

Introduction to functional programming with Haskell

Winter Term 2015

1. Theoretical Part

- λ -calculus
- category theory

2. Practical Part

- syntax of Haskell
- lists and list comprehension
- types and type classes
- recursion
- higher order functions

Books

- [1] Richard Bird: Thinking Functionally with Haskell, 2014
- [2] Miran Lipovača: [Learn you a Haskell for Great Good](#)

Introduction

Lecture 1 (12.10.2015)

Imperative programming	Functional programming
↑ Turing machine <code>int product = 0;</code> <code>for(int i = 2; i <= n; ++i)</code> <code>product *= i;</code> <code>return product;</code>	↑ λ -calculus <code>factorial(0) = 1</code> <code>factorial(n) = factorial(n-1)*n</code> Haskell: <code>product[1..n]</code>

$$f(x) = x^2 \qquad f : D \rightarrow F$$

Three keypoints of functional programming:

- FP is a method of program constructions that emphasize functions and their application rather than commands and their executions
 - PF uses “simple” mathematical notation that allows concise description of problems
 - PF has strong mathematical basis
-

The most important properties of Haskell:

1. High Order Functions (HOF) If functions are treated as first class values in a language - allowing them to be stored in data-structures, passed as arguments \rightarrow return as results \rightarrow referred as HOF. Function map takes a function and applies it to a list:
Haskell: `(a -> a) -> [a] -> [a]`
Example: consider the following function twice $f(x) = f(f(x))$
2. Nonstrict Semantic (Lazy Evaluation) Lazy evaluation or call by need is an evaluation strategy which delays the evaluation of an expression until its value is needed, and which also avoids repeated evaluations
 $f(2, \frac{3}{0})$ // something
3. Data abstraction
4. Equation and pattern matching

Further features

- pure functional
 - `"Hello" ++ "World"`
 - `"Hello" ++ getLine` // compiler Error -> `getLine` has type `IO-String`
- type inference
- something
- lazy
- packages

λ -calculus

What is a foundation of mathematics?

- Set theory was introduced by Georg Cantor in 1874-1884
 - $A = \{1, 2, 3, \dots\} \rightarrow \mathbb{N}$
 - $A = \{x | x \in F\}$ // A contains x for $x \in F$
 $\{x | x < 10 \ \&\& \ x > 2, x \in \mathbb{Z}\}$
 $\rightarrow \{3..9\}$
 - Russel Paradox founded in 1901
 - shows that the naive set theory is inconsistent
 - let R be a set of all sets that are not member of themselves
 - ‘The barber is a man in town who shaves all those, and only those, men in town who do not shave themselves’
 - Axiomatic set theory
 - ZFC(1908-1922) and NBG(1925-1945)
 - λ -calculus introduced in 1930 by Alonso Church
 - category theory by Eilenberg and MacLane (1942-1945)
 - Paper [A Short introduction to \$\lambda\$ -calculus](#) by A. Jung
-

Lecture 2 (19.10.15)

Typed and untyped:

- λ -calculus developed by Anlonzo Church in 1930s
 - One of original goals was to construct a formal system for the foundation of mathematics by having a system of functions together with a set of logical notions
 - When it tuned out that the constructed system was inconsistent Church this program was forgotten by him.
 - Then later Church separated the consistent subsystem and focussed on comptability.
 - known as untyped λ -calculus
 - Systems using λ -calculus:
 - LISP (not based on λ -calculus)
 - ML
 - Haskell
-

untyped λ -calculus

- The λ -calculus is a calculus of pure functions
- Let us consider the following function: $f(x) = x^2 + x + 1$
- We speak in terms of a name the argument and the result:
 - f is a function which, when applied to any argument x returns $x^2 + x + 1$
 - $g(x) = x^2 + x + 1 \quad \hat{=} \quad g(y) = y^2 + y + 1$
- can we describe and define functions without giving them names?
- λ -calculus allows us to do so
- λ -calculus is a notation for functions (short but cryptic) which takes its origin in mathematical logic
- Expressions in λ -calculus are written in the strict prefix form, that is, there are no infix and postfix operations.
- Furthermore functions and their arguments are simply written next to each other
 - examples:
 - * $\sin(x) \longrightarrow \sin x$
 - * $x + 4 \longrightarrow +x\ 4$
 - * $x^2 \longrightarrow * x\ x$
- Brackets are used only to enforce a special grouping:
 - example:
 - * $\sin(x) + 4 \longrightarrow (\sin x)\ 4$
- If an expression contains a variable (x), then one can form the function which obtains by considering the relationship between concrete values of x and the resulting value of expression
- In mathematics, function formation is sometimes written as an equation, $f(x) = 3x$, or sometimes as a mapping $f : x \rightarrow 3x$
- In the λ -calculus a special notation is available which dispenses the need to give a name to the function and which easily scales up to more complicated function definitions.
- In the example above, we will write $3x$ as $* 3\ x$ and turn it into a function by preceding $\lambda x. \rightarrow \lambda x. * 3\ x$
- The letter λ has a role similar to keyword **function** in some programming languages
- The variable which comes after λ is not part of an expression, but a formal parameter.
- The $.$ tells that here starts the function body.
 - We have λ variable . function body
- Let us consider a function $x^2 + y$
- This can be thought of as a function on x (y is fixed in this case), or it can be a function on y (x is fixed), and it can be a function on x and y

- To make a distinction between these cases, λ -calculus provides the so called abstraction.
 - To see how it works, let us consider a function $x \mapsto x^2 + y$
 - To make a clear distinction between the bound variable x and the unbound variable y we introduce an explicit variable binder, and we write:
 - $\lambda x. + (* x x)y$ // $x \rightarrow$ bounded variable
 - $\lambda z. + (* z z)y$
-

The pure λ -calculus

- The function which has been λ notation can itself be used in an expression.
- For example, the application of the function $3x$ to a value 4 is written as $(\lambda x. * 3 x)4$
- As we already mentioned the representation by means of λ bindings offers a general way to write functions.
- In case a function takes several arguments we need to use several bound variables.
 - example: $\lambda x. \lambda y. + (* x x)y$
 - $(\lambda x. \lambda y. + (* x x)y)3 \rightarrow \lambda y. (* 3 3)y$
- For applying a function to its argument we use juxtaposition:
 - if F is a function expression and A is an argument expression, then $(F A)$ denotes the application of F to A .
 - * example: $f(x) = x^2 + 1$ $A = \text{sin} y$
 - * $f(A) = (\text{sin} y)^2 + 1$
- Although it is not strictly necessary, it will be convenient to introduce an abbreviation for λ terms.
- If we abbreviate our function term to F .
 - $F =^{def} \lambda x. * 3 x$
- now we write $F 4$, instead of $(3. \text{formel})$
- Suppose the body of a function consists of another function:
 - $N =^{def} \lambda y. * (\lambda x. * y x)$
- If we apply this function to the value 3 then we get back $\lambda x. * 3 y$
- In other words N is a function which when applied to a number returns another function.
- If we want to underline that N is a function of two arguments then we should leave the brackets out
- That means we have $\lambda x. \lambda y. * y x$ or, as typically used we can omit the second λ :

– $\lambda y\ x.\ * \ y\ x$

- Function formation and function application are all there is in pure λ -calculus.
- They can be mixed freely and used as often as desired or needed
- Which is another way of saying that λ terms are constructed according to the grammar

– $M ::= c \mid x \mid M\ N \mid \lambda x.\ M$

- where c represents any constant we might use in a λ -term, such as numbers or arithmetic operations (a term without a constant is called pure $\lambda x.x$), the letter x represents any of infinitely many variables
- $M\ N$ is an application of one term M to another term N
- $\lambda x.\ M$ is an abstraction

The evaluation of λ -expressions

- We have seen how the λ -notations can be used to represent functional expressions and we are going to define conversion rules of λ -calculus which will describe how to evaluate an expression.
- or how to transform an expression from its initial state into final state
- The simplest type of λ -expression is a constant.
- Constants are self-defining, meaning that they cannot be transformed into any simpler expression
- The expression 5 produces the number 5
- The following table provides predefined constants:

Constant	Meaning
$0, 1, -1, -2, 2, \dots$	The set of integers
TRUE, FALSE	Boolean constants
$+, -, *$	mathematic operations
$=, \neq, <, >$	boolean functions
SUCC, PRED	successor and predecessor one int
EQO	returns TRUE if $int = 0$
COND	conditional functions
CONS	list constructor
NIL	empty list
HD	function returning head of a list
TL	function returning tail of a list
IE = NULL	if list is NIL returns TRUE
TUPLE	builds a n-Tuple of expressions
INDEX	tuple indexing function

Lecture 3 (26.10.2015)