Introduction to functional programming with Haskell

Winter Term 2015

1. Theoretical Part

- λ-calculus
- category theory

2. Practical Part

- syntax of Haskell
- lists and lists 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	λ-calculus
`int product = 0;`	`factorial(0) = 1`
`for(int i = 2; i <= n; ++i)`	`factorial(n) = factorial(n-1)*n`
`product *= i;`	Haskell: `product[1..n]`
`return product;`	

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

Three keypoints of functional progamming:

- FP is a method of progam constructions that empasize functions and their application rather than commands and their executions
- PF uses "simple" mathematical notation that allows concise description of problems
- PF has strong mathemathical 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 → return as results → referred as HOF. Function map takes a function and approximates 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
  - $A = \{1, 2, 3, ...\} \rightarrow$ IN
  - $A = \{x | x \in F\}$ //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 λ-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 $x^2 + x + 1$
  - $g(x) = x^2 + x + 1 \quad \widehat{=} \quad g(y) = y^2 + y + 1$
- 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 $f : x \to 3x$
- 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 $\lambda x. \to \lambda x. *\ 3\ x$
- 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 unbounded 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 funciton 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: $f(x) = x^2 + 1 \qquad A = siny$
    - $f(A) = (siny)^2 + 1$
- 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. 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 inifinitely manay 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
$0, 1, -1, -2, 2, ...$	The set of integers
`TRUE, FALSE`	Boolean constants
$+, -, *$	mathematic operations
$=, \neq, <, >$	boolean functions
`SUCC, PRED`	successor and predecessor one int
`EQ0`	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)

- An application of a constant function can be tranformed by using the builtin rules if a sufficient amount of arguments is provided for that constant.
- These builtin rules are often referred to as δ-rules.
  - Example: $+\ 1\ 3$ // read as "$+\ 1\ 3$ reduces to 4"
  - to evaluate this expression we need to apply a δ-rule for arithmetic function $+$ as follows:
  - $+\ 1\ 3 \rightarrow_\delta 4$
  - This process of simplification is called <u>reduction</u>
- In general the arguments of such a function may not be in the required form for the reduction to take place immediately
  - Example:
  - $*\ (+\ 1\ 2)(-4\ 1)$
  - to reduce this expression we must first reduce the argument expression of the multiplication function
  - It can be done as follows: $*\ (+\ 1\ 2)(-4\ 1) \rightarrow_\delta -x(+\ 1\ 2)B \rightarrow_\delta *\ 3\ 3 \rightarrow_\delta 9$
  - The most interesting reduction rule describes how to apply lambda abstraction.
- Let us consider the following application:
  - $(\lambda x.*\ x\ x)2$
  - In the LC the evaluation of such expressions consists in the textual replacement of a formal parameter in the body of a function by the actual parameter supplied.
  - In the above example we get: $(\lambda x.*\ x\ x)2 \rightarrow_\beta *\ 2\ 2\ \rightarrow_\delta 4$
  - <u>Def:</u> The process of copying the body of an abstraction, replacing all occurence of the bound variable by the argument expression is known as β-reduction
  - The term 'reduction' is used in a sense of simplification since we notionally cancel the λ, the bound variable and the argument expression and return just a modified form of a body.
- The reduction of an abstraction applied to an argument may yield another abstraction, in that case the process can maybe repeated
  - Example: In the folloring Expression :

    $*\ ((\lambda x.\lambda y.+\ x\ y)7)8$
  - we start by substituting the 7 for x in the body of outermost abstraction *i.e* $\lambda y.+\ x\ y$ yields:

    $*\ (\lambda y.+\ z\ y)8$

  - and end up with another abstraction to an argument expression.
  - We can reduce
  - The expression being reduced is called the redex(short for reducable expression), and for the time being we will state that the process of reducing a λ-expression consists of repeatedly reducing the redexes of the expression until no more redexes exist.

- We will sometimes qualify the word "redex" according to the type of the rule which can be used to simplify it δ-redex – δ-rule, β-redex – β-reduction
- Let us consider the following expression:
  - $(\lambda x.\ x)(+\ 2\ x)$
- This is clear thta the x within the inner abstraction $(\lambda x.x)$ refers to a different bound variable, than in its argument expression $(+\ 1\ x)$.
- If we apply this expression to an argument, (lets say 1) then it would be wrong to perform β-reduction als follows:
  - $(\lambda x.(\lambda x.x)(+\ 1\ x))1 \rightarrow (\lambda x.1)(+\ 1\ 1) \rightarrow 1$
- When we apply a β-reducion, we must be careful not to substitute inside an abstraction if the bound variable of that abstraction has the same name as the variable being substituted.
- If it does have the same name, then we must leave this (inner) abstraction unchanged . It is possible to avoid such type of name clash by renaming the variables concerned - in this case one of x's - so as to make each name unique.
- Such renaming is called α-conversion
- And expressions which are α-convertable, it means equal up to variable renaming, are called alphabetically equivalent.
- Let us consider the following example of β-reduction:
  - $(\lambda f.\lambda x.f\ 4\ x)(\lambda y.\lambda x.+\ x\ y)3$
- Step 1: Reduce the (only) redex by substituting the argument $(\lambda.(\lambda x\ x.x\ y))$ for f to the body of the abstraction $(\lambda f.\ \lambda x.\ f\ y\ x) \rightarrow (\lambda x(\lambda y.\ \lambda x.\ x\ x\ y)4x)3$
- Step 2: (Arbitory) chose redex which which results in 3 being substituted for x to the body expression $(\lambda y.$
  $lambdax.\ +\ x\ y)4x$, but do not substitute beyond the λx)written the inner abstraction $\rightarrow (\lambda y.\ \lambda x.\ +\ x\ y)4\ 3$
- Step 3: Reduce the (only) redex $(\lambda y.\ \lambda x.\ +\ x\ y)4 \rightarrow (\lambda x.\ x\ x\ 4)3$
- Step 4: Reduce the (only) redex:
  - $\rightarrow +\ 3\ 4$
- Step 5: Apply the DELTA-rule for " + "
  - $\rightarrow 7$
- Notice that the first reduction applied substituting f with the function $(\lambda x.\lambda y.\ +\ x\ y)$

- The abstraction containing f corresponds to a higher order function in the equivalent source program.
- At Step 2 there was a choice of two redexes to reduce and we arbitrarily chose chose to reduce the outer one first.
- However we could have chosen to do a β-reduction on the inner redex first as follows $(\lambda x(\lambda y.\ \lambda x.\ +\ x\ y)4x)3 \rightarrow (\lambda x.(\lambda x.\ +\ x\ 4)x)3 \rightarrow (\lambda x.\ +\ x\ 4)3 \rightarrow +\ 3\ 4 \rightarrow 7$

Reduction order and normal forms

- Def: A λ-expression is said to be in the normal form if it cannot be further reduced.
- In other words, it is in normal form if it contains no redexes.
- Normal form corresponds to the idea of the "end" of computation in the programming sense.
- This immediately suggests a naive evaluation scheme
```
while there are more redexes; do
reduce one of the redexes
end;
```
- The expression now is in normal form
- The problem with it is that there may be more than one redex within an expression
- This leads us to the question:
  - Which redex to reduce next?
  - Let us consider the following expression:
    - $(\lambda x.\ \lambda y.\ y)((\lambda z.\ z\ z)(\lambda z.\ z\ z))$
  - Here wre have to redexes:
    - $(\lambda x.)OANAFRAGEN$
  - and
    - $(\lambda z.\ z\ z)(\lambda z.\ z\ z)$
  - If we pick the second of these two to reduce first we get the following reduction sequence:
    - $(\lambda z.\ z\ z)(\lambda z.\ z\ z) \rightarrow (\lambda z.\ z\ z)(\lambda z.\ z\ z) \rightarrow \ldots$ which is not terminated
  - Which is not terminating
  - In the other hand if we chose the first redex we get
    - $((\lambda x.\ \lambda y\ y)(\lambda z.\ z\ z)(\lambda z.\ z\ z)) \rightarrow \lambda y.\ y\ \ldots$ which terminates in first step

- Before a discussion of reduciton order let us introduce some definitions
- Def:
  - The leftmost redex is that redex whose λ (or primitive function identifier in the case of DELTA-redex) is textually to the left of all other redexes within the expression Yeah, (Similary for the rightmost redex) AhaAha
- Def:
  - An outermost redex is defined to be a redex which is not contained within any other redex.
- Def:
  - An innermost redex is defined to be a redex which containes no other redex.
  - these are two of the most important reduction strategies:
  - Normal order:
    - Under this strategy the leftmost outermost redex is always reduced first. Normal order of reduction is normalizing strategy in the sense that a term as a normal form then this order of reduction will find it.

Applicative Order:

- Under this strategy the leftmost innermost redex is always reduced first.
- It turns out that applicative order of reduction is not normalization

Lecture (