Programming Paradigms

Summer Term 2017

9th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Programming Paradigms

Input and output in Haskell

Input/output in Haskell, very simple example

- Even in declarative languages, there should be some (disciplined) way to embed "imperative" commands like "print something to the screen".
- In pure functions, no such interaction with the OS / user / … is possible.
- But there is a special do-notation in Haskell that enables interaction, and from which one can call "normal" functions.

Simple example:

```
prod :: [Int] → Int
prod [ ]        =  1
prod (x : xs)  =  x * prod xs
```

pure function

```
main = do n ← readLn
          m ← readLn
          print (prod [n .. m])
```

"main program"

```
Input  → 5
Input  → 8
Output → 1680
```

program execution

Principles of input/output in Haskell: IO types …

- There is a predefined type constructor IO, such that for every concrete type like Int, Bool, [(Int, Tree Bool)] etc., the type IO Int, IO Bool, … can be built.

- The interpretation of a type IO a is that elements of that type are not themselves concrete values, but instead are (potentially arbitrarily complex) sequences of input and output operations, and computations depending on values read in, by which ultimately a value of type a is created.

- An (independently executable) Haskell program overall always has an "IO type", usually simply main :: IO ().

- To actually create "IO values", there are predefined primitives (and one can recognize their IO-related character based on their types):

```
getChar :: IO Char
getLine :: IO String
readLn :: Read a ⇒ IO a
```

```
putChar :: Char → IO ()
putStr, putStrLn :: String → IO ()
print :: Show a ⇒ a → IO ()
```

Principles of input/output in Haskell : … and do-notation

- To combine IO-computations (i.e., to build more complex action sequences based on the IO primitives), there is the do-notation.

- General form:

$$
\begin{array}{l}
\text{do } cmd_1 \\
\quad x_2 \leftarrow cmd_2 \\
\quad x_3 \leftarrow cmd_3 \\
\quad cmd_4 \\
\quad x_5 \leftarrow cmd_5 \\
\quad \ldots
\end{array}
$$

The do-block as a whole has the type of the last cmd_n. For that last command, generally no x_n is present.

where each cmd_i has an IO type and to each x_i (if explicitly present) a value of the type encapsulated in the cmd_i will be bound (and can, from this point onwards, be used in the whole do-block), namely exactly the result of executing cmd_i.

- Often also useful (e.g., at the end of a do-block): a predefined function return :: a $\rightarrow$ IO a that simply yields its argument, without any actual IO action.

Principles of input/output in Haskell: IO types and do-notation

- A slightly more complex example:

```
dialog = do putStr "Input: "
            s ← getLine
            if s == "end"
               then return ()
               else do let n = read s
                       putStrLn ("Output: " ++ show (n * n))
                       dialog
```

- What is never ever, at all, possible or allowed is to directly extract (beyond the explicit sequentialisation and binding structure within do-blocks) the encapsulated value from an IO computation, i.e., to simply turn an IO a value into an a value.

- Beside the shown example primitives for console input/output, there are primitives and libraries for file input/output, network communicaton, GUIs, …

- Of course, also in the context of IO related computations, all features and abstraction concepts of Haskell are available, so we define functions with recursion, use data types, polymorphism, higher-order, …

User defined “control structures”

- As emphasized, also in the context of IO related computations, all abstraction concepts of Haskell are available, particularly polymorphism and definition of higher-order functions.

- This can be employed for things like:

```
while :: (a → Bool) → (a → IO a) → (a → IO a)
while p body  =  loop
   where  loop x  =  if  p x  then  do  x' ← body x
                                        loop x'
                                 else  return x
```

- So what will be the behaviour/output of the following expression?

```
> while (< 10) (\n → do {print n; return (n + 1)}) 0
```

Functional programming in Haskell: summary (1)

- principle of functional programming:
  - specification = collection of function definitions
  - function definition = system of defining equations
  - operationalisiation = step-wise reduction of expressions to values

- expressions:
  - constants, variables, structured expressions: lists, tuples
  - function applications
  - list comprehensions

- systems of defining equations:
  - left- and right-hand sides with certain restrictions (e.g., concerning variable use)
  - multiple parameters, pattern matching
  - guards

- syntactic particularities of Haskell:
  - deviation from mathematical notation in function syntax
  - local definitions (let, where)
  - layout rule

Functional programming in Haskell: summary (2)

- reduction / evaluation:
  - pattern matching, selecting the case to use, recursion
  - lazy evaluation
  - special role of IO, do-blocks

- lists:
  - sequential notation vs. tree representation (:), pattern matching
  - specific list functions (e.g., length, ++, !!)
  - arithmetic sequences, infinite lists, list comprehensions

- types (strong typing, type checking, type inference):
  - data types
    - base types (Integer etc.)
    - structured types (lists, tuples)
    - algebraic data type declarations, constructors
  - polymorphic types, type variables
  - function types
    - function type declarations, currying
  - type classes, declarations, instance definitions

Functional programming in Haskell: summary (3)

- higher-order functions:
  - functions as parameters and/or as results
  - partial application, sections
  - lambda-expressions
  - higher-order functions on lists: map, filter, foldr, …

- use of explicit recursion schemes (capturing structural recursion as foldr)