B3CC: Concurrency

02: Haskell refresh

Ivo Gabe de Wolff

Utrecht University

B3CC: Concurrency

02: Haskell ~~refresh~~ crash course

Ivo Gabe de Wolff

Utrecht University

Warming up

Overview

- Haskell is a…
  - Purely functional (side effects are strictly controlled) …
  - Statically typed (every term has a type, inferred & checked by the compiler) …
  - Polymorphic (functions and data constructors can abstract over types) …
  - Non-strict/lazy (only compute what is needed) …
- … programming language

Haskell programming

- Code lives in a file with a `.hs` extension
- Can be compiled or interpreted in a REPL
  - On the command line `ghci`
  - In a cabal project (like the practicals) `cabal repl`
  - Load a file from within GHCi `:load Main.hs`
- REPL includes a debugger and other useful functions (see also `:help`)
  - Get information on a given name `:info <name>`
  - … or its documentation `:doc <name>`
  - … or the type of an expression `:type <expression>`

Simple expressions

- You can type most expressions directly into GHCi and get an answer

```
Prelude> 1024 * 768
786432

Prelude> let x = 3.0
Prelude> let y = 4.0
Prelude> sqrt (x^2 + y^2)
5.0

Prelude> (True && False) || False
False
```

Strings

- Strings are in “double quotes”
    - They can be concatenated with ++

```
Prelude> “henlo”
“henlo”

Prelude> “henlo” ++ “, infob3cc”
“henlo, infob3cc”
```

Functions

- Calling a function is done by putting the arguments directly after its name
  - No parentheses are necessary as part of the function call

```
Prelude> fromIntegral 6
6.0
Prelude> truncate 6.59
6
Prelude> round 6.59
7
Prelude> sqrt 2
1.4142135623730951
Prelude> not (5 < 3)
True
Prelude> gcd 21 14
7
```

Lists

- Built-in, perhaps the most common datatype
  - Elements must all be the same type
  - Comma separated and surrounded by square brackets [ ]
  - The empty list is simply [ ]

```
Prelude> [2, 9, 9, 7, 9]
[2,9,9,7,9]

Prelude> [“list”, “of”, “strings”]
[“list”, “of”, “strings”]
```

Lists

- Can be defined by enumeration
  - Start at zero, end at ten

```
Prelude> [0..10]
[0,1,2,3,4,5,6,7,8,9,10]
```

  - Start at one, increment by 0.25, end at 3

```
Prelude> [1, 1.25 .. 3.0]
[1.0,1.25,1.5,1.75,2.0,2.25,2.5,2.75,3.0]
```

Lists

- Lists can be constructed & destructed one element at a time using `:` and `[]`

```
Prelude> 0 : [1..10]
[0,1,2,3,4,5,6,7,8,9,10]
```

- Strings are just lists of characters, so `:` and `++` also work on them

```
Prelude> “woohoo” == ‘w’:’o’:’o’:’h’:’o’:’o’:[]
True

Prelude> [1,2] ++ [3..5]
[1,2,3,4,5]
```

List comprehensions

- Syntactic sugar for constructing lists

```
Prelude> import Data.Char
Prelude> let s = “haskell”
Prelude> [ toUpper c | c ← s ]
“HASKELL”
```

- There can be multiple generators, separated by commas
  - Each successive generator refines the results of the previous

```
Prelude> [ (i,j) | i ← [1..3], j ← [1..i] ]
[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]
```

List comprehensions

- The latter can also be written using a *guard*

```
Prelude> [ (i,j) | i ← [1..3], j ← [1..i] ]
[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]

Prelude> [ (i,j) | i ← [1..3], j ← [1..3], j ⩽ i ]
[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]
```

List comprehensions

- Boolean guards can be applied to filter elements

```
Prelude> [ n | n ← [0..10], even n ]
[0,2,4,6,8,10]
```

Types

- Everything in Haskell has a type
  - So far we haven't mentioned any, but they were always there!
- What is a type?
  - A set of values with common properties and operations on them
    - Integer
    - Double
    - [Char]
    - (Char, Bool)
    - …

Functions

- Functions describe how to produce an output from their inputs
  - The type signature says that `leftPad` accepts two arguments as input and produces a string as output
  - `::` can be read as "has type"

```
leftPad :: Int → String → String
leftPad n rest = replicate n ' ' ++ rest
```

- Functions only depend on their arguments
  - The type signature is a strong promise

https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code

Functions

- Functions describe how to produce an output from their inputs
  - *Pattern matching* is used to decompose datatypes

```
length :: [a] → Int
length xs =
  case xs of
    []     → 0
    (y:ys) → 1 + length ys
```

Functions

- Functions can have multiple patterns
  - Patterns are matched in order, top-to-bottom
  - Only the first match is evaluated
  - Each pattern has the same type

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

Functions

- Don't implement redundant cases:

```haskell
length :: [a] → Int
length []     = 0
length [x]    = 1     ✗ redundant case
length (_:xs) = 1 + length xs
```

- Since `[x] = x : []`, it is already handled correctly by the other two cases

Order of patterns

- The first pattern that matches is executed

```haskell
fibonacci :: Int → Int
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
fibonacci 0 = 1
fibonacci 1 = 1
```

✗ infinite loop

```haskell
fibonacci :: Int → Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

✓ ok

Functions

- There are many useful *higher-order* functions available on lists
  - These take functions as arguments
  - Some examples:

```
map :: (a → b) → [a] → [b]

zipWith :: (a → b → c) → [a] → [b] → [c]

foldl :: (b → a → b) → b → [a] → b

scanl :: (b → a → b) → b → [a] → [b]

filter :: (a → Bool) → [a] → [a]
```

Type classes

- A set of types which share a number of operations
  - Lets you generalise functions
  - Similar to interfaces in C# or traits in Rust
    - not to be confused with classes in OO languages

```
(==) :: Eq a => a -> a -> Bool
```

- If `a` is a member of type class `Eq`, then `==` can compare two values of this type for equality

Local definitions

- Local bindings can be declared in `let` or `where` clauses
  - Once defined, these bindings can not change (immutable!)
  - Order does not matter

```
slope (x1,y1) (x2,y2) =
  let dy = y2-y1
      dx = x2-x1
   in dy/dx


slope (x1,y1) (x2,y2) = dy/dx
  where
    dy = y2-y1
    dx = x2-x1
```

Syntactic peculiarities

- Case matters:
  - Types, data constructors, and typeclass names, start with an uppercase letter
  - Everything else (variables, function names…) start with a lowercase letter
- Indentation matters:
  - Code which is part of some expression should be indented further in than the beginning of that expression
  - Don't use tabs (ever)

```haskell
average x y = xy / 2
  where
    xy = x + y
```

✓ ok

```haskell
average x y = xy / 2
  where
xy = x + y
```

✗ syntax error

Example: BSN

- How many BSNs are there?
  - A valid BSN must pass the 11-test
  - For a 9-digit number ABCDEFGHI then:

```
9A + 8B + 7C + 6D + 5E + 4F + 3G + 2H + (-1)I
```

  - … must be a multiple of eleven

Data types

Types

- Basic types
  - Int, Float, Double, Char …
- Composite types
  - Tuples: (Int, Float), (Char, Bool, Int, Int)
  - Lists: [Int], [Float], [(Int, Float)]
- We can create new names (aliases) for existing types

```
type String = [Char]
```

Algebraic datatypes

- You can define your own datatypes
  - For well-structured code
  - For better readability
  - For increased type safety
- Enumeration types
  - Defines a *type* `Bool` and two new *type constructors* `False` and `True`

```
data Bool = False | True
  deriving (Show, Read, Eq, Ord)
```

Algebraic datatypes

- Datatypes can have type parameters

```haskell
data Vec2 a = Vec2 a a
  deriving (Eq, Show)
```

  - Write a function to point-wise add two vectors

Algebraic datatypes

- Data constructors can also have arguments

```
data Shape
  = Square Double
  | Rectangle Double Double      -- length, width
  | Circle Double                -- radius
  deriving (Eq)
```

- Write the function `area :: Shape → Double`

Algebraic datatypes

- Datatypes can be recursive

```
data Tree a
  = Node (Tree a) (Tree a)
  | Leaf a
```

- Write a function `sumTree` that sums all of the values stored in the tree

- Write a function `toList :: Tree a → [a]`

Monads

Monads

A ***monad*** *in X is just a* ***monoid*** *in the category of* ***endofunctors*** *of X, with product × replaced by* ***composition of endofunctors*** *and* ***unit*** *set by the* ***identity endofunctor****.*

— Mac Lane

Monads *are like* ***burritos***

— Mark Dominus

http://tiny.cc/b3d8fz
http://blog.plover.com/prog/burritos.html

Monads

Warm fuzzy thing

— Simon Peyton Jones

Monads

- Remember, Haskell is pure
  - Functions can't have side effects
  - Functions take in inputs and produce outputs
  - Nothing happens in-between (no modification of global variables)
- However, input/output is not at all pure

https://clips.twitch.tv/TawdryProductiveLobsterMingLee-FsU2cH2bDUeSkKwk

Input/Output

- The IO monad serves as a glue to bind together the actions of the program
  - Every IO action returns a value
  - The type is "tagged" with `IO` to distinguish actions from other values

```
getChar :: IO Char

putChar :: Char → IO ()
```

Input/Output

- The keyword `do` introduces a sequence of statements, executed in order
  - An action (such as `putChar`)
  - A pattern binding the result of an action with ← (such as `getChar`)
  - A set of local definitions introduced using `let`

```
main :: IO ()
main = do
  c1 ← getChar
  let c2 = chr (ord c1 + 1)
  putChar c2
```

- `main` is the entry point of the program and must have type `IO ()`

Input/Output

- We can invoke actions and examine their results using do-notation
  - We use `return :: a → IO a` to turn the ordinary value into an action
  - `return` is the opposite of ←

```
ready :: IO Bool
ready = do
  c ← getChar
  return (c == 'y')
```

Input/Output

- Each do introduces a single chain of statements. Any intervening construct must introduce a new `do` to initiate further sequences of actions

```
getLine :: IO String
getLine =
```

Input/Output

- `return` admits values into the realm of ordinary `IO` actions; can we go the other way?
  - No!
  - Consider the function:

```
f :: Int → Int → Int
```

  - It can not possibly do any `IO`, because that does not appear in the return type
    - Safe to execute concurrently!

Programming with actions

- `IO` actions are ordinary Haskell values
  - They can be passed to functions, stored in structures, etc…

```
todoList :: [IO ()]
todoList =
  [ putStr "henlo, "
  , do
      l ← getLine
      putStrLn l
  ]
```

  - This list does not invoke any actions, it simply holds them

```
sequence_ :: [IO ()] → IO ()
sequence_ = ...
```

Programming with actions

- Side effects are isolated into IO actions
- Pure code is separated from impure operations
- IO actions exist only within other IO actions

Photo by Justin Veenema