgs from Module System.Environment**

```
getArgs :: IO [String]
```

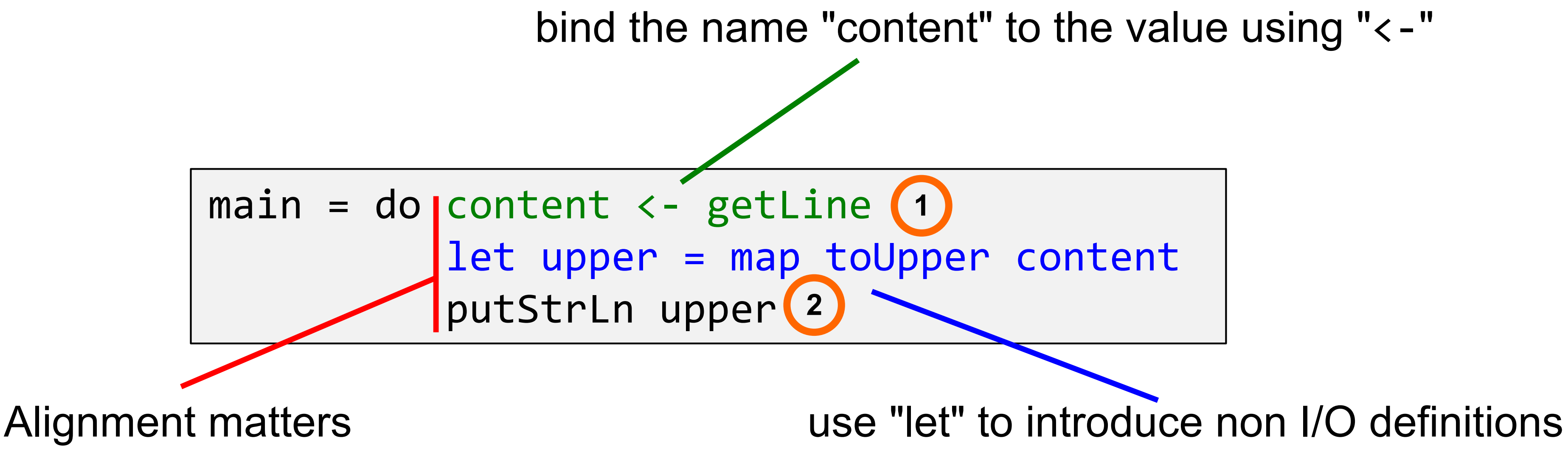
  - `getArgs` creates an IO action which returns a list of the command line arguments when executed

- **Example**

```
import System.Environment
import Data.List
main = do args <- getArgs
          putStrLn (intercalate "\n" args)
```

do Notation

- **do notation gives us a means to**
  - sequence I/O programs
  - bind names to the returned values



bind the name "content" to the value using "<-"

```
main = do content <- getLine
          let upper = map toUpper content
          putStrLn upper
```

Alignment matters

use "let" to introduce non I/O definitions

I/O programs are put in sequence: First 1 and then 2

do Notation Desugared

- **do notation is syntactic sugar for sequencing actions**
  - Original example

```
do putStr "Hi"
   name <- getLine
   putStrLn name
```

  - Rewritten using braces and semicolons

```
do { putStr "Hi" ;
     name <- getLine ;
     putStrLn name }
```

  - Desugared to applications of >> and >>=

```
(putStr "Hi") >>
getLine >>= \name ->
putStrLn name
```

do Notation Desugared cont.

- **(>>=) and (>>)**
  - describe what happens when effectful computations are sequenced
  - dubbed "programmable semicolons"

```
do {putStr "Hi" ;
    name <- getLine ;
    putStrLn name }
```

```
putStr "Hi" >> getLine >>= \name -> putStrLn name
```

- **`(>>) :: IO a -> IO b -> IO b`**
  - Sequencing actions, ignoring result of first action
- **`(>>=) :: IO a -> (a -> IO b) -> IO b`**
  - Sequencing actions, using result of first action to obtain second action

Preparations for Worksheet: Webclient

- **Start the build**

```
> cd WS_Webclient
> cabal build
```

```
On Windows use
"git bash" or "wsl"
```

```
cabal-version: >=1.10
name: WS-Webclient
version: 0.1.0.0
author: Daniel Kröni
build-type: Simple

executable webclient
  main-is: Main.hs
  build-depends: base >=4.13
                ,http-client-tls
                ,http-client
                ,bytestring
default-language: Haskell2010
```

`WS-Webclient.cabal` contains build related information

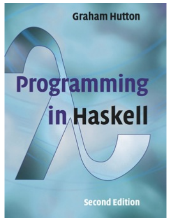
Name of the executable

Dependencies

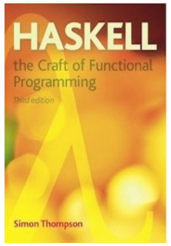
Worksheet: Webclient

```
$ ./webclient Brugg
Brugg: ⛅  +13°C
```

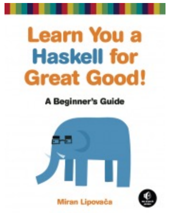
Further Reading



Chapter 10



Chapter 8



Book: Chapter 8

Web: http://learnyouahaskell.com/input-and-output